

Java Programming (R20CSE2204)

UNIT - I

Object-Oriented Thinking- A way of viewing world – Agents and Communities, messages and methods, Responsibilities, Classes and Instances, Class Hierarchies- Inheritance, Method binding, Overriding and Exceptions, Summary of Object-Oriented concepts. Java buzzwords, An Overview of Java, Data types, Variables and Arrays, operators, expressions, control statements, Introducing classes, Methods and Classes, String handling.

Inheritance– Inheritance concept, Inheritance basics, Member access, Constructors, Creating Multilevel hierarchy, super uses, using final with inheritance, Polymorphism-ad hoc polymorphism, pure polymorphism, method overriding, abstract classes, Object class, forms of inheritance- specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance.

What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Java OOPs Concepts

- Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.
- **Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.
- **Smalltalk** is considered the first truly object-oriented programming language.
- The popular object-oriented languages are **Java**, **C#**, **PHP**, **Python**, **C++**, etc
- The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

Editions of Java

Each edition of Java has different capabilities. There are three editions of Java:

- **Java Standard Editions (JSE):** It is used to create programs for a desktop computer.

- **Java Enterprise Edition (JEE):** It is used to create large programs that run on the server and manages heavy traffic and complex transactions.
- **Java Micro Edition (JME):** It is used to develop applications for small devices such as set-top boxes, phone, and appliances.

Types of Java Applications

There are four types of Java applications that can be created using Java programming:

- **Standalone Applications:** Java standalone applications uses GUI components such as AWT, Swing, and JavaFX. These components contain buttons, list, menu, scroll panel, etc. It is also known as desktop alienations.
- **Enterprise Applications:** An application which is distributed in nature is called enterprise applications.
- **Web Applications:** An applications that run on the server is called web applications. We use JSP, Servlet, Spring, and Hibernate technologies for creating web applications.
- **Mobile Applications:** Java ME is a cross-platform to develop mobile applications which run across smartphones. Java is a platform for App Development in Android.

Applications

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used.

Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irtc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

OOPs (Object-Oriented Programming System) / Object Oriented Thinking

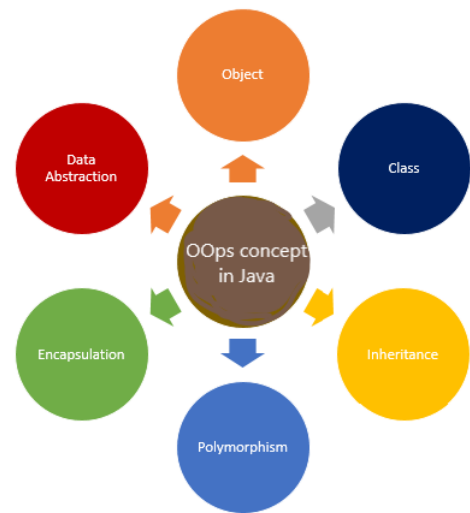
Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.

It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition



Object

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an **instance of a class**.
- An object contains an address and takes up some space in memory.
- Objects can communicate without knowing the details of each other's data or code.
- The only necessary thing is the type of message accepted and the type of response returned by the objects.



Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

- **Collection of objects** is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class doesn't consume any space.

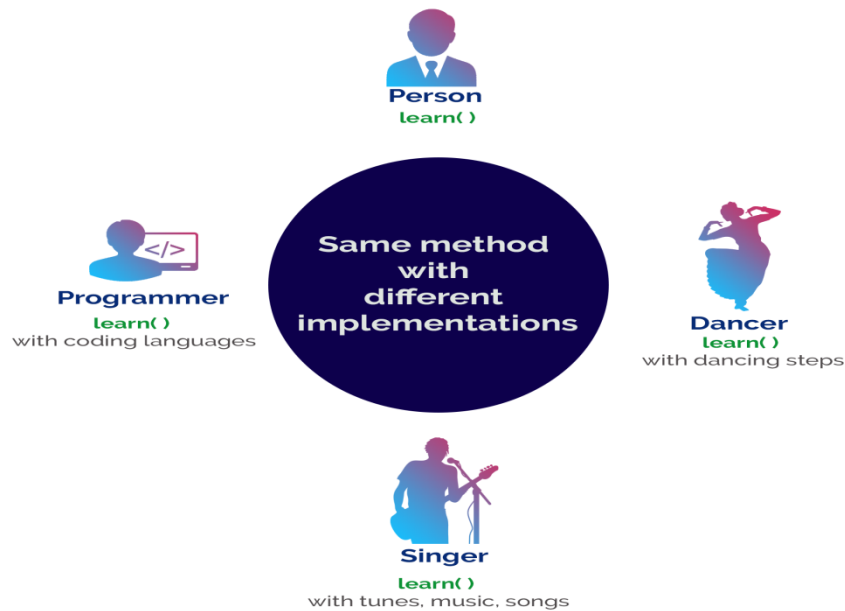
Inheritance

- When one object acquires all the properties and behaviors of a parent object, *it is known as inheritance*.
- It provides code reusability.
- It is used to achieve runtime polymorphism.



Polymorphism

- If *one task is performed in different ways*, it is known as polymorphism.
- For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.



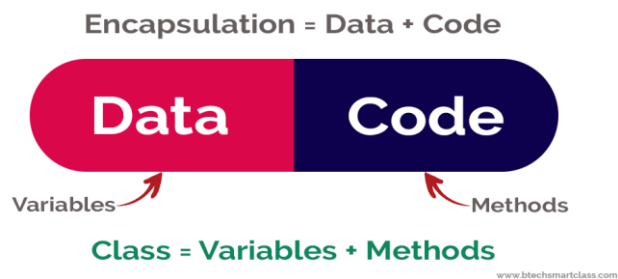
Abstraction

- *Hiding internal details and showing functionality* is known as abstraction.
- For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.



Encapsulation

- Binding (or wrapping) code and data together into a single unit are known as encapsulation.
- For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation.
- Java bean is the fully encapsulated class because all the data members are private here.
- Encapsulation is the process of combining data and code into a single unit (object / class). In OOP, every object is associated with its data and code.
- In programming, data is defined as variables and code is defined as methods. The java programming language uses the class concept to implement encapsulation.



Coupling

Coupling refers to the knowledge or information or **dependency of another class**. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

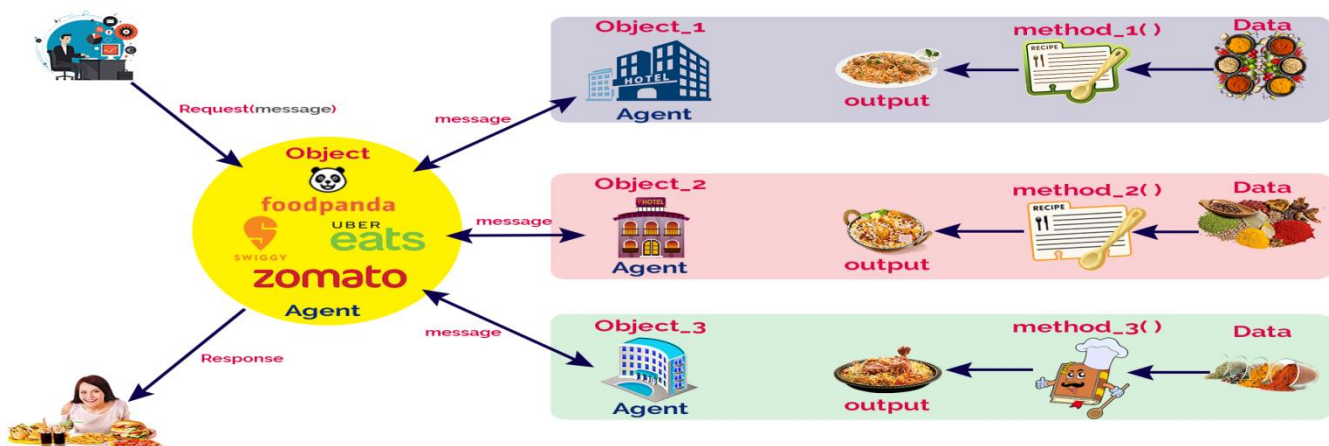
A way of viewing world

A way of viewing the world is an idea to illustrate the object-oriented programming concept with an example of a real-world situation.

Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?

To solve the problem, let me call zomato (an **agent** in food delivery community), tell them the variety and quantity of food and the hotel name from which I wish to deliver the food to my family members. Look at the following image.

A way of viewing world with OOP



Agents and Communities

To solve my food delivery problem, I used a solution by finding an appropriate agent (Zomato) and pass a message containing my request. It is the responsibility of the agent (Zomato) to satisfy my request. Here, the agent uses some method to do this. I do not need to know the method that the agent has used to solve my request. This is usually hidden from me.

So, in object-oriented programming, problem-solving is the solution to our problem which requires the help of many individuals in the community. We may describe agents and communities as follows.

An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community.

In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

Messages and Methods

To solve my problem, I started with a request to the agent zomato, which led to still more requestes among the members of the community until my request has done. Here, the members of a community interact with one another by making requests until the problem has satisfied.

In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action. The receiver is the object to whom the message was sent. In response to the message, the receiver performs some method to carry out the request. Every message may include any additional information as arguments.

In our example, I send a request to zomato with a message that contains food items, the quantity of food, and the hotel details. The receiver uses a method to food get delivered to my home.

Responsibilities

In object-oriented programming, behaviors of an object described in terms of responsibilities.

In our example, my request for action indicates only the desired outcome (food delivered to my family). The agent (zomato) free to use any technique that solves my problem. By discussing a problem in terms of responsibilities increases the level of abstraction. This enables more independence between the objects in solving complex problems.

Classes and Instances

In object-oriented programming, all objects are instances of a class. The method invoked by an object in response to a message is decided by the class. All the objects of a class use the same method in response to a similar message.

In our example, the zomato a class and all the hotels are sub-classes of it. For every request (message), the class creates an instance of it and uses a suitable method to solve the problem.

Classes Hierarchies

A graphical representation is often used to illustrate the relationships among the classes (objects) of a community. This graphical representation shows classes listed in a hierarchical tree-like structure. In this more abstract class listed near the top of the tree, and more specific classes in the middle of the tree, and the individuals listed near the bottom.

In object-oriented programming, classes can be organized into a hierarchical inheritance structure. A child class inherits properties from the parent class that higher in the tree.

Method Binding, Overriding, and Exception

In the class hierarchy, both parent and child classes may have the same method which implemented individually. Here, the implementation of the parent is overridden by the child. Or a class may provide multiple definitions to a single method to work with different arguments (overloading).

The search for the method to invoke in response to a request (message) begins with the class of this receiver. If no suitable method is found, the search is performed in the parent class of it. The search continues up the parent class chain until either a suitable method is found or the parent class chain is exhausted. If a suitable method is found, the method is executed. Otherwise, an error message is issued.

Summary of Object-Oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, identified the following characteristics as fundamental to OOP [Kay 1993]:

1. Everything is an object.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own memory, which consists of other objects.
4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

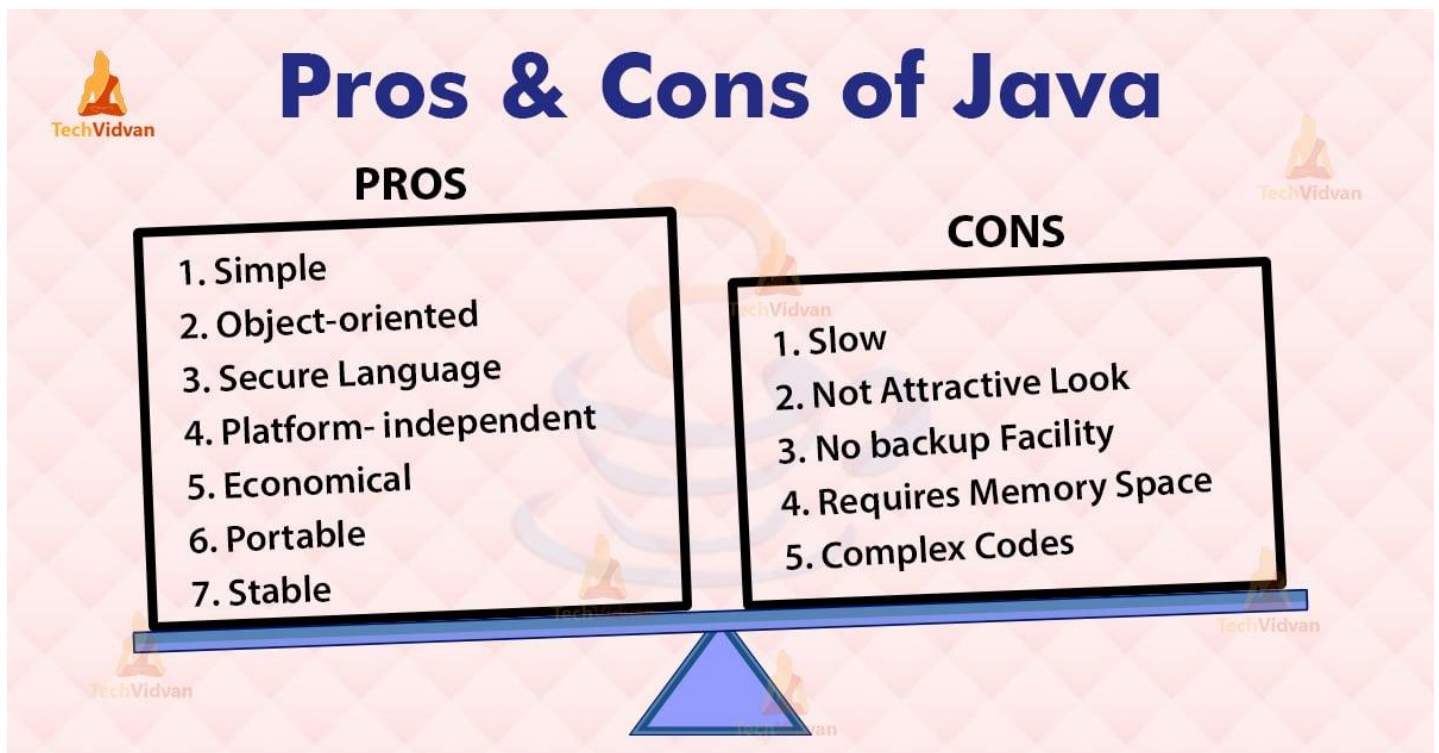
Advantages and Disadvantages of Java

Java has been consistently holding the top position of the TIOBE index among all other programming languages. Though many new languages have been discovered, the fame of Java never goes down. Java has been ruling over all other languages for more than 20 years.

The majority of experts cannot deny the fact that Java is one of the most powerful and effective languages ever created and is the most widely used programming language in many areas.

But, we also know that every coin has two sides; similarly, Java can not run away from this fact and therefore it has also got its own limitations and benefits; what we call it is a pros and cons of Java.

In this article, we will acquaint you with the prominent advantages and disadvantages of Java, which will help you have a clear vision of this language.



Advantages of Java

Java is an Object-Oriented and a general-purpose programming language that helps to create programs and applications on any platform. Java comes up with a bundle of advantages that lets you stick with it.

Let's discuss the pros of using Java programming language.

1. Java is Simple

Any language can be considered as simple if it is easy to learn and understand. The syntax of Java is straightforward, easy to write, learn, maintain, and understand, the code is easily debuggable.

Moreover, Java is less complex than the languages like C and C++, because many of the complex features of these languages are being removed from Java such as explicit pointers concept, storage classes, operator overloading, and many more.

2. Java is an Object-Oriented Programming language

Java is an object-oriented language that helps us to enhance the flexibility and reusability of the code. Using the OOPs concept, we can easily reuse the object in other programs.

It also helps us to increase security by binding the data and functions into a single unit and not letting it be accessed by the outside world. It also helps to organize the bigger modules into smaller ones so they are easy to understand.

3. Java is a secure language

Java reduces security threats and risks by avoiding the use of explicit pointers. A pointer stores the memory address of another value that can cause unauthorized access to memory.

This issue is resolved by removing the concept of pointers. Also, there is a Security manager in Java for each application that allows us to define the access rules for classes.

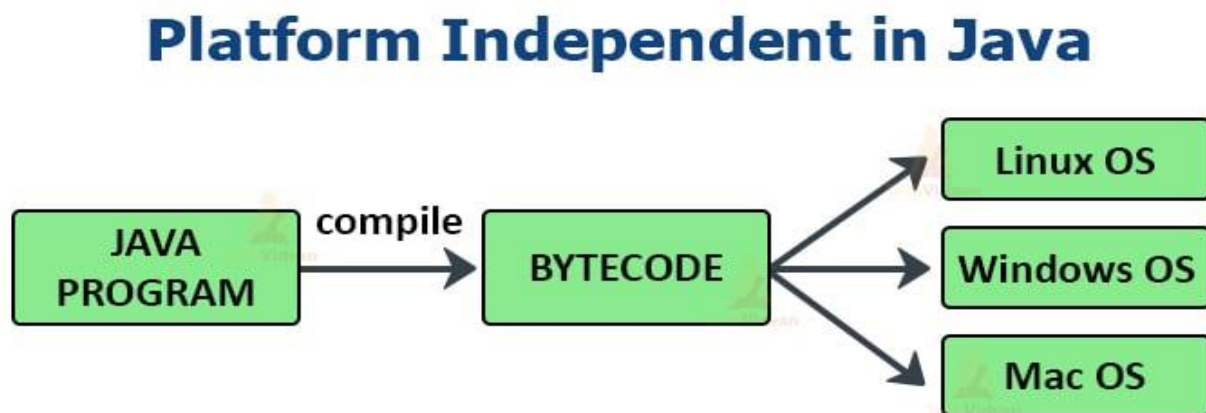
4. Java is cheap and economical to maintain

Java programs are cheap to develop and maintain as these programs are dependent on a specific hardware infrastructure to run. We can easily execute them on any machine that reduces the extra cost to maintain.

5. Java is platform-independent

Java offers a very effective boon to its users by providing the feature of platform independence that is Write Once Run Anywhere(WORA) feature.

The compiled code, i.e the byte code of java is platform-independent and can run on any machine irrespective of the operating system. We can run this code on any machine that supports the Java Virtual Machine(JVM) as shown in the figure below:



6. Java is a high-level programming language

Java is a high-level programming language as it is a human-readable language. It is similar to human language and has a very simple and easy to maintain syntax that is similar to the syntax of C++ language but in a simpler manner.

7. Java supports portability feature

Java is a portable language due to its platform independence feature. As the Java code can be run on any platform, it is portable and can be taken to any platform and can be executed on them. Therefore Java also provides the advantage of portability.

8. Java provides Automatic Garbage Collection

There is automatic memory management in Java that is managed by the Java Virtual Machine(JVM).

Whenever the objects are not used by programs anymore and they do not refer to anything that they do not need to be dereferenced or removed by the explicit programming.

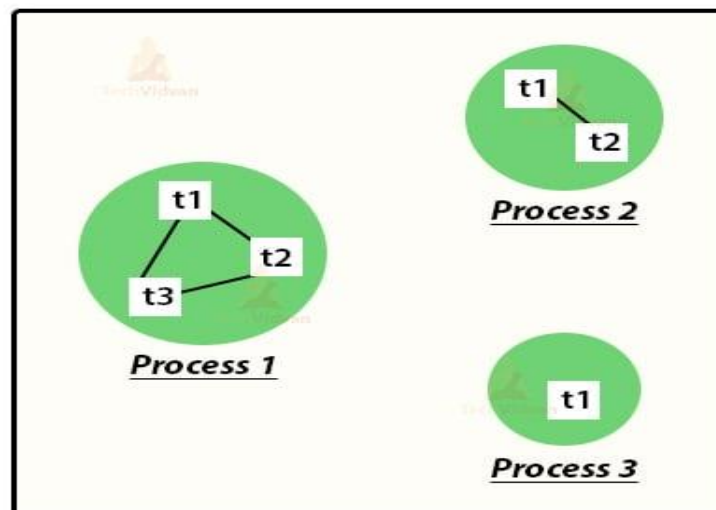
Java automatically removes the unused objects with the help of the automatic Garbage Collection process.

9. Java supports Multithreading

Java is a multithreaded language that is in Java more than one thread can run at the same time. A thread is the smallest unit of a process. Multithreading helps us to gain the maximum utilization of CPU.

Multiple threads share a common memory area and increase the efficiency and performance of the application. These threads run independently of each other without affecting each other.

Multi-threading Language in Java



10. Java is stable

Java programs are more stable as compared to programs of other languages. Moreover, a new version of Java is released in no time with more advanced features which makes it more stable.

11. Java is a distributed language

Java is a distributed language as it provides a mechanism for sharing data and programs among multiple computers that improve the performance and efficiency of the system.

The RMI(Remote Method Invocation) is something that supports the distributed processing in Java. Moreover, Java also supports Socket Programming and the CORBA technology that helps us to share objects in a distributed environment.

12. Java provides an efficient memory allocation strategy

Java has an efficient memory allocation strategy as it divides the memory mainly in two parts- Heap Area and Stack Area.

The JVM provides us the memory space for any variable either from the heap area or the stack area. Whenever we declare a variable JVM gives memory from either stack or heap space.

Disadvantages of Java

To start learning or working upon any programming language you must know its strengths and weaknesses so that you can utilize the best things out of it and avoid causing the circumstances that portray in the bad side of the language.

Java has also got some drawbacks that you should know before starting over. Let's discuss the cons of using Java.

1. Java is slow and has a poor performance

Java is memory-consuming and significantly slower than native languages such as C or C++. It is also slow compared to other languages like C and C++ because each code has to be interpreted to the machine level code.

This slow performance is due to the extra level of compilation and abstraction by the JVM. Moreover, sometimes the garbage collector leads in the poor performance of Java as it consumes more CPU time

2. Java provides not so attractive look and feels of the GUI

Though there are many GUI builders in Java for creating the graphical interface still they are not suitable for creating complicated UI. There are many inconsistencies while using them.

There are many popular frameworks such as Swing, SWT, JavaFX, JSF for creating GUI. But they are not mature enough to develop a complex UI. Choosing one of them which can be suitable for you may require additional research.

3. Java provides no backup facility

Java mainly works on storage and not focuses on the backup of data. This is a major drawback that makes it lose the interest and ratings among users.

4. Java requires significant memory space

Java requires a significant or major amount of memory space as compared to other languages like C and C++. During the execution of garbage collection, the memory efficiency and the performance of the system may be adversely affected.

5. Verbose and Complex codes

Java codes are verbose, meaning that there are many words in it and there are many long and complex sentences that are difficult to read and understand. This can reduce the readability of the code.

Java focuses on being more manageable but at the same time, it has to compromise it with the overly complex codes and long explanations for each thing.

Java Buzz Words

Java is the most popular object-oriented programming language. Java has many advanced features, a list of key features is known as Java Buzz Words. The java team has listed the following terms as java buzz words.

- **Simple**
- **Secure**
- **Portable**
- **Object-oriented**
- **Robust**
- **Architecture-neutral (or) Platform Independent**
- **Multi-threaded**
- **Interpreted**
- **High performance**
- **Distributed**
- **Dynamic**

Simple

Java programming language is very simple and easy to learn, understand, and code. Most of the syntaxes in java follow basic programming language C and object-oriented programming concepts are similar to C++. In a java programming language, many complicated features like pointers, operator overloading, structures, unions, etc. have been removed. One of the most useful features is the garbage collector it makes java more simple.

Secure

Java is said to be more secure programming language because it does not have pointers concept, java provides a feature "applet" which can be embedded into a web application. The applet in java does not allow access to other parts of the computer, which keeps away from harmful programs like viruses and unauthorized access.

Portable

Portability is one of the core features of java which enables the java programs to run on any computer or operating system. For example, an applet developed using java runs on a wide variety of CPUs, operating systems, and browsers connected to the Internet.

Object-oriented

Java is said to be a pure object-oriented programming language. In java, everything is an object. It supports all the features of the object-oriented programming paradigm. The primitive data types java also implemented as objects using wrapper classes, but still, it allows primitive data types to archive high-performance.

Robust

Java is more robust because the java code can be executed on a variety of environments, java has a strong memory management mechanism (garbage collector), java is a strictly typed language, it has a strong set of exception handling mechanism, and many more.

Architecture-neutral (or) Platform Independent

Java has invented to archive "write once; run anywhere, any time, forever". The java provides JVM (Java Virtual Machine) to to archive architectural-neutral or platform-independent. The JVM allows the java program created using one operating system can be executed on any other operating system.

Multi-threaded

Java supports multi-threading programming, which allows us to write programs that do multiple operations simultaneously.

Interpreted

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. The byte code is interpreted to any machine code so that it runs on the native machine.

High performance

Java provides high performance with the help of features like JVM, interpretation, and its simplicity.

Distributed

Java programming language supports TCP/IP protocols which enable the java to support the distributed environment of the Internet. Java also supports Remote Method Invocation (RMI), this feature enables a program to invoke methods across a network.

Dynamic

Java is said to be dynamic because the java byte code may be dynamically updated on a running system and it has a dynamic memory allocation and deallocation (objects and garbage collector).

Overview of Java

- Java is a computer programming language. Java was created based on C and C++.
- Java uses C syntax and many of the object-oriented features are taken from C++.
- Before Java was invented there were other languages like COBOL, FORTRAN, C, C++, Small Talk, etc.
- These languages had few disadvantages which were corrected in Java.
- Java also innovated many new features to solve the fundamental problems which the previous languages could not solve.
- Java was invented by a team of 13 employees of Sun Microsystems, Inc. which is lead by James Gosling, in 1991.
- The team includes persons like Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan, etc.,
- Java was developed as a part of the Green project.
- Initially, it was called Oak, later it was changed to Java in 1995.

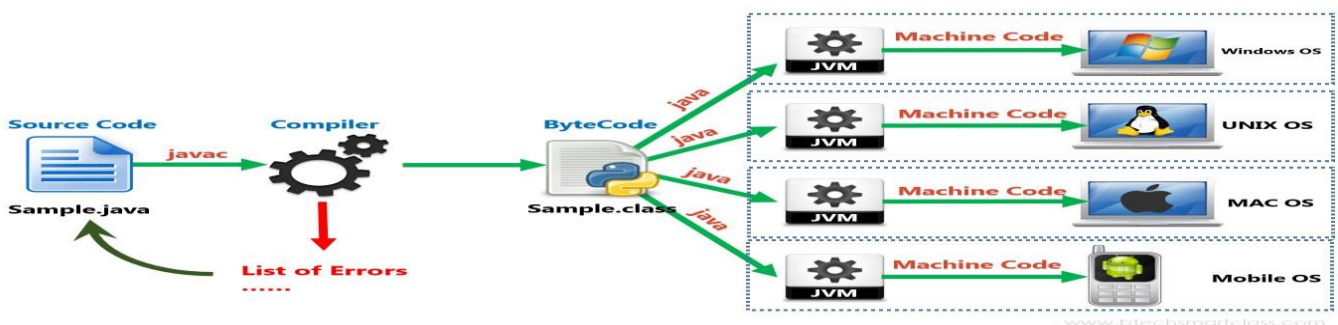
History of Java

- The C language developed in 1972 by Dennis Ritchie had taken a decade to become the most popular language.
- In 1979, Bjarne Stroustrup developed C++, an enhancement to the C language with included OOP fundamentals and features.
- A project named “Green” was initiated in December of 1990, whose aim was to create a programming tool that could render obsolete the C and C++ programming languages.
- Finally in the year of 1991 the Green Team was created a new Programming language named “OAK”.
- After some time they found that there is already a programming language with the name “OAK”.
- So, the green team had a meeting to choose a new name. After so many discussions they want to have a coffee. They went to a Coffee Shop which is just outside of the Gosling’s office and there they have decided name as “JAVA”.

Execution Process of Java Program

The following three steps are used to create and execute a java program.

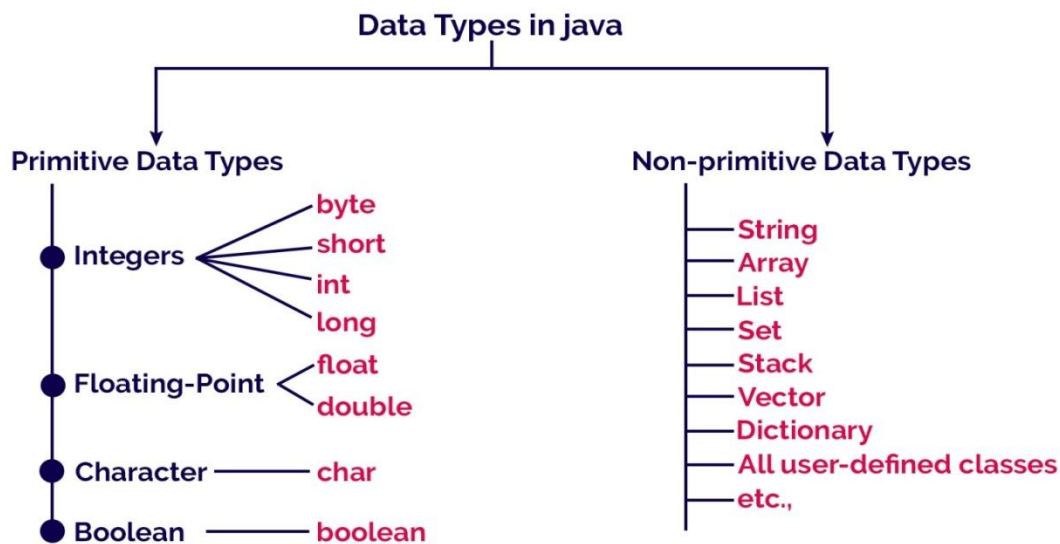
- Create a source code (.java file).
- Compile the source code using javac command.
- Run or execute .class file using java command.



Java Data Types

Java programming language has a rich set of data types. The data type is a category of data stored in variables. In java, data types are classified into two types and they are as follows.

- **Primitive Data Types**
- **Non-primitive Data Types**



www.btechsmartclass.com

Primitive Data Types

The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size. The primitive data types do not have any additional methods.

In java, primitive data types includes **byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**.

The following table provides more description of each primitive data type.

Data type	Meaning	Memory size	Range	Default Value
byte	Whole numbers	1 byte	-128 to +127	0
short	Whole numbers	2 bytes	-32768 to +32767	0
int	Whole numbers	4 bytes	-2,147,483,648 to +2,147,483,647	0

Data type	Meaning	Memory size	Range	Default Value
long	Whole numbers	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	Fractional numbers	4 bytes	-	0.0f
double	Fractional numbers	8 bytes	-	0.0d
char	Single character	2 bytes	0 to 65535	\u0000
boolean	unsigned char	1 bit	0 or 1	0 (false)

Non-primitive Data Types

In java, non-primitive data types are the reference data types or user-created data types. All non-primitive data types are implemented using object concepts. Every variable of the non-primitive data type is an object. The non-primitive data types may use additional methods to perform certain operations. The default value of non-primitive data type variable is null.

In java, examples of non-primitive data types are **String**, **Array**, **List**, **Queue**, **Stack**, **Class**, **Interface**, etc.

Java Variables

A variable is a named memory location used to store a data value. A variable can be defined as a container that holds a data value.

In java, we use the following syntax to create variables.

Syntax

```
data_type variable_name;
(or)
data_type variable_name_1, variable_name_2,...;
(or)
data_type variable_name = value;
(or)
data_type variable_name_1 = value, variable_name_2 = value,...;
```

In java programming language variables are classified as follows.

- **Local variables**
- **Instance variables or Member variables or Global variables**
- **Static variables or Class variables**
- **Final variables**

Local variables

The variables declared inside a method or a block are known as local variables. A local variable is visible within the method in which it is declared. The local variable is created when execution control enters into the method or block and destroyed after the method or block execution completed.

Let's look at the following example java program to illustrate local variable in java.

Example

```
public class LocalVariables
{
    public void show() {
        int a = 10;
        //static int x = 100;
        System.out.println("Inside show method, a = " + a);
    }
    public void display() {
        int b = 20;
        System.out.println("Inside display method, b = " + b);
        // trying to access variable 'a' - generates an ERROR
        System.out.println("Inside display method, a = " + a);
    }
    public static void main(String args[]) {
        LocalVariables obj = new LocalVariables();
        obj.show();
        obj.display();
    }
}
```

Instance variables or member variables or global variables

The variables declared inside a class and outside any method, constructor or block are known as instance variables or member variables. These variables are visible to all the methods of the class. The changes made to these variables by method affects all the methods in the class. These variables are created separate copy for every object of that class.

Let's look at the following example java program to illustrate instance variable in java.

Example

```
public class ClassVariables {  
  
    int x = 100;  
  
    public void show() {  
        System.out.println("Inside show method, x = " + x);  
        x = x + 100;  
    }  
    public void display() {  
        System.out.println("Inside display method, x = " + x);  
    }  
  
    public static void main(String[] args) {  
        ClassVariables obj = new ClassVariables();  
        obj.show();  
        obj.display();  
    }  
}
```

Static variables or Class variables

A static variable is a variable that declared using **static** keyword. The instance variables can be static variables but local variables can not. Static variables are initialized only once, at the start of the program execution. The static variable only has one copy per class irrespective of how many objects we create.

The static variable is access by using class name.

Let's look at the following example java program to illustrate static variable in java.

Example

```
public class StaticVariablesExample {

    int x, y; // Instance variables
    static int z; // Static variable

    StaticVariablesExample(int x, int y){
        this.x = x;
        this.y = y;
    }
    public void show() {
        int a; // Local variables
        System.out.println("Inside show method,");
        System.out.println("x = " + x + ", y = " + y + ", z = " + z);
    }
    public static void main(String[] args) {
        StaticVariablesExample obj_1 = new StaticVariablesExample(10, 20);
        StaticVariablesExample obj_2 = new StaticVariablesExample(100, 200);
        obj_1.show();
        StaticVariablesExample.z = 1000;
        obj_2.show();
    }
}
```


Final variables

A final variable is a variable that declared using **final** keyword. The final variable is initialized only once, and does not allow any method to change its value again. The variable created using **final** keyword acts as constant. All variables like local, instance, and static variables can be final variables.

Let's look at the following example java program to illustrate final variable in java.

Example

```
public class FinalVariableExample {  
  
    final int a = 10;  
  
    void show() {  
        System.out.println("a = " + a);  
        a = 20; //Error due to final variable can't be modified  
    }  
  
    public static void main(String[] args) {  
  
        FinalVariableExample obj = new FinalVariableExample();  
        obj.show();  
  
    }  
  
}
```

Java Arrays

- An array is a collection of similar data values with a single name.
- An array can also be defined as, a special type of variable that holds multiple values of the same data type at a time.
- In java, arrays are objects and they are created dynamically using **new** operator.
- Every array in java is organized using index values.
- The index value of an array starts with '0' and ends with 'size-1'.
- We use the index value to access individual elements of an array.

In java, there are two types of arrays and they are as follows.

- **One Dimensional Array**
- **Multi Dimensional Array**

Creating an array

In the java programming language, an array must be created using new operator and with a specific size. The size must be an integer value but not a byte, short, or long. We use the following syntax to create an array.

Syntax

```
data_type array_name[ ] = new data_type[size];
```

(or)

```
data_type[ ] array_name = new data_type[size];
```

Let's look at the following example program.

Example

```
public class ArrayExample {  
  
    public static void main(String[] args) {  
  
        int list[] = new int[5];  
  
        list[0] = 10;  
        System.out.println("Value at index 0 - " + list[0]);  
        System.out.println("Length of the array - " + list.length);  
    }  
  
}
```

In java, an array can also be initialized at the time of its declaration. When an array is initialized at the time of its declaration, it need not specify the size of the array and use of the new operator.

Here, the size is automatically decided based on the number of values that are initialized.

Example

```
int list[] = { 10, 20, 30, 40, 50};
```

NullPointerException with Arrays

In java, an array created without size and initialized to null remains null only. It does not allow us to assign a value. When we try to assign a value it generates a **NullPointerException**.

Look at the following example program.

Example

```
public class ArrayExample {  
    public static void main(String[] args) {  
        short list[] = null;  
        list[0] = 10;  
        System.out.println("Value at index 0 - " + list[0]);  
    }  
}
```

ArrayIndexOutOfBoundsException with Arrays

In java, the JVM (Java Virtual Machine) throws **ArrayIndexOutOfBoundsException** when an array is trying to access with an index value of negative value, value equal to array size, or value more than the array size.

Look at the following example program.

Example

```
public class ArrayExample {  
    public static void main(String[] args) {  
        short list[] = { 10, 20, 30};  
        list[4] = 10;  
        System.out.println("Value at index 0 - " + list[0]);  
    }  
}
```

Multidimensional Array

In java, we can create an array with multiple dimensions. We can create 2-dimensional, 3-dimensional, or any dimensional array.

In Java, multidimensional arrays are arrays of arrays. To create a multidimensional array variable, specify each additional index using another set of square brackets. We use the following syntax to create two-dimensional array.

Syntax

```
data_type array_name[ ][ ] = new data_type[rows][columns];
```

(or)

```
data_type[ ][ ] array_name = new data_type[rows][columns];
```

When we create a two-dimensional array, it created with a separate index for rows and columns. The individual element is accessed using the respective row index followed by the column index. A multidimensional array can be initialized while it has created using the following syntax.

Syntax

```
data_type array_name[ ][ ] = { {value1, value2}, {value3, value4}, {value5, value6},...};
```

When an array is initialized at the time of declaration, it need not specify the size of the array and use of the new operator. Here, the size is automatically decided based on the number of values that are initialized.

Example

```
int matrix_a[ ][ ] = { {1, 2}, {3, 4}, {5, 6} };
```

The above statement creates a two-dimensional array of three rows and two columns.

Java Operators

An operator is a symbol used to perform arithmetic and logical operations. Java provides a rich set of operators.

In java, operators are classified into the following four types.

- **Arithmetic Operators**
- **Relational (or) Comparison Operators**
- **Logical Operators**
- **Assignment Operators**
- **Bitwise Operators**
- **Conditional Operators**

Let's look at each operator in detail.

Arithmetic Operators

In java, arithmetic operators are used to performing basic mathematical operations like addition, subtraction, multiplication, division, modulus, increment, decrement, etc.,

Operator	Meaning	Example
+	Addition	$10 + 5 = 15$
-	Subtraction	$10 - 5 = 5$
*	Multiplication	$10 * 5 = 50$
/	Division	$10 / 5 = 2$
%	Modulus - Remainder of the Division	$5 \% 2 = 1$
++	Increment	a++
--	Decrement	a--

- ☐ The addition operator can be used with numerical data types and character or string data type. When it is used with numerical values, it performs mathematical addition and when it is used with character or string data type values, it performs concatenation (appending).
- ☐ The modulus (remainder of the division) operator is used with integer data type only.
- ☐ The increment and decrement operators are used as pre-increment or pre-decrement and post-increment or post-decrement.
- ☐ When they are used as pre, the value is get modified before it is used in the actual expresion and when it is used as post, the value is get modified after the the actual expression evaluation.

Let's look at the following example program.

Example

```
public class ArithmeticOperators {  
  
    public static void main(String[] args) {
```

```

int a = 10, b = 20, result;

System.out.println("a = " + a + ", b = " + b);
result = a + b;
System.out.println("Addition : " + a + " + " + b + " = " + result);
result = a - b;
System.out.println("Subtraction : " + a + " - " + b + " = " + result);
result = a * b;
System.out.println("Multiplucation : " + a + " * " + b + " = " + result);
result = b / a;
System.out.println("Division : " + b + " / " + a + " = " + result);
result = b % a;
System.out.println("Modulus : " + b + " % " + a + " = " + result);
result = ++a;
System.out.println("Pre-increment : ++a = " + result);
result = b--;
System.out.println("Post-decrement : b-- = " + result);
}
}

```

Relational Operators (<, >, <=, >=, ==, !=)

The relational operators are the symbols that are used to compare two values. That means the relational operators are used to check the relationship between two values. Every relational operator has two possible results either **TRUE** or **FALSE**. In simple words, the relational operators are used to define conditions in a program. The following table provides information about relational operators.

Operator	Meaning	Example
<	Returns TRUE if the first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Returns TRUE if the first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE
<=	Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE

Operator	Meaning	Example
>=	Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE

Look at the following example program.

Example

```
public class RelationalOperators {
    public static void main(String[] args) {
        boolean a;
        a = 10<5;
        System.out.println("10 < 5 is " + a);
        a = 10>5;
        System.out.println("10 > 5 is " + a);
        a = 10<=5;
        System.out.println("10 <= 5 is " + a);
        a = 10>=5;
        System.out.println("10 >= 5 is " + a);
        a = 10==5;
        System.out.println("10 == 5 is " + a);
        a = 10!=5;
        System.out.println("10 != 5 is " + a);
    }
}
```

Logical Operators

The logical operators are the symbols that are used to combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
&	Logical AND - Returns TRUE if all conditions are TRUE otherwise returns FALSE	false & true => false
	Logical OR - Returns FALSE if all conditions are FALSE otherwise returns TRUE	false true => true
^	Logical XOR - Returns FALSE if all conditions are same otherwise returns TRUE	true ^ true => false
!	Logical NOT - Returns TRUE if condition is FALSE and returns FALSE if it is TRUE	!false => true
&&	short-circuit AND - Similar to Logical AND (&), but once a decision is finalized it does not evaluate remaining.	false & true => false
	short-circuit OR - Similar to Logical OR (), but once a decision is finalized it does not evaluate remaining.	false true => true

□ The operators &, |, and ^ can be used with both boolean and integer data type values. When they are used with integers, performs bitwise operations and with boolean, performs logical operations.

□ Logical operators and Short-circuit operators both are similar, but in case of short-circuit operators once the decision is finalized it does not evaluate remaining expressions.

Look at the following example program.

Example

```
public class LogicalOperators {
    public static void main(String[] args) {
        int x = 10, y = 20, z = 0;
        boolean a = true;
        a = x > y && (z = x + y) > 15;
        System.out.println("a = " + a + ", and z = " + z);
        a = x > y & (z = x + y) > 15;
        System.out.println("a = " + a + ", and z = " + z);    } }
```


Assignment Operators

The assignment operators are used to assign right-hand side value (Rvalue) to the left-hand side variable (Lvalue). The assignment operator is used in different variants along with arithmetic operators. The following table describes all the assignment operators in the java programming language.

Operator	Meaning	Example
=	Assign the right-hand side value to left-hand side variable	A = 15
+=	Add both left and right-hand side values and store the result into left-hand side variable	A += 10
-=	Subtract right-hand side value from left-hand side variable value and store the result into left-hand side variable	A -= B
*=	Multiply right-hand side value with left-hand side variable value and store the result into left-hand side variable	A *= B
/=	Divide left-hand side variable value with right-hand side variable value and store the result into the left-hand side variable	A /= B
%=	Divide left-hand side variable value with right-hand side variable value and store the remainder into the left-hand side variable	A %= B
&=	Logical AND assignment	-
=	Logical OR assignment	-
^=	Logical XOR assignment	-

Look at the following example program.

Example

```
public class AssignmentOperators {  
    public static void main(String[] args) {  
        int a = 10, b = 20, c;  
        boolean x = true;
```

```

        System.out.println("a = " + a + ", b = " + b);
        a += b;
        System.out.println("a = " + a);
        a -= b;
        System.out.println("a = " + a);
        a *= b;
        System.out.println("a = " + a);
        a /= b;
        System.out.println("a = " + a);
        a %= b;
        System.out.println("a = " + a);
        x |= (a>b);
        System.out.println("x = " + x);
        x &= (a>b);
        System.out.println("x = " + x);
    }
}

```

Bitwise Operators

The bitwise operators are used to perform bit-level operations in the java programming language.

When we use the bitwise operators, the operations are performed based on binary values.

The following table describes all the bitwise operators in the java programming language. Let us consider two variables A and B as A = 25 (11001) and B = 20 (10100).

Operator	Meaning	Example
&	the result of Bitwise AND is 1 if all the bits are 1 otherwise it is 0	A & B ⇒ 16 (10000)
	the result of Bitwise OR is 1 if all the bits are 1 otherwise it is 0	A B ⇒ 29 (11101)

Operator	Meaning	Example
<code>^</code>	the result of Bitwise XOR is 0 if all the bits are same otherwise it is 1	$A \wedge B$ $\Rightarrow 13 (01101)$
<code>~</code>	the result of Bitwise once complement is negation of the bit (Flipping)	$\sim A$ $\Rightarrow 6 (00110)$
<code><<</code>	the Bitwise left shift operator shifts all the bits to the left by the specified number of positions	$A \ll 2$ $\Rightarrow 100 (1100100)$
<code>>></code>	the Bitwise right shift operator shifts all the bits to the right by the specified number of positions	$A \gg 2$ $\Rightarrow 6 (00110)$

Look at the following example program.

Example

```
public class BitwiseOperators {
    public static void main(String[] args) {
        int a = 25, b = 20;
        System.out.println(a + " & " + b + " = " + (a & b));
        System.out.println(a + " | " + b + " = " + (a | b));
        System.out.println(a + " ^ " + b + " = " + (a ^ b));
        System.out.println("~" + a + " = " + ~a);
        System.out.println(a + ">>" + 2 + " = " + (a>>2));
        System.out.println(a + "<<" + 2 + " = " + (a<<2));
        System.out.println(a + ">>>" + 2 + " = " + (a>>>2));
    }
}
```

Conditional Operators

The conditional operator is also called a **ternary operator** because it requires three operands. This operator is used for decision making. In this operator, first, we verify a condition, then we perform one operation out of the two operations based on the condition result. If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed. The conditional operator is used with the following syntax.

Syntax

Condition ? TRUE Part : FALSE Part;

Look at the following example program.

Example

```
public class ConditionalOperator {  
    public static void main(String[] args) {  
        int a = 10, b = 20, c;  
        c = (a>b)? a : b;  
        System.out.println("c = " + c);  
    }  
}
```

Java Expressions

- In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task.
- These set of symbols makes an expression.
In the java programming language, an expression is defined as follows.

An expression is a collection of operators and operands that represents a specific value.

- In the above definition, an **operator** is a symbol that performs tasks like arithmetic operations, logical operations, and conditional operations, etc.
- **Operands** are the values on which the operators perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

In the java programming language, expressions are divided into THREE types. They are as follows.

- **Infix Expression**
- **Postfix Expression**
- **Prefix Expression**

The above classification is based on the operator position in the expression.

Infix Expression

The expression in which the operator is used between operands is called infix expression. The infix expression has the following general structure.

Example



Postfix Expression

The expression in which the operator is used after operands is called postfix expression. The postfix expression has the following general structure.

Example



Prefix Expression

The expression in which the operator is used before operands is called a prefix expression. The prefix expression has the following general structure.

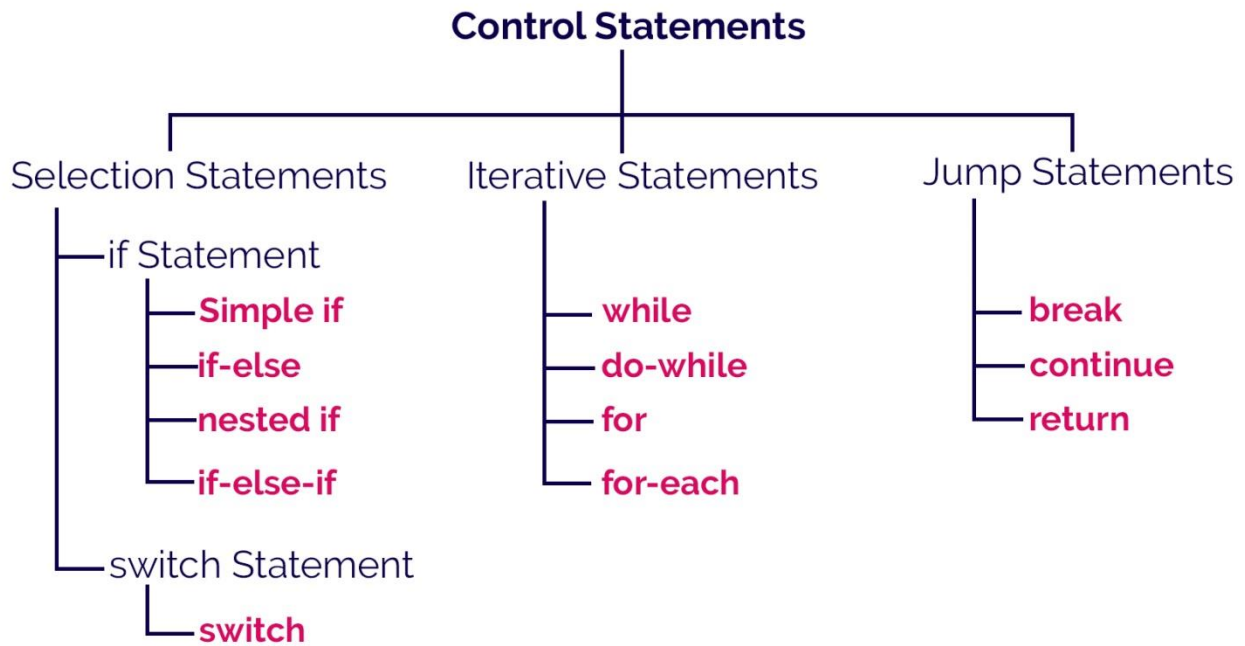
Example



Java Control Statements

In java, the default execution flow of a program is a sequential order. But the sequential order of execution flow may not be suitable for all situations. Sometimes, we may want to jump from line to another line, we may want

to skip a part of the program, or sometimes we may want to execute a part of the program again and again. To solve this problem, java provides control statements.



www.btechsmartclass.com

In java, the control statements are the statements which will tell us that in which order the instructions are getting executed. The control statements are used to control the order of execution according to our requirements. Java provides several control statements, and they are classified as follows.

Types of Control Statements

In java, the control statements are classified as follows.

- Selection Control Statements (Decision Making Statements)
- Iterative Control Statements (Looping Statements)
- Jump Statements

Let's look at each type of control statements in java.

Selection Control Statements

In java, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition. Java provides the following selection statements.

- if statement
- if-else statement
- if-elif statement
- nested if statement

- switch statement

Iterative Control Statements

In java, the iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as the given condition is True. Using iterative statements reduces the size of the code, reduces the code complexity, makes it more efficient, and increases the execution speed. Java provides the following iterative statements.

- while statement
- do-while statement
- for statement
- for-each statement

Jump Statements

In java, the jump statements are used to terminate a block or take the execution control to the next iteration. Java provides the following jump statements.

- break
- continue
- return

In java, the selection statements are also known as decision making statements or branching statements or conditional control statements. The selection statements are used to select a part of the program to be executed based on a condition. Java provides the following selection statements.

- if statement
- if-else statement
- nested if statement
- if-else if statement
- switch statement

Selection Control Statements

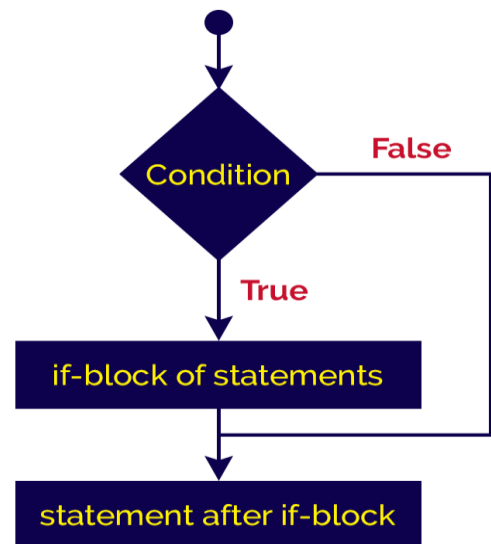
if statement in java

In java, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result. The if statement checks, the given condition then decides the execution of a block of statements. If the condition is True, then the block of statements is executed and if it is False, then the block of statements is ignored. The syntax and execution flow of if the statement is as follows.

Syntax

```
if(condition){  
    if-block of statements;  
    ...  
}  
statement after if-block;  
...
```

Flow of execution



www.btechsmartclass.com

Let's look at the following example java code.

Java Program

```
import java.util.Scanner;  
public class IfStatementTest {  
    public static void main(String[] args) {  
        Scanner read = new Scanner(System.in);  
        System.out.print("Enter any number: ");  
        int num = read.nextInt();  
        if((num % 5) == 0) {  
            System.out.println("We are inside the if-block!");  
            System.out.println("Given number is divisible by 5!!!");  
        }  
        System.out.println("We are outside the if-block!!!");  
    }  
}
```

In the above execution, the number 12 is not divisible by 5. So, the condition becomes False and the condition is evaluated to False. Then the if statement ignores the execution of its block of statements.

When we enter a number which is divisible by 5, then it produces the output as follows.

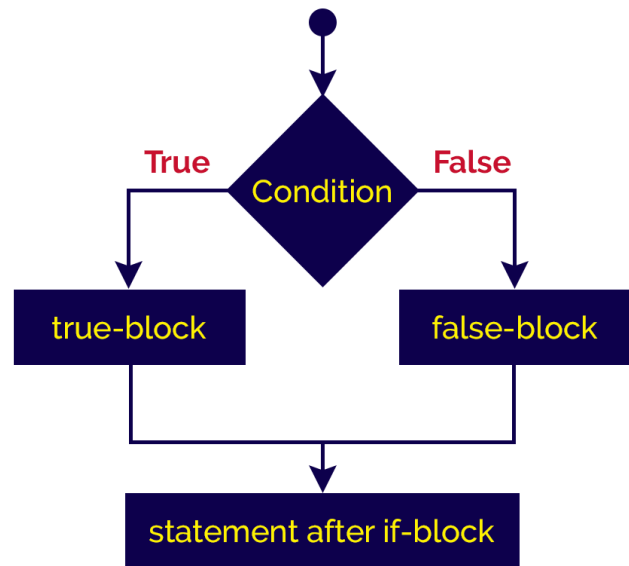
if-else statement in java

In java, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result. The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result. If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed. The syntax and execution flow of if-else statement is as follows.

Syntax

```
if(condition){  
    true-block of statements;  
    ...  
}  
else{  
    false-block of statements;  
    ...  
}  
statement after if-block;  
...
```

Flow of execution



www.btechsmartclass.com

Let's look at the following example java code.

Java Program

```
import java.util.Scanner;  
  
public class IfElseStatementTest {  
    public static void main(String[] args) {  
  
        Scanner read = new Scanner(System.in);  
        System.out.print("Enter any number: ");  
        int num = read.nextInt();  
        if((num % 2) == 0) {  
            System.out.println("We are inside the true-block!");  
            System.out.println("Given number is EVEN number!!");  
        }  
    }  
}
```

```

        else {
            System.out.println("We are inside the false-block!");
            System.out.println("Given number is ODD number!!!");
        }
        System.out.println("We are outside the if-block!!!");
    }
}

```

Nested if statement in java

Writing an if statement inside another if-statement is called nested if statement. The general syntax of the nested if-statement is as follows.

Syntax

```

if(condition_1){
    if(condition_2){
        inner if-block of statements;
        ...
    }
    ...
}

```

Let's look at the following example java code.

Java Program

```

import java.util.Scanner;
public class NestedIfStatementTest {
    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.print("Enter any number: ");
        int num = read.nextInt();
        if (num < 100) {

```

```

        System.out.println("\nGiven number is below 100");
        if (num % 2 == 0)
            System.out.println("And it is EVEN");
        else
            System.out.println("And it is ODD");
    } else
        System.out.println("Given number is not below 100");

    System.out.println("\nWe are outside the if-block!!!");
}
}

```

if-else if statement in java

Writing an if-statement inside else of an if statement is called if-else-if statement. The general syntax of the an if-else-if statement is as follows.

Syntax

```

if(condition_1){
    condition_1 true-block;
    ...
}
else if(condition_2){
    condition_2 true-block;
    condition_1 false-block too;
    ...
}

```

Let's look at the following example java code.

Java Program

```

import java.util.Scanner;
public class IfElseIfStatementTest {
    public static void main(String[] args) {

```

```

int num1, num2, num3;

Scanner read = new Scanner(System.in);

System.out.print("Enter any three numbers: ");

num1 = read.nextInt();
num2 = read.nextInt();
num3 = read.nextInt();

if( num1>=num2 && num1>=num3)

    System.out.println("\nThe largest number is " + num1) ;

    else if (num2>=num1 && num2>=num3)

System.out.println("\nThe largest number is " + num2) ;

    else

System.out.println("\nThe largest number is " + num3) ;

System.out.println("\nWe are outside the if-block!!!");

}

}

```

switch statement in java

Using the switch statement, one can select only one option from more number of options very easily. In the switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches the value associated with an option, the execution starts from that option. In the switch statement, every option is defined as a **case**.

The switch statement has the following syntax and execution flow diagram.

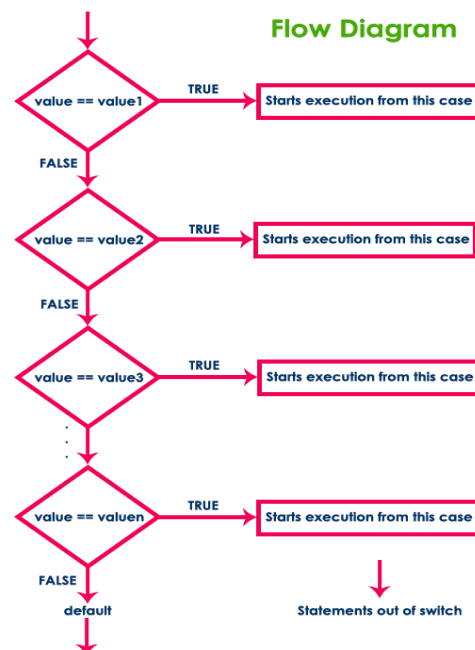
Syntax

```

switch ( expression or value )
{
    case value1: set of statements;
    ....
    case value2: set of statements;
    ....
    case value3: set of statements;
    ....
    case value4: set of statements;
    ....
    case value5: set of statements;
    ....
    .
    .
    default: set of statements;
}

```

Flow Diagram



Let's look at the following example java code.

Java Program

```
import java.util.Scanner;

public class SwitchStatementTest {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);

        System.out.print("Press any digit: ");

        int value = read.nextInt();

        switch( value )

        {

            case 0: System.out.println("ZERO") ; break ;
            case 1: System.out.println("ONE") ; break ;
            case 2: System.out.println("TWO") ; break ;
            case 3: System.out.println("THREE") ; break ;
            case 4: System.out.println("FOUR") ; break ;
            case 5: System.out.println("FIVE") ; break ;
            case 6: System.out.println("SIX") ; break ;
            case 7: System.out.println("SEVEN") ; break ;
            case 8: System.out.println("EIGHT") ; break ;
            case 9: System.out.println("NINE") ; break ;
            default: System.out.println("Not a Digit") ;

        }

    }

}
```

Java Iterative Statements

The java programming language provides a set of iterative statements that are used to execute a statement or a block of statements repeatedly as long as the given condition is true. The iterative statements are also known as looping statements or repetitive statements. Java provides the following iterative statements.

- while statement
- do-while statement
- for statement
- for-each statement

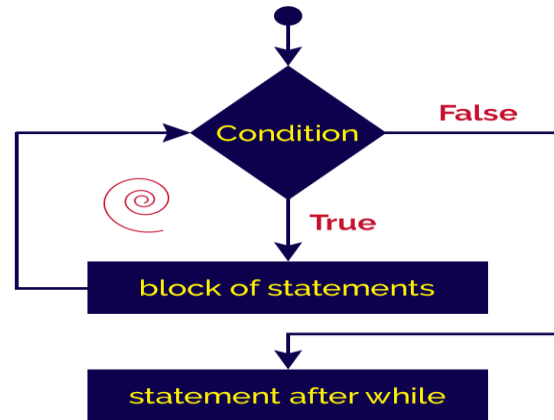
while statement in java

The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement. The syntax and execution flow of while statement is as follows.

Syntax

```
while(boolean-expression){  
    block of statements;  
    ...  
}  
statement after while;  
...
```

Flow of execution



www.btechsmartclass.com

Let's look at the following example java code.

Java Program

```
public class WhileTest {  
    public static void main(String[] args) {  
        int num = 1;  
        while(num <= 10) {  
            System.out.println(num);  
            num++;  
        }  
        System.out.println("Statement after while!");  
    }  
}
```

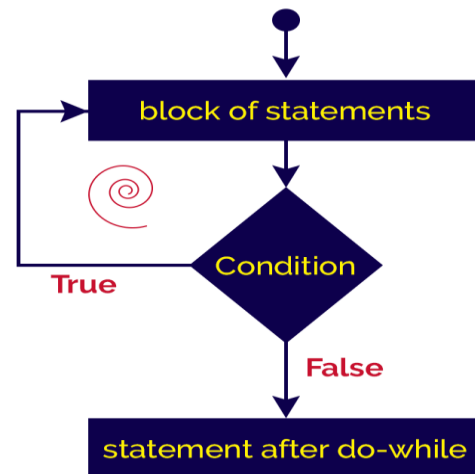
do-while statement in java

The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as the **Exit control looping statement**. The do-while statement has the following syntax.

Syntax

```
do{  
    block of statements;  
}while(boolean-expression);  
statement after do-while;  
...
```

Flow of execution



Let's look at the following example java code.

Java Program

```
public class DoWhileTest {  
    public static void main(String[] args) {  
        int num = 1;  
        do {  
            System.out.println(num);  
            num++;  
        } while(num <= 10);  
        System.out.println("Statement after do-while!");  
    }  
}
```

for statement in java

- ❖ The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition TRUE.
- ❖ In for-statement, the execution begins with the **initialization** statement.
- ❖ After the initialization statement, it executes **Condition**.

- ❖ If the condition is evaluated to true, then the block of statements executed otherwise it terminates the for-statement.
- ❖ After the block of statements execution, the **modification** statement gets executed, followed by condition again.

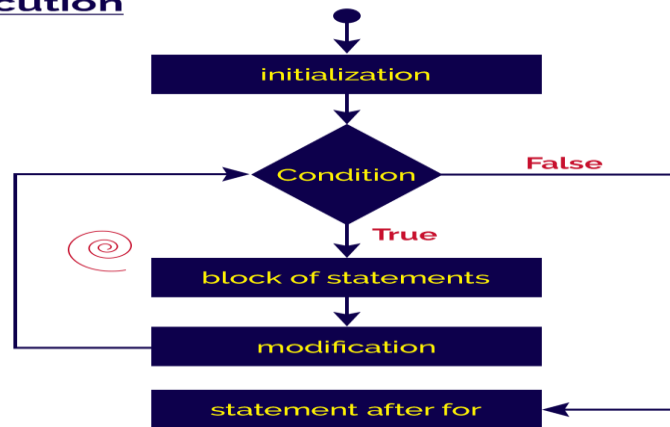
is

T

Syntax

```
for(initialization; boolean-expression; modification){
    block of statements;
    ...
}
statement after for;
...
```

Flow of execution



www.btechsmartclass.com

Let's look at the following example java code.

Java Program

```
public class ForTest {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
        System.out.println("Statement after for!");
    }
}
```

for-each statement in java

The Java for-each statement was introduced since Java 5.0 version. It provides an approach to traverse through an array or collection in Java. The for-each statement also known as **enhanced for** statement. The for-each statement executes the block of statements for each element of the given array or collection.

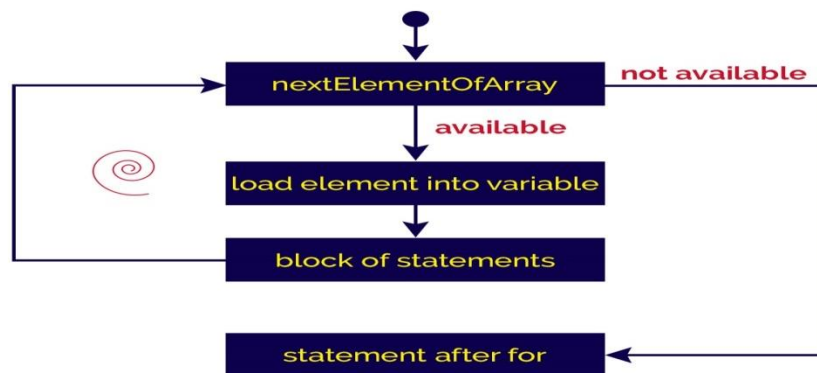
- In for-each statement, we can not skip any element of given array or collection.

The for-each statement has the following syntax and execution flow diagram.

Syntax

```
for( dataType variableName : Array ){  
    block of statements;  
    ...  
}  
statement after for;  
...
```

Flow of execution



www.btechsmartclass.com

Let's look at the following example java code.

Java Program

```
public class ForEachTest {  
    public static void main(String[] args) {  
        int[] arrayList = {10, 20, 30, 40, 50};  
        for(int i : arrayList) {  
            System.out.println("i = " + i);  
        }  
        System.out.println("Statement after for-each!");  
    }  
}
```

Java Jump Statements

The java programming language supports jump statements that used to transfer execution control from one line to another line. The java programming language provides the following jump statements.

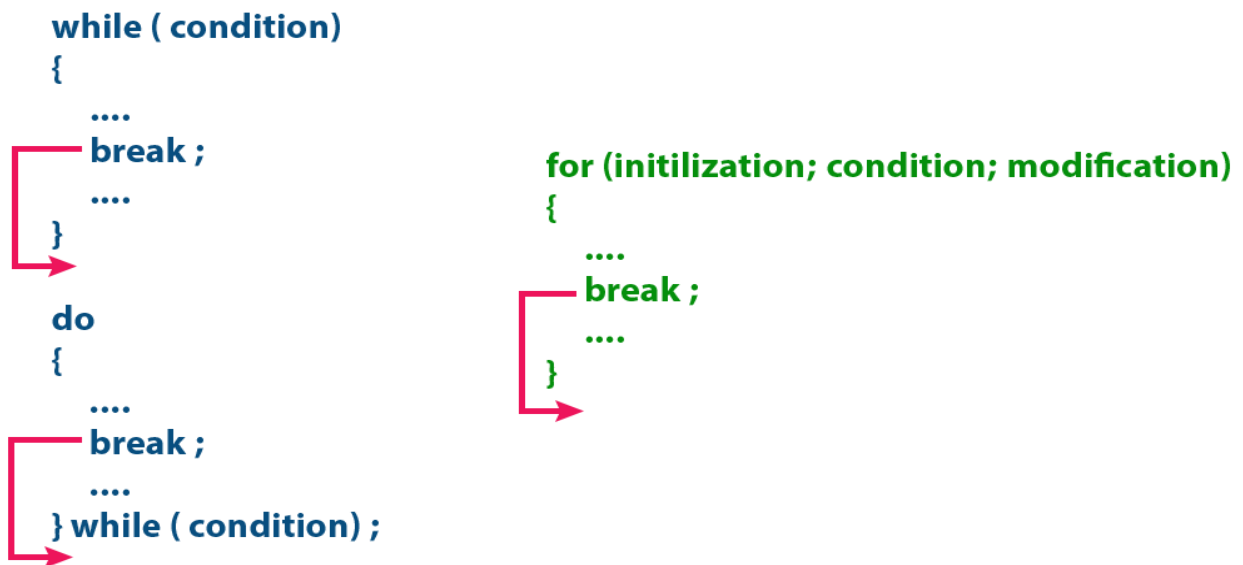
- break statement
- continue statement
- labelled break and continue statements
- return statement

break statement in java

The break statement in java is used to terminate a switch or looping statement. That means the break statement is used to come out of a switch statement and a looping statement like while, do-while, for, and for-each.

□ Using the break statement outside the switch or loop statement is not allowed.

The following picture depicts the execution flow of the break statement.



Let's look at the following example java code.

Java Program

```
public class JavaBreakStatement {
    public static void main(String[] args) {
        int list[] = { 10, 20, 30, 40, 50 };
        for(int i : list) {
```

```

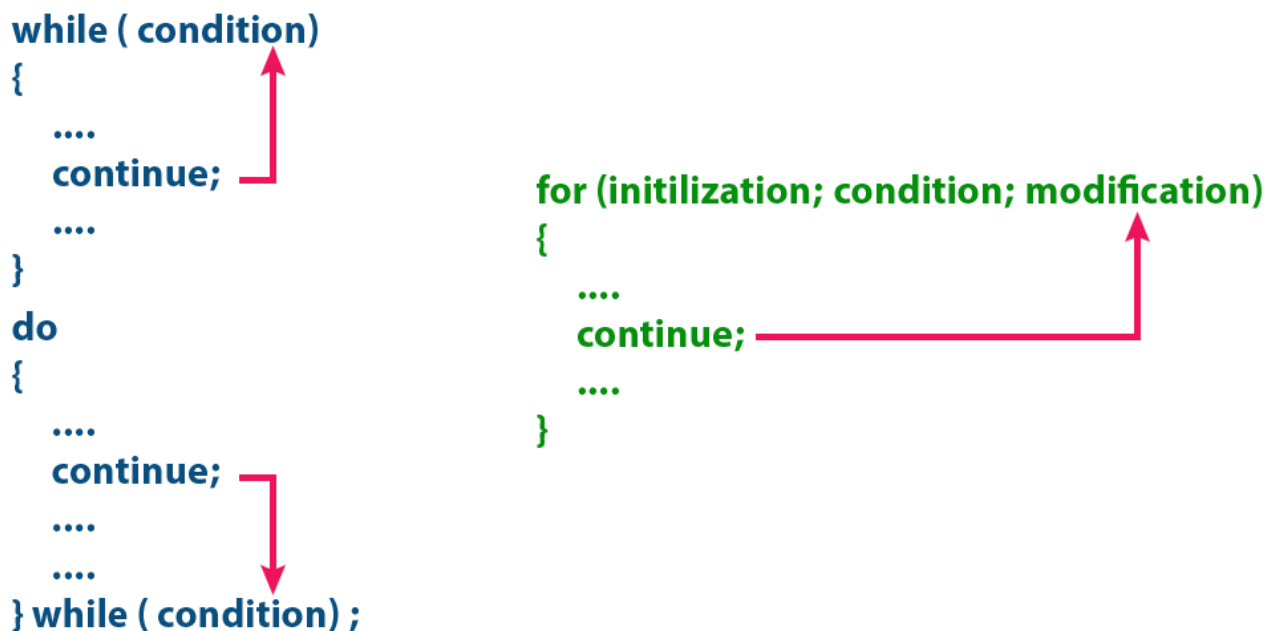
        if(i == 30)
            break;
        System.out.println(i);
    }
}

```

continue statement in java

The continue statement is used to move the execution control to the beginning of the looping statement. When the continue statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The continue statement can be used with looping statements like while, do-while, for, and for-each.

When we use continue statement with while and do-while statements, the execution control directly jumps to the condition. When we use continue statement with for statement the execution control directly jumps to the modification portion (increment/decrement/any modification) of the for loop. The continue statement flow of execution is as shown in the following figure.



Let's look at the following example java code.

Java Program

```
public class JavaContinueStatement {  
    public static void main(String[] args) {  
        int list[] = {10, 20, 30, 40, 50};  
        for(int i : list) {  
            if(i == 30)  
                continue;  
            System.out.println(i);  
        }  
    }  
}
```

Labelled break and continue statement in java

The java programming language does not support **goto** statement, alternatively, the break and continue statements can be used with label.

The labelled break statement terminates the block with specified label. The labelled continue statement takes the execution control to the beginning of a loop with specified label.

Let's look at the following example java code.

Java Program

```
import java.util.Scanner;  
public class JavaLabelledStatement {  
    public static void main(String args[]) {  
        Scanner read = new Scanner(System.in);  
        reading: for (int i = 1; i <= 3; i++) {  
            System.out.print("Enter a even number: ");  
            int value = read.nextInt();  
            verify: if (value % 2 == 0) {  
                System.out.println("\nYou won!!!");  
                System.out.println("Your score is " + i*10 + " out of 30.");  
            }  
        }  
    }  
}
```

```

        break reading;
    } else {
        System.out.println("\nSorry try again!!!");
        System.out.println("You let with " + (3-i) + " more options...");
        continue reading;
    }
}
}
}
}

```

return statement in java

In java, the return statement used to terminate a method with or without a value. The return statement takes the execution control to the calling function. That means the return statement transfer the execution control from called function to the calling function by carrying a value.

□ Java allows the use of return-statement with both, with and without return type methods.

In java, the return statement used with both methods with and without return type. In the case of a method with the return type, the return statement is mandatory, and it is optional for a method without return type.

When a return statement used with a return type, it carries a value of return type. But, when it is used without a return type, it does not carry any value. Instead, simply transfers the execution control.

Let's look at the following example java code.

Java Program

```

import java.util.Scanner;
public class JavaReturnStatementExample {
    int value;        int readValue() {
        Scanner read = new Scanner(System.in);
        System.out.print("Enter any number: ");
        return this.value=read.nextInt();    }
    void showValue(int value) {
        for(int i = 0; i <= value; i++) {
            if(i == 5) return;

```

```

        System.out.println(i);
    }
}

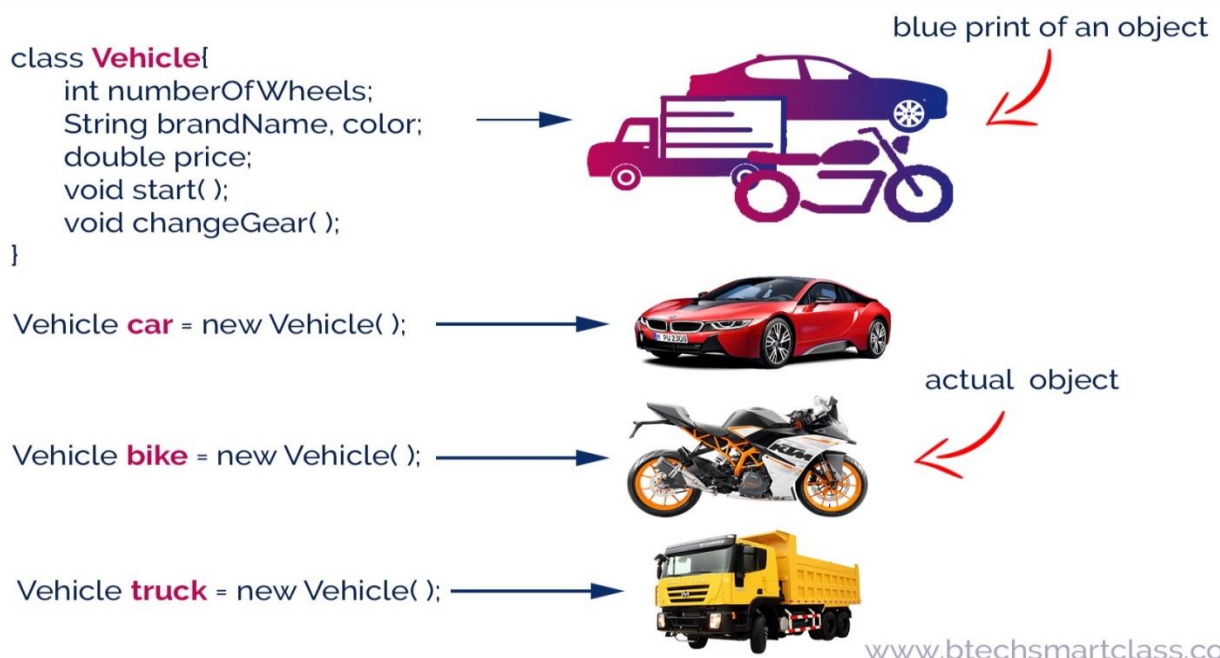
public static void main(String[] args) {
    JavaReturnStatementExample obj = new JavaReturnStatementExample();
    obj.showValue(obj.readValue());
}
}

```

Java Classes

- Java is an object-oriented programming language, so everything in java program must be based on the object concept. In a java programming language, the class concept defines the skeleton of an object.
- The java class is a template of an object. The class defines the blueprint of an object. Every class in java forms a new data type.
- Once a class got created, we can generate as many objects as we want. Every class defines the properties and behaviors of an object. All the objects of a class have the same properties and behaviors that were defined in the class.
- Every class of java programming language has the following characteristics.
 - **Identity** - It is the name given to the class.
 - **State** - Represents data values that are associated with an object.
 - **Behavior** - Represents actions can be performed by an object.

Look at the following picture to understand the class and object concept.



Creating a Class

In java, we use the keyword class to create a class. A class in java contains properties as variables and behaviors as methods. Following is the syntax of class in the java.

Syntax

```
class <ClassName>{  
    data members declaration;  
    methods defination;  
}
```

- ☐ The ClassName must begin with an alphabet, and the Upper-case letter is preferred.
- ☐ The ClassName must follow all naming rules.

Creating an Object

In java, an object is an instance of a class. When an object of a class is created, the class is said to be instantiated. All the objects that are created using a single class have the same properties and methods. But the value of properties is different for every object. Following is the syntax of class in the java.

Syntax

```
<ClassName> <objectName> = new <ClassName>();
```

- ☐ The objectName must begin with an alphabet, and a Lower-case letter is preferred.
- ☐ The objectName must follow all naming rules.

Java Methods

A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task. The major advantage of methods is code re-usability (define the code once, and use it many times).

In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.

Every method in java must be declared inside a class.

Every method declaration has the following characteristics.

- **returnType** - Specifies the data type of a return value.
- **name** - Specifies a unique name to identify it.
- **parameters** - The data values it may accept or receive.
- **{ }** - Defines the block belongs to the method.

Creating a method

A method is created inside the class and it may be created with any access specifier. However, specifying access specifier is optional.

Following is the syntax for creating methods in java.

Syntax

```
class <ClassName>{  
    <accessSpecifier> <returnType> <methodName>( parameters ){  
        ...  
        block of statements;  
        ...  
    }  
}
```

- ❑ The methodName must begin with an alphabet, and the Lower-case letter is preferred.
- ❑ The methodName must follow all naming rules.
- ❑ If you don't want to pass parameters, we ignore it.
- ❑ If a method defined with return type other than void, it must contain the return statement, otherwise, it may be ignored.

Calling a method

In java, a method call precedes with the object name of the class to which it belongs and a dot operator. It may call directly if the method defined with the static modifier. Every method call must be made, as to the method name with parentheses (), and it must terminate with a semicolon.

Syntax


```
<objectName>.<methodName>( actualArguments );
```

- ❑ The method call must pass the values to parameters if it has.
- ❑ If the method has a return type, we must provide the receiver.

Let's look at the following example java code.

Example

```
import java.util.Scanner;
public class JavaMethodsExample {
    int sNo;
    String name;
    Scanner read = new Scanner(System.in);
    void readData() {
        System.out.print("Enter Serial Number: ");
        sNo = read.nextInt();
        System.out.print("Enter the Name: ");
        name = read.next();
    }
    static void showData(int sNo, String name) {
        System.out.println("Hello, " + name + "! your serial number is " + sNo);
    }
    public static void main(String[] args) {
        JavaMethodsExample obj = new JavaMethodsExample();
        obj.readData(); // method call using object
        showData(obj.sNo, obj.name); // method call without using object
    }
}
```

- ❑ The objectName must begin with an alphabet, and a Lower-case letter is preferred.
- ❑ The objectName must follow all naming rules.

Variable arguments of a method

In java, a method can be defined with a variable number of arguments. That means creating a method that receives any number of arguments of the same data type.

Syntax

```
<returnType> <methodName>(<dataType...parameterName>);
```

Let's look at the following example java code.

Example

```
public class JavaMethodWithVariableArgs {  
    void diaplay(int...list) {  
        System.out.println("\nNumber of arguments: " + list.length);  
  
        for(int i : list) {  
            System.out.print(i + "\t");  
        }  
    }  
    public static void main(String[] args) {  
        JavaMethodWithVariableArgs obj = new JavaMethodWithVariableArgs();  
  
        obj.diaplay(1, 2);  
        obj.diaplay(10, 20, 30, 40, 50);  
    }  
}
```

- ❑ When a method has both the normal parameter and variable-argument, then the variable argument must be specified at the end in the parameters list.

Constructor

A constructor is a special method of a class that has the same name as the class name. The constructor gets executed automatically on object creation. It does not require the explicit method call. A constructor may have parameters and access specifiers too. In java, if you do not provide any constructor the compiler automatically creates a default constructor.

Let's look at the following example java code.

Example

```
public class ConstructorExample {  
  
    ConstructorExample() {  
        System.out.println("Object created!");  
    }  
  
    public static void main(String[] args) {  
  
        ConstructorExample obj1 = new ConstructorExample();  
        ConstructorExample obj2 = new ConstructorExample();  
  
    }  
  
}
```

- ❑ A constructor can not have return value.

Java String Handling

A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class **String**.

In the background, the string values are organized as an array of a character data type.

The string created using a character array can not be extended. It does not allow to append more characters after its definition, but it can be modified.

Let's look at the following example java code.

Example

```
char[] name = {'J', 'a', 'v', 'a', ' ', 'T', 'u', 't', 'o', 'r', 'i', 'a', 'l', 's'};  
//name[14] = '@'; //ArrayIndexOutOfBoundsException  
name[5] = '-';
```

```
System.out.println(name);
```

The **String** class defined in the package **java.lang** package. The **String** class implements **Serializable**, **Comparable**, and **CharSequence** interfaces.

The string created using the **String** class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Let's look at the following example java code.

Example

```
String siteName = "btechsmartclass.com";  
siteName = "www.btechsmartclass.com";
```

Creating String object in java

In java, we can use the following two ways to create a string object.

- Using string literal
- Using String constructor

Let's look at the following example java code.

Example

```
String title = "Java Tutorials";           // Using literals  
  
String siteName = new String("www.btechsmartclass.com"); // Using constructor
```

□ The **String** class constructor accepts both string and character array as an argument.

String handling methods

In java programming language, the **String** class contains various methods that can be used to handle string data values. It contains methods like `concat()`, `compareTo()`, `split()`, `join()`, `replace()`, `trim()`, `length()`, `intern()`, `equals()`, `comparison()`, `substring()`, etc.

The following table depicts all built-in methods of **String** class in java.

Method	Description	Return Value
charAt(int)	Finds the character at given index	char
length()	Finds the length of given string	int
compareTo(String)	Compares two strings	int
compareToIgnoreCase(String)	Compares two strings, ignoring case	int
concat(String)	Concatenates the object string with argument string.	String
contains(String)	Checks whether a string contains sub-string	boolean
contentEquals(String)	Checks whether two strings are same	boolean
equals(String)	Checks whether two strings are same	boolean
equalsIgnoreCase(String)	Checks whether two strings are same, ignoring case	boolean
startsWith(String)	Checks whether a string starts with the specified string	boolean
endsWith(String)	Checks whether a string ends with the specified string	boolean
getBytes()	Converts string value to bytes	byte[]
hashCode()	Finds the hash code of a string	int
indexOf(String)	Finds the first index of argument string in object string	int
lastIndexOf(String)	Finds the last index of argument string in object string	int

Method	Description	Return Value
isEmpty()	Checks whether a string is empty or not	boolean
replace(String, String)	Replaces the first string with second string	String
replaceAll(String, String)	Replaces the first string with second string at all occurrences.	String
substring(int, int)	Extracts a sub-string from specified start and end index values	String
toLowerCase()	Converts a string to lower case letters	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends	String
toString(int)	Converts the value to a String object	String
split(String)	splits the string matching argument string	String[]
intern()	returns string from the pool	String
join(String, String, ...)	Joins all strings, first string as delimiter.	String

Let's look at the following example java code.

Java Program

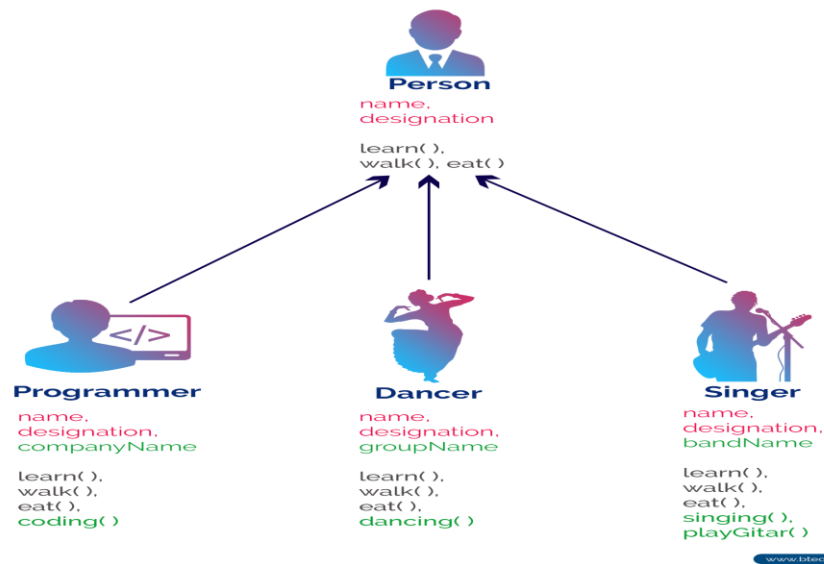
```
public class JavaStringExample {
```

```
public static void main(String[] args) {  
    String title = "Java Tutorials";  
    String siteName = "www.btechsmartclass.com";  
  
    System.out.println("Length of title: " + title.length());  
    System.out.println("Char at index 3: " + title.charAt(3));  
    System.out.println("Index of 'T': " + title.indexOf("T"));  
    System.out.println("Last index of 'a': " + title.lastIndexOf('a'));  
    System.out.println("Empty: " + title.isEmpty());  
    System.out.println("Ends with '.com': " + siteName.endsWith(".com"));  
    System.out.println("Equals: " + siteName.equals(title));  
    System.out.println("Sub-string: " + siteName.substring(9, 14));  
    System.out.println("Upper case: " + siteName.toUpperCase());  
}  
  
}
```

Inheritance Inheritance concept

Inheritance Concept

- The inheritance is a very useful and powerful concept of object-oriented programming. In java, using the inheritance concept, we can use the existing features of one class in another class.
- The inheritance provides a great advantage called code re-usability.
- With the help of code re-usability, the commonly used code in an application need not be written again and again.



The inheritance can be defined as follows.

The inheritance is the process of acquiring the properties of one class to another class.

Inheritance Basics

- In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.
- The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.
- The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

There are five types of inheritances, and they are as follows.

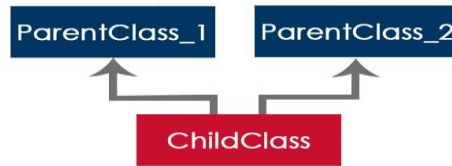
- **Simple Inheritance (or) Single Inheritance**
- **Multiple Inheritance**
- **Multi-Level Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

The following picture illustrates how various inheritances are implemented.

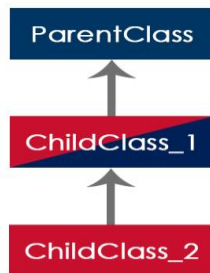
Simple Inheritance



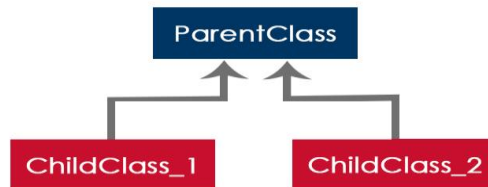
Multiple Inheritance



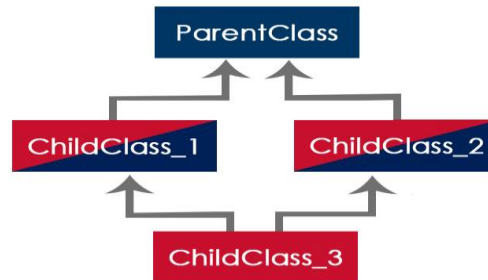
Multi Level Inheritance



Hierarchical Inheritance



Hybrid Inheritance



www.btechsmartclass.com

The java programming language does not support multiple inheritance type. However, it provides an alternate with the concept of interfaces.

Creating Child Class in java

In java, we use the keyword **extends** to create a child class. The following syntax used to create a child class in java.

Syntax

```
class <ChildClassName> extends <ParentClassName>{  
    ...  
    //Implementation of child class  
    ... }
```

In a java programming language, a class extends only one class. Extending multiple classes is not allowed in java.

Let's look at individual inheritance types and how they get implemented in java with an example.

Single Inheritance in java

In this type of inheritance, one child class derives from one parent class. Look at the following example code.

Example

```
class ParentClass{
    int a;
    void setData(int a) {
        this.a = a;
    }
}
class ChildClass extends ParentClass{
    void showData() {
        System.out.println("Value of a is " + a);
    }
}
public class SingleInheritance {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.setData(100);
        obj.showData();
    }
}
```

Multi-level Inheritance in java

In this type of inheritance, the child class derives from a class which already derived from another class. Look at the following example java code.

Example

```
class ParentClass{
    int a;
    void setData(int a) {
        this.a = a;
    }
}
class ChildClass extends ParentClass{
    void showData() {
        System.out.println("Value of a is " + a);
    }
}
```

```

}

class ChildChildClass extends ChildClass{
    void display() {
        System.out.println("Inside ChildChildClass!");
    }
}

public class MultipleInheritance {
    public static void main(String[] args) {
        ChildChildClass obj = new ChildChildClass();
        obj.setData(100);
        obj.showData();
        obj.display();
    }
}

```

Hierarchical Inheritance in java

In this type of inheritance, two or more child classes derive from one parent class. Look at the following example java code.

Example

```

class ParentClass{
    int a;
    void setData(int a) {
        this.a = a;
    }
}

class ChildClass extends ParentClass{
    void showData() {
        System.out.println("Inside ChildClass!");
        System.out.println("Value of a is " + a);
    }
}

class ChildClassToo extends ParentClass{
    void display() {
        System.out.println("Inside ChildClassToo!");
    }
}

```

```

        System.out.println("Value of a is " + a);
    }
}

public class HierarchicalInheritance {

    public static void main(String[] args) {

        ChildClass child_obj = new ChildClass();
        child_obj.setData(100);
        child_obj.showData();
        ChildClassToo childToo_obj = new ChildClassToo();
        childToo_obj.setData(200);
        childToo_obj.display();

    }
}

```

Hybrid Inheritance in java

The hybrid inheritance is the combination of more than one type of inheritance. We may use any combination as a single with multiple inheritances, multi-level with multiple inheritances, etc.,

Java Access Modifiers / Member Access

In Java, the access specifiers (also known as access modifiers) used to restrict the scope or accessibility of a class, constructor, variable, method or data member of class and interface. There are four access specifiers, and their list is below.

- **default (or) no modifier**
- **public**
- **protected**
- **private**

In java, we cannot employ all access specifiers on everything. The following table describes where we can apply the access specifiers.

Access Specifier Item	Default	Public	Protected	Private
Class	Yes	Yes	No	No
Inner Class	Yes	Yes	Yes	Yes
Interface	Yes	Yes	No	No
Interface Inside Class	Yes	Yes	Yes	Yes
enum	Yes	Yes	No	No
enum Inside Class	Yes	Yes	Yes	Yes
enum inside Interface	Yes	No	No	No
Constructor	Yes	Yes	Yes	Yes
methods & data inside class	Yes	Yes	Yes	Yes
methods & data inside Interface	Yes	No	No	No

www.btechsmartclass.com

Let's look at the following example java code, which generates an error because a class does not allow private access specifier unless it is an inner class.

Example

```
private class Sample{
    ...
}
```

In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Access control for members of class and interface in java

Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

www.btechsmartclass.com

- ❑ The **public** members can be accessed everywhere.
- ❑ The **private** members can be accessed only inside the same class.
- ❑ The **protected** members are accessible to every child class (same package or other packages).
- ❑ The **default** members are accessible within the same package but not outside the package.

Let's look at the following example java code.

Example

```
class ParentClass{
    int a = 10;
    public int b = 20;
    protected int c = 30;
    private int d = 40;

    void showData() {
        System.out.println("Inside ParentClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}

class ChildClass extends ParentClass{
    void accessData() {
        System.out.println("Inside ChildClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        //System.out.println("d = " + d);    // private member can't be accessed
    }
}

public class AccessModifiersExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.showData();
        obj.accessData();
    } }
```

Java Constructors in Inheritance

It is very important to understand how the constructors get executed in the inheritance concept. In the inheritance, the constructors never get inherited to any child class.

In java, the default constructor of a parent class called automatically by the constructor of its child class. That means when we create an object of the child class, the parent class constructor executed, followed by the child class constructor executed.

Let's look at the following example java code.

Example

```
class ParentClass{
    int a;
    ParentClass(){
        System.out.println("Inside ParentClass constructor!");
    }
}
class ChildClass extends ParentClass{
    ChildClass(){
        System.out.println("Inside ChildClass constructor!!");
    }
}
class ChildChildClass extends ChildClass{
    ChildChildClass(){
        System.out.println("Inside ChildChildClass constructor!!!");
    }
}
public class ConstructorInInheritance {
    public static void main(String[] args) {
        ChildChildClass obj = new ChildChildClass();
    }
}
```

However, if the parent class contains both default and parameterized constructor, then only the default constructor called automatically by the child class constructor.

Let's look at the following example java code.

Example

```
class ParentClass{
    int a;
```

```

    ParentClass(int a){
        System.out.println("Inside ParentClass parameterized constructor!");
        this.a = a;
    }
    ParentClass(){
        System.out.println("Inside ParentClass default constructor!");
    }
}
class ChildClass extends ParentClass{
    ChildClass(){
        System.out.println("Inside ChildClass constructor!!");
    }
}
public class ConstructorInInheritance {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
    }
}

```

The parameterized constructor of parent class must be called explicitly using the **super** keyword.

Java super keyword

- In java, super is a keyword used to refers to the parent class object.
- The super keyword came into existence to solve the naming conflicts in the inheritance.
- When both parent class and child class have members with the same name, then the super keyword is used to refer to the parent class version.

In java, the super keyword is used for the following purposes.

- **To refer parent class data members**
- **To refer parent class methods**
- **To call parent class constructor**

□ The **super** keyword is used inside the child class only.

super to refer parent class data members

When both parent class and child class have data members with the same name, then the super keyword is used to refer to the parent class data member from child class.

Let's look at the following example java code.

Example

```
class ParentClass{
    int num = 10;
}
class ChildClass extends ParentClass{
    int num = 20;
    void showData() {
        System.out.println("Inside the ChildClass");
        System.out.println("ChildClass num = " + num);
        System.out.println("ParentClass num = " + super.num);
    }
}
public class SuperKeywordExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.showData();
        System.out.println("\nInside the non-child class");
        System.out.println("ChildClass num = " + obj.num);
        //System.out.println("ParentClass num = " + super.num); //super can't be used here    }    }
```

super to refer parent class method

When both parent class and child class have method with the same name, then the super keyword is used to refer to the parent class method from child class.

Let's look at the following example java code.

Example

```
class ParentClass{
```

```

        int num1 = 10;
        void showData() {
            System.out.println("\nInside the ParentClass showData method");
            System.out.println("ChildClass num = " + num1);
        }
    }

class ChildClass extends ParentClass{
    int num2 = 20;
    void showData() {
        System.out.println("\nInside the ChildClass showData method");
        System.out.println("ChildClass num = " + num2);
        super.showData();
    }
}

public class SuperKeywordExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.showData();
        //super.showData();           // super can't be used here
    }
}

```

super to call parent class constructor

When an object of child class is created, it automatically calls the parent class default-constructor before it's own. But, the parameterized constructor of parent class must be called explicitly using the **super** keyword inside the child class constructor.

Let's look at the following example java code.

Example

```

class ParentClass{
    int num1;
    ParentClass(){
        System.out.println("\nInside the ParentClass default constructor");
    }
}

```

```

        num1 = 10;
    }
    ParentClass(int value){
        System.out.println("\nInside the ParentClass parameterized constructor");
        num1 = value;
    }
}

class ChildClass extends ParentClass{
    int num2;
    ChildClass(){
        super(100);
        System.out.println("\nInside the ChildClass constructor");
        num2 = 200;
    }
}

public class SuperKeywordExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
    }
}

```

To call the parameterized constructor of the parent class, the super keyword must be the first statement inside the child class constructor, and we must pass the parameter values.

Java final keyword

In java, the final is a keyword and it is used with the following things.

- **With variable (to create constant)**
- **With method (to avoid method overriding)**
- **With class (to avoid inheritance)**

Let's look at each of the above.

final with variables

- When a variable defined with the **final** keyword, it becomes a constant, and it does not allow us to modify the value.
- The variable defined with the final keyword allows only a one-time assignment, once a value assigned to it, never allows us to change it again.
- Let's look at the following example java code.

Example

```
public class FinalVariableExample {  
    public static void main(String[] args) {  
        final int a = 10;  
        System.out.println("a = " + a);  
        a = 100; // Can't be modified  
    }  
}
```

final with methods

- When a method defined with the **final** keyword, it does not allow it to override.
- The final method extends to the child class, but the child class can not override or re-define it.
- It must be used as it has implemented in the parent class.

Let's look at the following example java code.

Example

```
class ParentClass{  
    int num = 10;  
    final void showData() {  
        System.out.println("Inside ParentClass showData() method");  
        System.out.println("num = " + num);  
    }  
}  
  
class ChildClass extends ParentClass{  
    void showData() {  
        System.out.println("Inside ChildClass showData() method");  
        System.out.println("num = " + num);  
    }  
}  
  
public class FinalKeywordExample {  
    public static void main(String[] args) {  
        ChildClass obj = new ChildClass();  
        obj.showData();  
    }  
}
```

final with class

When a class defined with final keyword, it can not be extended by any other class.

Let's look at the following example java code.

Example

```
final class ParentClass{
    int num = 10;
    void showData() {
        System.out.println("Inside ParentClass showData() method");
        System.out.println("num = " + num);
    }
}

class ChildClass extends ParentClass{
}

public class FinalKeywordExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
    }
}
```

Java Polymorphism

- The polymorphism is the process of defining same method with different implementation.
- That means creating multiple methods with different behaviors.
- In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept.

In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments. The ad hoc polymorphism implemented within the class only.

Let's look at the following example java code.

Example

```
import java.util.Arrays;
public class AdHocPolymorphismExample {
    void sorting(int[] list) {
        Arrays.parallelSort(list);
        System.out.println("Integers after sort: " + Arrays.toString(list) );
    }
    void sorting(String[] names) {
        Arrays.parallelSort(names);
        System.out.println("Names after sort: " + Arrays.toString(names) );
    }
    public static void main(String[] args) {
        AdHocPolymorphismExample obj = new AdHocPolymorphismExample();
        int list[] = {2, 3, 1, 5, 4};
        obj.sorting(list);    // Calling with integer array
        String[] names = {"rama", "raja", "shyam", "seeta"};
        obj.sorting(names);    // Calling with String array
    }
}
```

Pure polymorphism

The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept.

In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.

When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.

The pure polymorphism implemented in the inheritance concept only.

Let's look at the following example java code.

Example

```
class ParentClass{
    int num = 10;
    void showData() {
        System.out.println("Inside ParentClass showData() method");
        System.out.println("num = " + num);
    }
}

class ChildClass extends ParentClass{
    void showData() {
        System.out.println("Inside ChildClass showData() method");
        System.out.println("num = " + num);
    }
}

public class PurePolymorphism {
    public static void main(String[] args) {
        ParentClass obj = new ParentClass();
        obj.showData();
        obj = new ChildClass();
        obj.showData();
    }
}
```

Java Method Overriding

The method overriding is the process of re-defining a method in a child class that is already defined in the parent class. When both parent and child classes have the same method, then that method is said to be the overriding method.

The method overriding enables the child class to change the implementation of the method which acquired from parent class according to its requirement.

In the case of the method overriding, the method binding happens at run time. The method binding which happens at run time is known as late binding. So, the method overriding follows late binding. The method overriding is also known as **dynamic method dispatch** or **run time polymorphism** or **pure polymorphism**.

Let's look at the following example java code.

Example

```
class ParentClass{
    int num = 10;
    void showData() {
        System.out.println("Inside ParentClass showData() method");
        System.out.println("num = " + num);
    }
}

class ChildClass extends ParentClass{
    void showData() {
        System.out.println("Inside ChildClass showData() method");
        System.out.println("num = " + num);
    }
}

public class PurePolymorphism {
    public static void main(String[] args) {
        ParentClass obj = new ParentClass();
        obj.showData();
        obj = new ChildClass();
        obj.showData();
    }
}
```

10 Rules for method overriding

While overriding a method, we must follow the below list of rules.

- Static methods cannot be overridden.
- Final methods cannot be overridden.
- Private methods cannot be overridden.
- Constructor cannot be overridden.
- An abstract method must be overridden.
- Use super keyword to invoke overridden method from child class.
- The return type of the overriding method must be same as the parent has it.

- The access specifier of the overriding method can be changed, but the visibility must increase but not decrease. For example, a protected method in the parent class can be made public, but not private, in the child class.
- If the overridden method does not throw an exception in the parent class, then the child class overriding method can only throw the unchecked exception, throwing a checked exception is not allowed.
- If the parent class overridden method does throw an exception, then the child class overriding method can only throw the same, or subclass exception, or it may not throw any exception.

Java Abstract Class

An abstract class is a class that created using abstract keyword. In other words, a class prefixed with abstract keyword is known as an abstract class.

In java, an abstract class may contain abstract methods (methods without implementation) and also non-abstract methods (methods with implementation).

We use the following syntax to create an abstract class.

Syntax

```
abstract class <ClassName>{  
    ...  
}
```

Let's look at the following example java code.

Example

```
import java.util.*;  
abstract class Shape {  
    int length, breadth, radius;  
    Scanner input = new Scanner(System.in);  
    abstract void printArea();  
}  
class Rectangle extends Shape {  
    void printArea() {  
        System.out.println("*** Finding the Area of Rectangle ***");  
        System.out.print("Enter length and breadth: ");  
        length = input.nextInt();  
        breadth = input.nextInt();  
    }  
}
```

```

        System.out.println("The area of Rectangle is: " + length * breadth);
    }
}

class Triangle extends Shape {
    void printArea() {
        System.out.println("\n*** Finding the Area of Triangle ***");
        System.out.print("Enter Base And Height: ");
        length = input.nextInt();
        breadth = input.nextInt();
        System.out.println("The area of Triangle is: " + (length * breadth) / 2);
    }
}

class Cricle extends Shape {
    void printArea() {
        System.out.println("\n*** Finding the Area of Cricle ***");
        System.out.print("Enter Radius: ");
        radius = input.nextInt();
        System.out.println("The area of Cricle is: " + 3.14f * radius * radius);
    }
}

public class AbstractClassExample {
    public static void main(String[] args) {
        Rectangle rec = new Rectangle();
        rec.printArea();
        Triangle tri = new Triangle();
        tri.printArea();
        Cricle cri = new Cricle();
        cri.printArea();
    }
}

```

□ An abstract class can not be instantiated but can be referenced. That means we can not create an object of an abstract class, but base reference can be created.

In the above example program, the child class objects are created to invoke the overridden abstract method. But we may also create base class reference and assign it with child class instance to invoke the same. The main method of the above program can be written as follows that produce the same output.

Example

```
public static void main(String[] args) {  
    Shape obj = new Rectangle(); //Base class reference to Child class instance  
    obj.printArea();  
    obj = new Triangle();  
    obj.printArea();  
    obj = new Circle();  
    obj.printArea();  
}
```

8 Rules for method overriding

An abstract class must follow the below list of rules.

- An abstract class must be created with abstract keyword.
- An abstract class can be created without any abstract method.
- An abstract class may contain abstract methods and non-abstract methods.
- An abstract class may contain final methods that can not be overridden.
- An abstract class may contain static methods, but the abstract method can not be static.
- An abstract class may have a constructor that gets executed when the child class object created.
- An abstract method must be overridden by the child class, otherwise, it must be defined as an abstract class.
- An abstract class can not be instantiated but can be referenced.

Java Object Class

In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the **java.lang** package.

Every class in the java programming language is a subclass of Object class by default.

The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Methods of Object class

The following table depicts all built-in methods of Object class in java.

Method	Description	Return Value
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int
equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	eturns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void

Method	Description	Return Value
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

Java Forms of Inheritance

- The inheritance concept used for the number of purposes in the java programming language.
- One of the main purposes is substitutability.
- The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object.
- For example, if B is a child class of A, anywhere we expect an instance of A we can use an instance of B.
- The substitutability can achieve using inheritance, whether using extends or implements keywords.

The following are the different forms of inheritance in java.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

Specification

This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them. The java provides concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.

Construction

This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

Eextension

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

Combination

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

Inheritance is used in a variety of way and for a variety of different purposes .

- Inheritance for Specialization
- Inheritance for Specification
- Inheritance for Construction
- Inheritance for Extension
- Inheritance for Limitation
- Inheritance for Combination

One or many of these forms may occur in a single case.

Forms of Inheritance (- Inheritance for Specialization -)

Most commonly used inheritance and sub classification is for specialization.

Always creates a subtype, and the principles of substitutability is explicitly upheld.

It is the most ideal form of inheritance.

An example of subclassification for specialization is;

```
public class PinBallGame extends Frame {  
    // body of class  
}
```

Specialization

- By far the most common form of inheritance is for specialization.
 - Child class is a specialized form of parent class
 - Principle of substitutability holds
- A good example is the Java hierarchy of Graphical components in the AWT:
 - Component
 - Label
 - Button
 - TextComponent
 - TextArea
 - TextField
 - CheckBox
 - ScrollBar

Forms of Inheritance (- Inheritance for Specification -)

This is another most common use of inheritance. Two different mechanisms are provided by Java, **interface** and **abstract**, to make use of *subclassification for specification*. Subtype is formed and substitutability is explicitly upheld.

Mostly, not used for refinement of its parent class, but instead is used for definitions of the properties provided by its parent.

```
class FireButtonListener implements ActionListener {  
    // body of class  
}  
class B extends A {  
    // class A is defined as abstract specification class  
}
```

Specification

- The next most common form of inheritance involves specification. The parent class specifies some behavior, but does not implement the behavior
 - Child class implements the behavior
 - Similar to Java interface or abstract class
 - When parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes
- Example, Java 1.1 Event Listeners:
ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

Forms of Inheritance (- Inheritance for Construction -)

Child class inherits most of its functionality from parent, but may change the name or parameters of methods inherited from parent class to form its interface.

This type of inheritance is also widely used for code reuse purposes. It simplifies the construction of newly formed abstraction but is not a form of subtype, and often violates substitutability.

Example is **Stack** class defined in Java libraries.

Construction

- The parent class is used only for its behavior, the child class has no *is-a* relationship to the parent.
 - Child modify the arguments or names of methods
- An example might be subclassing the idea of a *Set* from an existing *List* class.
 - Child class is not a more specialized form of parent class; no substitutability

Forms of Inheritance (- Inheritance for Extension -)

Subclassification for extension occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes.

Such subclasses are always subtypes, and substitutability can be used.

Example of this type of inheritance is done in the definition of the class Properties which is an extension of the class Hashtable.

Generalization or Extension

- The child class generalizes or extends the parent class by providing more functionality
 - In some sense, opposite of subclassing for specialization
- The child doesn't change anything inherited from the parent, it simply adds new features
 - Often used when we cannot modify existing base parent class
- Example, ColoredWindow inheriting from Window
 - Add additional data fields
 - Override window display methods

Forms of Inheritance (- Inheritance for Limitation -)

Subclassification for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of its parent class.

Like subclassification for extension, this form of inheritance occurs most frequently when a programmer is building on a base of existing classes.

Is not a subtype, and substitutability is not proper.

Limitation

- The child class limits some of the behavior of the parent class.
- Example, you have an existing List data type, and you want a Stack
- Inherit from List, but override the methods that allow access to elements other than top so as to produce errors.

Forms of Inheritance (- Inheritance for Combination -)

This types of inheritance is known as *multiple inheritance* in Object Oriented Programming.

Although the Java does not permit a subclass to be formed by inheritance from more than one parent class, several approximations to the concept are possible.

Example of this type is Hole class defined as;

```
class Hole extends Ball implements PinBallTarget{  
// body of class  
}
```

Combination

- Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.
- Example: Mouse and TouchPad and JoyStick
- Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

Summary of Forms of Inheritance

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.
- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- Generalization. The child class modifies or overrides some of the methods of the parent class.
- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.
- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

The Benefits of Inheritance

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

The Costs of Inheritance

- Execution Speed
- Program Size
- Message-Passing Overhead
- Program Complexity (in overuse of inheritance)

Java Programming (R20CSE2204)

UNIT - II

Packages- Defining a Package, CLASSPATH, Access protection, importing packages.
Interfaces- defining an interface, implementing interfaces, Nested interfaces, applying interfaces, variables in interfaces and extending interfaces.

Stream based I/O (java.io)– The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and writing Files, Random access file operations, The Console class, Serialization, Enumerations, auto boxing, generics. Database Connectivity.

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of using packages in Java

1. **Maintenance:** Java packages are used for proper maintenance. If any developer newly joined a company, he can easily reach to files needed.
2. **Reusability:** We can place the common code in a common folder so that everybody can check that folder and use it whenever needed.
3. **Name conflict:** Packages help to resolve the naming conflict between the two classes with the same name. Assume that there are two classes with the same name Student.java. Each class will be stored in its own packages such as stdPack1 and stdPack2 without having any conflict of names.
4. **Organized:** It also helps in organizing the files within our project.
5. **Access Protection:** A package provides access protection. It can be used to provide visibility control. The members of the class can be defined in such a manner that they will be visible only to elements of that package.

Types of Packages in Java

There are two different types of packages in Java. They are:

1. User-defined Package

The package which is defined by the user is called a User-defined package. It contains user-defined classes and interfaces.

Creating package in Java

Java supports a keyword called “package” which is used to create user-defined packages in java programming. It has the following general form:

```
package packageName;
```

Here, packageName is the name of package. The package statement must be the first line in a java source code file followed by one or more classes.

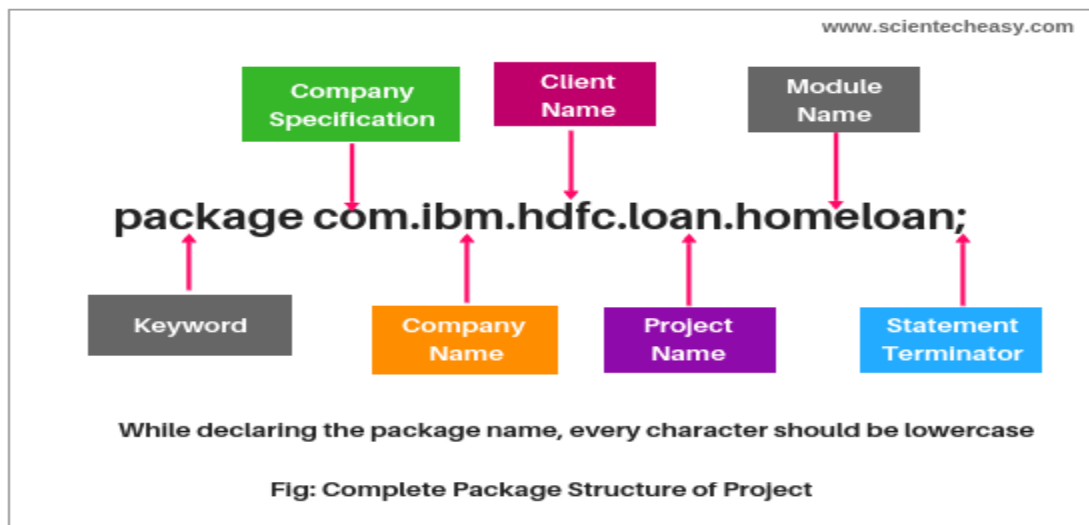
For example:

```
package myPackage;
public class A {
    // class body
}
```

Naming Convention to declare User-defined Package in Real-time Project

While developing your project, you must follow some naming conventions regarding packages declaration. Let’s take an example to understand the convention.

See below a complete package structure of the project.



1. Suppose you are working in IBM and the domain name of IBM is www.ibm.com. You can declare the package by reversing the domain like this:

```
package com.ibm;
```

where,

- com → It is generally company specification name and the folder starts with com which is called root folder.
 - ibm → Company name where the product is developed. It is the subfolder.
2. hdfc → Client name for which we are developing our product or working for the project.
3. loan → Name of the project.

4. homeloan → It is the name of the modules of the loan project. There are a number of modules in the loan project like a Home loan, Car loan, or Personal loan. Suppose you are working for Home loan module.

This is a complete packages structure like a professional which is adopted in the company. Another example is:

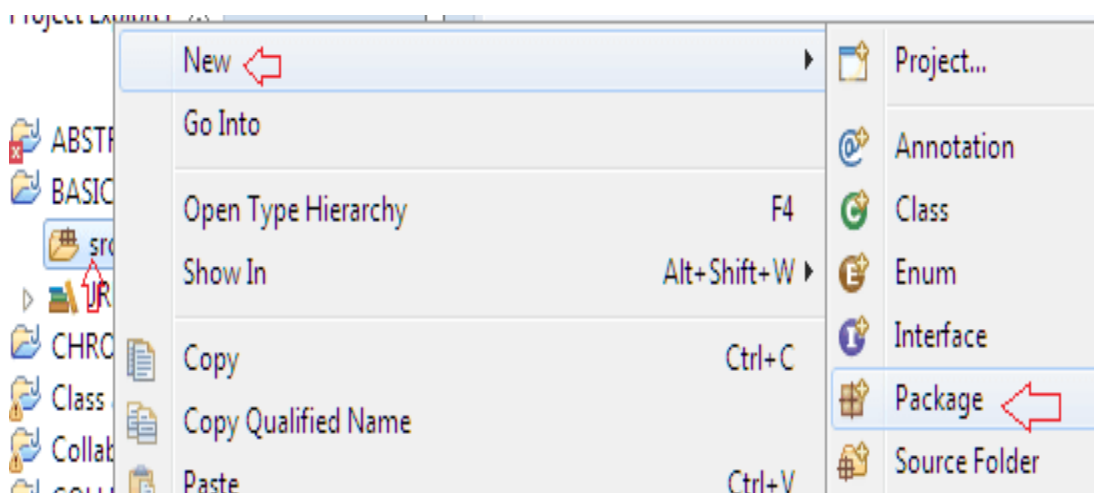
```
package com.tcs.icici.loan.carloan.penalty;
```

Note: Keep in mind Root folder should be always the same for all the classes.

How to create Package in Eclipse IDE?

In Eclipse IDE, there are the following steps to create a package in java. They are as follows:

1. Right-click on the src folder as shown in the below screenshot.



2. Go to New option and then click on package.

3. A window dialog box will appear where you have to enter the package name according to the naming convention and click on Finish button.

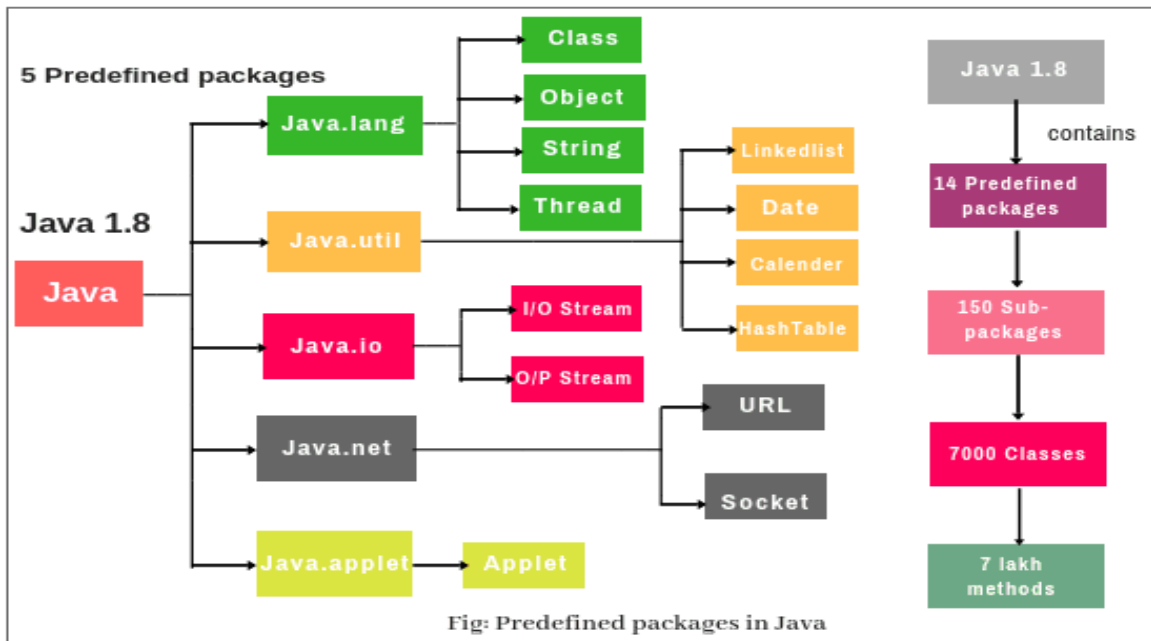
Once the package is created, a package folder will be created in your file system where you can create classes and interfaces.

2. Predefined Packages in Java (Built-in Packages)

Predefined packages in java are those which are developed by Sun Microsystems. They are also called built-in packages in java.

These packages consist of a large number of predefined classes, interfaces, and methods that are used by the programmer to perform any task in his programs.

Java APIs contains the following predefined packages, as shown in the below figure:



Core packages:

1. **Java.lang:** lang stands for language. The Java language package consists of java classes and interfaces that form the core of the Java language and the JVM. It is a fundamental package that is useful for writing and executing all Java programs.

Examples are classes, objects, String, Thread, predefined data types, etc. It is imported automatically into the Java programs.

2. **Java.io:** io stands for input and output. It provides a set of I/O streams that are used to read and write data to files. A stream represents a flow of data from one place to another place.

3. **Java.util:** util stands for utility. It contains a collection of useful utility classes and related interfaces that implement data structures like LinkedList, Dictionary, HashTable, stack, vector, Calendar, data utility, etc.

4. **Java.net:** net stands for network. It contains networking classes and interfaces for networking operations. The programming related to client-server can be done by using this package.

Window Toolkit and Applet:

1. **Java.awt:** awt stands for abstract window toolkit. The Abstract window toolkit packages contain the GUI(Graphical User Interface) elements such as buttons, lists, menus, and text areas. Programmers can develop programs with colorful screens, paintings, and images, etc using this package.

2. **Java.awt.image:** It contains classes and interfaces for creating images and colors.

3. **Java.applet:** It is used for creating applets. Applets are programs that are executed from the server into the client machine on a network.

4. **Java.text:** This package contains two important classes such as DateFormat and NumberFormat. The class DateFormat is used to format dates and times. The NumberFormat is used to format numeric values.

5. **Java.sql:** SQL stands for the structured query language. This package is used in a Java program to connect databases like Oracle or Sybase and retrieve the data from them.

Access protection in java packages

- In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public.
- In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.
- In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

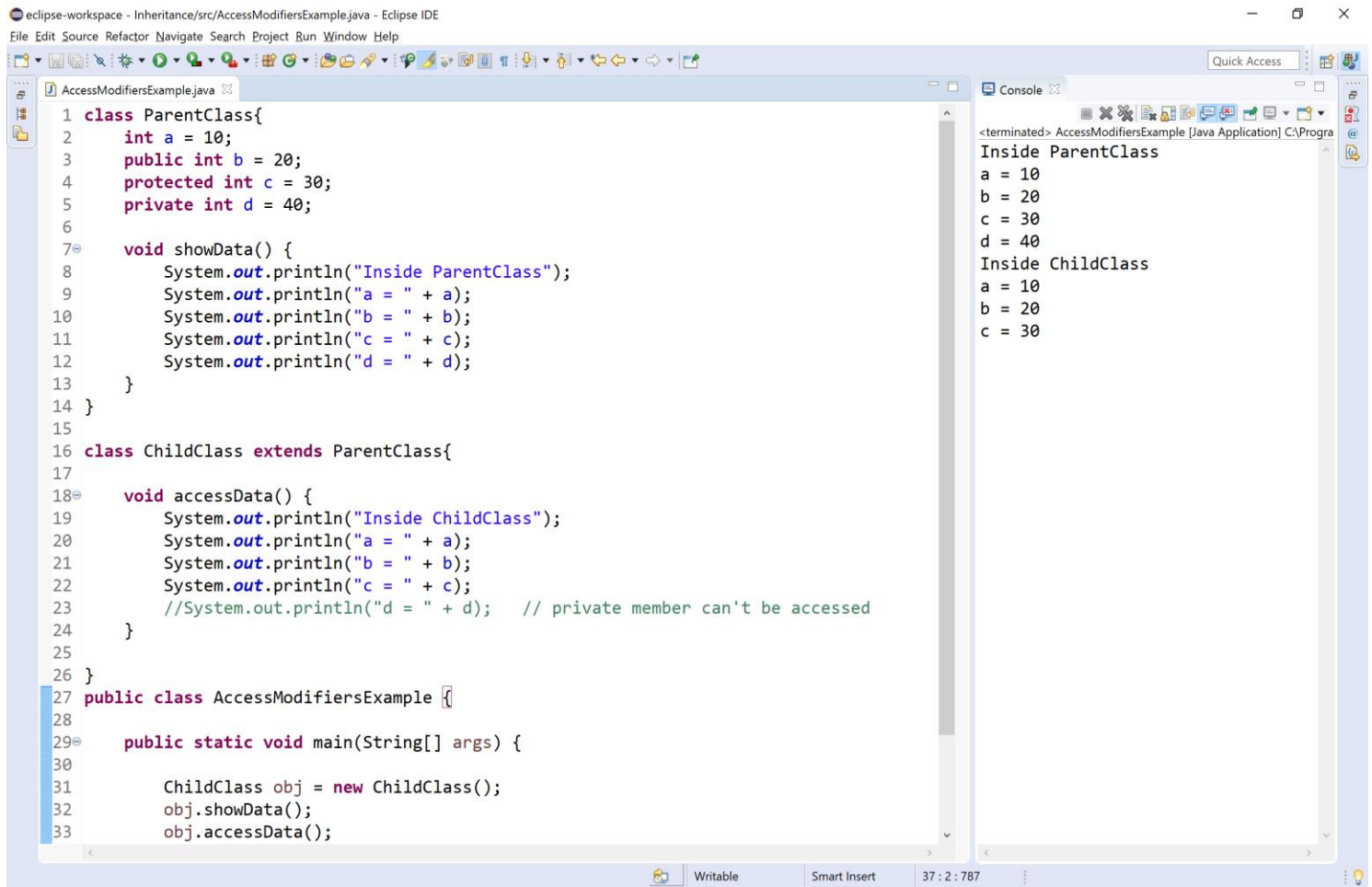
Access control for members of class and interface in java

Access Specifier \ Accessibility Location	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

www.btechsmartclass.com

- ☐ The **public** members can be accessed everywhere.
- ☐ The **private** members can be accessed only inside the same class.
- ☐ The **protected** members are accessible to every child class (same package or other packages).
- ☐ The **default** members are accessible within the same package but not outside the package.

Example When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE with a Java project named 'Inheritance/src/AccessModifiersExample.java'. The code defines two classes: 'ParentClass' and 'ChildClass'. 'ParentClass' has four attributes: 'a' (int, 10), 'b' (public int, 20), 'c' (protected int, 30), and 'd' (private int, 40). It has a 'showData()' method that prints the values of these attributes. 'ChildClass' extends 'ParentClass' and has an 'accessData()' method that prints the values of 'a', 'b', and 'c', but it has a commented-out line for 'd' because it is private. The 'AccessModifiersExample' class has a 'main' method that creates a 'ChildClass' object and calls both 'showData()' and 'accessData()' methods. The console output shows the results of these calls: 'Inside ParentClass' followed by 'a = 10', 'b = 20', 'c = 30', 'd = 40', and 'Inside ChildClass' followed by 'a = 10', 'b = 20', 'c = 30'.

```
1 class ParentClass{
2     int a = 10;
3     public int b = 20;
4     protected int c = 30;
5     private int d = 40;
6
7     void showData() {
8         System.out.println("Inside ParentClass");
9         System.out.println("a = " + a);
10        System.out.println("b = " + b);
11        System.out.println("c = " + c);
12        System.out.println("d = " + d);
13    }
14 }
15
16 class ChildClass extends ParentClass{
17
18     void accessData() {
19         System.out.println("Inside ChildClass");
20         System.out.println("a = " + a);
21         System.out.println("b = " + b);
22         System.out.println("c = " + c);
23         //System.out.println("d = " + d); // private member can't be accessed
24     }
25 }
26
27 public class AccessModifiersExample {
28
29     public static void main(String[] args) {
30
31         ChildClass obj = new ChildClass();
32         obj.showData();
33         obj.accessData();
34     }
35 }
```

Console Output:

```
<terminated> AccessModifiersExample [Java Application] C:\Progra
Inside ParentClass
a = 10
b = 20
c = 30
d = 40
Inside ChildClass
a = 10
b = 20
c = 30
```

Importing Packages in java

- In java, the **import** keyword used to import built-in and user-defined packages.
- When a package has imported, we can refer to all the classes of that package using their name directly.
- The import statement must be after the package statement, and before any other statement.
- Using an import statement, we may import a specific class or all the classes from a package.

- Using one import statement, we may import only one package or a class.
- Using an import statement, we can not import a class directly, but it must be a part of a package.
- A program may contain any number of import statements.

Importing specific class

Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.

Syntax

```
import packageName.ClassName;
```

Let's look at an import statement to import a built-in package and Scanner class.

Example

```
package myPackage;
import java.util.Scanner;
public class ImportingExample {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        int i = read.nextInt();
        System.out.println("You have entered a number " + i);
    }
}
```

In the above code, the class **ImportingExample** belongs to **myPackage** package, and it also importing a class called **Scanner** from **java.util** package.

Importing all the classes

Using an importing statement, we can import all the classes of a package. To import all the classes of the package, we use * symbol. The following syntax is employed to import all the classes of a package.

Syntax

```
import packageName.*;
```

Let's look at an import statement to import a built-in package.

Example


```
package myPackage;
import java.util.*;
public class ImportingExample {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        int i = read.nextInt();
        System.out.println("You have entered a number " + i);
        Random rand = new Random();
        int num = rand.nextInt(100);
        System.out.println("Randomly generated number " + num);
    }
}
```

In the above code, the class **ImportingExample** belongs to **myPackage** package, and it also importing all the classes like Scanner, Random, Stack, Vector, ArrayList, HashSet, etc. from the **java.util** package.

□ The import statement imports only classes of the package, but not sub-packages and its classes.

□ We may also import sub-packages by using a symbol '.' (dot) to separate parent package and sub-package.

Consider the following import statement.

```
import java.util.*;
```

The above import statement **util** is a sub-package of **java** package. It imports all the classes of **util** package only, but not classes of **java** package.

Defining an interface in java

- In java, an **interface** is similar to a class, but it contains abstract methods and static final variables only. The interface in Java is another mechanism to achieve abstraction.
- We may think of an interface as a completely abstract class. None of the methods in the interface has an implementation, and all the variables in the interface are constants.
- All the methods of an interface, implemented by the class that implements it.

- The interface in java enables java to support multiple-inheritance. An interface may extend only one interface, but a class may implement any number of interfaces.

- ☐ An interface is a container of abstract methods and static final variables.
- ☐ An interface, implemented by a class. (**class implements interface**).
- ☐ An interface may extend another interface. (**Interface extends Interface**).
- ☐ An interface never implements another interface, or class.
- ☐ A class may implement any number of interfaces.
- ☐ We can not instantiate an interface.
- ☐ Specifying the keyword abstract for interface methods is optional, it automatically added.
- ☐ All the members of an interface are public by default.

Defining an interface is similar to that of a class. We use the keyword interface to define an interface. All the members of an interface are public by default. The following is the syntax for defining an interface.

Syntax

```
interface InterfaceName{  
    ...  
    members declaration;    ...  
}
```

Let's look at an example code to define an interface.

Example

```
interface HumanInterfaceExample {  
    void learn(String str);  
    void work();  
    int duration = 10;  
}
```

In the above code defines an interface **HumanInterfaceExample** that contains two abstract methods learn(), work() and one constant duration.

Every interface in Java is auto-completed by the compiler. For example, in the above example code, no member is defined as public, but all are public automatically.

The above code automatically converted as follows.

Converted code

```
interface HumanInterfaceExample {  
    public abstract void learn(String str);  
    public abstract void work();  
    public static final int duration = 10;  
}
```

In the next tutorial, we will learn how a class implements an interface to make use of the interface concept.

Implementing an Interface in java

- In java, an **interface** is implemented by a class. The class that implements an interface must provide code for all the methods defined in the interface, otherwise, it must be defined as an abstract class.
- The class uses a keyword **implements** to implement an interface. A class can implement any number of interfaces. When a class wants to implement more than one interface, we use the **implements** keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defining a class that implements an interface.

Syntax

```
class className implements InterfaceName{  
    ...  
    boby-of-the-class  
    ...  
}
```

Let's look at an example code to define a class that implements an interface.

Example

```
interface Human {  
    void learn(String str);  
    void work();  
    int duration = 10;  
}  
  
class Programmer implements Human{  
    public void learn(String str) {  
        System.out.println("Learn using " + str);  
    }  
    public void work() {  
        System.out.println("Develop applications");  
    }  
}  
  
public class HumanTest {  
    public static void main(String[] args) {  
        Programmer trainee = new Programmer();  
        trainee.learn("coding");  
        trainee.work();  
    }  
}
```

In the above code defines an interface **Human** that contains two abstract methods learn(), work() and one constant duration. The class Programmer implements the interface. As it implementing the Human interface it must provide the body of all the methods those defined in the Human interface.

Implementing multiple Interfaces

When a class wants to implement more than one interface, we use the **implements** keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defining a class that implements multiple interfaces.

Syntax

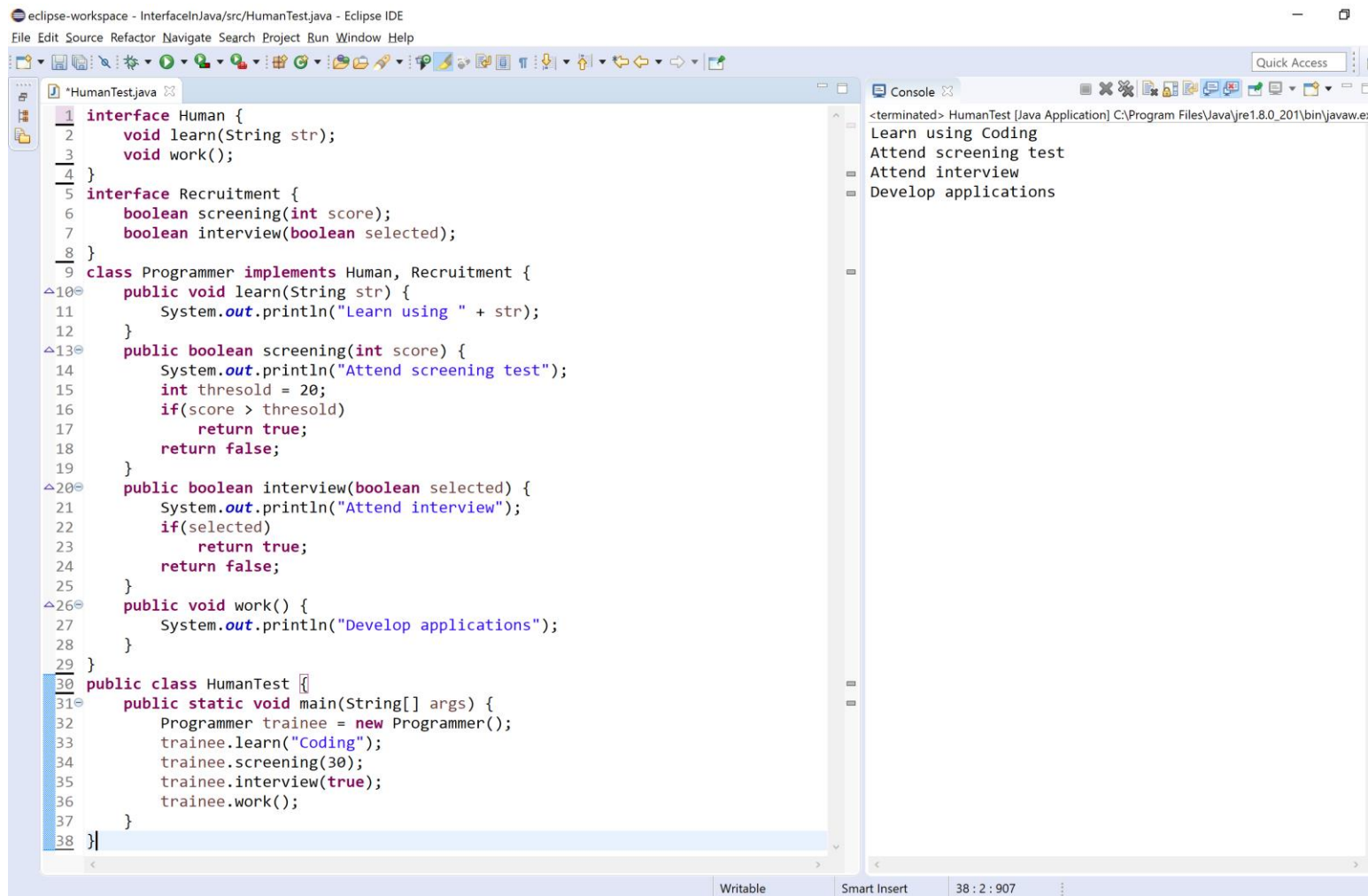
```
class className implements InterfaceName1, InterfaceName2, ...{  
  
    ...  
  
    boby-of-the-class  
  
    ...  
  
}
```

Let's look at an example code to define a class that implements multiple interfaces.

Example

In the code defines two interfaces **Human** and **Recruitment**, and a class **Programmer** implements both the interfaces.

When we run the above program, it produce the following output.



The screenshot shows the Eclipse IDE with a Java project named 'InterfaceInJava'. The main editor displays the file 'HumanTest.java' with the following code:

```
1 interface Human {  
2     void learn(String str);  
3     void work();  
4 }  
5 interface Recruitment {  
6     boolean screening(int score);  
7     boolean interview(boolean selected);  
8 }  
9 class Programmer implements Human, Recruitment {  
10     public void learn(String str) {  
11         System.out.println("Learn using " + str);  
12     }  
13     public boolean screening(int score) {  
14         System.out.println("Attend screening test");  
15         int threshold = 20;  
16         if(score > threshold)  
17             return true;  
18         return false;  
19     }  
20     public boolean interview(boolean selected) {  
21         System.out.println("Attend interview");  
22         if(selected)  
23             return true;  
24         return false;  
25     }  
26     public void work() {  
27         System.out.println("Develop applications");  
28     }  
29 }  
30 public class HumanTest {  
31     public static void main(String[] args) {  
32         Programmer trainee = new Programmer();  
33         trainee.learn("Coding");  
34         trainee.screening(30);  
35         trainee.interview(true);  
36         trainee.work();  
37     }  
38 }
```

The console window on the right shows the output of the program:

```
<terminated> HumanTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe  
Learn using Coding  
Attend screening test  
Attend interview  
Develop applications
```

The status bar at the bottom indicates the file is 'Writable', 'Smart Insert' is active, and the cursor is at line 38, column 2, row 907.

Nested Interfaces in java

- In java, an interface may be defined inside another interface, and also inside a class.
- The interface that defined inside another interface or a class is known as nested interface.
- The nested interface is also referred as inner interface.

- The nested interface declared within an interface is public by default.
- The nested interface declared within a class can be with any access modifier.
- Every nested interface is static by default.

The nested interface cannot be accessed directly. We can only access the nested interface by using outer interface or outer class name followed by dot(.), followed by the nested interface name.

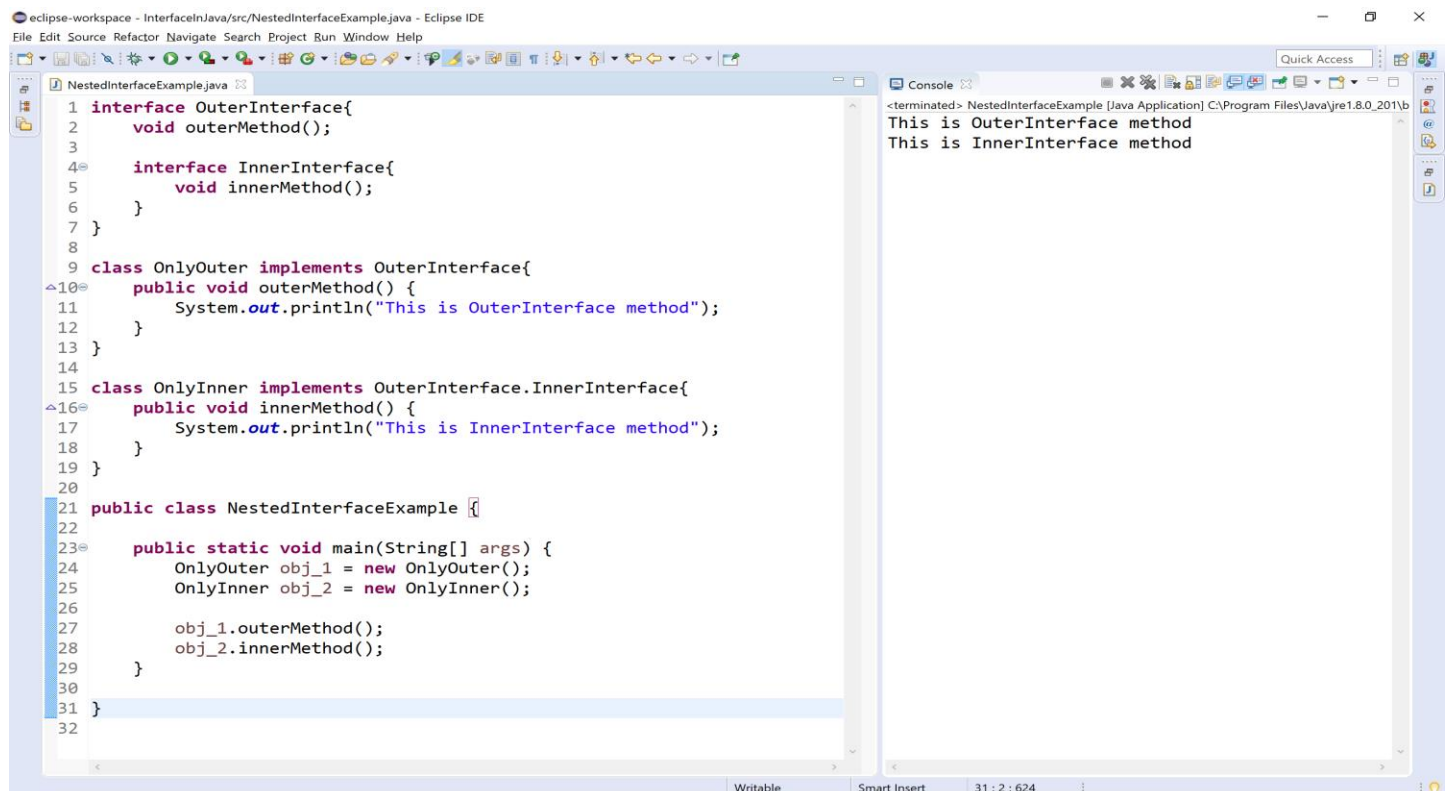
Nested interface inside another interface

The nested interface that defined inside another interface must be accessed as **OuterInterface.InnerInterface**.

Let's look at an example code to illustrate nested interfaces inside another interface.

Example

When we run the above program, it produces the following output.



The screenshot shows the Eclipse IDE with a Java project. The editor displays the following code:

```
1 interface OuterInterface{
2     void outerMethod();
3
4     interface InnerInterface{
5         void innerMethod();
6     }
7 }
8
9 class OnlyOuter implements OuterInterface{
10     public void outerMethod() {
11         System.out.println("This is OuterInterface method");
12     }
13 }
14
15 class OnlyInner implements OuterInterface.InnerInterface{
16     public void innerMethod() {
17         System.out.println("This is InnerInterface method");
18     }
19 }
20
21 public class NestedInterfaceExample {
22
23     public static void main(String[] args) {
24         OnlyOuter obj_1 = new OnlyOuter();
25         OnlyInner obj_2 = new OnlyInner();
26
27         obj_1.outerMethod();
28         obj_2.innerMethod();
29     }
30 }
31 }
32
```

The console output shows the following lines:

```
<terminated> NestedInterfaceExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
This is OuterInterface method
This is InnerInterface method
```

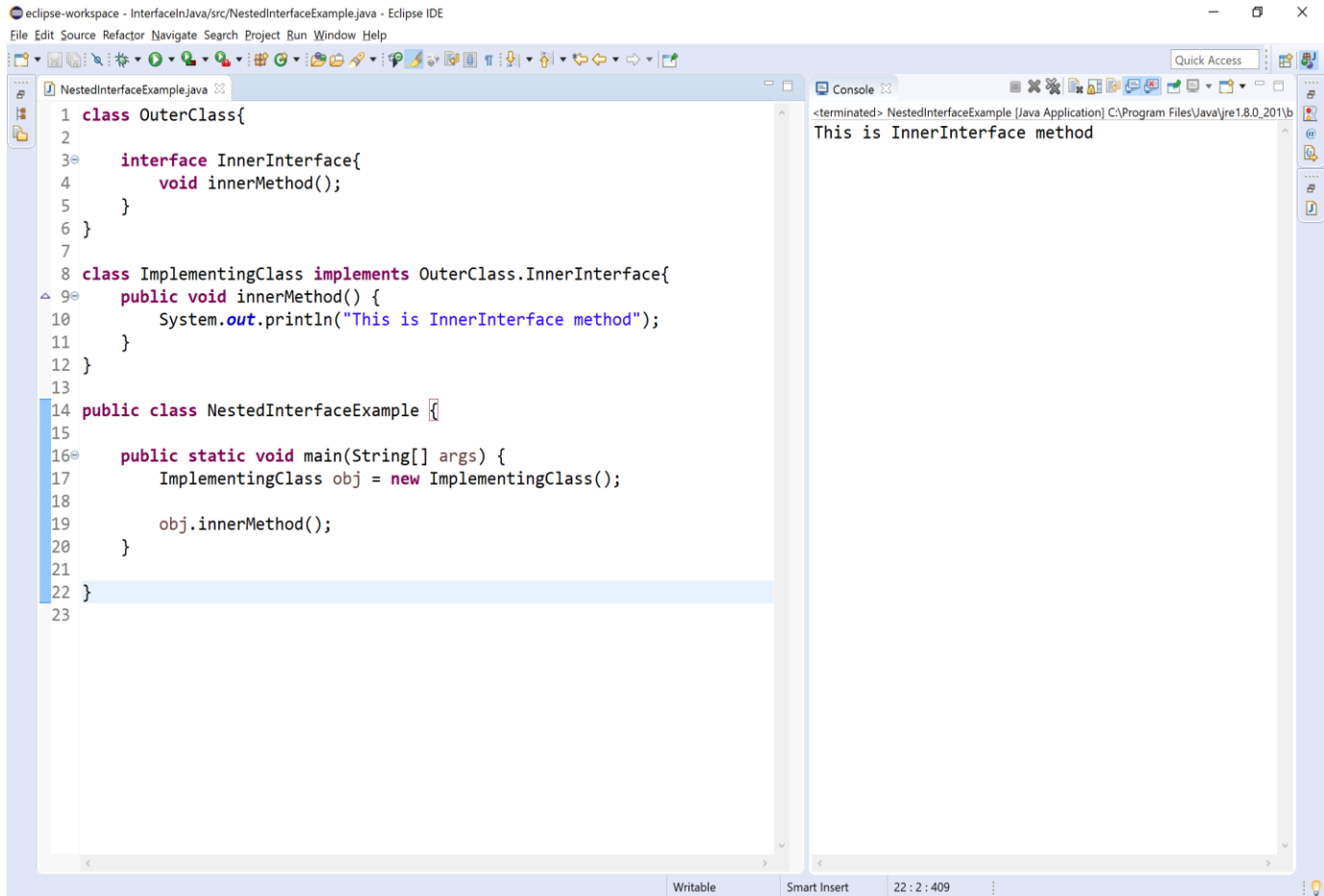
Nested interface inside a class

The nested interface that defined inside a class must be accessed as **ClassName.InnerInterface**.

Let's look at an example code to illustrate nested interfaces inside a class.

Example

When we run the above program, it produce the following output.



The screenshot shows the Eclipse IDE with a Java project named 'InterfaceInJava'. The main editor displays the file 'NestedInterfaceExample.java' with the following code:

```
1 class OuterClass{
2
3     interface InnerInterface{
4         void innerMethod();
5     }
6 }
7
8 class ImplementingClass implements OuterClass.InnerInterface{
9     public void innerMethod() {
10         System.out.println("This is InnerInterface method");
11     }
12 }
13
14 public class NestedInterfaceExample {
15
16     public static void main(String[] args) {
17         ImplementingClass obj = new ImplementingClass();
18
19         obj.innerMethod();
20     }
21 }
22
23
```

The console on the right shows the output of the program:

```
<terminated> NestedInterfaceExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
This is InnerInterface method
```

Variables in Java Interfaces

- In java, an interface is a completely abstract class.
- An interface is a container of abstract methods and static final variables.
- The interface contains the static final variables.
- The variables defined in an interface cannot be modified by the class that implements the interface, but it may use as it defined in the interface.

- The variable in an interface is public, static, and final by default.
- If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.

- ❑ No access modifier is allowed except the public for interface variables.
- ❑ Every variable of an interface must be initialized in the interface itself.
- ❑ The class that implements an interface can not modify the interface variable, but it may use as it defined in the interface.

Let's look at an example code to illustrate variables in an interface.

Example

The screenshot shows the Eclipse IDE with a Java project. The main editor displays the file `InterfaceVariablesExample.java`. The code defines an interface `SampleInterface` with a public static variable `UPPER_LIMIT` set to 100. A comment indicates that `LOWER_LIMIT` must be initialized. Below the interface, a class `InterfaceVariablesExample` implements `SampleInterface`. Its `main` method prints the value of `UPPER_LIMIT` and includes a commented-out line that attempts to modify it, which is noted as not being allowed. The console on the right shows the output: `UPPER LIMIT = 100`.

```
1 interface SampleInterface{
2
3     int UPPER_LIMIT = 100;
4
5     //int LOWER_LIMIT; // Error - must be initialised
6
7 }
8
9 public class InterfaceVariablesExample implements SampleInterface{
10
11     public static void main(String[] args) {
12
13         System.out.println("UPPER LIMIT = " + UPPER_LIMIT);
14
15         // UPPER_LIMIT = 150; // Can not be modified
16     }
17
18 }
19
```

Console output: `UPPER LIMIT = 100`

Extending an Interface in java

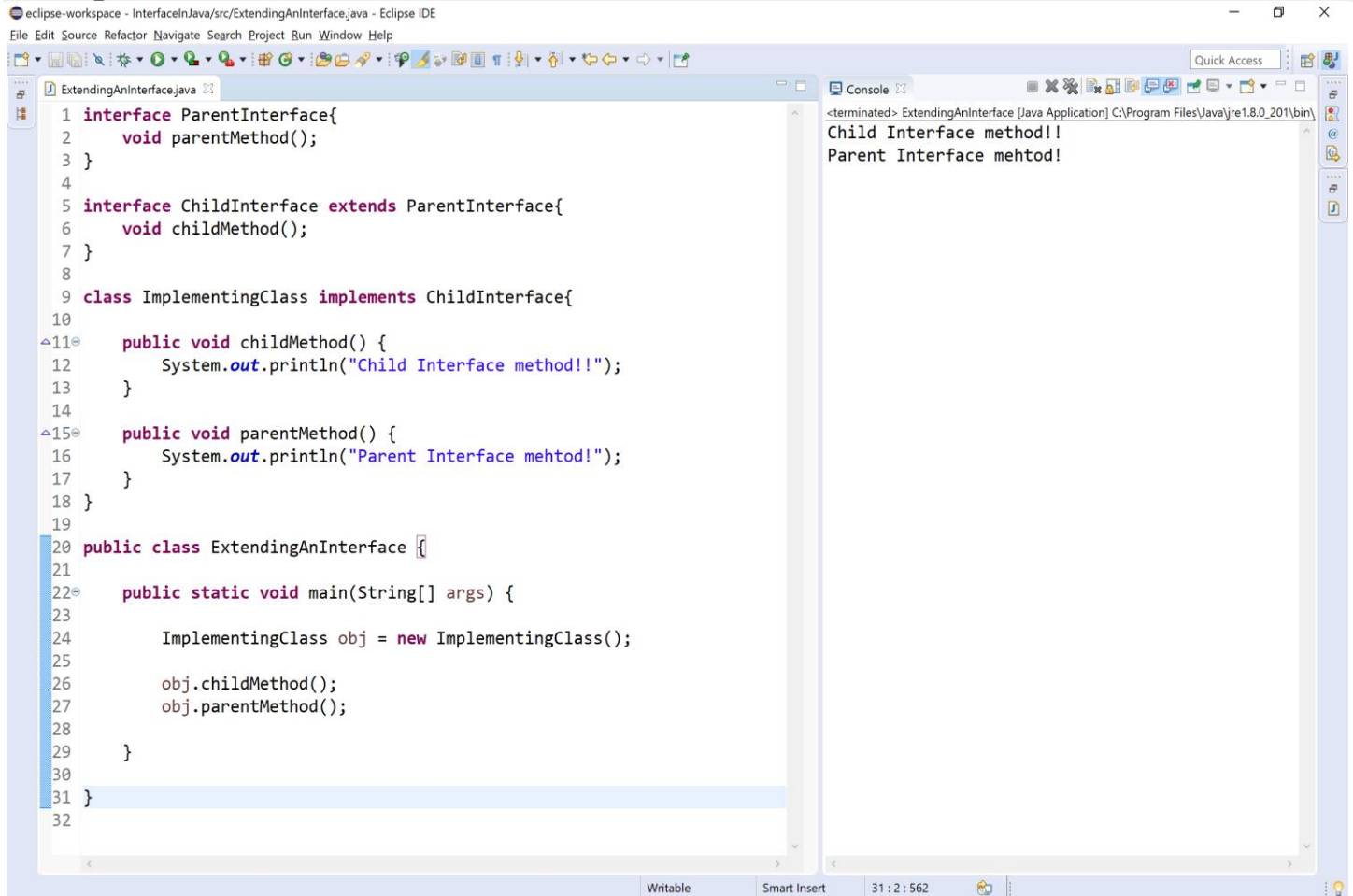
- In java, an interface can extend another interface.
- When an interface wants to extend another interface, it uses the keyword **extends**.
- The interface that extends another interface has its own members and all the members defined in its parent interface too.
- The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces, otherwise, it needs to be defined as abstract class.

- ❑ An interface can extend another interface.
- ❑ An interface can not extend multiple interfaces.
- ❑ An interface can implement neither an interface nor a class.

❑ The class that implements child interface needs to provide code for all the methods defined in both child and parent interfaces.

Let's look at an example code to illustrate extending an interface.

Example



The screenshot shows the Eclipse IDE with a Java project. The main editor displays the file `ExtendingAnInterface.java` with the following code:

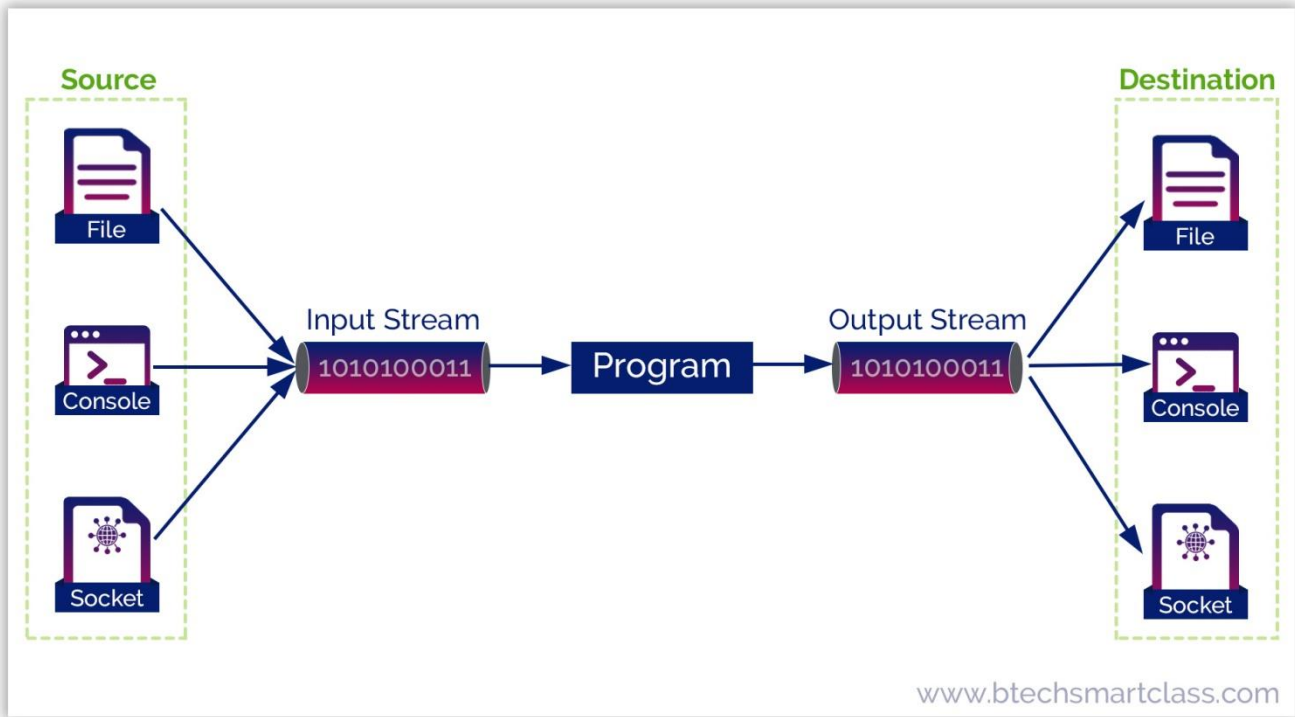
```
1 interface ParentInterface{
2     void parentMethod();
3 }
4
5 interface ChildInterface extends ParentInterface{
6     void childMethod();
7 }
8
9 class ImplementingClass implements ChildInterface{
10
11     public void childMethod() {
12         System.out.println("Child Interface method!!");
13     }
14
15     public void parentMethod() {
16         System.out.println("Parent Interface mehtod!");
17     }
18 }
19
20 public class ExtendingAnInterface {
21
22     public static void main(String[] args) {
23
24         ImplementingClass obj = new ImplementingClass();
25
26         obj.childMethod();
27         obj.parentMethod();
28     }
29 }
30
31 }
32
```

The console on the right shows the output of the program:

```
<terminated> ExtendingAnInterface [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\
Child Interface method!!
Parent Interface mehtod!
```

Stream in java

- In java, the IO operations are performed using the concept of streams.
- Generally, a stream means a continuous flow of data.
- In java, a stream is a logical container of data that allows us to read from and write to it.
- A stream can be linked to a data source, or data destination, like a console, file or network connection by java IO system.
- The stream-based IO operations are faster than normal IO operations.
- The Stream is defined in the `java.io` package.
- To understand the functionality of java streams, look at the following picture.

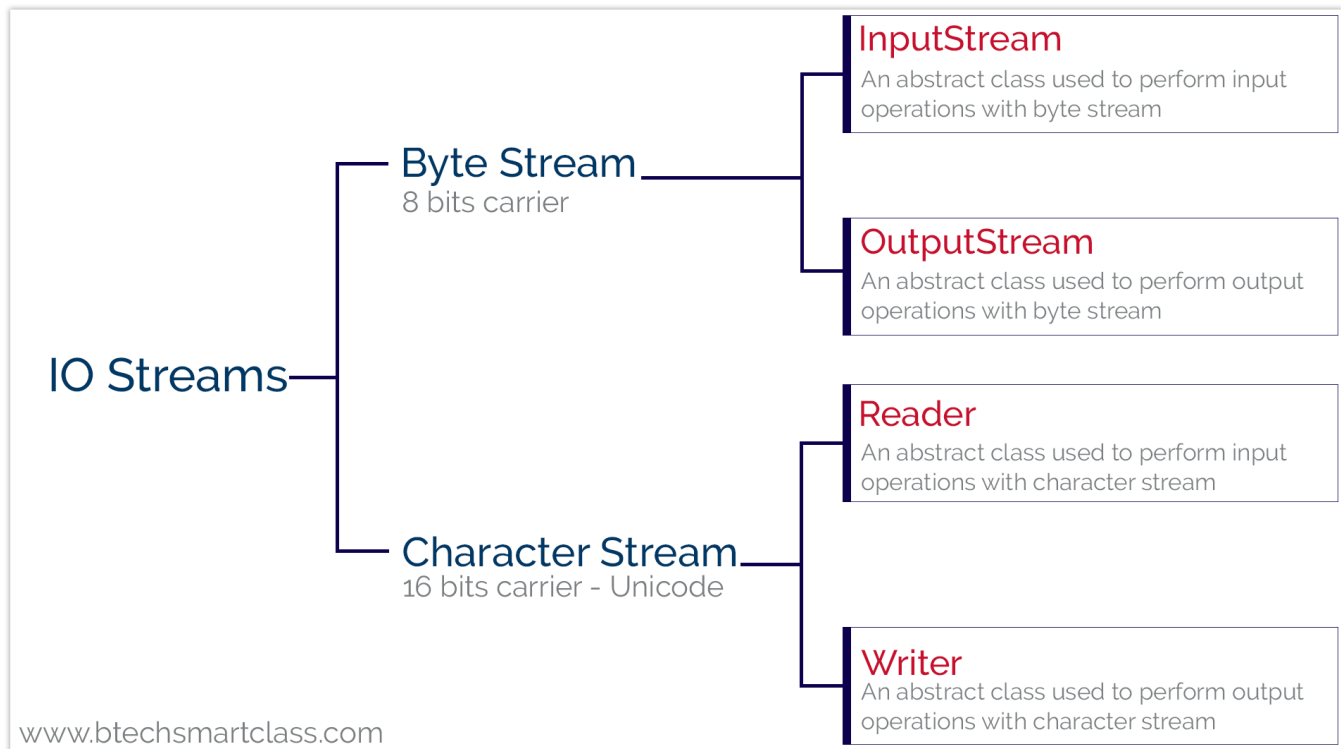


- In java, the stream-based IO operations are performed using two separate streams input stream and output stream.
- The input stream is used for input operations, and the output stream is used for output operations.
- The java stream is composed of bytes.
- In Java, every program creates 3 streams automatically, and these streams are attached to the console.
- **System.out**: standard output stream for console output operations.
- **System.in**: standard input stream for console input operations.
- **System.err**: standard error stream for console error output operations.
- The Java streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

Java provides two types of streams, and they are as follows.

- **Byte Stream**
- **Character Stream**

The following picture shows how streams are categorized, and various built-in classes used by the java IO system.



Both character and byte streams essentially provides a convenient and efficient way to handle data streams in Java.

Byte Stream in java

- In java, the byte stream is an 8 bits carrier.
- The byte stream in java allows us to transmit 8 bits of data.
- In Java 1.0 version all IO operations were byte oriented, there was no other stream (character stream).
- The java byte stream is defined by two abstract classes, **InputStream** and **OutputStream**.
- The InputStream class used for byte stream based input operations, and the OutputStream class used for byte stream based output operations.
- The InputStream and OutputStream classes have several concrete classes to perform various IO operations based on the byte stream.
- The following picture shows the classes used for byte stream operations.

Byte Stream

8 bits carrier

InputStream

- BufferedInputStream**
Used for Buffered Input Stream
- ByteArrayInputStream**
Used for reading from a byte array
- DataInputStream**
Used for reading java standard data type
- ObjectInputStream** - Input stream for objects
- FileInputStream** - Used for reading from a File
- PipedInputStream** - Input pipe
- InputStream** - Describe stream input
- FilterInputStream** - Implements **InputStream**

OutputStream

- BufferedOutputStream**
Used for Buffered Output Stream
- ByteArrayOutputStream**
Used for writing into a byte array
- DataOutputStream**
Used for writing java standard data type
- ObjectOutputStream** - Output stream for objects
- FileOutputStream** - Used for writing into a File
- PipedOutputStream** - Output pipe
- OutputStream** - Describe stream output
- FilterOutputStream** - Implements **OutputStream**
- PrintStream** - Contains **print()** and **println()**

read() and **write()** both are key methods of byte stream

www.btechsmartclass.com

InputStream class

The InputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	int available() It returns the number of bytes that can be read from the input stream.
2	int read() It reads the next byte from the input stream.
3	int read(byte[] b) It reads a chunk of bytes from the input stream and store them in its byte array, b.
4	void close() It closes the input stream and also frees any resources connected with this input stream.

OutputStream class

The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

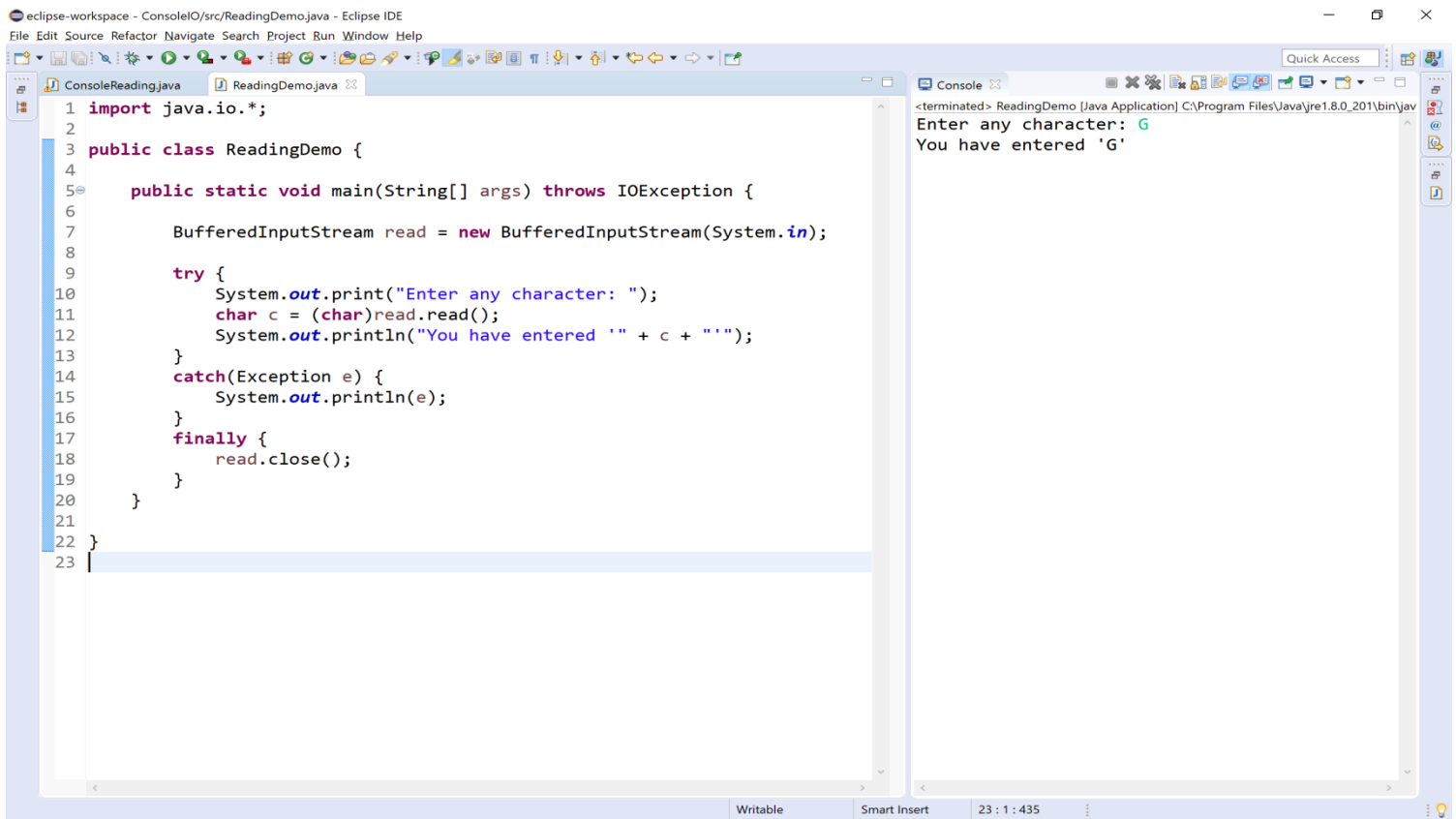
S.No.	Method with Description
1	void write(int n) It writes byte(contained in an int) to the output stream.
2	void write(byte[] b) It writes a whole byte array(b) to the output stream.
3	void flush() It flushes the output steam by forcing out buffered bytes to be written out.
4	void close() It closes the output stream and also frees any resources connected with this output stream.

Reading data using BufferedInputStream

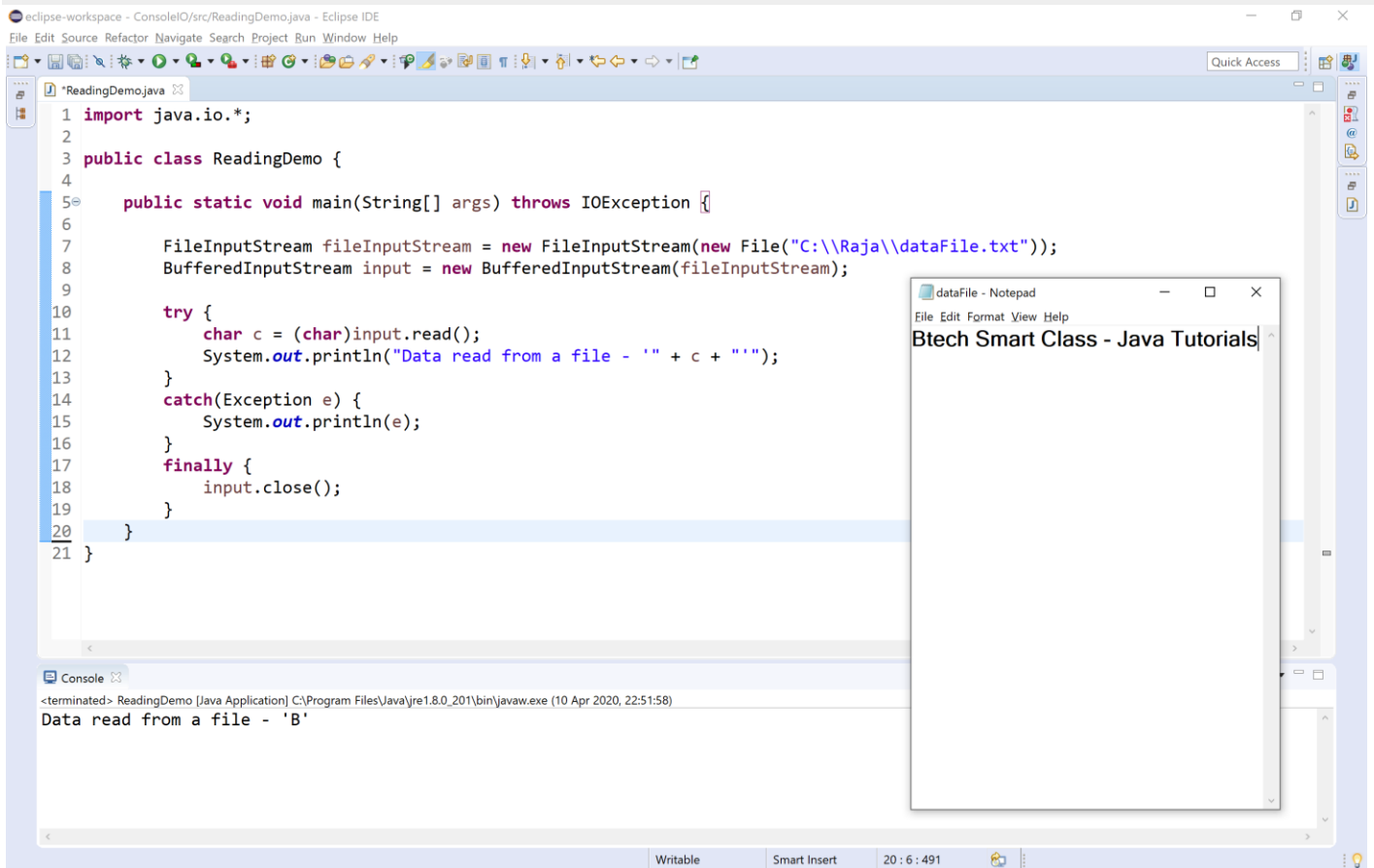
We can use the BufferedInputStream class to read data from the console. The BufferedInputStream class use a method read() to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using BufferedInputStream.

Example 1 - Reading from console



Example 2 - Reading from a file

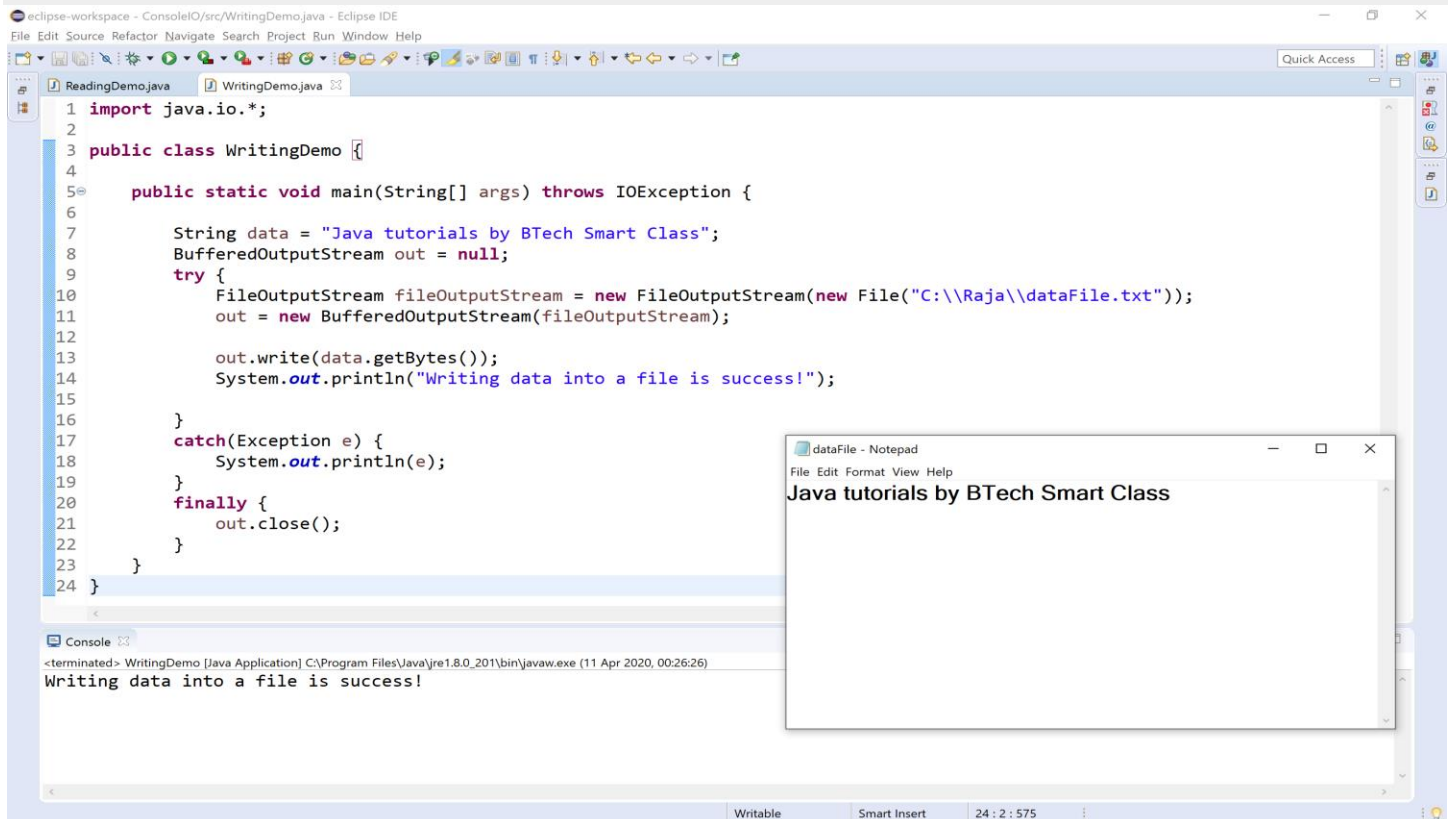


Writing data using BufferedOutputStream

We can use the BufferedOutputStream class to write data into the console, file, socket. The BufferedOutputStream class use a method write() to write data.

Let's look at an example code to illustrate writing data into a file using BufferedOutputStream.

Example - Writing data into a file



The screenshot shows the Eclipse IDE with a Java file named `WritingDemo.java`. The code uses `BufferedOutputStream` to write data to a file. A Notepad window titled `dataFile - Notepad` shows the text written to the file. The console output also confirms the successful write.

```
1 import java.io.*;
2
3 public class WritingDemo {
4
5     public static void main(String[] args) throws IOException {
6
7         String data = "Java tutorials by BTech Smart Class";
8         BufferedOutputStream out = null;
9         try {
10             FileOutputStream fileOutputStream = new FileOutputStream(new File("C:\\Raja\\dataFile.txt"));
11             out = new BufferedOutputStream(fileOutputStream);
12
13             out.write(data.getBytes());
14             System.out.println("Writing data into a file is success!");
15
16         }
17         catch(Exception e) {
18             System.out.println(e);
19         }
20         finally {
21             out.close();
22         }
23     }
24 }
```

dataFile - Notepad
File Edit Format View Help
Java tutorials by BTech Smart Class

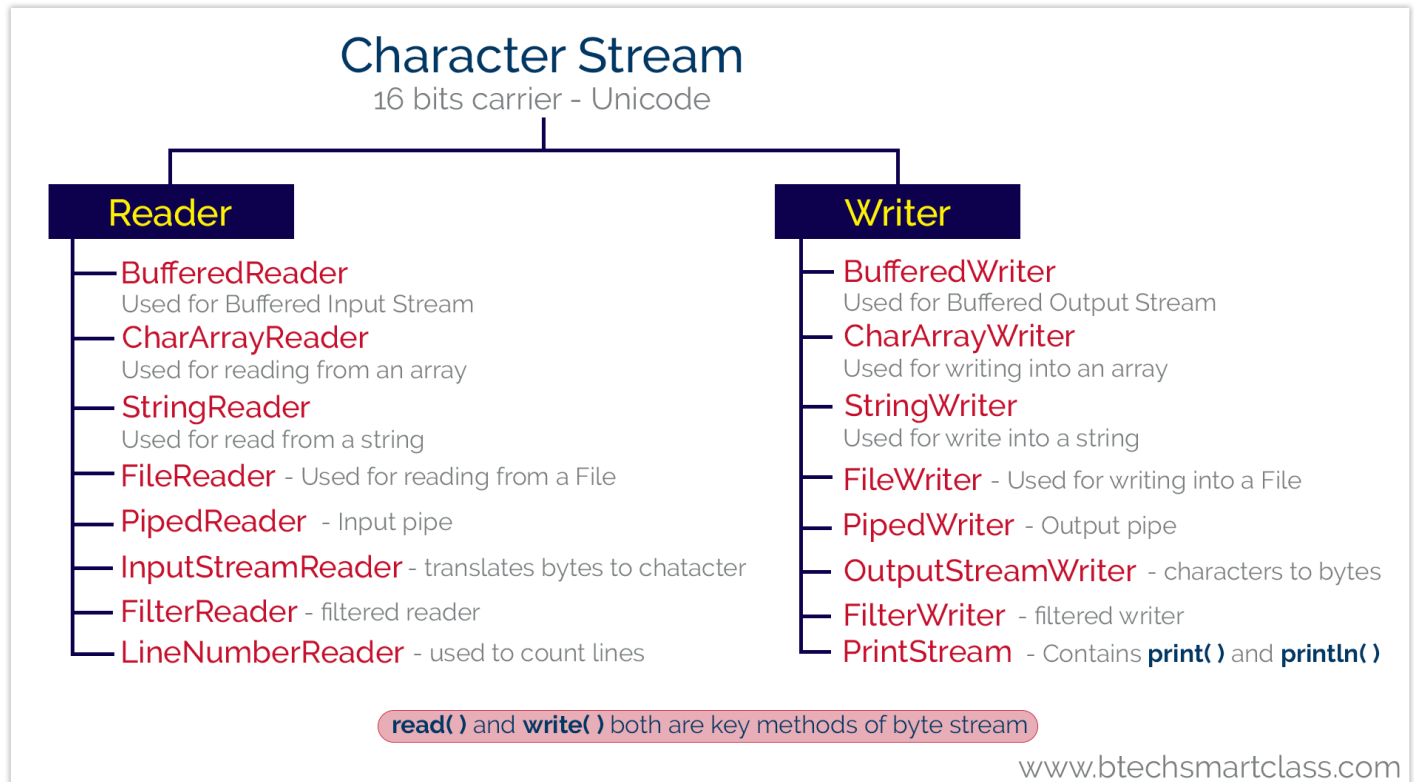
Console
<terminated> WritingDemo [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (11 Apr 2020, 00:26:26)
Writing data into a file is success!

Character Stream in java

- In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream.
- The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.
- In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data.
- The character stream was introduced in Java 1.1 version. The character stream

- The java character stream is defined by two abstract classes, **Reader** and **Writer**. The Reader class used for character stream based input operations, and the Writer class used for character stream based output operations.
- The Reader and Writer classes have several concrete classes to perform various IO operations based on the character stream.

The following picture shows the classes used for character stream operations.



Reader class

The Reader class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	int read() It reads the next character from the input stream.
2	int read(char[] cbuffer) It reads a chunk of characters from the input stream and store them in its byte array, cbuffer.

S.No.	Method with Description
3	int read(char[] cbuf, int off, int len) It reads characters into a portion of an array.
4	int read(CharBuffer target) It reads characters into the specified character buffer.
5	String readLine() It reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a line feed.
6	boolean ready() It tells whether the stream is ready to be read.
7	void close() It closes the input stream and also frees any resources connected with this input stream.

Writer class

The Writer class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	void flush() It flushes the output stream by forcing out buffered bytes to be written out.
2	void write(char[] cbuf) It writes a whole array(cbuf) to the output stream.
3	void write(char[] cbuf, int off, int len) It writes a portion of an array of characters.

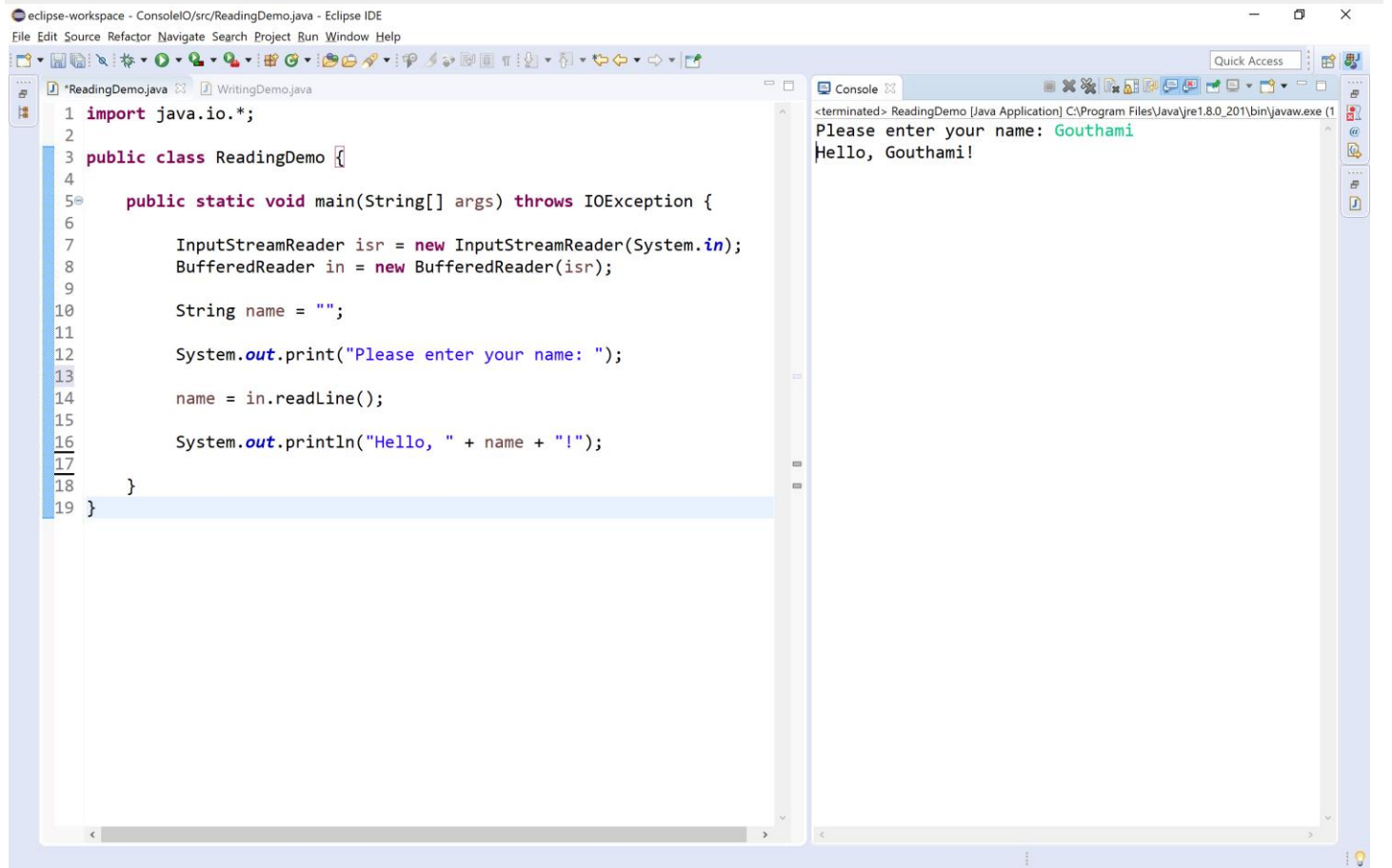
S.No.	Method with Description
4	void write(int c) It writes single character.
5	void write(String str) It writes a string.
6	void write(String str, int off, int len) It writes a portion of a string.
7	Writer append(char c) It appends the specified character to the writer.
8	Writer append(CharSequence csq) It appends the specified character sequence to the writer
9	Writer append(CharSequence csq, int start, int end) It appends a subsequence of the specified character sequence to the writer.
10	void close() It closes the output stream and also frees any resources connected with this output stream.

Reading data using **BufferedReader**

We can use the `BufferedReader` class to read data from the console. The `BufferedReader` class needs `InputStreamReader` class. The `BufferedReader` use a method `read()` to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using `BufferedReader`.

Example 1 - Reading from console

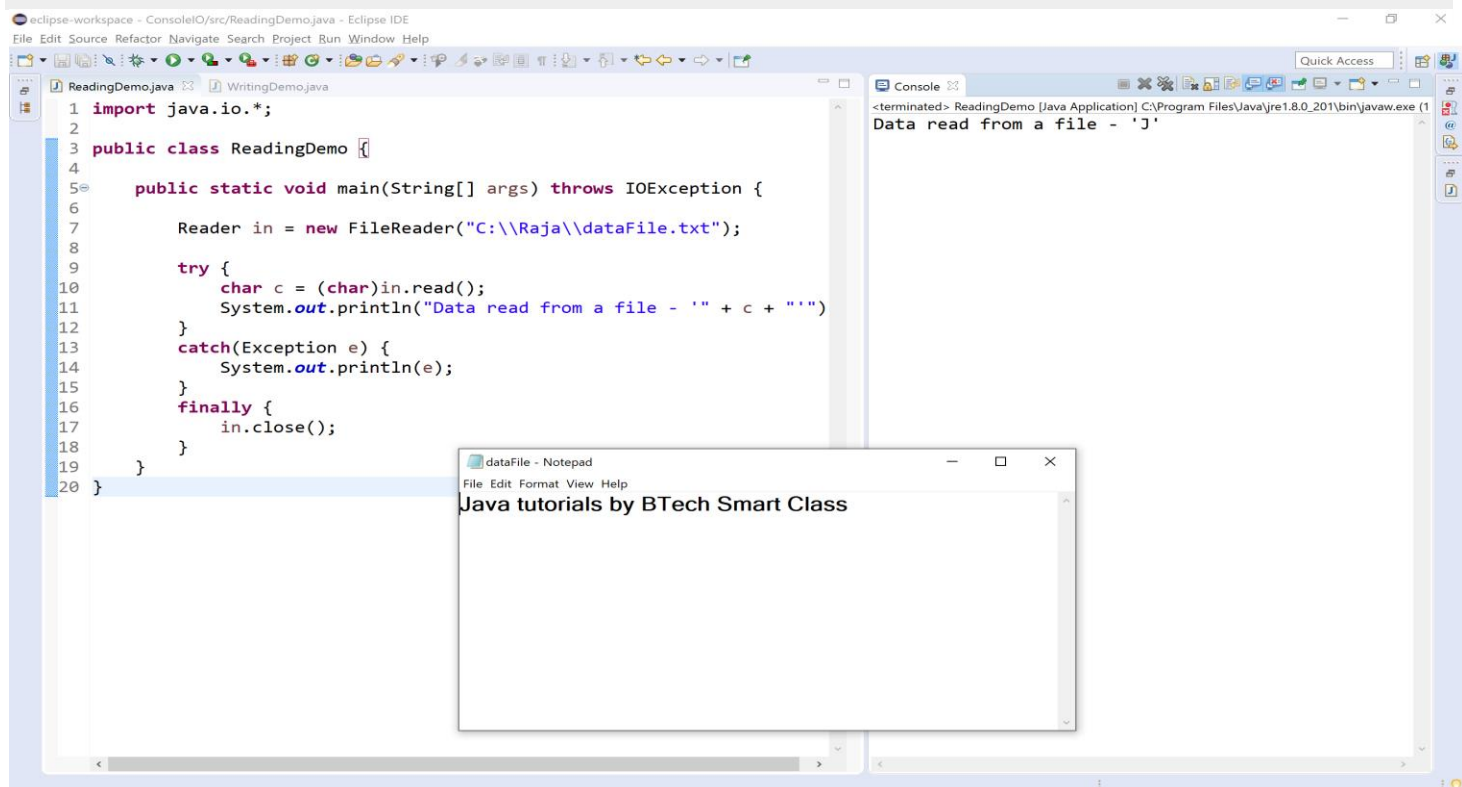


The screenshot shows the Eclipse IDE with a Java project named 'ConsoleIO/src'. The main editor displays the file 'ReadingDemo.java' with the following code:

```
1 import java.io.*;
2
3 public class ReadingDemo {
4
5     public static void main(String[] args) throws IOException {
6
7         InputStreamReader isr = new InputStreamReader(System.in);
8         BufferedReader in = new BufferedReader(isr);
9
10        String name = "";
11
12        System.out.print("Please enter your name: ");
13
14        name = in.readLine();
15
16        System.out.println("Hello, " + name + "!");
17    }
18 }
19 }
```

The right-hand side of the IDE shows the 'Console' window. It displays the output of the program: '<terminated> ReadingDemo [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (1)' followed by the prompt 'Please enter your name: Gouthami' and the response 'Hello, Gouthami!'.

Example 2 - Reading from a file



The screenshot shows the Eclipse IDE with the same project. The main editor displays the file 'ReadingDemo.java' with the following code:

```
1 import java.io.*;
2
3 public class ReadingDemo {
4
5     public static void main(String[] args) throws IOException {
6
7         Reader in = new FileReader("C:\\Raja\\dataFile.txt");
8
9         try {
10            char c = (char)in.read();
11            System.out.println("Data read from a file - " + c + "");
12        }
13        catch(Exception e) {
14            System.out.println(e);
15        }
16        finally {
17            in.close();
18        }
19    }
20 }
```

The right-hand side of the IDE shows the 'Console' window. It displays the output of the program: '<terminated> ReadingDemo [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (1)' followed by the message 'Data read from a file - 'J''.

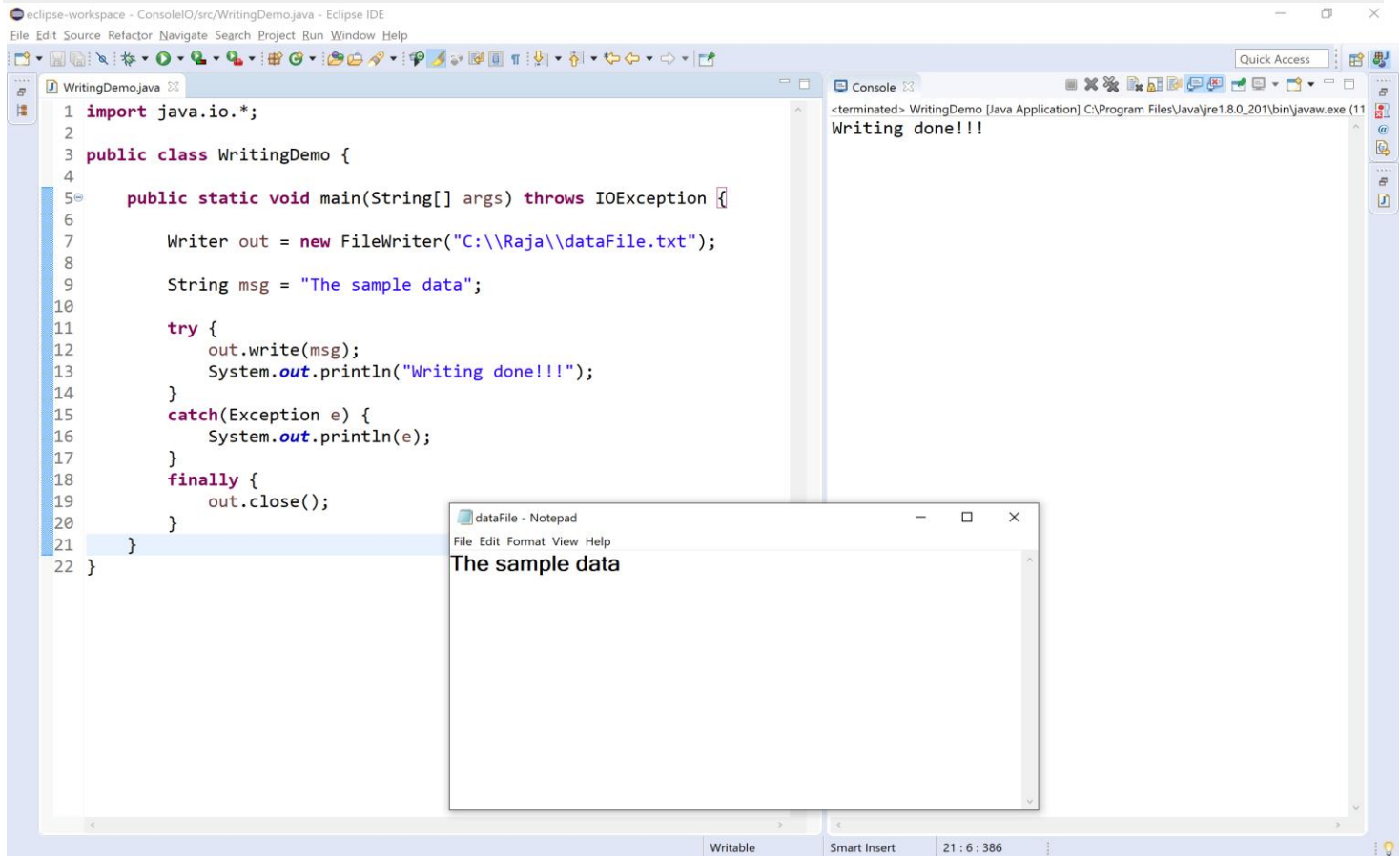
In the foreground, there is a Notepad window titled 'dataFile - Notepad'. It shows the content of the file 'dataFile.txt', which is 'Java tutorials by BTech Smart Class'.

Writing data using FileWriter

We can use the FileWriter class to write data into the file. The FileWriter class use a method write() to write data.

Let's look at an example code to illustrate writing data into a file using FileWriter.

Example - Writing data into a file



Console IO Operations in Java

Reading console input in java

In java, there are three ways to read console input. Using the 3 following ways, we can read input data from the console.

- Using `BufferedReader` class
- Using `Scanner` class
- Using `Console` class

Let's explore the each method to read data with example.

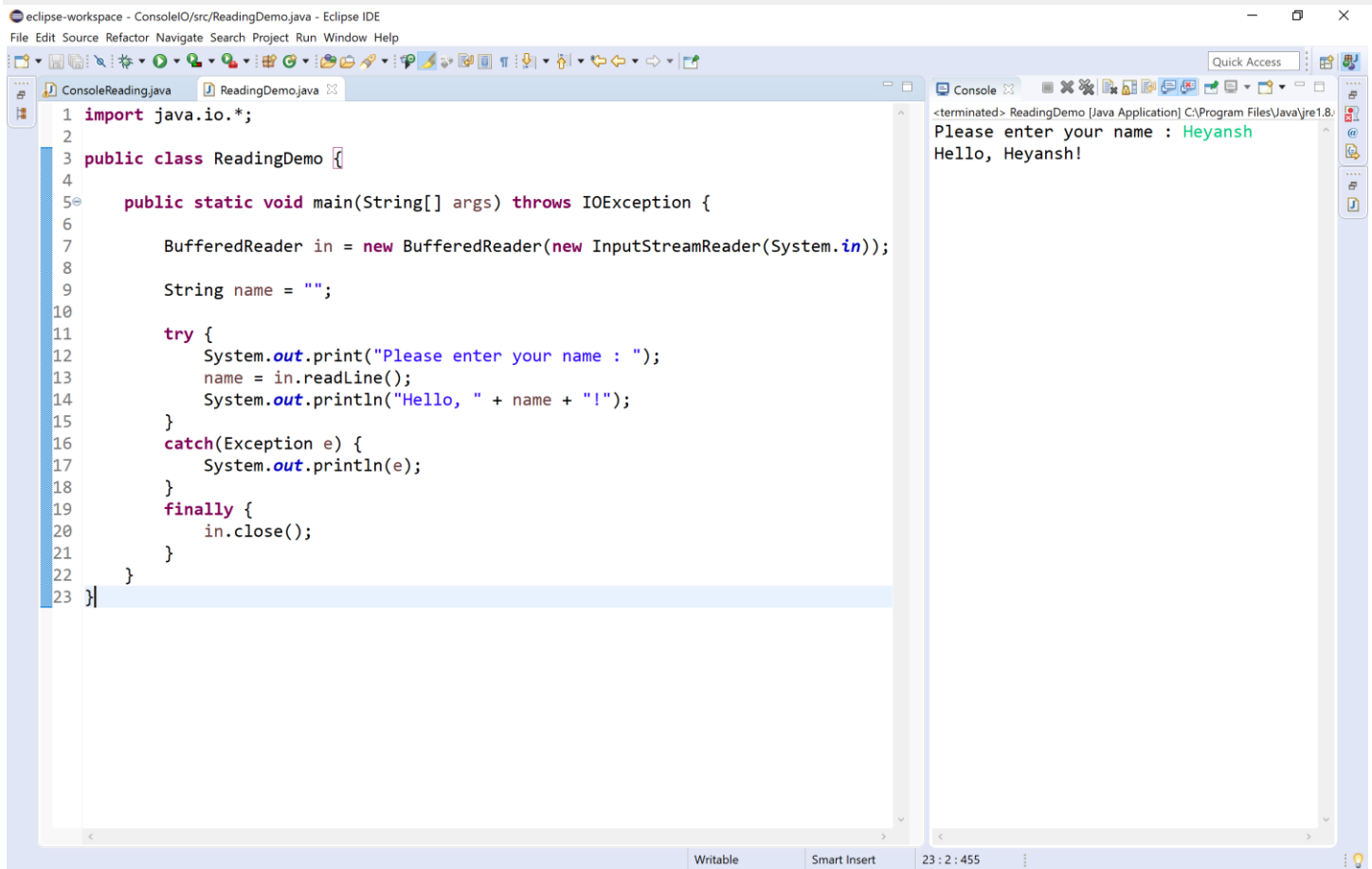
1. Reading console input using BufferedReader class in java

Reading input data using the **BufferedReader** class is the traditional technique. This way of the reading method is used by wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a **BufferedReader**, we can read input from the console.

The **BufferedReader** class has defined in the **java.io** package.

Consider the following example code to understand how to read console input using BufferedReader class.

Example



The screenshot shows the Eclipse IDE with a Java project named 'ConsoleReading.java'. The main editor displays the source code for 'ReadingDemo.java'. The code imports 'java.io.*', defines a 'public class ReadingDemo', and implements a 'main' method that uses 'BufferedReader' to read input from 'System.in'. The console window on the right shows the program's execution: it prompts 'Please enter your name :', the user enters 'Heyansh', and the program outputs 'Hello, Heyansh!'.

```
1 import java.io.*;
2
3 public class ReadingDemo {
4
5     public static void main(String[] args) throws IOException {
6
7         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
8
9         String name = "";
10
11         try {
12             System.out.print("Please enter your name : ");
13             name = in.readLine();
14             System.out.println("Hello, " + name + "!");
15         }
16         catch(Exception e) {
17             System.out.println(e);
18         }
19         finally {
20             in.close();
21         }
22     }
23 }
```

Console Output:

```
<terminated> ReadingDemo [Java Application] C:\Program Files\Java\jre1.8.
Please enter your name : Heyansh
Hello, Heyansh!
```

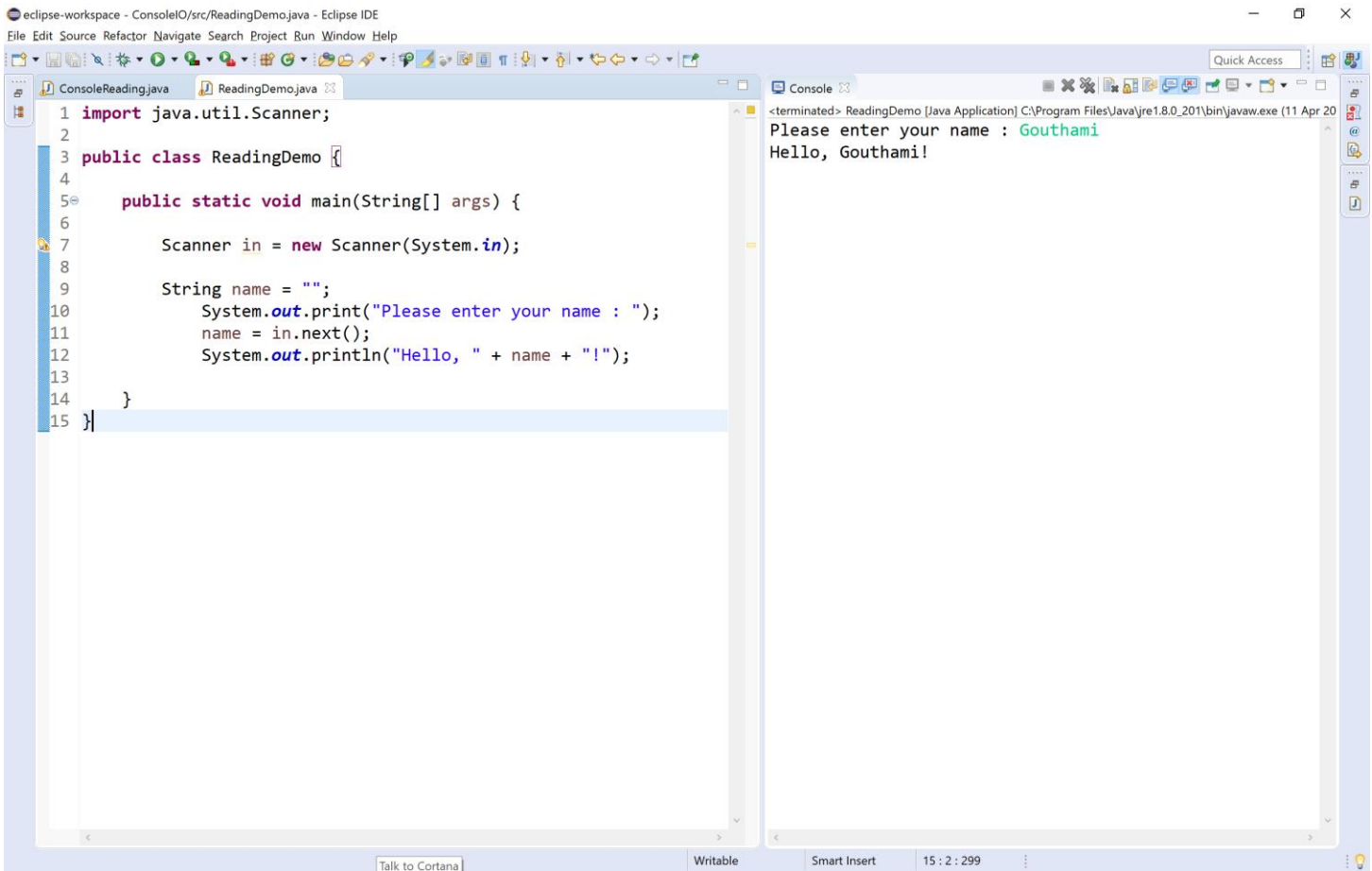
2. Reading console input using Scanner class in java

Reading input data using the **Scanner** class is the most commonly used method. This way of the reading method is used by wrapping the **System.in** (standard input stream) which is wrapped in a **Scanner**, we can read input from the console.

The **Scanner** class has defined in the **java.util** package.

Consider the following example code to understand how to read console input using Scanner class.

Example



3. Reading console input using Console class in java

Reading input data using the **Console** class is the most commonly used method. This class was introduced in Java 1.6 version.

The **Console** class has defined in the **java.io** package.

Consider the following example code to understand how to read console input using Console class.

Example

```
import java.io.*;

public class ReadingDemo {

    public static void main(String[] args) {

        String name;

        Console con = System.console();

        if(con != null) {

            name = con.readLine("Please enter your name : ");

            System.out.println("Hello, " + name + "!!!");

        }

        else {
```

```
        System.out.println("Console not available.");
    }
}
}
```

Writing console output in java

In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

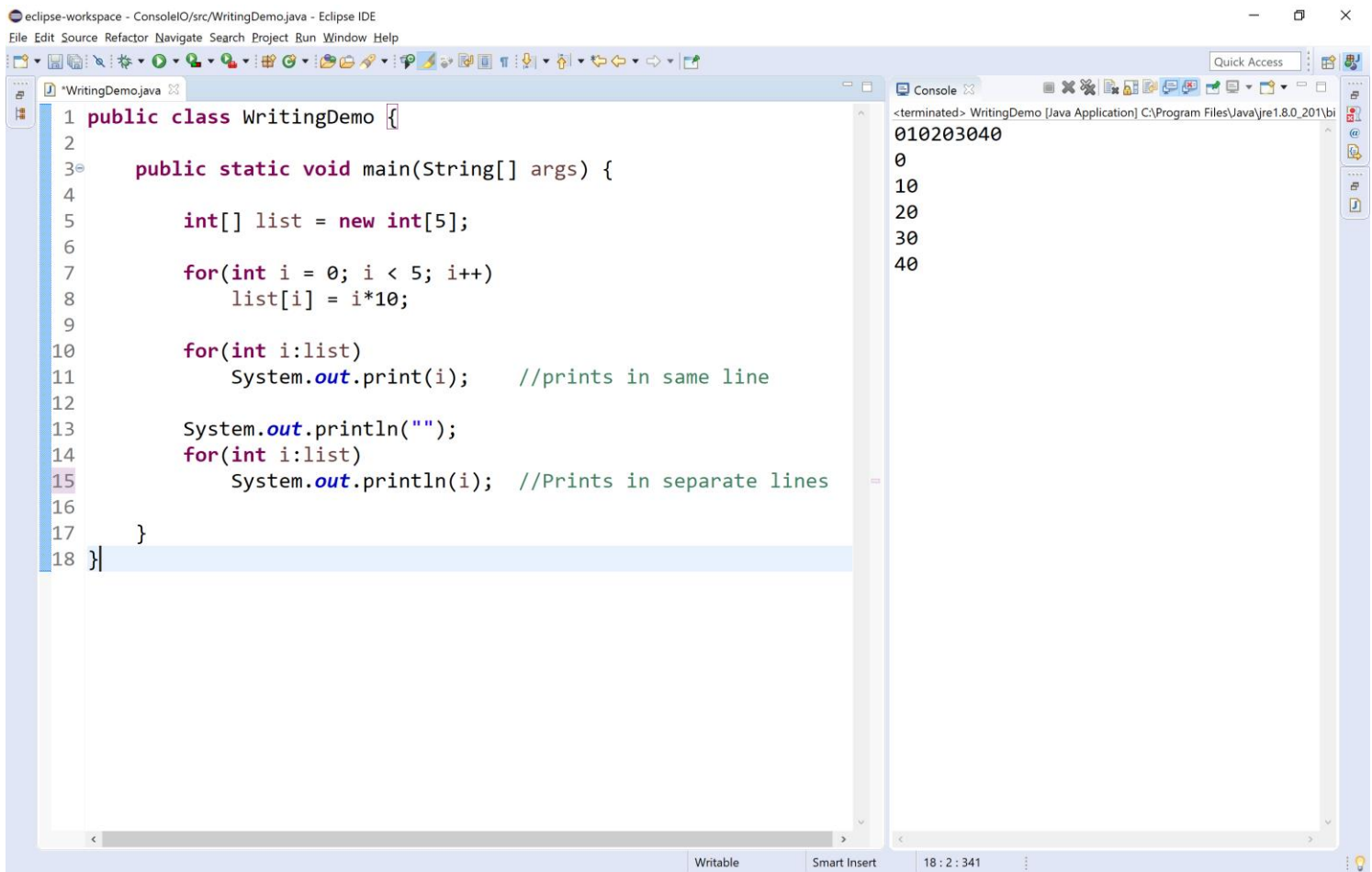
- Using print() and println() methods
- Using write() method

Let's explore the each method to write data with example.

- **Writing console output using print() and println() methods**
- The PrintStream is a built-in class that provides two methods print() and println() to write console output. The print() and println() methods are the most widely used methods for console output.
- Both print() and println() methods are used with System.out stream.
- The print() method writes console output in the same line. This method can be used with console output only.
- The println() method writes console output in a separate line (new line). This method can be used with console and also with other output sources.

Let's look at the following code to illustrate print() and println() methods.

Example



The screenshot shows the Eclipse IDE interface. The main editor window displays a Java file named `WritingDemo.java` with the following code:

```
1 public class WritingDemo {
2
3     public static void main(String[] args) {
4
5         int[] list = new int[5];
6
7         for(int i = 0; i < 5; i++)
8             list[i] = i*10;
9
10        for(int i:list)
11            System.out.print(i);    //prints in same line
12
13        System.out.println("");
14        for(int i:list)
15            System.out.println(i); //Prints in separate lines
16
17    }
18 }
```

The right-hand side of the IDE shows the Console window. It displays the output of the program, which is the concatenation of the values in the array `list` (0, 10, 20, 30, 40) followed by a newline character, resulting in the text `010203040` on the first line and `0` on the second line.

2. Writing console output using write() method

Alternatively, the `PrintStream` class provides a method `write()` to write console output.

The `write()` method takes an integer as an argument, and writes its ASCII equivalent character on to the console; it also accepts escape sequences.

Let's look at the following code to illustrate the `write()` method.

Example

The screenshot shows the Eclipse IDE with a Java file named `WritingDemo.java` and a console window. The Java code defines a `WritingDemo` class with a `main` method. The `main` method creates an integer array `list` of size 26, initializes it with values from 65 to 90 (i.e., `i + 65`), and then iterates over the array using a `for` loop, printing each element to the console. The console output shows the letters A through Z, which correspond to the values 65 through 90.

```
1 public class WritingDemo {
2
3
4     public static void main(String[] args) {
5
6         int[] list = new int[26];
7
8         for(int i = 0; i < 26; i++) {
9             list[i] = i + 65;
10        }
11
12        for(int i:list) {
13            System.out.write(i);
14            System.out.write('\n');
15        }
16    }
17 }
```

Console Output:

```
<terminated> WritingDemo [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
```

File class in Java

- The **File** is a built-in class in Java. In java, the File class has been defined in the **java.io** package.
- The File class represents a reference to a file or directory.
- The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory.
- The File class in java has the following constructors.

S.No.	Constructor with Description
1	File(String pathname) It creates a new File instance by converting the givenpathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.
2	File(String parent, String child) It Creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.

S.No. Constructor with Description	
3	File(File parent, String child) It creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
4	File(URI uri) It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class in java has the following methods.

S.No. Methods with Description	
1	String getName() It returns the name of the file or directory that referenced by the current File object.
2	String getParent() It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.
3	String getPath() It returns the path of current File.
4	File getParentFile() It returns the path of the current file's parent; or null if it does not exist.
5	String getAbsolutePath() It returns the current file or directory path from the root.
6	boolean isAbsolute() It returns true if the current file is absolute, false otherwise.
7	boolean isDirectory() It returns true, if the current file is a directory; otherwise returns false.
8	boolean isFile() It returns true, if the current file is a file; otherwise returns false.

S.No.	Methods with Description
9	boolean exists() It returns true if the current file or directory exist; otherwise returns false.
10	boolean canRead() It returns true if and only if the file specified exists and can be read by the application; false otherwise.
11	boolean canWrite() It returns true if and only if the file specified exists and the application is allowed to write to the file; false otherwise.
12	long length() It returns the length of the current file.
13	long lastModified() It returns the time that specifies the file was last modified.
14	boolean createNewFile() It returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	boolean delete() It deletes the file or directory. And returns true if and only if the file or directory is successfully deleted; false otherwise.
16	void deleteOnExit() It sends a requests that the file or directory needs be deleted when the virtual machine terminates.
17	boolean mkdir() It returns true if and only if the directory was created; false otherwise.
18	boolean mkdirs() It returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
19	boolean renameTo(File dest) It renames the current file. And returns true if and only if the renaming succeeded; false

S.No.	Methods with Description
	otherwise.
20	boolean setLastModified(long time) It sets the last-modified time of the file or directory. And returns true if and only if the operation succeeded; false otherwise.
21	boolean setReadOnly() It sets the file permission to only read operations; Returns true if and only if the operation succeeded; false otherwise.
22	String[] list() It returns an array of strings containing names of all the files and directories in the current directory.
23	String[] list(FilenameFilter filter) It returns an array of strings containing names of all the files and directories in the current directory that satisfy the specified filter.
24	File[] listFiles() It returns an array of file references containing names of all the files and directories in the current directory.
25	File[] listFiles(FileFilter filter) It returns an array of file references containing names of all the files and directories in the current directory that satisfy the specified filter.
26	boolean equals(Object obj) It returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.
27	int compareTo(File pathname) It Compares two abstract pathnames lexicographically. It returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
28	int compareTo(File pathname) Compares this abstract pathname to another object. Returns zero if the argument is

S.No. Methods with Description

equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.

Let's look at the following code to illustrate file operations.

Example

```
import java.io.*;

public class FileClassTest {

    public static void main(String args[]) {

        File f = new File("C:\\Raja\\datFile.txt");

        System.out.println("Executable File : " + f.canExecute());
        System.out.println("Name of the file : " + f.getName());
        System.out.println("Path of the file : " + f.getAbsolutePath());
        System.out.println("Parent name : " + f.getParent());
        System.out.println("Write mode : " + f.canWrite());
        System.out.println("Read mode : " + f.canRead());
        System.out.println("Existance : " + f.exists());
        System.out.println("Last Modified : " + f.lastModified());
        System.out.println("Length : " + f.length());
        //f.createNewFile()
        //f.delete();
        //f.setReadOnly()

    }

}
```

Let's look at the following java code to list all the files in a directory including the files present in all its subdirectories.

Example

```
import java.util.Scanner;
import java.io.*;
public class ListingFiles {
    public static void main(String[] args) {
        String path = null;
        Scanner read = new Scanner(System.in);
        System.out.print("Enter the root directory name: ");
        path = read.next() + "\\";
        File f_ref = new File(path);
        if (!f_ref.exists()) {
            printLine();
            System.out.println("Root directory does not exists!");
            printLine();
        } else {
            String ch = "y";
            while (ch.equalsIgnoreCase("y")) {
                printFiles(path);
                System.out.print("Do you want to open any sub-directory
                                (Y/N): ");
                ch = read.next().toLowerCase();
                if (ch.equalsIgnoreCase("y")) {
                    System.out.print("Enter the sub-directory name: ");
                    path = path + "\\\\" + read.next();
                    File f_ref_2 = new File(path);
                    if (!f_ref_2.exists()) {
                        printLine();
                        System.out.println("The sub-directory does not
                                exists!");
                        printLine();
                    }
                }
            }
        }
    }
}
```

```

        int lastIndex = path.lastIndexOf("\\");
        path = path.substring(0, lastIndex);
    }
}

}

}

System.out.println("***** Program Closed *****");
}

```

```

public static void printFiles(String path) {
    System.out.println("Current Location: " + path);
    File f_ref = new File(path);
    File[] filesList = f_ref.listFiles();
    for (File file : filesList) {
        if (file.isFile())
            System.out.println("- " + file.getName());
        else
            System.out.println("> " + file.getName());
    }
}

```

```

public static void printLine() {
    System.out.println("-----");
}

}

```

File Reading & Writing in Java

In java, there multiple ways to read data from a file and to write data to a file. The most commonly used ways are as follows.

- **Using Byte Stream (FileInputStream and FileOutputStream)**
- **Using Character Stream (FileReader and FileWriter)**

Let's look each of these ways.

File Handling using Byte Stream

In java, we can use a byte stream to handle files. The byte stream has the following built-in classes to perform various operations on a file.

- **FileInputStream** - It is a built-in class in java that allows reading data from a file. This class has implemented based on the byte stream. The FileInputStream class provides a method **read()** to read data from a file byte by byte.
- **FileOutputStream** - It is a built-in class in java that allows writing data to a file. This class has implemented based on the byte stream. The FileOutputStream class provides a method **write()** to write data to a file byte by byte.

Let's look at the following example program that reads data from a file and writes the same to another file using FileInoutStream and FileOutputStream classes.

Example

```
import java.io.*;

public class FileReadingTest {

    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("C:\\Raja\\Input-File.txt");
            out = new FileOutputStream("C:\\Raja\\Output-File.txt");

            int c;
            while ((c = in.read()) != -1) {
```



```

        out.write(c);
    }
    System.out.println("Reading and Writing has been success!!!");
}
catch(Exception e){
    System.out.println(e);
}finally {
    if (in != null) {
        in.close();
    }
    if (out != null) {
        out.close();
    }
}
}
}

```

File Handling using Character Stream

In java, we can use a character stream to handle files. The character stream has the following built-in classes to perform various operations on a file.

- **FileReader** - It is a built-in class in java that allows reading data from a file. This class has implemented based on the character stream. The FileReader class provides a method **read()** to read data from a file character by character.
- **FileWriter** - It is a built-in class in java that allows writing data to a file. This class has implemented based on the character stream. The FileWriter class provides a method **write()** to write data to a file character by character.

Let's look at the following example program that reads data from a file and writes the same to another file using FileReader and FileWriter classes.

Example

```
import java.io.*;

public class FileIO {

    public static void main(String args[]) throws IOException {

        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("C:\\Raja\\Input-File.txt");
            out = new FileWriter("C:\\Raja\\Output-File.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
            System.out.println("Reading and Writing in a file is done!!!");
        }
        catch(Exception e) {
            System.out.println(e);
        }
        finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

RandomAccessFile in Java

In java, the **java.io** package has a built-in class **RandomAccessFile** that enables a file to be accessed randomly. The RandomAccessFile class has several methods used to move the cursor position in a file.

A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The RandomAccessFile class in java has the following constructors.

S.No.	Constructor with Description
1	RandomAccessFile(File fileName, String mode) It creates a random access file stream to read from, and optionally to write to, the file specified argument.
2	RandomAccessFile(String fileName, String mode) It creates a random access file stream to read from, and optionally to write to, a file with the fileName.

Access Modes

Using the RandomAccessFile, a file may be created in the following modes.

- **r** - Creates the file with read mode; Calling write methods will result in an IOException.
- **rw** - Creates the file with read and write mode.
- **rwd** - Creates the file with read and write mode - synchronously. All updates to file content are written to the disk synchronously.
- **rws** - Creates the file with read and write mode - synchronously. All updates to file content or meta data are written to the disk synchronously.

RandomAccessFile methods

The RandomAccessFile class in java has the following methods.

S.No.	Methods with Description
1	int read() It reads byte of data from a file. The byte is returned as an integer in the range 0-255.

S.No.	Methods with Description
2	int read(byte[] b) It reads byte of data from file upto b.length, -1 if end of file is reached.
3	int read(byte[] b, int offset, int len) It reads bytes initialising from offset position upto b.length from the buffer.
4	boolean readBoolean() It reads a boolean value from from the file.
5	byte readByte() It reads signed eight-bit value from file.
6	char readChar() It reads a character value from file.
7	double readDouble() It reads a double value from file.
8	float readFloat() It reads a float value from file.
9	long readLong() It reads a long value from file.
10	int readInt() It reads a integer value from file.
11	void readFully(byte[] b) It reads bytes initialising from offset position upto b.length from the buffer.

S.No.	Methods with Description
12	void readFully(byte[] b, int offset, int len) It reads bytes initialising from offset position upto b.length from the buffer.
13	String readUTF() It reads in a string from the file.
14	void seek(long pos) It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs.
15	long length() It returns the length of the file.
16	void write(int b) It writes the specified byte to the file from the current cursor position.
17	void writeFloat(float v) It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
18	void writeDouble(double v) It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

Let's look at the following example program.

Example

```
import java.io.*;

public class RandomAccessFileDemo
{
    public static void main(String[] args)
    {
        try
        {
            double d = 1.5;
            float f = 14.56f;

            // Creating a new RandomAccessFile - "F2"
            RandomAccessFile f_ref = new RandomAccessFile("C:\\Raja\\Input-File.txt", "rw");

            // Writing to file
            f_ref.writeUTF("Hello, Good Morning!");

            // File Pointer at index position - 0
            f_ref.seek(0);

            // read() method :
            System.out.println("Use of read() method : " + f_ref.read());

            f_ref.seek(0);

            byte[] b = { 1, 2, 3 };

            // Use of .read(byte[] b) method :
            System.out.println("Use of .read(byte[] b) : " + f_ref.read(b));
```

```
// readBoolean() method :
System.out.println("Use of readBoolean() : " + f_ref.readBoolean());

// readByte() method :
System.out.println("Use of readByte() : " + f_ref.readByte());

f_ref.writeChar('c');
f_ref.seek(0);

// readChar() :
System.out.println("Use of readChar() : " + f_ref.readChar());

f_ref.seek(0);
f_ref.writeDouble(d);
f_ref.seek(0);

// read double
System.out.println("Use of readDouble() : " + f_ref.readDouble());

f_ref.seek(0);
f_ref.writeFloat(f);
f_ref.seek(0);

// readFloat() :
System.out.println("Use of readFloat() : " + f_ref.readFloat());

f_ref.seek(0);
// Create array upto geek.length
byte[] arr = new byte[(int) f_ref.length()];
// readFully() :
f_ref.readFully(arr);
```

```

String str1 = new String(arr);
System.out.println("Use of readFully() : " + str1);

f_ref.seek(0);

// readFully(byte[] b, int off, int len) :
f_ref.readFully(arr, 0, 8);

String str2 = new String(arr);
System.out.println("Use of readFully(byte[] b, int off, int len) : " + str2);
}
catch (IOException ex)
{
    System.out.println("Something went Wrong");
    ex.printStackTrace();
}
}
}

```

Console class in Java

- In java, the **java.io** package has a built-in class **Console** used to read from and write to the console, if one exists.
- This class was added to the Java SE 6. The Console class implements the **Flushable** interface.
- In java, most the input functionalities of Console class available through **System.in**, and the output functionalities available through **System.out**.

Console class Constructors

The Console class does not have any constructor. We can obtain the Console class object by calling **System.console()**.

Console class methods

The Console class in java has the following methods.

S.No.	Methods with Description
1	void flush() It causes buffered output to be written physically to the console.
2	String readLine() It reads a string value from the keyboard, the input is terminated on pressing enter key.
3	String readLine(String promptingString, Object...args) It displays the given promptingString, and reads a string from the keyboard; input is terminated on pressing Enter key.
4	char[] readPassword() It reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
5	char[] readPassword(String promptingString, Object...args) It displays the given promptingString, and reads a string value from the keyboard, the string is not displayed; the input is terminated on pressing enter key.
6	Console printf(String str, Object....args) It writes the given string to the console.
7	Console format(String str, Object....args) It writes the given string to the console.
8	Reader reader() It returns a reference to a Reader connected to the console.
9	PrintWriter writer()

S.No. Methods with Description

It returns a reference to a Writer connected to the console.

Let's look at the following example program for reading a string using Console class.

Example

```
import java.io.*;

public class ReadingDemo {

    public static void main(String[] args) {

        String name;

        Console con = System.console();

        if(con != null) {

            name = con.readLine("Please enter your name : ");

            System.out.println("Hello, " + name + "!!");

        }

        else {

            System.out.println("Console not available.");

        }

    }

}
```

Let's look at the following example program for writing to the console using Console class.

Example

```
import java.io.*;

public class WritingDemo {

    public static void main(String[] args) {

        int[] list = new int[26];

        for(int i = 0; i < 26; i++) {

            list[i] = i + 65;

        }

    }

}
```

```
    }  
    for(int i:list) {  
        System.out.write(i);  
        System.out.write("\n");  
    }  
}  
}
```

Serialization and Deserialization in Java

- In java, the **Serialization** is the process of converting an object into a byte stream so that it can be stored on to a file, or memory, or a database for future access.
- The deserialization is reverse of serialization. The deserialization is the process of reconstructing the object from the serialized state.
- Using serialization and deserialization, we can transfer the Object Code from one Java Virtual machine to another.

Serialization in Java

- In a java programming language, the Serialization is achieved with the help of interface **Serializable**.
- The class whose object needs to be serialized must implement the Serializable interface.
- We use the **ObjectOutputStream** class to write a serialized object to write to a destination.
- The ObjectOutputStream class provides a method **writeObject()** to serializing an object.
- We use the following steps to serialize an object.
- **Step 1** - Define the class whose object needs to be serialized; it must implement Serializable interface.
- **Step 2** - Create a file reference with file path using FileOutputStream class.

- **Step 3** - Create reference to ObjectOutputStream object with file reference.
- **Step 4** - Use writeObject(object) method by passing the object that wants to be serialized.
- **Step 5** - Close the FileOutputStream and ObjectOutputStream.

Let's look at the following example program for serializing an object.

Example

```
import java.io.*;

public class SerializationExample {

    public static void main(String[] args) {
        Student stud = new Student();
        stud.studName = "Rama";
        stud.studBranch = "IT";
        try {
            FileOutputStream fos = new FileOutputStream("my_data.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(stud);
            oos.close();
            fos.close();
            System.out.println("The object has been saved to my_data file!");
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Deserialization in Java

In a java programming language, the Deserialization is achieved with the help of class **ObjectInputStream**. This class provides a method **readObject()** to deserializing an object.

We use the following steps to serialize an object.

- **Step 1** - Create a file reference with file path in which serialized object is available using `FileInputStream` class.
- **Step 2** - Create reference to `ObjectInputStream` object with file reference.
- **Step 3** - Use `readObject()` method to access serialized object, and typecast it to destination type.
- **Step 4** - Close the `FileInputStream` and `ObjectInputStream`.

Let's look at the following example program for deserializing an object.

Example

```
import java.io.*;

public class DeserializationExample {
    public static void main(String[] args) throws Exception {
        try {
            FileInputStream fis = new FileInputStream("my_data.txt");
            ObjectInputStream ois = new ObjectInputStream(fis);

            Student stud2 = (Student) ois.readObject();
            System.out.println("The object has been deserialized.");

            fis.close();
            ois.close();

            System.out.println("Name = " + stud2.studName);
            System.out.println("Department = " + stud2.studBranch);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
}
```

Enum in Java

- In java, an **Enumeration** is a list of named constants.
- The enum concept was introduced in Java SE 5 version.
- The enum in Java programming the concept of enumeration is greatly expanded with lot more new features compared to the other languages like C, and C++.
- In java, the enumeration concept was defined based on the class concept.
- When we create an enum in java, it converts into a class type. This concept enables the java enum to have constructors, methods, and instance variables.
- All the constants of an enum are **public**, **static**, and **final**. As they are static, we can access directly using enum name.
- The main objective of enum is to define our own data types in Java, and they are said to be enumeration data types.

Creating enum in Java

To create enum in Java, we use the keyword **enum**. The syntax for creating enum is similar to that of class.

In java, an enum can be defined outside a class, inside a class, but not inside a method.

Let's look at the following example program for creating a basic enum.

Example

```
enum WeekDay{
    MONDAY, TUESDAY, WEDNESSDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY;
}

public class EnumerationExample {
    public static void main(String[] args) {
        WeekDay day = WeekDay.FRIDAY;
        System.out.println("Today is " + day);
    }
}
```

```

        System.out.println("\nAll WeekDays: ");
        for(WeekDay d:WeekDay.values())
            System.out.println(d);
    }
}

```

- ❑ Every enum is converted to a class that extends the built-in class **Enum**.
- ❑ Every constant of an enum is defined as an object.
- ❑ As an enum represents a class, it can have methods, constructors. It also gets a few extra methods from the Enum class, and one of them is the **values()** method.

Constructors in Java enum

In a java programming language, an enum can have both the type of constructors default and parameterized. The execution of constructor depends on the constants we defined in the enum. For every constant in an enum, the constructor is executed.

If an enum has a parameterized constructor, the parameter value should be passed when we define constant.

Let's look at the following example program for illustrating constructors in enum.

Example

```

enum WeekDay{
    MONDAY, TUESDAY, WEDNESSDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY("Holiday");

    String msg;
    WeekDay(){
        System.out.println("Default constructor!");
    }
    WeekDay(String str){
        System.out.println("Parameterized constructor!");
        msg = str;
    }
}

```

```

}
public class EnumerationExample {
    public static void main(String[] args) {
        WeekDay day = WeekDay.SUNDAY;
        System.out.println("\nToday is " + day + " and its " + day.msg);
    }
}

```

In the above example, the constant **SUNDAY** is created by calling the parameterized constructor, other constants created by calling default constructor.

Methods in Java enum

In a java programming language, an enum can have methods. These methods are similar to the methods of a class. All the methods defined in an enum can be used with all the constants defined by the same enum.

Let's look at the following example program for illustrating methods in enum.

Example

```

enum WeekDay{
    MONDAY, TUESDAY, WEDNESSDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY("Holiday");

    String msg;
    WeekDay(){
        System.out.println("Default constructor!");
    }
    WeekDay(String str){
        System.out.println("Parameterized constructor!");
        msg = str;
    }
    void printMessage() {
        System.out.println("Today is also " + msg);
    }
}

```



```

public class EnumerationExample {
    public static void main(String[] args) {
        WeekDay day = WeekDay.SUNDAY;
        System.out.println("\nToday is " + day);
        day.printMessage();
    }
}

```

When we run the above program, it produce the following output.

- ☐ In java, enum can not extend another enum and a class.
- ☐ In java, enum can implement interfaces.
- ☐ In java, enum does not allow to create an object of it.
- ☐ In java, every enum extends a built-in class **Enum** by default.

Autoboxing and Unboxing in Java

- In java, all the primitive data types have defined using the class concept, these classes known as **wrapper classes**. In java, every primitive type has its corresponding wrapper class.
- All the wrapper classes in Java were defined in the **java.lang** package.

The following table shows the primitive type and its corresponding wrapper class.

S.No.	Primitive Type	Wrapper class
1	byte	Byte
2	short	Short
3	int	Interger
4	long	Long

S.No.	Primitive Type	Wrapper class
5	float	Float
6	double	Double
7	char	Character
8	boolean	Boolean

The Java 1.5 version introduced a concept that converts primitive type to corresponding wrapper type and reverses of it.

Autoboxing in Java

In java, the process of converting a primitive type value into its corresponding wrapper class object is called autoboxing or simply boxing.

For example, converting an int value to an Integer class object.

The compiler automatically performs the autoboxing when a primitive type value has assigned to an object of the corresponding wrapper class.

□ We can also perform autoboxing manually using the method `valueOf()`, which is provided by every wrapper class.

Let's look at the following example program for autoboxing.

Example - Autoboxing

```
import java.lang.*;
public class AutoBoxingExample {
    public static void main(String[] args) {
        // Auto boxing : primitive to Wrapper
        int num = 100;
        Integer i = num;
        Integer j = Integer.valueOf(num);
        System.out.println("num = " + num + ", i = " + i + ", j = " + j);
    }
}
```

```
}  
  
}
```

Auto un-boxing in Java

In java, the process of converting an object of a wrapper class type to a primitive type value is called auto un-boxing or simply unboxing. For example, converting an Integer object to an int value.

The compiler automatically performs the auto un-boxing when a wrapper class object has assigned to a primitive type.

□ We can also perform auto un-boxing manually using the method `intValue()`, which is provided by Integer wrapper class. Similarly every wrapper class has a method for auto un-boxing.

Let's look at the following example program for autoboxing.

Example - Auto unboxing

```
import java.lang.*;  
  
public class AutoUnboxingExample {  
    public static void main(String[] args) {  
        // Auto un-boxing : Wrapper to primitive  
        Integer num = 200;  
        int i = num;  
        int j = num.intValue();  
        System.out.println("num = " + num + ", i = " + i + ", j = " + j);  
    }  
}
```

Generics in Java

- The java generics is a language feature that allows creating methods and class which can handle any type of data values.
- The generic programming is a way to write generalized programs, java supports it by java generics.
- The java generics is similar to the templates in the C++ programming language.

- Most of the collection framework classes are generic classes.
- The java generics allows only non-primitive type, it does not support primitive types like int, float, char, etc.

The java generics feature was introduced in Java 1.5 version.

In java, generics used angular brackets “< >”. In java, the generics feature implemented using the following.

- **Generic Method**
- **Generic Classe**

Generic methods in Java

- The java generics allows creating generic methods which can work with a different type of data values.
- Using a generic method, we can create a single method that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Let's look at the following example program for generic method.

Example - Generic method

```
public class GenericFunctions {  
    public <T, U> void displayData(T value1, U value2) {  
        System.out.println("(" + value1.getClass().getName() + ", " +  
            value2.getClass().getName() + ")");  
    }  
}
```

```

public static void main(String[] args) {
    GenericFunctions obj = new GenericFunctions();
    obj.displayData(45.6f, 10);
    obj.displayData(10, 10);
    obj.displayData("Hi", 'c');
}
}

```

In the above example code, the method `displayData()` is a generic method that allows a different type of parameter values for every function call.

Generic Class in Java

- In java, a class can be defined as a generic class that allows creating a class that can work with different types.
- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

Let's look at the following example program for generic class.

Example - Generic class

```

public class GenericsExample<T> {
    T obj;
    public GenericsExample(T anotherObj) {
        this.obj = anotherObj;
    }
    public T getData() {
        return this.obj;
    }

    public static void main(String[] args) {

        GenericsExample<Integer> actualObj1 = new
            GenericsExample<Integer>(100);
        System.out.println(actualObj1.getData());
    }
}

```

```
GenericsExample<String> actualObj2 = new  
    GenericsExample<String>("Java");  
System.out.println(actualObj2.getData());  
  
GenericsExample<Float> actualObj3 = new GenericsExample<Float>(25.9f);  
System.out.println(actualObj3.getData());  
  
}  
  
}
```

Database Access

Database Programming using JDBC:

JDBC is an international standard for programming access to SQL databases. It was developed by *JavaSoft*, a subsidiary of *Sun Microsystems*.

Relational Database Management System supports SQL. As we know that Java is platform independent, JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

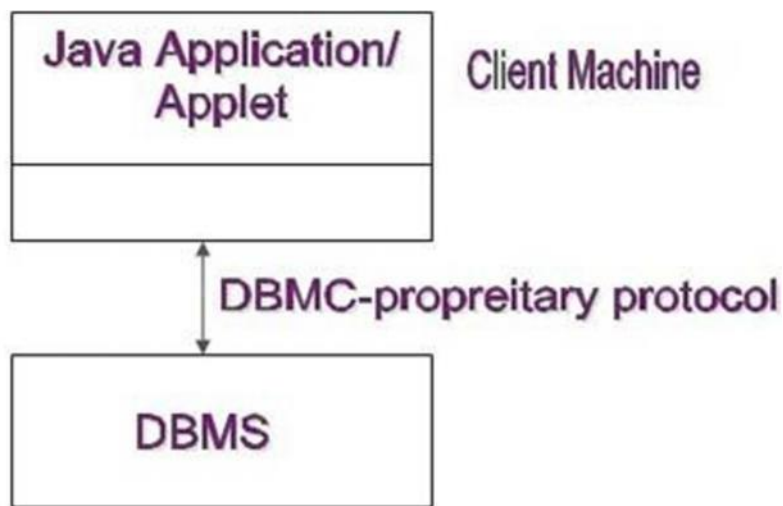
Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not dependent upon any language.

JDBC Driver Model

JDBC supports two-tier and three-tier model:

I. Two-tier Model

In this model the Java applets and application are directly connected with any type of database. The client directly communicates with database server through JDBC driver.

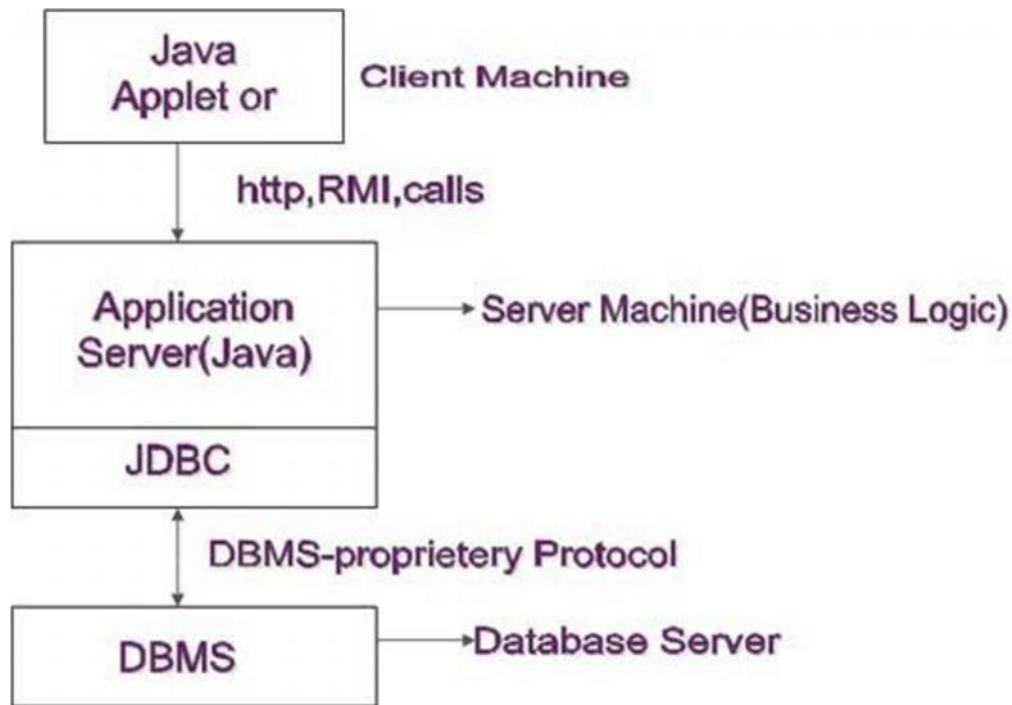


The JDBC ManagerDriver API sends it to the various SQL statements.

As another layer, the Manager should communicate with various third party drivers that actually connect to the database and return information from the query or perform action specified by the query.

II. Three-tier Model

In this model, client connects with database server through a middle-tier server, which is used for various purposes. Middle-tier server performs various functions.



It extracts the SQL command from the client and sends these commands to the database server. Also, it extracts the result from the database server and submits the same to the client.

Exploring JDBC Architecture

JDBC is open specification given by Sun Microsystems having rules and guidelines to develop JDBC driver. JDBC driver is a bridge software between Java application and database software. It converts Java call to database call and vice versa. Java application talks with database server using JDBC driver.

- JDBC call converts database call.
- JDBC driver connects to database server and sends command to the database **server**.
- Database executes the statement.
- Sends the output back to JDBC driver.
- JDBC driver sends back to java application.

Each JDBC driver is specific to one database software.

We can get the JDBC driver from three parties:

1. Sun Microsystems
2. Database vendor
3. Third party vendor

It is recommended to use database vendor- supplied JDBC drivers.

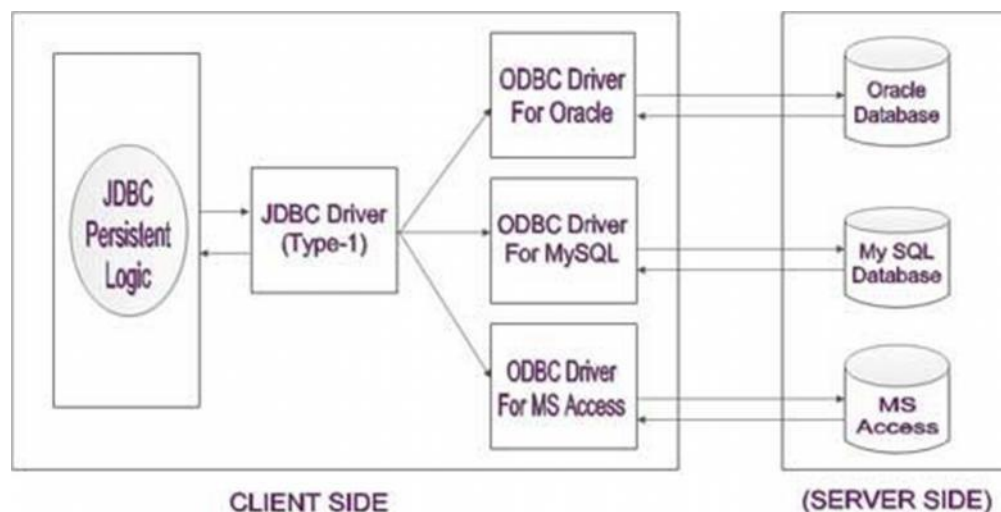
Non-Java applications use ODBC drivers to interact with database software.

There are four different methodologies or mechanisms to develop JDBC drivers based on the rules and guidelines of JDBC specification. These are:

- i. *Type-1(JDBC-ODBC Bridge Driver)*
- ii. *Type-2 Native API/ partly Java Driver)*
- iii. *Type-3(Net-Protocol/All Java Driver)*
- iv. *Type-4(Native- Protocol/All Java Driver)*

i. Type-1(JDBC-ODBC Bridge Driver)

The type-1 mechanism-based drivers are given drivers to take the support of ODBC drivers while introducing to database software. In this process, the type 1 driver takes support of the native code to communicate with ODBC drivers.



JDBC Type-1 driver is not specific to any database software because it does not interact with database software directly. It interacts with database software by using the database software-specific ODBC driver.

Type-1 driver is supplied only by Sun Microsystems. It has a built-in JDK software. JDK software supplies a basic and built-in service called DriverManager which serves to manage a set of JDBC drivers and to establish the connection with database software by using JDBC driver.

Type-1 driver class name is: *sun.jdbc.odbc.JdbcOdbcDriver*

Every JDBC driver must be registered with DriverManager service, as this creates JDBC class object in DriverManager service.

Advantages of JDBC-ODBC Bridge Driver

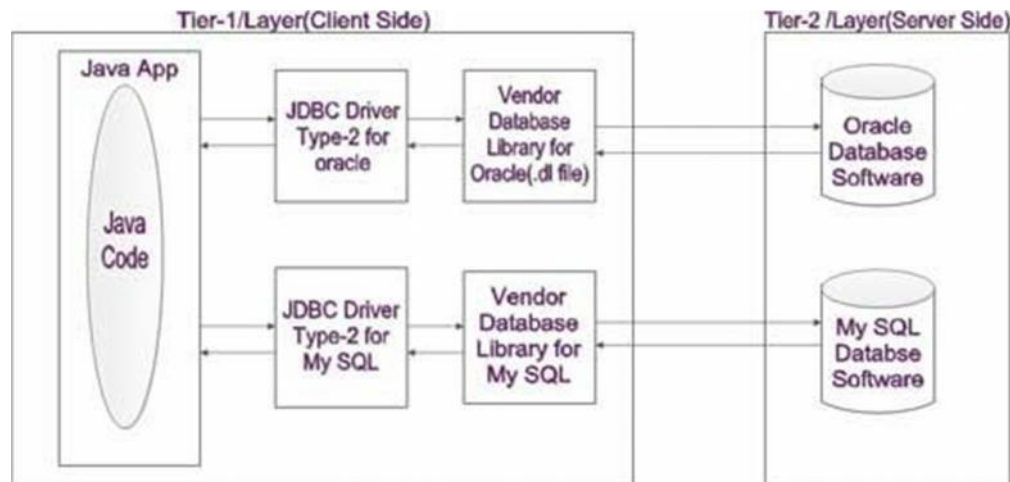
- Serves as a single driver that can be used to interact with different data stores.
- Allows you to communicate with all the databases supported by the ODBC driver.
- Represents a vendor-independent driver and is available with JDK

Disadvantages of JDBC-ODBC Bridge Driver

- Decreases the execution speed due to more number of transactions. (Include JDBC ODBC DB Native call)
- Depends on the ODBC driver due to which Java application indirectly becomes dependent on ODBC drivers.

ii. Type-2(JAVA to Native API)

Type-2 driver converts JDBC calls in a client machine. It uses native code to communicate with vendor database library.



Type-2 JDBC driver takes the support of vendor database software. In the figure, the java application is programmed using JDBC API, making JDBC calls. These JDBC calls are then converted into database specific native calls and the request is then dispatched to the database specific native libraries.

Type-2 drivers are suitable to use with server-side application. It is not recommended to use type-2 drivers with client-side application since native libraries for the client platform should be installed on the client machines.

Studying Javax.sql.* package:

Steps to connect Database:

1. Loading the Driver

To begin with, you first need load the driver or register it before using it in the program . Registration is to be done once in your program. You can register a driver in one of two ways mentioned below :

- **Class.forName() :** Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- **DriverManager.registerDriver():** DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time . The following example uses DriverManager.registerDriver()to register the Oracle driver

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

2. Create the connections

After loading the driver, establish connections using :

```
Connection con = DriverManager.getConnection(url,user,password)
```

user – username from which your sql command prompt can be accessed.

password – password from which your sql command prompt can be accessed.

con: is a reference to Connection interface.

url : Uniform Resource Locator. It can be created as follows:

```
String url = "jdbc:oracle:thin:@localhost:1521:xe"
```

Where oracle is the database used, thin is the driver used , @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function. Use of this can be referred from final code.

3. Create a statement

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.

Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step .

4. Execute the query

Now comes the most important part i.e executing the query. Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:

- Query for updating / inserting table in a database.
- Query for retrieving data .

The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.

The executeUpdate(sql query) method ofStatement interface is used to execute queries of updating/inserting .

Example:

```
int m = st.executeUpdate(sql);
if (m==1)
```

```
System.out.println("inserted successfully : "+sql);  
else  
System.out.println("insertion failed");
```

Here sql is sql query of the type String

5.Close the connections

So finally we have sent the data to the specified location and now we are at the verge of completion of our task .

By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Example :

```
import java.sql.*;  
  
import java.util.*;  
  
class Main  
{  
  
    public static void main(String a[])  
    {  
  
        //Creating the connection  
  
        String url = "jdbc:oracle:thin:@localhost:1521:xe";  
  
        String user = "system";  
  
        String pass = "12345";  
  
  
        //Entering the data  
  
        Scanner k = new Scanner(System.in);  
  
        System.out.println("enter name");  
  
        String name = k.next();  
  
        System.out.println("enter roll no");  
  
        int roll = k.nextInt();
```

```

System.out.println("enter class");

String cls = k.next();

//Inserting data using SQL query

String sql = "insert into student1 values('"+name+"','"+roll+"','"+cls+"')";

Connection con=null;

try
{
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    //Reference to connection interface

    con = DriverManager.getConnection(url,user,pass);

    Statement st = con.createStatement();

    int m = st.executeUpdate(sql);

    if (m == 1)

        System.out.println("inserted successfully : "+sql);

    else

        System.out.println("insertion failed");

    con.close();

}

catch(Exception ex)
{
    System.err.println(ex);

}

}
}

```

```

C:\Windows\System32\cmd.exe

E:\>javac Main.java

E:\>java Main
enter name
Abc
enter roll no
14
enter class
6a
inserted successfully : insert into student1 values('Abc',14,'6a')

E:\>

```

Java Programming (R20CSE2204)

UNIT - III

Exception handling - Fundamentals of exception handling, Exception types, Termination or resumptive models, Uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws and finally, built- in exceptions, creating own exception sub classes.

Multithreading- Differences between thread-based multitasking and process-based multitasking, Java thread model, creating threads, thread priorities, synchronizing threads, inter thread communication.

Exception Handling in Java

- An exception in java programming is an abnormal situation that is araised during the program execution. In simple words, an exception is a problem that arises at the time of program execution.
- When an exception occurs, it disrupts the program execution flow. When an exception occurs, the program execution gets terminated, and the system generates an error. We use the exception handling mechanism to avoid abnormal termination of program execution.
- Java programming language has a very powerful and efficient exception handling mechanism with a large number of built-in classes to handle most of the exceptions automatically.
- Java programming language has the following class hierarchy to support the exception handling mechanism.

Reasons for Exception Occurrence

Several reasons lead to the occurrence of an exception. A few of them are as follows.

- When we try to open a file that does not exist may lead to an exception.
- When the user enters invalid input data, it may lead to an exception.
- When a network connection has lost during the program execution may lead to an exception.
- When we try to access the memory beyond the allocated range may lead to an exception.
- The physical device problems may also lead to an exception.

Types of Exception

In java, exceptions have categorized into two types, and they are as follows.

- **Checked Exception** - An exception that is checked by the compiler at the time of compilation is called a checked exception.
- **Unchecked Exception** - An exception that can not be caught by the compiler but occurs at the time of program execution is called an unchecked exception.

How exceptions handled in Java?

In java, the exception handling mechanism uses five keywords namely `try`, `catch`, `finally`, `throw`, and `throws`.

Exception Types in Java

In java, exceptions are mainly categorized into two types, and they are as follows.

- **Checked Exceptions**
- **Unchecked Exceptions**

Checked Exceptions

The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.

The checked exceptions are generally caused by faults outside of the code itself like missing resources, networking errors, and problems with threads come to mind.

The following are a few built-in classes used to handle checked exceptions in java.

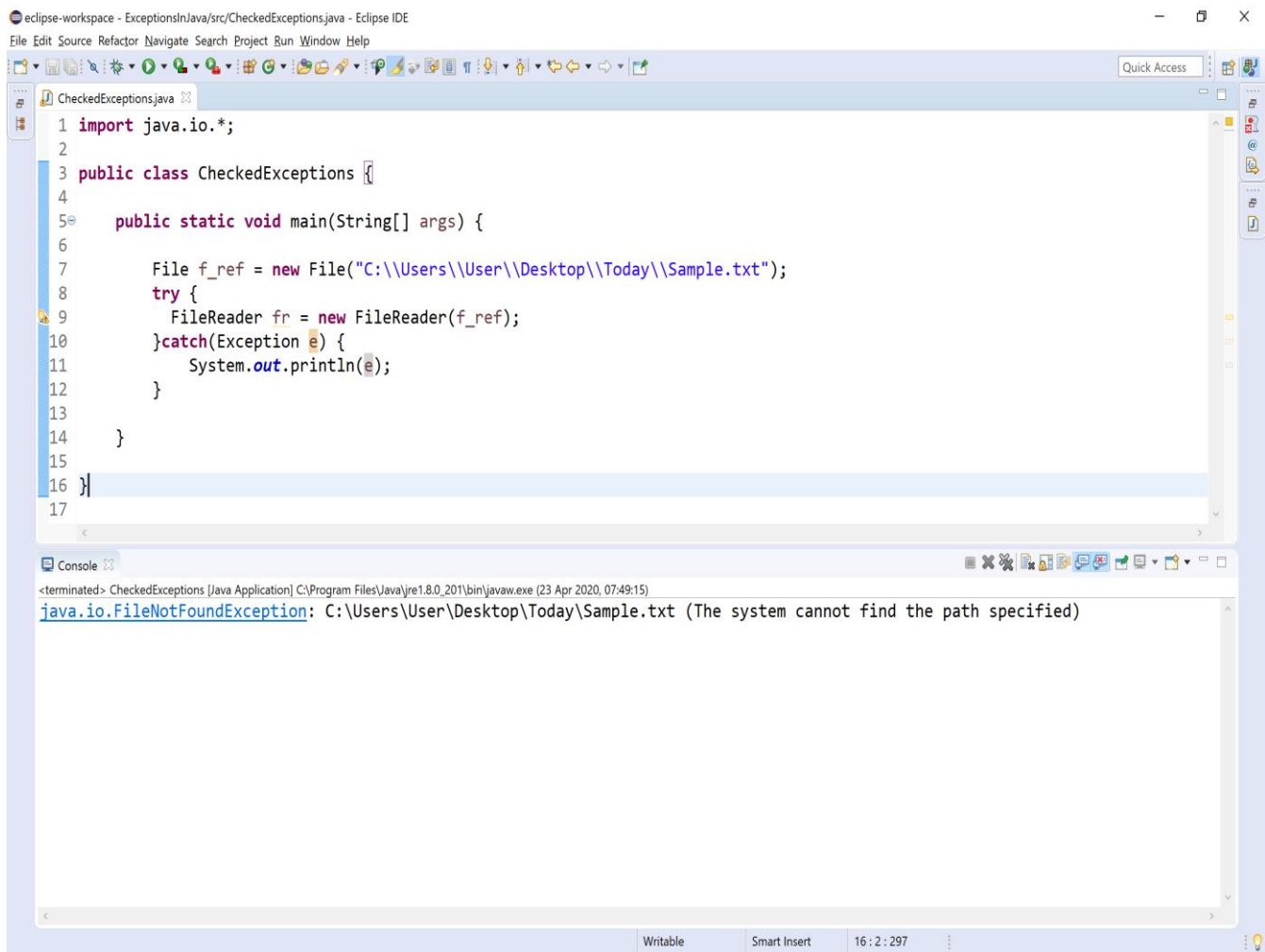
- IOException
- FileNotFoundException
- ClassNotFoundException
- SQLException
- DataAccessException
- InstantiationException
- UnknownHostException

□ In the exception class hierarchy, the checked exception classes are the direct children of the Exception class.

The checked exception is also known as a compile-time exception.

Let's look at the following example program for the checked exception method.

Example - Checked Exceptions



```
eclipse-workspace - ExceptionsInJava/src/CheckedExceptions.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

CheckedExceptions.java
1 import java.io.*;
2
3 public class CheckedExceptions {
4
5     public static void main(String[] args) {
6
7         File f_ref = new File("C:\\Users\\User\\Desktop\\Today\\Sample.txt");
8         try {
9             FileReader fr = new FileReader(f_ref);
10        } catch (Exception e) {
11            System.out.println(e);
12        }
13    }
14 }
15
16 }
17

Console
<terminated> CheckedExceptions [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (23 Apr 2020, 07:49:15)
java.io.FileNotFoundException: C:\Users\User\Desktop\Today\Sample.txt (The system cannot find the path specified)
```

Unchecked Exceptions

The unchecked exception is an exception that occurs at the time of program execution. The unchecked exceptions are not caught by the compiler at the time of compilation.

The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.

The following are a few built-in classes used to handle unchecked exceptions in java.

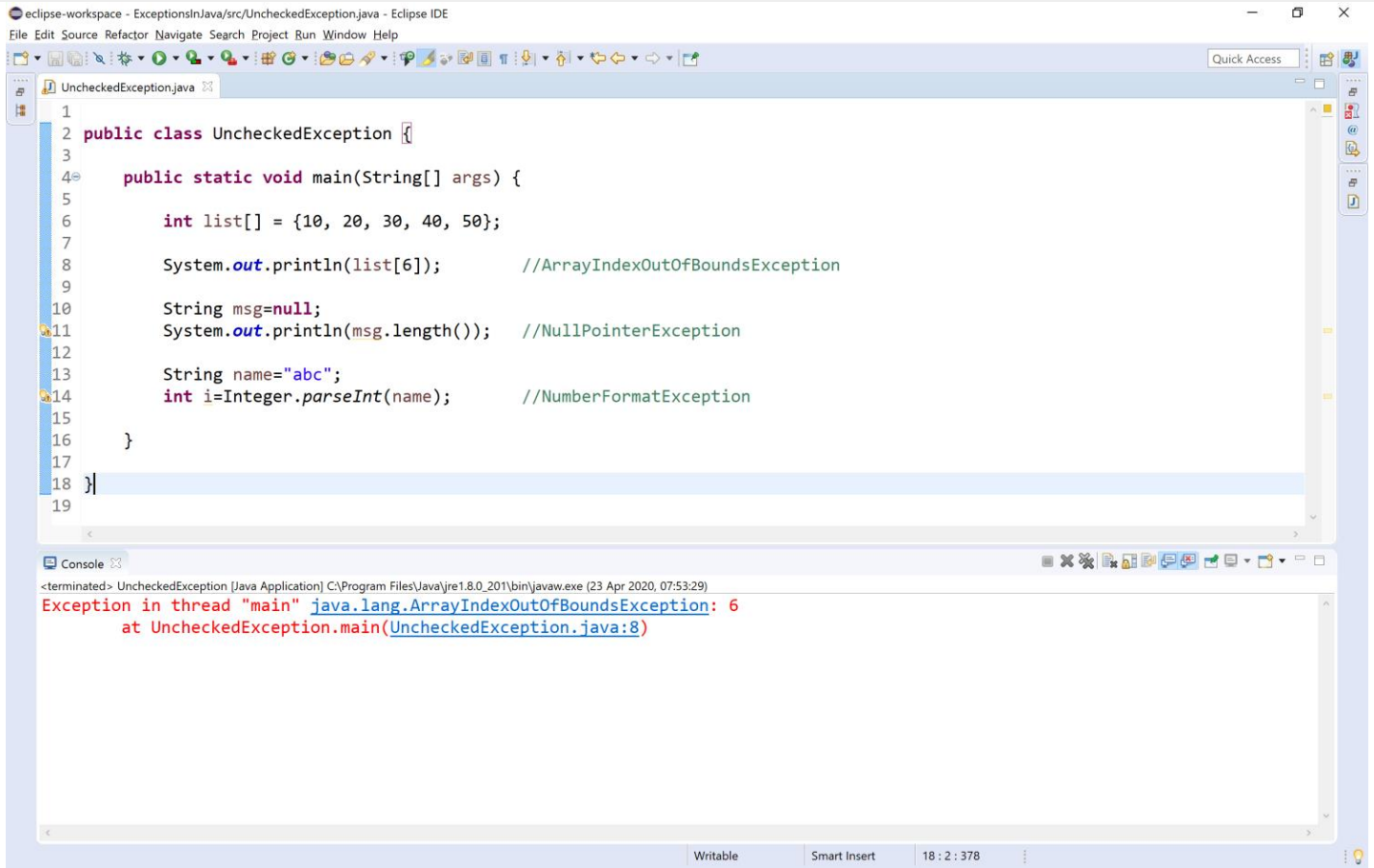
- ArithmeticException
- NullPointerException
- NumberFormatException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException

□ In the exception class hierarchy, the unchecked exception classes are the children of RuntimeException class, which is a child class of Exception class.

The unchecked exception is also known as a runtime exception.

Let's look at the following example program for the unchecked exceptions.

Example - Unchecked Exceptions



The screenshot shows the Eclipse IDE with a Java file named `UncheckedException.java`. The code defines a `public class UncheckedException` with a `main` method. Inside `main`, it declares an array `list` with values `{10, 20, 30, 40, 50}`. It then attempts to access `list[6]`, which throws a `java.lang.ArrayIndexOutOfBoundsException`. Next, it sets `String msg=null` and calls `msg.length()`, which throws a `NullPointerException`. Finally, it tries to parse the string `"abc"` as an integer using `Integer.parseInt(name)`, which throws a `NumberFormatException`. The console output shows the first exception: `Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6 at UncheckedException.main(UncheckedException.java:8)`.

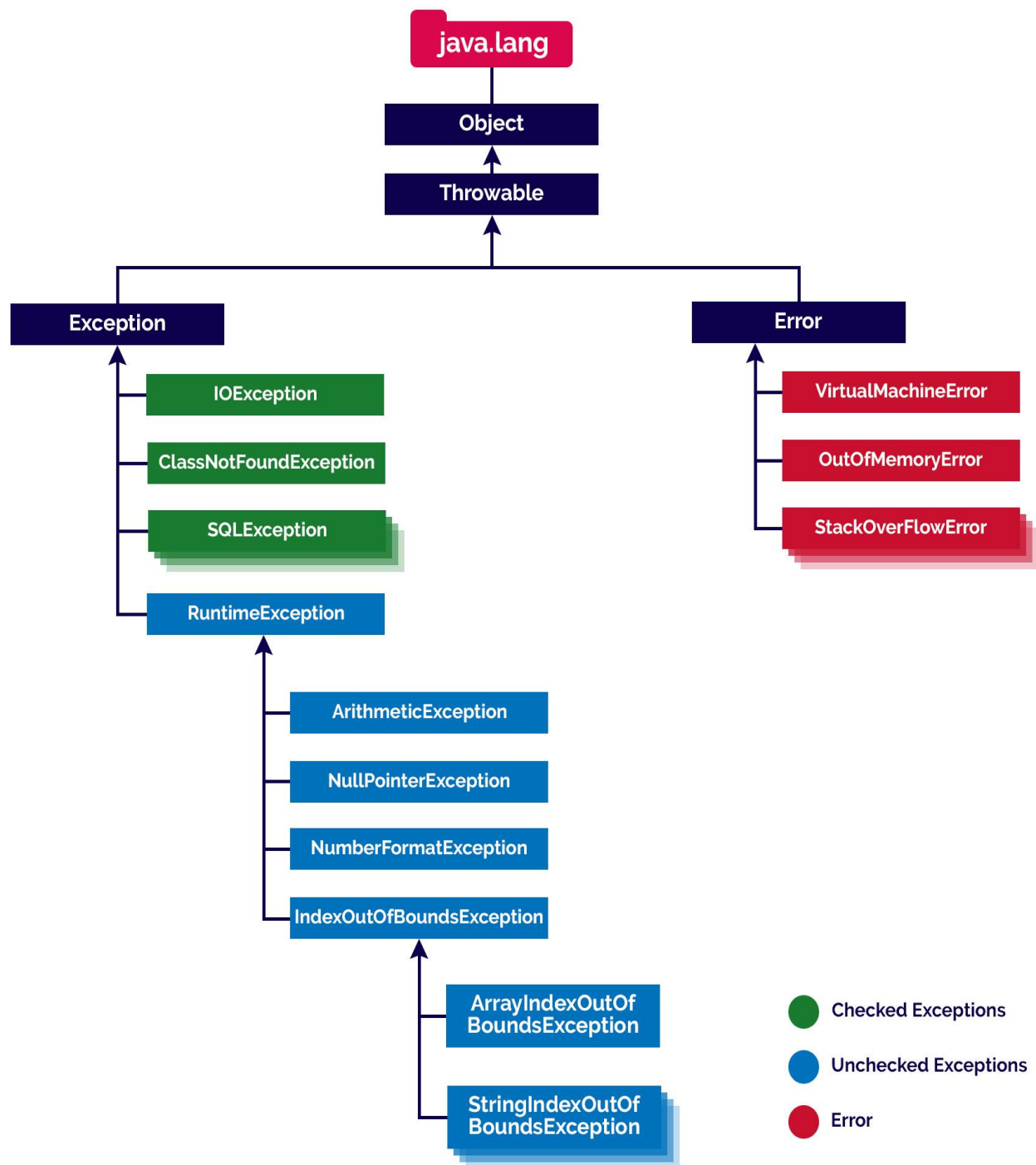
```
1 public class UncheckedException {
2
3
4     public static void main(String[] args) {
5
6         int list[] = {10, 20, 30, 40, 50};
7
8         System.out.println(list[6]);           //ArrayIndexOutOfBoundsException
9
10        String msg=null;
11        System.out.println(msg.length());      //NullPointerException
12
13        String name="abc";
14        int i=Integer.parseInt(name);         //NumberFormatException
15
16    }
17
18 }
19
```

Console Output:

```
<terminated> UncheckedException [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (23 Apr 2020, 07:53:29)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
    at UncheckedException.main(UncheckedException.java:8)
```

Exception class hierarchy

In java, the built-in classes used to handle exceptions have the following class hierarchy.



Exception Models in Java

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java supports are as follows.

- **Termination Model**
- **Resumptive Model**

Let's look into details of each exception model.

Termination Model

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

Resumptive Model

The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stabilize the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that wants resumption-like behavior. We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

Uncaught Exceptions in Java

- In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints an exception message with the help of an uncaught exception handler.

- The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.
- Java programming language has a very strong exception handling mechanism. It allow us to handle the exception use the keywords like try, catch, finally, throw, and throws.
- When an uncaught exception occurs, the JVM calls a special private method known **dispatchUncaughtException()**, on the Thread class in which the exception occurs and terminates the thread.
- The Division by zero exception is one of the example for uncaught exceptions. Look at the following code.

Example

When we execute the above code, it produce the following output for the value

a = 10 and b = 0.

The screenshot shows the Eclipse IDE with a Java file named `UncaughtExceptionExample.java`. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class UncaughtExceptionExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.println("Enter the a and b values: ");
9         int a = read.nextInt();
10        int b = read.nextInt();
11        int c = a / b;
12        System.out.println(a + "/" + b + " = " + c);
13    }
14 }
15 }
```

The console output shows the program execution:

```
<terminated> UncaughtExceptionExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 12:34:25 AM - 12:34:42 AM)
Enter the a and b values:
10
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at UncaughtExceptionExample.main(UncaughtExceptionExample.java:11)
```

The status bar at the bottom indicates 'Writable', 'Smart Insert', and the time '13:11:333'. An 'Activate Windows' watermark is visible in the bottom right corner.

In the above example code, we are not using try and catch blocks, but when the value of b is zero the division by zero exception occurs and it is caught by the default exception handler.

Try and Catch in Java

In Java, the **try** and **catch**, both are the keywords used for exception handling.

The keyword **try** is used to define a block of code that will be tested for the occurrence of an exception. The keyword **catch** is used to define a block of code that handles the exception occurred in the respective try block.

The uncaught exceptions are the exceptions that are not caught by the compiler but are automatically caught and handled by the Java built-in exception handler.

Both **try** and **catch** are used as a pair. Every try block must have one or more catch blocks. We cannot use **try** without at least one **catch**, and **catch** alone cannot be used (**catch** without **try** is not allowed).

The following is the syntax of try and catch blocks.

Syntax

```
try{
    ...
    code to be tested
    ...
}
catch(ExceptionType object){
    ...
    code for handling the exception
    ...
}
```

Consider the following example code to illustrate try and catch blocks in Java.

Example

The screenshot shows the Eclipse IDE with a Java file named `TryCatchExample.java`. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class TryCatchExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.println("Enter the a and b values: ");
9         try {
10             int a = read.nextInt();
11             int b = read.nextInt();
12             int c = a / b;
13             System.out.println(a + "/" + b + " = " + c);
14         }
15         catch(ArithmeticException ae) {
16             System.out.println("Problem info: Value of divisor can not be ZERO");
17         }
18     }
19 }
```

The console output shows the program execution:

```
<terminated> UncaughtExceptionExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 1:08:45 AM - 1:08:51 AM)
Enter the a and b values:
10
0
Problem info: Value of divisor can not be ZERO
```

In the above example code, when an exception occurs in the try block the execution control transferred to the catch block and the catch block handles it.

Multiple catch clauses

In java programming language, a try block may has one or more number of catch blocks. That means a single try statement can have multiple catch clauses.

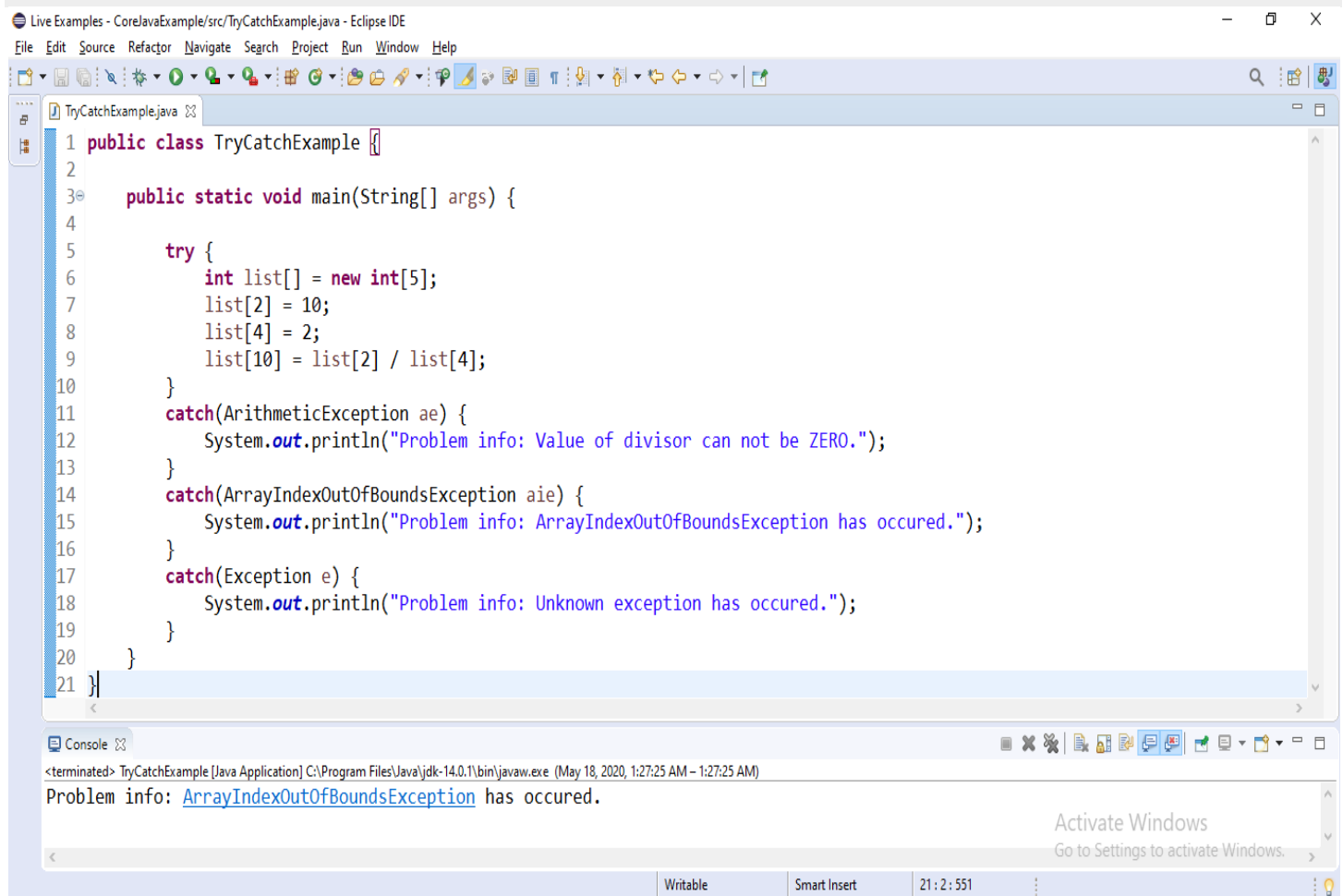
When a try block has more than one catch block, each catch block must contain a different exception type to be handled.

The multiple catch clauses are defined when the try block contains the code that may lead to different type of exceptions.

- ❑ The try block generates only one exception at a time, and at a time only one catch block is executed.
- ❑ When there are multiple catch blocks, the order of catch blocks must be from the most specific exception handler to most general.
- ❑ The catch block with Exception class handler must be defined at the last.

Let's look at the following example Java code to illustrate multiple catch clauses.

Example



The screenshot shows the Eclipse IDE with a Java file named `TryCatchExample.java`. The code defines a public class `TryCatchExample` with a `main` method. Inside the `main` method, a `try` block contains several lines of code: `int list[] = new int[5];`, `list[2] = 10;`, `list[4] = 2;`, and `list[10] = list[2] / list[4];`. This is followed by three `catch` blocks: `catch(ArithmeticException ae)`, `catch(ArrayIndexOutOfBoundsException aie)`, and `catch(Exception e)`. Each catch block prints a message to the console. The console output shows the message: "Problem info: [ArrayIndexOutOfBoundsException](#) has occurred." The status bar at the bottom indicates the file is writable, smart insert is on, and the time is 21:2:551.

```
1 public class TryCatchExample {
2
3     public static void main(String[] args) {
4
5         try {
6             int list[] = new int[5];
7             list[2] = 10;
8             list[4] = 2;
9             list[10] = list[2] / list[4];
10        }
11        catch(ArithmeticException ae) {
12            System.out.println("Problem info: Value of divisor can not be ZERO.");
13        }
14        catch(ArrayIndexOutOfBoundsException aie) {
15            System.out.println("Problem info: ArrayIndexOutOfBoundsException has occurred.");
16        }
17        catch(Exception e) {
18            System.out.println("Problem info: Unknown exception has occurred.");
19        }
20    }
21 }
```

Console Output:

```
<terminated> TryCatchExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 1:27:25 AM - 1:27:25 AM)
Problem info: ArrayIndexOutOfBoundsException has occurred.
```

Nested try statements

The java allows to write a try statement inside another try statement. A try block within another try block is known as nested try block.

When there are nested try blocks, each try block must have one or more separate catch blocks.

Let's look at the following example Java code to illustrate nested try statements.

Example

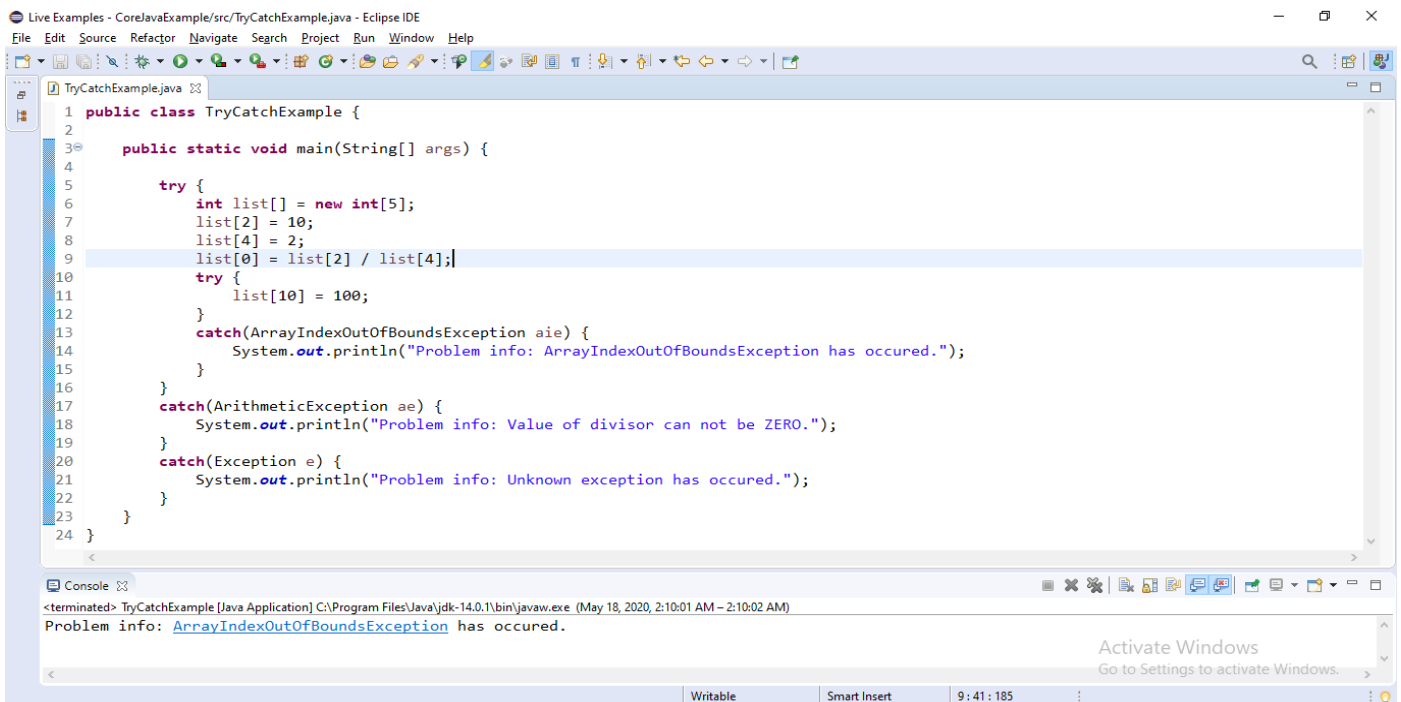
```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int list[] = new int[5];
            list[2] = 10;
            list[4] = 2;
```

```

list[0] = list[2] / list[4];
try {
    list[10] = 100;
}
catch(ArrayIndexOutOfBoundsException aie) {
    System.out.println("Problem info: ArrayIndexOutOfBoundsException has occurred.");
}
}
catch(ArithmeticException ae) {
    System.out.println("Problem info: Value of divisor can not be ZERO.");
}
catch(Exception e) {
    System.out.println("Problem info: Unknown exception has occurred.");
}
}
}

```

When we run the above code, it produce the following output.



The screenshot shows the Eclipse IDE with the file `TryCatchExample.java` open. The code is as follows:

```

1 public class TryCatchExample {
2
3     public static void main(String[] args) {
4
5         try {
6             int list[] = new int[5];
7             list[2] = 10;
8             list[4] = 2;
9             list[0] = list[2] / list[4];
10            try {
11                list[10] = 100;
12            }
13            catch(ArrayIndexOutOfBoundsException aie) {
14                System.out.println("Problem info: ArrayIndexOutOfBoundsException has occurred.");
15            }
16        }
17        catch(ArithmeticException ae) {
18            System.out.println("Problem info: Value of divisor can not be ZERO.");
19        }
20        catch(Exception e) {
21            System.out.println("Problem info: Unknown exception has occurred.");
22        }
23    }
24 }

```

The console output at the bottom shows:

```

<terminated> TryCatchExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 2:10:01 AM – 2:10:02 AM)
Problem info: ArrayIndexOutOfBoundsException has occurred.

```

The status bar at the bottom indicates the time is 9:41:185.

❑ In case of nested try blocks, if an exception occurred in the inner try block and its catch blocks are unable to handle it then it transfers the control to the outer try's catch block to handle it.

throw, throws, and finally keywords in Java

In java, the keywords throw, throws, and finally are used in the exception handling concept. Let's look at each of these keywords.

throw keyword in Java

The throw keyword is used to throw an exception instance explicitly from a try block to corresponding catch block. That means it is used to transfer the control from try block to corresponding catch block.

The throw keyword must be used inside the try block. When JVM encounters the throw keyword, it stops the execution of try block and jump to the corresponding catch block.

- ❑ Using throw keyword only object of Throwable class or its sub classes can be thrown.
- ❑ Using throw keyword only one exception can be thrown.
- ❑ The throw keyword must followed by an throwable instance.

The following is the general syntax for using throw keyword in a try block.

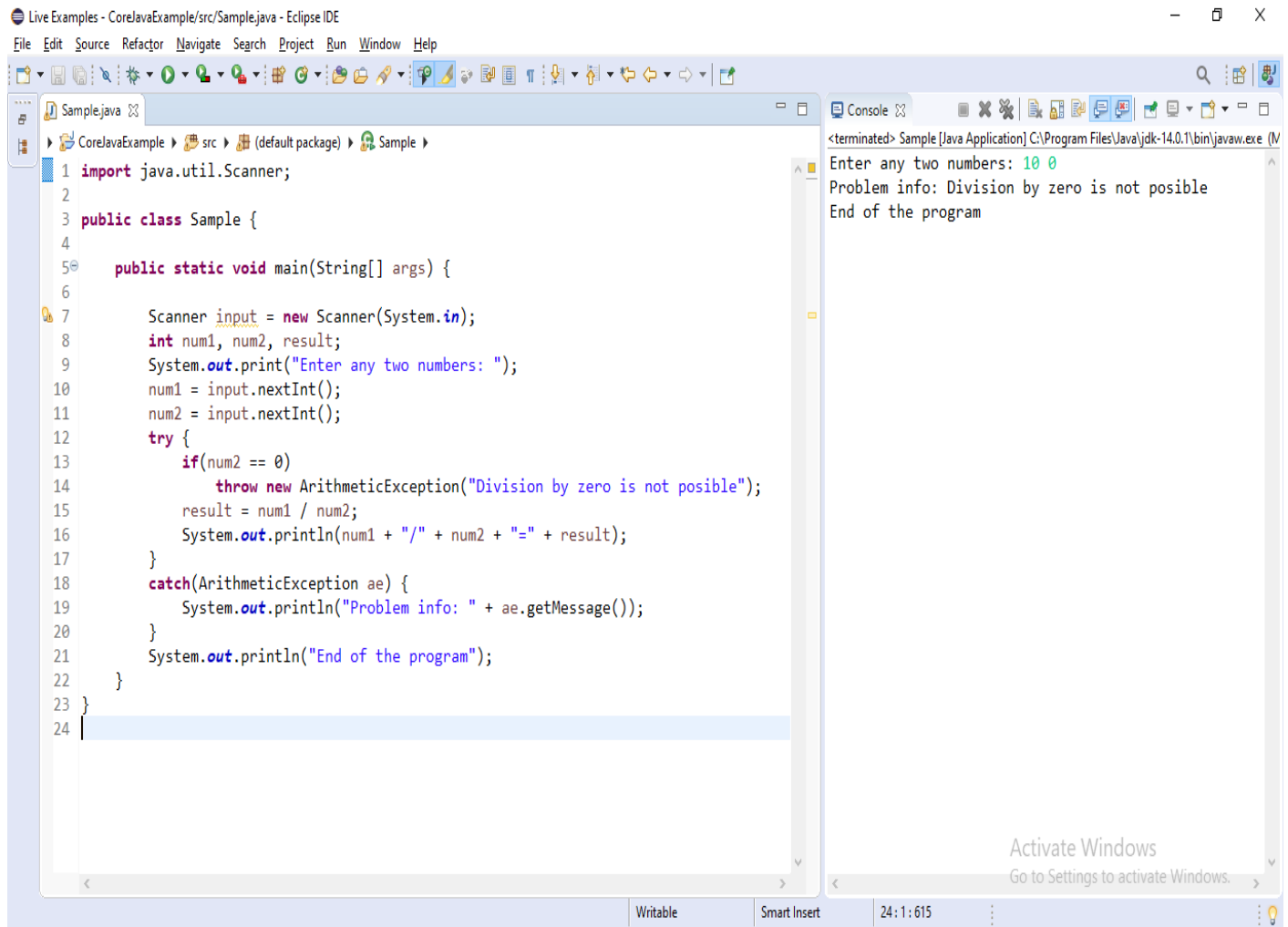
Syntax

```
throw instance;
```

Here the instance must be throwable instance and it can be created dynamically using new operator.

Let's look at the following example Java code to illustrate throw keyword.

Example



throws keyword in Java

The throws keyword specifies the exceptions that a method can throw to the default handler and does not handle itself. That means when we need a method to throw an exception automatically, we use throws keyword followed by method declaration

□ When a method throws an exception, we must put the calling statement of method in try-catch block.

Let's look at the following example Java code to illustrate throws keyword.

Example

```
import java.util.Scanner;  
public class ThrowsExample {
```

```
int num1, num2, result;
Scanner input = new Scanner(System.in);

void division() throws ArithmeticException {
    System.out.print("Enter any two numbers: ");
    num1 = input.nextInt();
    num2 = input.nextInt();
    result = num1 / num2;
    System.out.println(num1 + "/" + num2 + "=" + result);
}

public static void main(String[] args) {

    try {
        new ThrowsExample().division();
    }
    catch(ArithmeticException ae) {
        System.out.println("Problem info: " + ae.getMessage());
    }
    System.out.println("End of the program");
}
```

When we run the above code, it produce the following output.

The screenshot shows the Eclipse IDE with a project named 'CoreJavaExample'. The 'src' folder contains a package 'ThrowsExample' with a file 'ThrowsExample.java'. The code in the file is as follows:

```
1 import java.util.Scanner;
2
3 public class ThrowsExample {
4
5
6     int num1, num2, result;
7     Scanner input = new Scanner(System.in);
8
9     void division() throws ArithmeticException {
10         System.out.print("Enter any two numbers: ");
11         num1 = input.nextInt();
12         num2 = input.nextInt();
13         result = num1 / num2;
14         System.out.println(num1 + "/" + num2 + "=" + result);
15     }
16
17     public static void main(String[] args) {
18
19         try {
20             new ThrowsExample().division();
21         }
22         catch(ArithmeticException ae) {
23             System.out.println("Problem info: " + ae.getMessage());
24         }
25         System.out.println("End of the program");
26     }
27 }
28
29
```

The console output on the right shows the program execution:

```
<terminated> ThrowsExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw
Enter any two numbers: 20 0
Problem info: / by zero
End of the program
```

finally keyword in Java

The finally keyword is used to define a block that must be executed irrespective of exception occurrence.

The basic purpose of the finally keyword is to cleanup resources allocated by the try block, such as closing a file, closing a database connection, etc.

- ❑ Only one finally block is allowed for each try block.
- ❑ Use of the finally block is optional.

Let's look at the following example Java code to illustrate the throws keyword.

Example

The screenshot shows the Eclipse IDE with a project named 'CoreJavaExample'. The 'src' folder contains a package 'FinallyExample' with a file 'FinallyExample.java'. The code in the file is as follows:

```
1 import java.util.Scanner;
2
3 public class FinallyExample {
4
5     public static void main(String[] args) {
6         int num1, num2, result;
7         Scanner input = new Scanner(System.in);
8         System.out.print("Enter any two numbers: ");
9         num1 = input.nextInt();
10        num2 = input.nextInt();
11        try {
12            if(num2 == 0)
13                throw new ArithmeticException("Division by zero");
14            result = num1 / num2;
15            System.out.println(num1 + "/" + num2 + "=" + result);
16        }
17        catch(ArithmeticException ae) {
18            System.out.println("Problem info: " + ae.getMessage());
19        }
20        finally {
21            System.out.println("The finally block executes always");
22        }
23        System.out.println("End of the program");
24    }
25 }
26
27
```

The console output on the right shows the program execution:

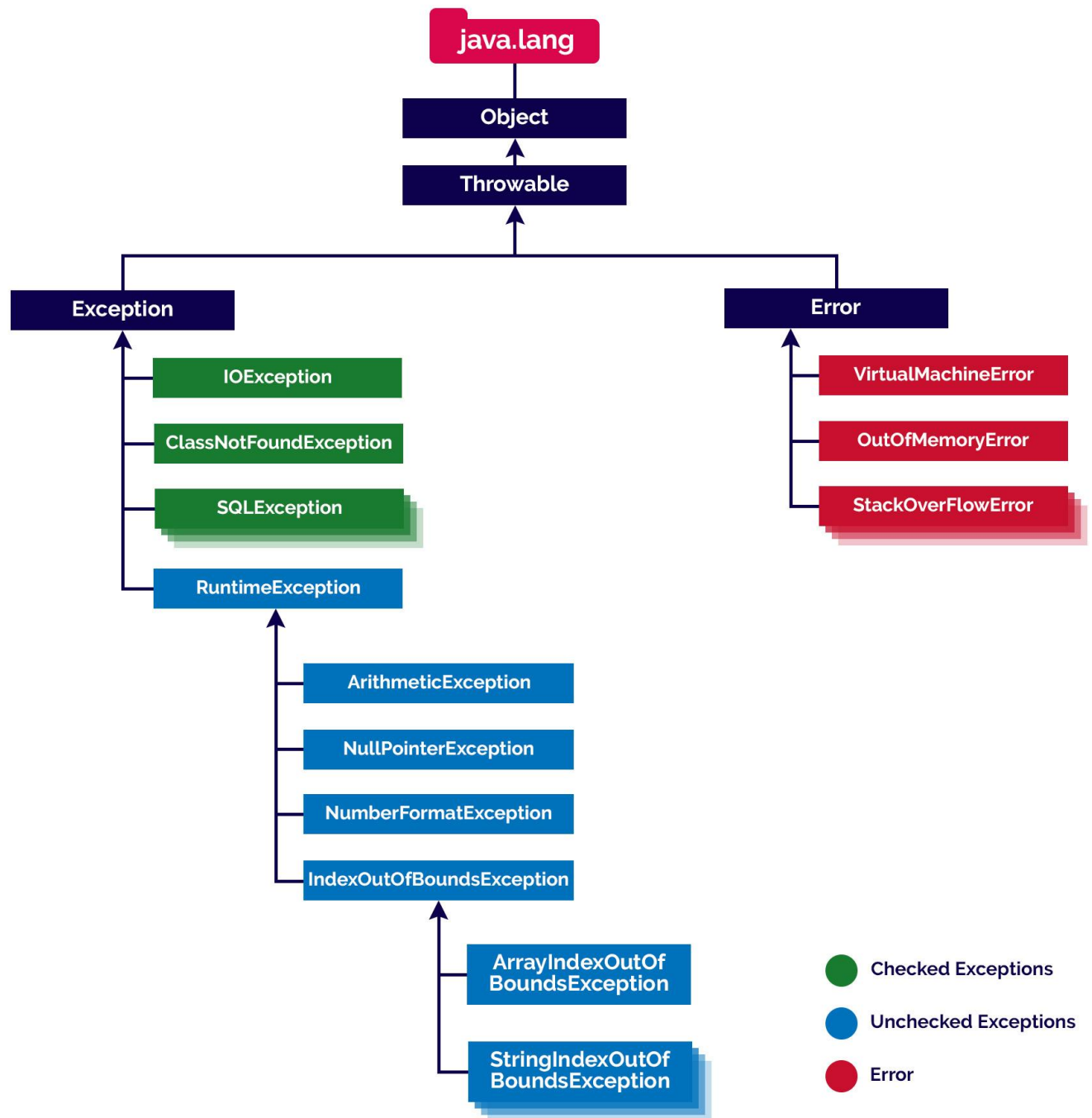
```
<terminated> FinallyExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw
Enter any two numbers: 20 0
Problem info: Division by zero
The finally block executes always
End of the program
```

Built-in Exceptions in Java

The Java programming language has several built-in exception class that support exception handling. Every exception class is suitable to explain certain error situations at run time.

All the built-in exception classes in Java were defined a package **java.lang**.

Few built-in exceptions in Java are shown in the following image.



List of checked exceptions in Java

The following table shows the list of several checked exceptions.

S. No.	Exception Class with Description
1	ClassNotFoundException It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.
2	CloneNotSupportedException Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.
3	IllegalAccessException It is thrown when one attempts to access a method or member that visibility qualifiers do not allow.
4	InstantiationException It is thrown when an application tries to create an instance of a class using the newInstance method in class Class, but the specified class object cannot be instantiated because it is an interface or is an abstract class.
5	InterruptedException It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.
6	NoSuchFieldException It indicates that the class doesn't have a field of a specified name.
7	NoSuchMethodException It is thrown when some JAR file has a different version at runtime than it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

List of unchecked exceptions in Java

The following table shows the list of several unchecked exceptions.

S. No.	Exception Class with Description
1	ArithmeticException It handles the arithmetic exceptions like division by zero
2	ArrayIndexOutOfBoundsException It handles the situations like an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3	ArrayStoreException It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects
4	AssertionError It is used to indicate that an assertion has failed
5	ClassCastException It handles the situation when we try to improperly cast a class from one type to another.
6	IllegalArgumentException This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument.
7	IllegalMonitorStateException This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.
8	IllegalStateException It signals that a method has been invoked at an illegal or inappropriate time.

S.
No. Exception Class with Description

9 `IllegalThreadStateException`

It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.

10 `IndexOutOfBoundsException`

It is thrown when attempting to access an invalid index within a collection, such as an array , vector , string , and so forth

11 `NegativeArraySizeException`

It is thrown if an applet tries to create an array with negative size.

12 `NullPointerException`

it is thrown when program attempts to use an object reference that has the null value.

13 `NumberFormatException`

It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.

14 `SecurityException`

It is thrown by the Java Card Virtual Machine to indicate a security violation.

15 `StringIndexOutOfBoundsException`

It is thrown by the methods of the String class, in order to indicate that an index is either negative, or greater than the size of the string itself.

16 `UnsupportedOperationException`

It is thrown to indicate that the requested operation is not supported.

Creating Own Exceptions in Java

- The Java programming language allow us to create our own exception classes which are basically subclasses built-in class **Exception**.
- To create our own exception class simply create a class as a subclass of built-in Exception class.
- We may create constructor in the user-defined exception class and pass a string to Exception class constructor using **super()**. We can use **getMessage()** method to access the string.
- Let's look at the following Java code that illustrates the creation of user-defined exception.

Example

```
import java.util.Scanner;

class NotEligibleException extends Exception{
    NotEligibleException(String msg){
        super(msg);    }    }
class VoterList{
    int age;
    VoterList(int age){    this.age = age;    }
    void checkEligibility() {
        try {
            if(age < 18) {
                throw new NotEligibleException("Error: Not eligible for vote due to under age.");
            }
            System.out.println("Congrates! You are eligible for vote.");
        }
        catch(NotEligibleException nee) {    System.out.println(nee.getMessage());    }    }
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);    System.out.println("Enter your age in years: ");
        int age = input.nextInt();    VoterList person = new VoterList(age);
        person.checkEligibility();
    }    }
```

Multithreading in java

- The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time.
- This type of program is known as a multithreading program.
- Each part of this program is called a thread.
- Every thread defines a separate path of execution in java.
- A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

- Process-based multitasking
- Thread-based multitasking

It is important to know the difference between process-based and thread-based multitasking. Let's distinguish both.

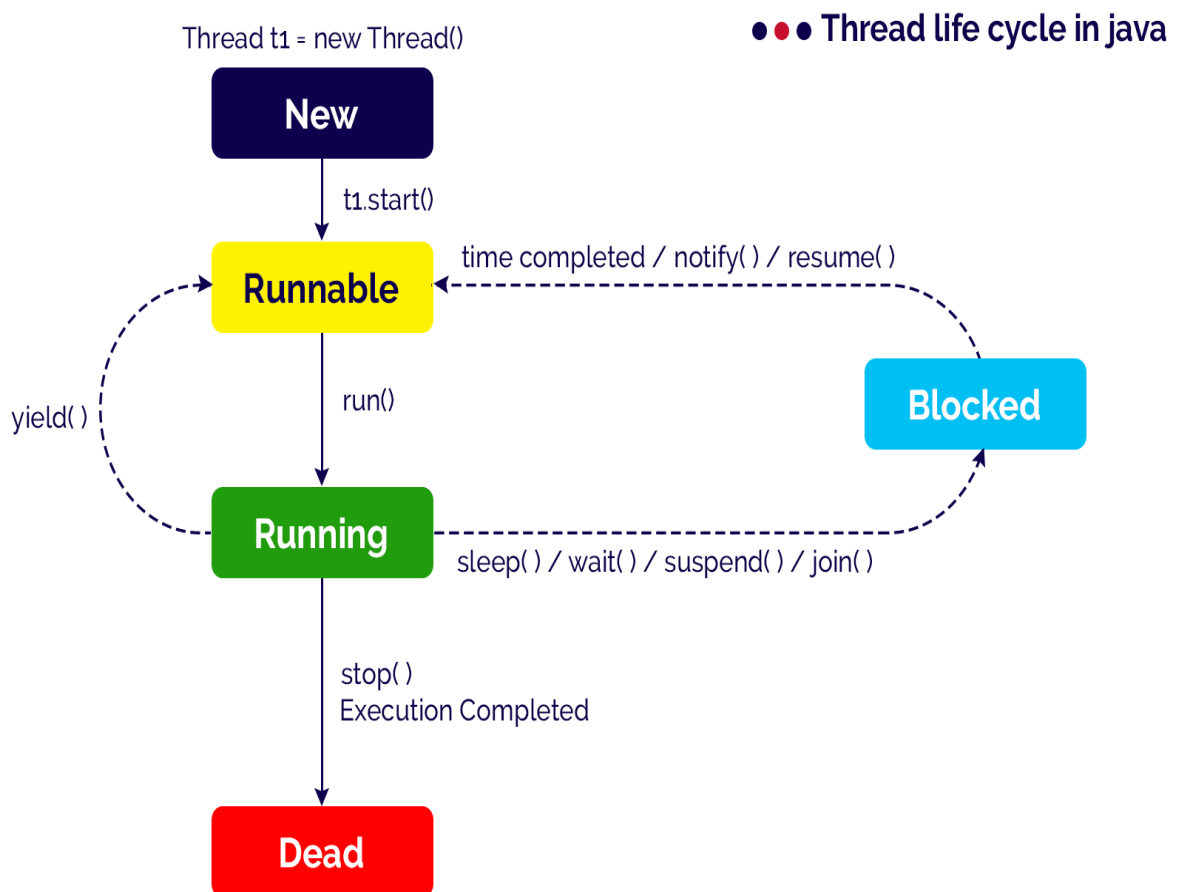
Process-based multitasking	Thread-based multitasking
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.
Process requires separate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not

expensive.

Java Thread Model

- The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time.
- This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java.
- A thread is explained in different ways, and a few of them are as specified below.

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



New

When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

Example

```
Thread t1 = new Thread();
```

Runnable / Ready

When a thread calls start() method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

Example

```
t1.start();
```

Running

When a thread calls run() method, then the thread is said to be Running. The run() method of a thread called automatically by the start() method.

Blocked / Waiting

A thread in the Running state may move into the blocked state due to various reasons like sleep() method called, wait() method called, suspend() method called, and join() method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify() or notifyAll() method called, resume() method called, etc.

Example

```
Thread.sleep(1000);  
wait(1000);  
wait(); suspend(); notify(); notifyAll(); resume();
```

Dead / Terminated

A thread in the Running state may move into the dead state due to either its execution completed or the stop() method called. The dead state is also known as the terminated state.

Creating threads in java

- In java, a thread is a lightweight process.
- Every java program executes by a thread called the main thread.
- When a java program gets executed, the main thread created automatically.
- All other threads called from the main thread.
- The java programming language provides two methods to create threads, and they are listed below.
- **Using Thread class (by extending Thread class)**
- **Using Runnable interface (by implementing Runnable interface)**

Let's see how to create threads using each of the above.

Extending Thread class

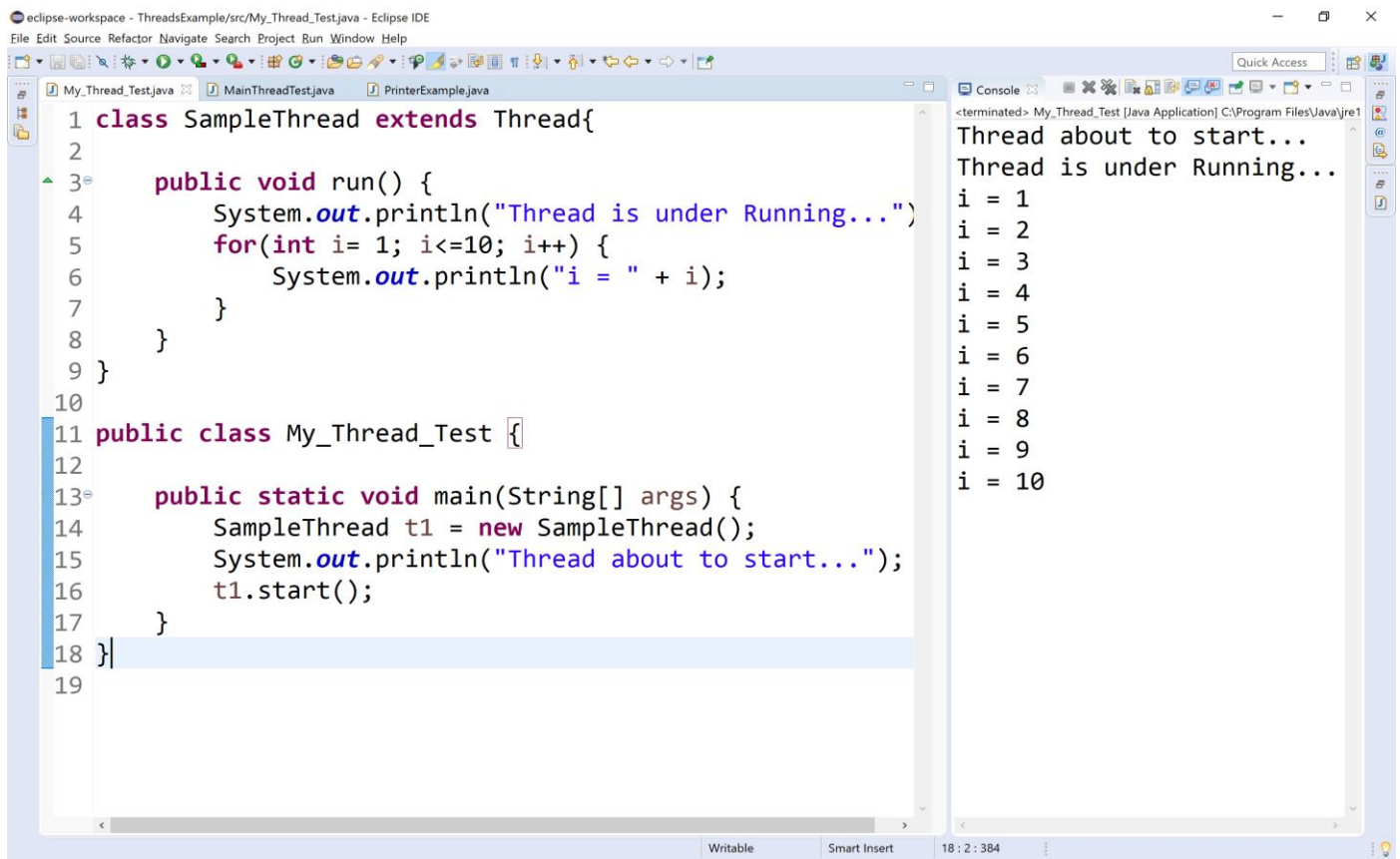
The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

- **Step-1:** Create a class as a child of Thread class. That means, create a class that extends Thread class.
- **Step-2:** Override the run() method with the code that is to be executed by the thread. The run() method must be public while overriding.
- **Step-3:** Create the object of the newly created class in the main() method.
- **Step-4:** Call the start() method on the object created in the above step.

Look at the following example program.

Example



```
1 class SampleThread extends Thread{
2
3     public void run() {
4         System.out.println("Thread is under Running...")
5         for(int i= 1; i<=10; i++) {
6             System.out.println("i = " + i);
7         }
8     }
9 }
10
11 public class My_Thread_Test {
12
13     public static void main(String[] args) {
14         SampleThread t1 = new SampleThread();
15         System.out.println("Thread about to start...");
16         t1.start();
17     }
18 }
19
```

```
<terminated> My_Thread_Test [Java Application] C:\Program Files\Java\jre1
Thread about to start...
Thread is under Running...
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

Implementng Runnable interface

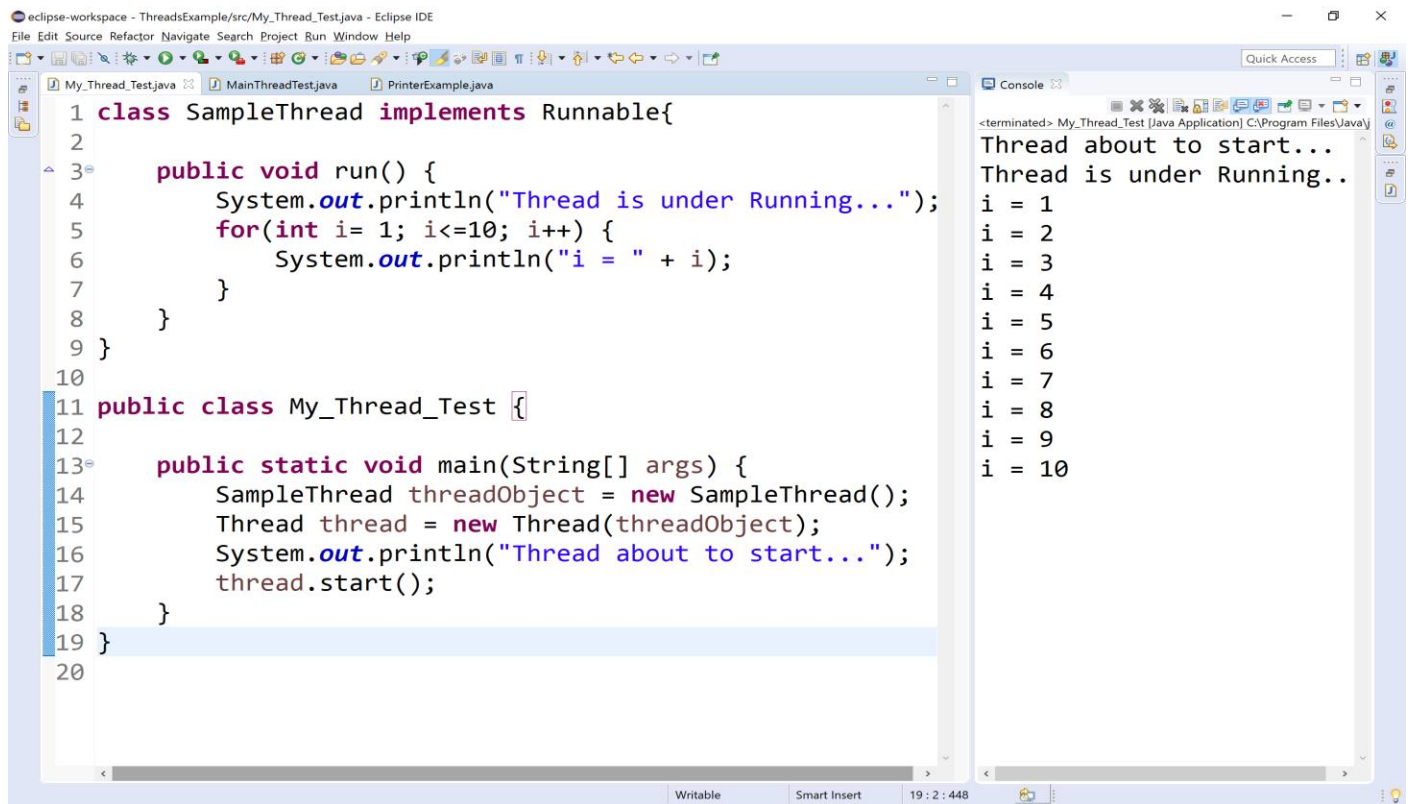
The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.

- **Step-1:** Create a class that implements Runnable interface.
- **Step-2:** Override the run() method with the code that is to be executed by the thread. The run() method must be public while overriding.
- **Step-3:** Create the object of the newly created class in the main() method.
- **Step-4:** Create the Thread class object by passing above created object as parameter to the Thread class constructor.
- **Step-5:** Call the start() method on the Thread class object created in the above step.

Look at the following example program.

Example



```
1 class SampleThread implements Runnable{
2
3     public void run() {
4         System.out.println("Thread is under Running...");
5         for(int i= 1; i<=10; i++) {
6             System.out.println("i = " + i);
7         }
8     }
9 }
10
11 public class My_Thread_Test {
12
13     public static void main(String[] args) {
14         SampleThread threadObject = new SampleThread();
15         Thread thread = new Thread(threadObject);
16         System.out.println("Thread about to start...");
17         thread.start();
18     }
19 }
20
```

```
<terminated> My_Thread_Test [Java Application] C:\Program Files\Java\
Thread about to start...
Thread is under Running..
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

More about Thread class

The Thread class in java is a subclass of Object class and it implements Runnable interface. The Thread class is available inside the java.lang package. The Thread class has the following syntax.

```
class Thread extends Object implements Runnable{
...
}
```

The Thread class has the following constructors.

- Thread()
- Thread(String threadName)
- Thread(Runnable objectName)
- Thread(Runnable objectName, String threadName)

The Thread class contains the following methods.

Method	Description	Return Value
run()	Defines actual task of the thread.	void
start()	It moves the thread from Ready state to Running state by calling run() method.	void
setName(String)	Assigns a name to the thread.	void
getName()	Returns the name of the thread.	String
setPriority(int)	Assigns priority to the thread.	void
getPriority()	Returns the priority of the thread.	int
getId()	Returns the ID of the thread.	long
activeCount()	Returns total number of thread under active.	int
currentThread()	Returns the reference of the thread that is currently in running state.	void
sleep(long)	moves the thread to blocked state till the specified number of milliseconds.	void
isAlive()	Tests if the thread is alive.	boolean

Method	Description	Return Value
yield()	Tells to the scheduler that the current thread is willing to yield its current use of a processor.	void
join()	Waits for the thread to end.	void
<input type="checkbox"/> The Thread class in java also contains methods like stop() , destroy() , suspend() , and resume() . But they are deprecated.		

Java Thread Priority

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.

The java programming language Thread class provides two methods **setPriority(int)**, and **getPriority()** to handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

- **MAX_PRIORITY** - It has the value 10 and indicates highest priority.
- **NORM_PRIORITY** - It has the value 5 and indicates normal priority.
- **MIN_PRIORITY** - It has the value 1 and indicates lowest priority.

☐ The default priority of any thread is 5 (i.e. NORM_PRIORITY).

setPriority() method

The setPriority() method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

The regular use of the setPriority() method is as follows.

Example

```
threadObject.setPriority(4); or
threadObject.setPriority(MAX_PRIORITY);
```

getPriority() method

The getPriority() method of Thread class used to access the priority of a thread. It does not takes any argument and returns name of the thread as String.

The regular use of the getPriority() method is as follows.

Example

```
String threadName = threadObject.getPriority();
```

Look at the following example program.

Example

```
class SampleThread extends Thread{
    public void run() {
        System.out.println("Inside SampleThread");
        System.out.println("Current Thread: " + Thread.currentThread().getName());
    } }

public class My_Thread_Test {
    public static void main(String[] args) {
        SampleThread threadObject1 = new SampleThread();
        SampleThread threadObject2 = new SampleThread();
        threadObject1.setName("first");
        threadObject2.setName("second");
        threadObject1.setPriority(4);
        threadObject2.setPriority(Thread.MAX_PRIORITY);
        threadObject1.start();
        threadObject2.start();
    } }
```

The screenshot shows the Eclipse IDE with a Java project named 'ThreadsExample'. The main editor displays the file 'My_Thread_Test.java'. The code defines a 'SampleThread' class that extends 'Thread' and implements the 'run()' method to print 'Inside SampleThread' and the current thread's name. The 'My_Thread_Test' class has a 'main' method that creates two 'SampleThread' objects, 'threadObject1' and 'threadObject2'. 'threadObject1' is named 'first' and 'threadObject2' is named 'second'. 'threadObject1' is given a priority of 4, and 'threadObject2' is given the priority of 'Thread.MAX_PRIORITY'. Both threads are started. The console on the right shows the output of the program, which is terminated. The output shows the execution of 'threadObject2' (named 'second') first, followed by 'threadObject1' (named 'first').

```
1 class SampleThread extends Thread{
2     public void run() {
3         System.out.println("Inside SampleThread");
4         System.out.println("Current Thread: " + Thread.currentThread().getName());
5     }
6 }
7
8 public class My_Thread_Test {
9
10    public static void main(String[] args) {
11        SampleThread threadObject1 = new SampleThread();
12        SampleThread threadObject2 = new SampleThread();
13        threadObject1.setName("first");
14        threadObject2.setName("second");
15
16        threadObject1.setPriority(4);
17        threadObject2.setPriority(Thread.MAX_PRIORITY);
18
19        threadObject1.start();
20        threadObject2.start();
21
22    }
23 }
24
```

Console Output:

```
<terminated> My_Thread_Test [Java Application] C:\Program Fi
Inside SampleThread
Current Thread: second
Inside SampleThread
Current Thread: first
```

□ In java, it is not guaranteed that threads execute according to their priority because it depends on JVM specification that which scheduling it chooses.

Java Thread Synchronisation

- The java programming language supports multithreading.
- The problem of shared resources occurs when two or more threads get execute at the same time.
- In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.

□ The synchronization is the process of allowing only one thread to access a shared resource at a time.

In java, the synchronization is achieved using the following concepts.

- Mutual Exclusion
- Inter thread communication

In this tutorial, we discuss mutual exclusion only, and the interthread communication will be discussed in the next tutorial.

Mutual Exclusion

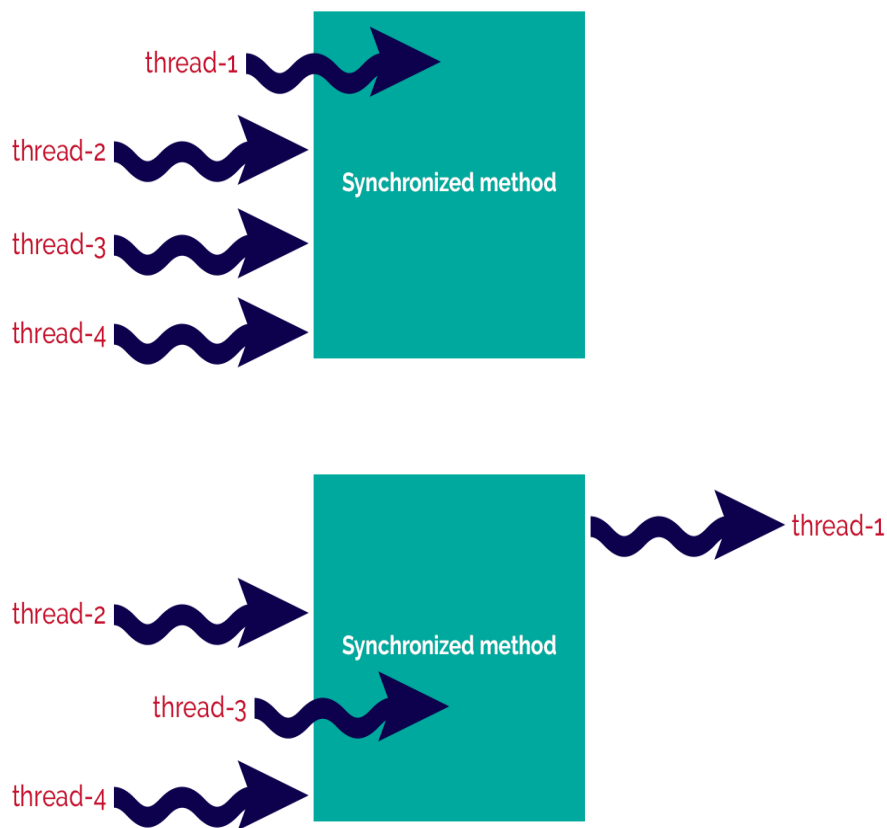
Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

- Synchronized method
- Synchronized block

Synchronized method

When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.

●●● Java thread execution with synchronized method



www.btechsmartclass.com

In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method). When thread-1 completes its task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example

```
class Table{
    synchronized void printTable(int n) {
        for(int i = 1; i <= 10; i++)
            System.out.println(n + " * " + i + " = " + i*n);
    }
}
```

```
class MyThread_1 extends Thread{
    Table table = new Table();
    int number;
    MyThread_1(Table table, int number){
        this.table = table;
        this.number = number;
    }
    public void run() {
        table.printTable(number);
    }
}
```

```
class MyThread_2 extends Thread{

    Table table = new Table();
    int number;
    MyThread_2(Table table, int number){
        this.table = table;
        this.number = number;
    }
    public void run() {
        table.printTable(number);
    }
}
```

```

}

public class ThreadSynchronizationExample {

    public static void main(String[] args) {
        Table table = new Table();
        MyThread_1 thread_1 = new MyThread_1(table, 5);
        MyThread_2 thread_2 = new MyThread_2(table, 10);
        thread_1.start();
        thread_2.start();
    }
}

```

When we run this code, it produce the following output.

The screenshot shows the Eclipse IDE with the file `ThreadSynchronizationExample.java` open. The code defines a `Table` class with a `synchronized printTable(int n)` method that prints multiplication results for `i` from 1 to `n`. Two threads, `MyThread_1` (with `number = 5`) and `MyThread_2` (with `number = 10`), are created and started. The console output shows the results of these threads running in parallel, interleaved.

```

1 class Table{
2     synchronized void printTable(int n) {
3         for(int i = 1; i <= 10; i++)
4             System.out.println(n + " * " + i + " = " + i*n);
5     }
6 }
7
8
9 class MyThread_1 extends Thread{
10     Table table = new Table();
11     int number;
12     MyThread_1(Table table, int number){
13         this.table = table;
14         this.number = number;
15     }
16     public void run() {
17         table.printTable(number);
18     }
19 }
20
21 class MyThread_2 extends Thread{
22
23     Table table = new Table();
24     int number;
25     MyThread_2(Table table, int number){
26         this.table = table;

```

Console Output:

```

<terminated> ThreadSynchronizationExample (Java Application) C:\
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100

```

Synchronized block

The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.

The following syntax is used to define a synchronized block.

Syntax

```
synchronized(object){  
    ...  
    block code  
    ...  
}
```

Look at the following example code to illustrate synchronized block.

Example

```
import java.util.*;  
class NameList {  
    String name = "";  
    public int count = 0;  
    public void addName(String name, List<String> namesList){  
        synchronized(this){  
            this.name = name;  
            count++;  
        }    namesList.add(name);  
    }  
    public int getCount(){  
        return count;  
    }  
}  
  
public class SynchronizedBlockExample {  
    public static void main (String[] args)
```

```

{
    NameList namesList_1 = new NameList();
    NameList namesList_2 = new NameList();
    List<String> list = new ArrayList<String>();
    namesList_1.addName("Rama", list);
    namesList_2.addName("Seetha", list);
    System.out.println("Thread1: " + namesList_1.name + ", " + namesList_1.getCount() +
"\n");
    System.out.println("Thread2: " + namesList_2.name + ", " + namesList_2.getCount() +
"\n");
} }

```

The screenshot shows the Eclipse IDE with a Java file named `SynchronizedBlockExample.java`. The code defines a `NameList` class with a `name` string and a `count` integer. It has two methods: `addName(String name, List<String> namesList)` which is synchronized and increments the count, and `getCount()` which returns the count. The `main` method creates two `NameList` objects, adds "Rama" to the first and "Seetha" to the second, and prints their names and counts. The console output shows:

Thread1: Rama, 1

Thread2: Seetha, 1

□ The complete code of a method may be written inside the synchronized block, where it works similarly to the synchronized method.

Java Inter Thread Communication

- Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**.

- In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
- That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task. T
- he inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- wait()
- notify()
- notifyAll()

The following table gives detailed description about the above methods.

Method	Description
void wait()	It makes the current thread to pause its execution until other thread in the same monitor calls notify()
void notify()	It wakes up the thread that called wait() on the same object.
void notifyAll()	It wakes up all the threads that called wait() on the same object.

□ Calling notify() or notifyAll() does not actually give up a lock on a resource.

Let's look at an example problem of producer and consumer. The producer produces the item and the consumer consumes the same. But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced.

So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same. Here we use the inter-thread communication to implement the producer and consumer problem.

The sample implementation of producer and consumer problem is as follows.

Example

```
class ItemQueue {
    int item;
    boolean valueSet = false;
    synchronized int getItem()
```

```

{
    while (!valueSet)
        try { wait(); } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

    System.out.println("Consummed:" + item);
    valueSet = false;
    try { Thread.sleep(1000);
    } catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
    } notify(); return item; }

synchronized void putItem(int item) {
    while (valueSet)
        try { wait();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

    this.item = item; valueSet = true;
    System.out.println("Produced: " + item);
    try { Thread.sleep(1000);
    } catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
    }
    notify();
} }

class Producer implements Runnable{
    ItemQueue itemQueue;
    Producer(ItemQueue itemQueue){
        this.itemQueue = itemQueue;
        new Thread(this, "Producer").start();
    }

    public void run() {

```

```

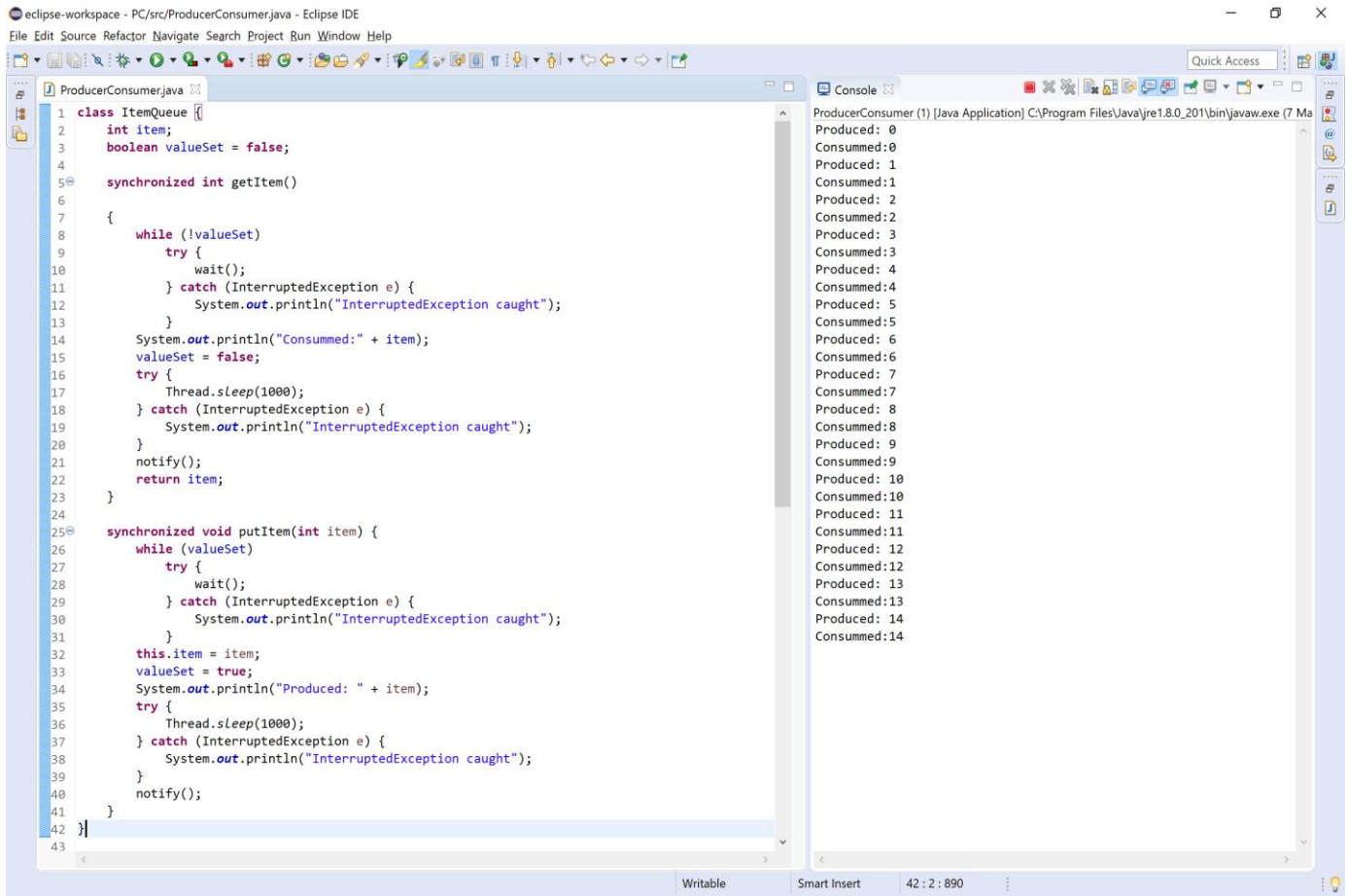
        int i = 0;
        while(true) {
            itemQueue.putItem(i++);
        } } }

class Consumer implements Runnable{
    ItemQueue itemQueue;
    Consumer(ItemQueue itemQueue){
        this.itemQueue = itemQueue;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) { itemQueue.getItem(); }
    } }

class ProducerConsumer{
    public static void main(String args[]) {
        ItemQueue itemQueue = new ItemQueue();
        new Producer(itemQueue);
        new Consumer(itemQueue);
    }
}

```

When we run this code, it produce the following output.



```
1 class ItemQueue {
2     int item;
3     boolean valueSet = false;
4
5     synchronized int getItem()
6
7     {
8         while (!valueSet)
9             try {
10                 wait();
11             } catch (InterruptedException e) {
12                 System.out.println("InterruptedException caught");
13             }
14         System.out.println("Consumed: " + item);
15         valueSet = false;
16         try {
17             Thread.sleep(1000);
18         } catch (InterruptedException e) {
19             System.out.println("InterruptedException caught");
20         }
21         notify();
22         return item;
23     }
24
25     synchronized void putItem(int item) {
26         while (valueSet)
27             try {
28                 wait();
29             } catch (InterruptedException e) {
30                 System.out.println("InterruptedException caught");
31             }
32         this.item = item;
33         valueSet = true;
34         System.out.println("Produced: " + item);
35         try {
36             Thread.sleep(1000);
37         } catch (InterruptedException e) {
38             System.out.println("InterruptedException caught");
39         }
40         notify();
41     }
42 }
43 }
```

Console Output:

```
Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
Produced: 11
Consumed: 11
Produced: 12
Consumed: 12
Produced: 13
Consumed: 13
Produced: 14
Consumed: 14
```

□ All the methods wait(), notify(), and notifyAll() can be used only inside the synchronized methods only.

Java Programming (R20CSE2204)

UNIT - IV

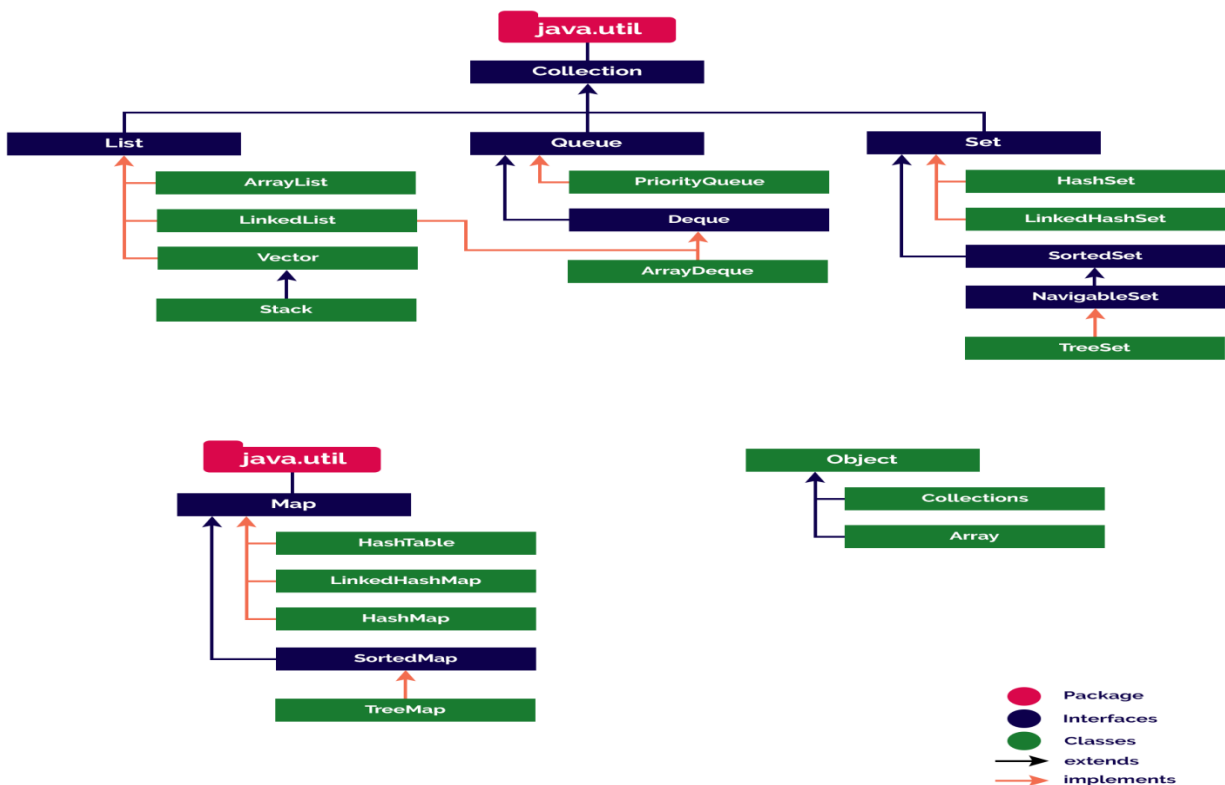
The Collections Framework (java.util)- Collections overview, Collection Interfaces, The Collection classes- Array List, Linked List, Hash Set, Tree Set, Priority Queue, Array Deque. Accessing a Collection via an Iterator, Using an Iterator, The For-Each alternative, Map Interfaces and Classes, Comparators, Collection algorithms, Arrays, The Legacy Classes and Interfaces- Dictionary, Hashtable, Properties, Stack, Vector
More Utility classes, String Tokenizer, Bit Set, Date, Calendar, Random, Formatter, Scanner

Java Collection Framework

- Java collection framework is a collection of interfaces and classes used to storing and processing a group of individual objects as a single unit.
- The java collection framework holds several classes that provide a large number of methods to store and process a group of objects.
- These classes make the programmer task super easy and fast.

Java collection framework was introduced in java 1.2 version.

● ● ● Java Collection Framework



- Before the collection framework in java (before java 1.2 version), there was a set of classes like **Array**, **Vector**, **Stack**, **HashTable**. These classes are known as **legacy classes**.
- The java collection framework contains List, Queue, Set, and Map as top-level interfaces.
- The List, Queue, and Set stores single value as its element, whereas Map stores a pair of a key and value as its element.

Java Collection Interface

- The **Collection** interface is the root interface for most of the interfaces and classes of collection framework.
- The Collection interface is available inside the **java.util** package.
- It defines the methods that are commonly used by almost all the collections.
- The Collection interface defines the following methods.

• • • Methods of **Collection** interface in java



www.btechsmartclass.com

□ The Collection interface extends **Iterable** interface.

Let's consider an example program on ArrayList to illustrate the methods of Collection interface.

Example

Eclipse-workspace - JavaCollectionFramework/src/CollectionInterfaceExample.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

The screenshot displays the Eclipse IDE interface. The left pane shows the source code for `CollectionInterfaceExample.java`. The right pane shows the console output of the program.

Source Code:

```
1 import java.util.*;
2
3 public class CollectionInterfaceExample {
4
5     public static void main(String[] args) {
6
7         List list_1 = new ArrayList();
8         List<String> list_2 = new ArrayList<String>();
9
10        list_1.add(10);
11        list_1.add(20);
12        list_2.add("BTech");
13        list_2.add("Smart");
14        list_2.add("Class");
15
16        list_1.addAll(list_2);
17        System.out.println("Elements of list_1: " + list_1);
18
19        System.out.println("Search for BTech: " + list_1.contains("BTech"));
20        System.out.println("Search for list_2 in list_1: " + list_1.containsAll(list_2));
21
22        System.out.println("Check whether list_1 and list_2 are equal: " + list_1.equals(list_2));
23
24        System.out.println("Check is list_1 empty: " + list_1.isEmpty());
25
26        System.out.println("Size of list_1: " + list_1.size());
27
28        System.out.println("Hashcode of list_1: " + list_1.hashCode());
29
30        list_1.remove(0);
31        System.out.println(list_1);
32
33        list_1.retainAll(list_2);
34        System.out.println(list_1);
35
36        list_1.removeAll(list_2);
37        System.out.println(list_1);
38
39        list_2.clear();
40        System.out.println(list_2);
41    }
42 }
43
```

Console Output:

```
<terminated> CollectionInterfaceExample [Java Application] C:\Program Files\Java\jre1
Elements of list_1: [10, 20, BTech, Smart, Class]
Search for BTech: true
Search for list_2 in list_1: true
Check whether list_1 and list_2 are equal: false
Check is list_1 empty: false
Size of list_1: 5
Hashcode of list_1: -764556324
[20, BTech, Smart, Class]
[BTech, Smart, Class]
[]
[]
```

The bottom status bar of the IDE shows "Writable", "Smart Insert", and the cursor position "42:2:1114".

Java ArrayList Class

- The **ArrayList** class is a part of java collection framework.
- It is available inside the **java.util** package. The ArrayList class extends AbstractList class and implements List interface.
- The elements of ArrayList are organized as an array internally. The default size of an ArrayList is 10.
- The ArrayList class is used to create a dynamic array that can grow or shrunk as needed.

- The ArrayList is a child class of **AbstractList**
- The ArrayList implements interfaces like **List**, **Serializable**, **Cloneable**, and **RandomAccess**.
- The ArrayList allows to store duplicate data values.
- The ArrayList allows to access elements randomly using index-based accessing.
- The ArrayList maintains the order of insertion.

ArrayList class declaration

The ArrayList class has the following declaration.

Example

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList class constructors

The ArrayList class has the following constructors.

- **ArrayList()** - Creates an empty ArrayList.
- **ArrayList(Collection c)** - Creates an ArrayList with given collection of elements.
- **ArrayList(int size)** - Creates an empty ArrayList with given size (capacity).

Operations on ArrayList

The ArrayList class allow us to perform several operations like adding, accessing, deleting, updating, looping, etc. Let's look at each operation with examples.

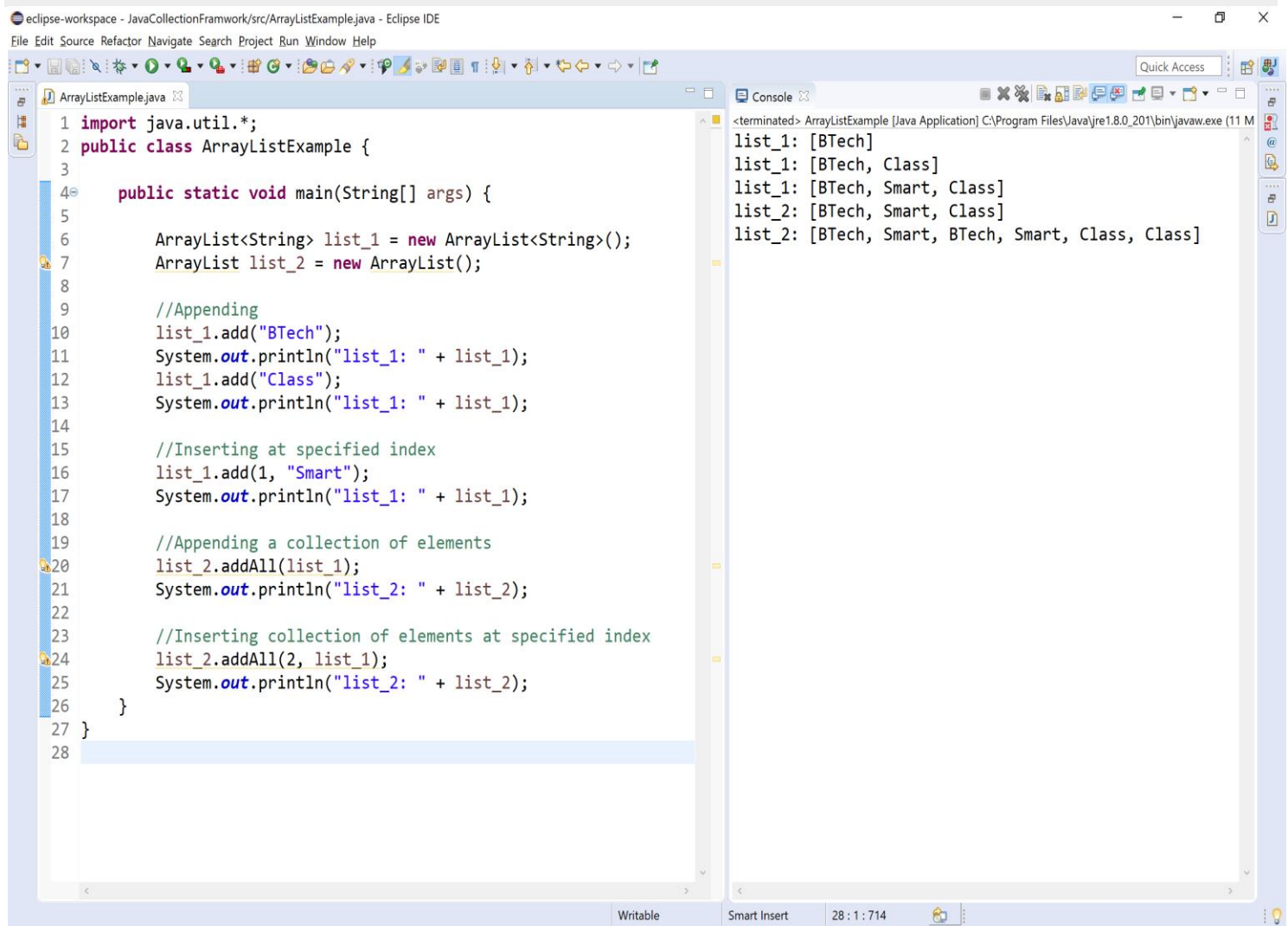
Adding Items

The ArrayList class has the following methods to add items.

- **boolean add(E element)** - Appends given element to the ArrayList.
- **boolean addAll(Collection c)** - Appends given collection of elements to the ArrayList.
- **void add(int index, E element)** - Inserts the given element at specified index.
- **boolean addAll(int index, Collection c)** - Inserts the given collection of elements at specified index.

Let's consider an example program to illustrate adding items to the ArrayList.

Example



```
eclipse-workspace - JavaCollectionFramework/src/ArrayListExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

ArrayListExample.java
1 import java.util.*;
2 public class ArrayListExample {
3
4     public static void main(String[] args) {
5
6         ArrayList<String> list_1 = new ArrayList<String>();
6         ArrayList list_2 = new ArrayList();
7
8         //Appending
9         list_1.add("BTech");
10        System.out.println("list_1: " + list_1);
11        list_1.add("Class");
12        System.out.println("list_1: " + list_1);
13
14        //Inserting at specified index
15        list_1.add(1, "Smart");
16        System.out.println("list_1: " + list_1);
17
18        //Appending a collection of elements
19        list_2.addAll(list_1);
20        System.out.println("list_2: " + list_2);
21
22        //Inserting collection of elements at specified index
23        list_2.addAll(2, list_1);
24        System.out.println("list_2: " + list_2);
25    }
26 }
27
28

Console
<terminated> ArrayListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (11 M
list_1: [BTech]
list_1: [BTech, Class]
list_1: [BTech, Smart, Class]
list_2: [BTech, Smart, Class]
list_2: [BTech, Smart, BTech, Smart, Class, Class]
```

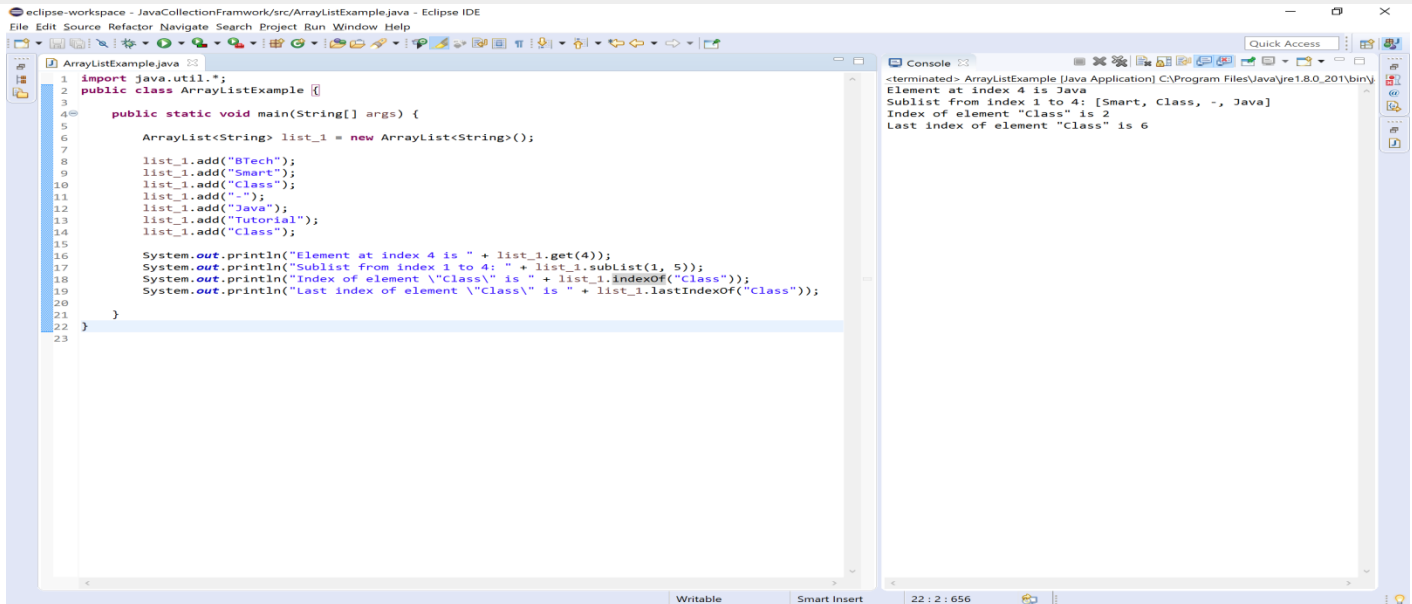
Accessing Items

The ArrayList class has the following methods to access items.

- **E get(int index)** - Returns element at specified index from the ArrayList.
- **ArrayList subList(int startIndex, int lastIndex)** - Returns an ArrayList that contains elements from specified startIndex to lastIndex-1 from the invoking ArrayList.
- **int indexOf(E element)** - Returns the index value of given element first occurrence in the ArrayList.
- **int lastIndexOf(E element)** - Returns the index value of given element last occurrence in the ArrayList.

Let's consider an example program to illustrate accessing items from the ArrayList.

Example



The screenshot shows the Eclipse IDE with a Java project named 'JavaCollectionFramework'. The file 'ArrayListExample.java' is open in the editor. The code defines a public class 'ArrayListExample' with a main method. In the main method, an 'ArrayList<String>' named 'list_1' is created and populated with the elements 'BTech', 'Smart', 'Class', '-', 'Java', 'Tutorial', and 'Class'. The program then prints the element at index 4, a sublist from index 1 to 4, the index of the first 'Class' element, and the last index of the 'Class' element. The console on the right shows the output of these operations.

```
1 import java.util.*;
2 public class ArrayListExample {
3
4     public static void main(String[] args) {
5
6         ArrayList<String> list_1 = new ArrayList<String>();
7
8         list_1.add("BTech");
9         list_1.add("Smart");
10        list_1.add("Class");
11        list_1.add("-");
12        list_1.add("Java");
13        list_1.add("Tutorial");
14        list_1.add("Class");
15
16        System.out.println("Element at index 4 is " + list_1.get(4));
17        System.out.println("Sublist from index 1 to 4: " + list_1.subList(1, 5));
18        System.out.println("Index of element \"Class\" is " + list_1.indexOf("Class"));
19        System.out.println("Last index of element \"Class\" is " + list_1.lastIndexOf("Class"));
20
21    }
22 }
23
```

Console Output:

```
-terminated- ArrayListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
Element at index 4 is Java
Sublist from index 1 to 4: [Smart, Class, -, Java]
Index of element "Class" is 2
Last index of element "Class" is 6
```

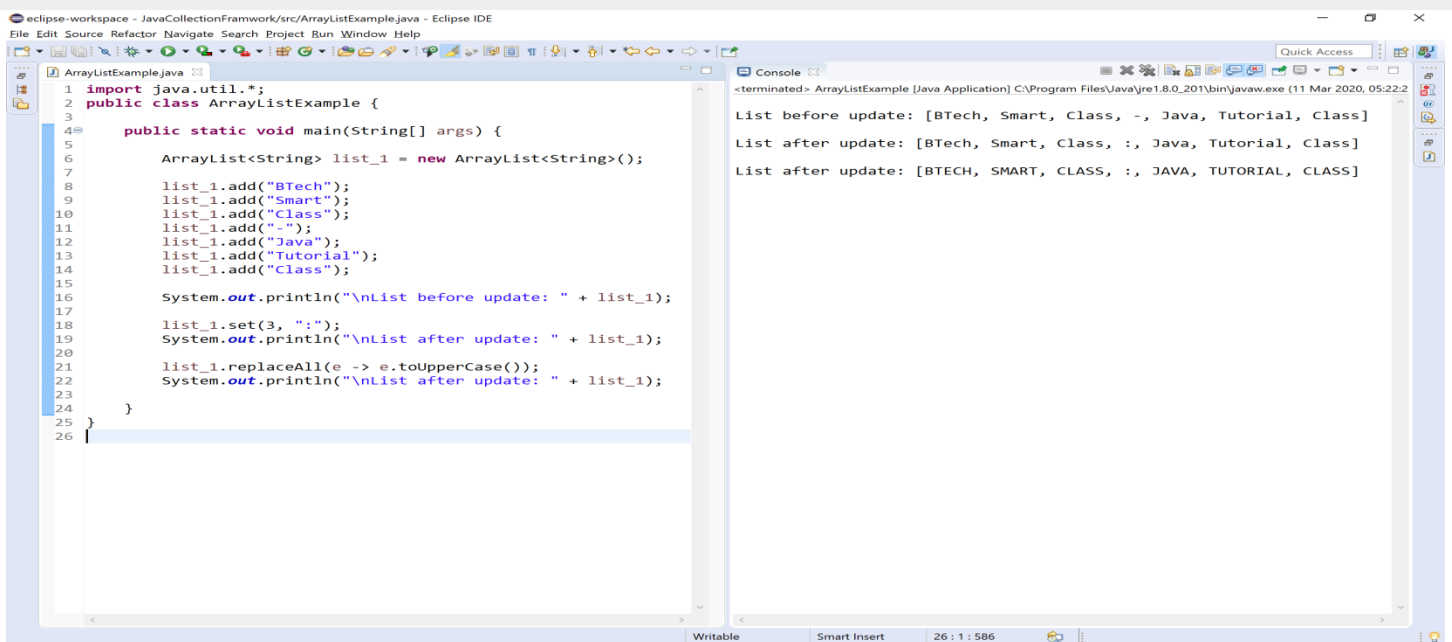
Updating Items

The ArrayList class has the following methods to update or change items.

- **set(int index, E newElement)** - Replace the element at specified index with newElement in the invoking ArrayList.
- **replaceAll(UnaryOperator e)** - Replaces each element of invoking ArrayList with the result of applying the operator to that element.

Let's consider an example program to illustrate updating items in the ArrayList.

Example



The screenshot shows the Eclipse IDE with the same 'JavaCollectionFramework' project. The file 'ArrayListExample.java' is open. The code is similar to the first example but includes updates to the 'list_1' ArrayList. It prints the list before and after setting the element at index 3 to ':', and after replacing all elements with their uppercase versions. The console on the right shows the output of these operations.

```
1 import java.util.*;
2 public class ArrayListExample {
3
4     public static void main(String[] args) {
5
6         ArrayList<String> list_1 = new ArrayList<String>();
7
8         list_1.add("BTech");
9         list_1.add("Smart");
10        list_1.add("Class");
11        list_1.add("-");
12        list_1.add("Java");
13        list_1.add("Tutorial");
14        list_1.add("Class");
15
16        System.out.println("\nList before update: " + list_1);
17
18        list_1.set(3, ":");
19        System.out.println("\nList after update: " + list_1);
20
21        list_1.replaceAll(e -> e.toUpperCase());
22        System.out.println("\nList after update: " + list_1);
23
24    }
25 }
26
```

Console Output:

```
<terminated- ArrayListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (11 Mar 2020, 05:22:2
List before update: [BTech, Smart, Class, -, Java, Tutorial, Class]
List after update: [BTech, Smart, Class, :, Java, Tutorial, Class]
List after update: [BTECH, SMART, CLASS, :, JAVA, TUTORIAL, CLASS]
```

Removing Items

The ArrayList class has the following methods to remove items.

- **E remove(int index)** - Removes the element at specified index in the invoking ArrayList.
- **boolean remove(Object element)** - Removes the first occurrence of the given element from the invoking ArrayList.
- **boolean removeAll(Collection c)** - Removes the given collection of elements from the invoking ArrayList.
- **void retainAll(Collection c)** - Removes all the elements except the given collection of elements from the invoking ArrayList.
- **boolean removeIf(Predicate filter)** - Removes all the elements from the ArrayList that satisfies the given predicate.
- **void clear()** - Removes all the elements from the ArrayList.

Let's consider an example program to illustrate removing items from the ArrayList.

Example

```
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list_1 = new ArrayList<String>();
        ArrayList list_2 = new ArrayList();
        ArrayList list_3 = new ArrayList();

        list_1.add("BTech");
        list_1.add("Smart");
        list_1.add("Class");
        list_1.add("-");
        list_1.add("Java");
        list_1.add("Tutorial");
        list_1.add("Classes");
        list_1.add("on");
        list_1.add("Collection");
        list_1.add("framwork");
        list_1.add("-");
        list_1.add("ArrayList");

        list_2.add("Tutorial");
        list_2.add("Java");

        list_3.add("BTech");
```

```

list_3.add("Smart");
list_3.add("Class");
System.out.println("\nList_1 before remove:\n" + list_1);
System.out.println("\nList_2 before remove:\n" + list_2);
System.out.println("\nList_3 before remove:\n" + list_3);
list_1.remove(3);
System.out.println("\nList after removing element from index 3:\n" + list_1);
list_1.remove("Tutorial");
System.out.println("\nList after removing \"Tutorial\" element:\n" + list_1);
list_1.removeAll(list_2);
System.out.println("\nList after removing all elements of list_2 from list_1:\n" + list_1);
list_1.removeIf(n -> n.equals("Classes"));
System.out.println("\nList after removing all elements that are equal to \"Classes\":\n" + list_1);
list_1.retainAll(list_3);
System.out.println("\nList after removing all elements from list_1 except elements of list_3:\n" + list_1);
list_1.clear();
System.out.println("\nList after removing all elements from list_1:\n" + list_1);
}
}

```

The screenshot shows the Eclipse IDE interface with the file `ArrayListExample.java` open. The console window displays the following output:

```

List_1 before remove:
[BTech, Smart, Class, -, Java, Tutorial, Classes, on, Collection, framework, -, ArrayList]

List_2 before remove:
[Tutorial, Java]

List_3 before remove:
[BTech, Smart, Class]

List after removing element from index 3:
[BTech, Smart, Class, Java, Tutorial, Classes, on, Collection, framework, -, ArrayList]

List after removing 'Tutorial' element:
[BTech, Smart, Class, Java, Classes, on, Collection, framework, -, ArrayList]

List after removing all elements of list_2 from list_1:
[BTech, Smart, Class, Classes, on, Collection, framework, -, ArrayList]

List after removing all elements that are equal to 'Classes':
[BTech, Smart, Class, on, Collection, framework, -, ArrayList]

List after removing all elements from list_1 except elements of list_3:
[BTech, Smart, Class]

List after removing all elements from list_1:
[]

```

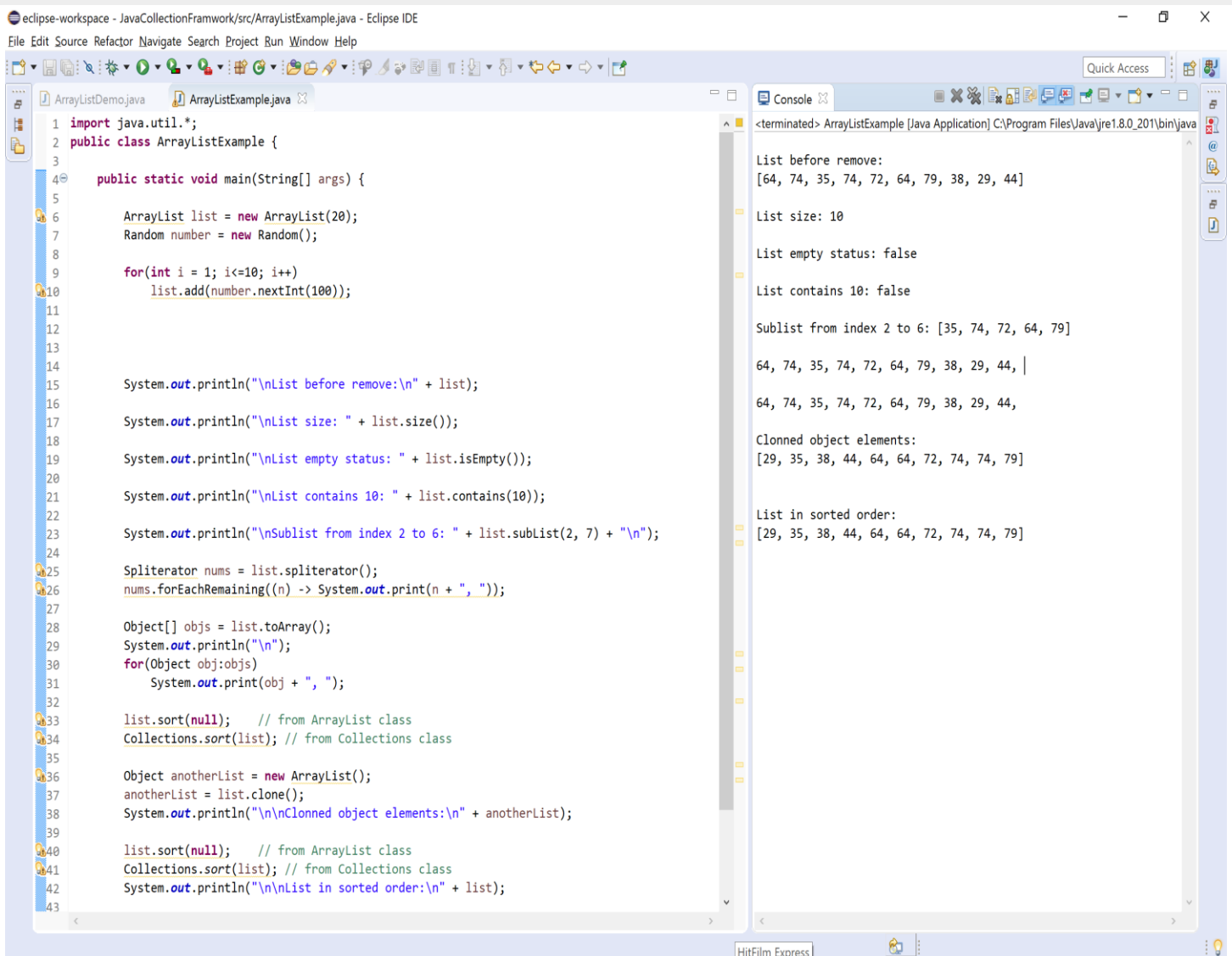
Other utility methods

The ArrayList class has the following methods to work with elements of it.

- **int size()** - Returns the total number of elements in the invoking ArrayList.
- **boolean isEmpty()** - Returns true if the list is empty otherwise returns false.
- **boolean contains(Object element)** - Returns true if the list contains given element otherwise returns false.
- **void sort(Comparator c)** - Sorts all the elements of invoking list based on the given comparator.
- **List[] subList(int startIndex, int endIndex)** - Returns list of elements starting from startIndex to endIndex-1.
- **Object clone()** - Returns a shallow copy of an ArrayList.
- **Object[] toArray()** - Returns an array of Object instances that contains all the elements from invoking ArrayList.
- **Splitterator spliterator()** - Creates spliterator over the elements in a list.
- **void trimToSize()** - Used to trim a ArrayList instance to the number of elements it contains.

Let's consider an example program to illustrate other utility methods of the ArrayList.

Example



```
eclipse-workspace - JavaCollectionFramework/src/ArrayListExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

ArrayListDemo.java ArrayListExample.java

1 import java.util.*;
2 public class ArrayListExample {
3
4     public static void main(String[] args) {
5
6         ArrayList list = new ArrayList(20);
7         Random number = new Random();
8
9         for(int i = 1; i<=10; i++)
10             list.add(number.nextInt(100));
11
12
13
14
15         System.out.println("\nList before remove:\n" + list);
16
17         System.out.println("\nList size: " + list.size());
18
19         System.out.println("\nList empty status: " + list.isEmpty());
20
21         System.out.println("\nList contains 10: " + list.contains(10));
22
23         System.out.println("\nSublist from index 2 to 6: " + list.subList(2, 7) + "\n");
24
25         Spliterator nums = list.spliterator();
26         nums.forEachRemaining((n) -> System.out.print(n + ", "));
27
28         Object[] objs = list.toArray();
29         System.out.println("\n");
30         for(Object obj:objs)
31             System.out.print(obj + ", ");
32
33         list.sort(null); // from ArrayList class
34         Collections.sort(list); // from Collections class
35
36         Object anotherList = new ArrayList();
37         anotherList = list.clone();
38         System.out.println("\n\nCloned object elements:\n" + anotherList);
39
40         list.sort(null); // from ArrayList class
41         Collections.sort(list); // from Collections class
42         System.out.println("\n\nList in sorted order:\n" + list);
43     }
44 }
```

Console Output:

```
<terminated> ArrayListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\java

List before remove:
[64, 74, 35, 74, 72, 64, 79, 38, 29, 44]

List size: 10

List empty status: false

List contains 10: false

Sublist from index 2 to 6: [35, 74, 72, 64, 79]

64, 74, 35, 74, 72, 64, 79, 38, 29, 44, |
64, 74, 35, 74, 72, 64, 79, 38, 29, 44,

Cloned object elements:
[29, 35, 38, 44, 64, 64, 72, 74, 74, 79]

List in sorted order:
[29, 35, 38, 44, 64, 64, 72, 74, 74, 79]
```

Consolidated list of methods

The following table provides a consolidated view of all methods of ArrayList.

Method	Description
boolean add(E element)	Appends given element to the ArrayList.
boolean addAll(Collection c)	Appends given collection of elements to the ArrayList.
void add(int index, E element)	Inserts the given element at specified index.
boolean addAll(int index, Collection c)	Inserts the given collection of elements at specified index.
E get(int index)	Returns element at specified index from the ArrayList.
ArrayList subList(int startIndex, int lastIndex)	Returns an ArrayList that contains elements from specified startIndex to lastIndex-1 from the invoking ArrayList.
int indexOf(E element)	Returns the index value of given element first occurrence in the ArrayList.
int lastIndexOf(E element)	Returns the index value of given element last occurrence in the ArrayList.
E set(int index, E newElement)	Replace the element at specified index with newElement in the invoking ArrayList.
ArrayList replaceAll(UnaryOperator e)	Replaces each element of invoking ArrayList with the result of applying the operator to that element.
E remove(int index)	Removes the element at specified index in the invoking ArrayList.
boolean remove(Object element)	Removes the first occurrence of the given element from the invoking ArrayList.

Method	Description
boolean removeAll(Collection c)	Removes the given collection of elements from the invoking ArrayList.
void retainAll(Collection c)	Removes all the elements except the given collection of elements from the invoking ArrayList.
boolean removeIf(Predicate filter)	Removes all the elements from the ArrayList that satisfies the given predicate.
void clear()	Removes all the elements from the ArrayList.
int size()	Returns the total number of elements in the invoking ArrayList.
boolean isEmpty()	Returns true if the list is empty otherwise returns false.
boolean contains(Object element)	Returns true if the list contains given element otherwise returns false.
void sort(Comparator c)	Sorts all the elements of invoking list based on the given comparator.
List< E > subList(int startIndex, int endIndex)	Returns list of elements starting from startIndex to endIndex-1.
Object clone()	Returns a shallow copy of an ArrayList.
Object[] toArray()	Returns an array of Object instances that contains all the elements from invoking ArrayList.
Splitter splitter()	Creates splitter over the elements in a list.
void trimToSize()	Used to trim a ArrayList instance to the number of elements it contains.

Java LinkedList Class

- The **LinkedList** class is a part of java collection framework.
- It is available inside the **java.util** package.
- The LinkedList class extends **AbstractSequentialList** class and implements **List** and **Deque** interface.
- The elements of LinkedList are organized as the elements of linked list data structure.
- The LinkedList class is used to create a dynamic list of elements that can grow or shrunk as needed.

- The LinkedList is a child class of **AbstractSequentialList**
- The LinkedList implements interfaces like **List**, **Deque**, **Cloneable**, and **Serializable**.
- The LinkedList allows to store duplicate data values.
- The LinkedList maintains the order of insertion.

LinkedList class declaration

The LinkedList class has the following declaration.

Example

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList class constructors

The LinkedList class has the following constructors.

- **LinkedList()** - Creates an empty List.
- **LinkedList(Collection c)** - Creates a List with given collection of elements.

Operations on LinkedList

The LinkedList class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The LinkedList class has the following methods to add items.

- **boolean add(E element)** - Appends given element to the List.
- **boolean addAll(Collection c)** - Appends given collection of elements to the List.
- **void add(int position, E element)** - Inserts the given element at specified position.
- **boolean addAll(int position, Collection c)** - Inserts the given collection of elements at specified position.
- **void addFirst(E element)** - Inserts the given element at beggining of the list.
- **void addLast(E element)** - Inserts the given element at end of the list.
- **boolean offer(E element)** - Inserts the given element at end of the list.
- **boolean offerFirst(E element)** - Inserts the given element at beggining of the list.

- **boolean offerLast(E element)** - Inserts the given element at end of the list.
- **void push(E element)** - Inserts the given element at beginning of the list.

Let's consider an example program to illustrate adding items to the LinkedList.

Example

```

1 import java.util.*;
2
3 public class LinkedListExample {
4
5     public static void main(String[] args) {
6
7         LinkedList<String> list_1 = new LinkedList<String>();
8         LinkedList list_2 = new LinkedList();
9
10        list_2.add(10);
11        list_2.add(20);
12        list_2.addFirst(5);
13        list_2.addLast(25);
14        list_2.offer(2);
15        list_2.offerFirst(1);
16        list_2.offerLast(10);
17        list_2.push(40);
18
19        list_1.addAll(list_2);
20
21        System.out.println("List_1: " + list_1);
22
23        System.out.println("List_2: " + list_2);
24    }
25 }
26
27 }
28

```

Console Output:

```

<terminated> LinkedListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
List_1: [40, 1, 5, 10, 20, 25, 2, 10]
List_2: [40, 1, 5, 10, 20, 25, 2, 10]

```

Accessing Items

The LinkedList class has the following methods to access items.

- **E get(int position)** - Returns element at specified position from the LinkedList.
- **E element()** - Returns the first element from the invoking LinkedList.
- **E getFirst()** - Returns the first element from the invoking LinkedList.
- **E getLast()** - Returns the last element from the invoking LinkedList.
- **E peek()** - Returns the first element from the invoking LinkedList.
- **E peekFirst()** - Returns the first element from the invoking LinkedList, and returns null if list is empty.
- **E peekLast()** - Returns the last element from the invoking LinkedList, and returns null if list is empty.
- **int indexOf(E element)** - Returns the index value of given element first occurrence in the LinkedList.
- **int lastIndexOf(E element)** - Returns the index value of given element last occurrence in the LinkedList.
- **E pop()** - Returns the first element from the invoking LinkedList.

Let's consider an example program to illustrate accessing items from the LinkedList.

Example

The screenshot shows the Eclipse IDE with a Java project named 'JavaCollectionFramework'. The file 'LinkedListExample.java' is open in the editor. The code defines a 'LinkedList' class with a 'main' method that creates a 'LinkedList' object, adds elements from 1 to 10, and then prints various details about the list. The console on the right shows the output of the program, which includes the list elements and the results of several method calls.

```
1 import java.util.*;
2
3 public class LinkedListExample {
4
5     public static void main(String[] args) {
6
7         LinkedList list_1 = new LinkedList();
8
9         for(int i = 1; i <= 10; i++)
10             list_1.add(i);
11
12         System.out.println("List is " + list_1 + "\n");
13
14         System.out.println("get(position) - " + list_1.get(3));
15         System.out.println("getFirst() - " + list_1.getFirst());
16         System.out.println("getLast() - " + list_1.getLast());
17         System.out.println("element() - " + list_1.element());
18         System.out.println("peek() - " + list_1.peek());
19         System.out.println("peekFirst() - " + list_1.peekFirst());
20         System.out.println("peekLast() - " + list_1.peekLast());
21         System.out.println("pop() - " + list_1.pop());
22         System.out.println("indexOf(element) - " + list_1.indexOf(5));
23         System.out.println("lastIndexOf(element) - " + list_1.lastIndexOf(5));
24
25     }
26 }
27
28
```

Console Output:

```
<terminated> LinkedListExample [Java Application] C:\Program Files\Java\jre1.8.0
List is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

get(position) - 4
getFirst() - 1
getLast() - 10
element() - 1
peek() - 1
peekFirst() - 1
peekLast() - 10
pop() - 1
indexOf(element) - 3
lastIndexOf(element) - 3
```

Updating Items

The LinkedList class has the following methods to update or change items.

- **E set(int index, E newElement)** - Replace the element at specified index with newElement in the invoking LinkedList.

Let's consider an example program to illustrate updating items in the LinkedList.

Example

The screenshot shows the Eclipse IDE with the same 'JavaCollectionFramework' project. The file 'LinkedListExample.java' is open, but the code has been modified to demonstrate the 'set' method. The 'main' method now includes a call to 'list_1.set(3, 50);' after the initial list is created. The console output shows the list after this update, where the element at index 3 has been replaced with 50.

```
1 import java.util.*;
2
3 public class LinkedListExample {
4
5     public static void main(String[] args) {
6
7         LinkedList list_1 = new LinkedList();
8
9         for(int i = 1; i <= 10; i++)
10             list_1.add(i);
11
12         System.out.println("List is " + list_1 + "\n");
13
14         list_1.set(3, 50);
15
16         System.out.println("List after update at index 3 is\n" + list_1 + "\n");
17
18     }
19 }
20
21
```

Console Output:

```
<terminated> LinkedListExample [Java Application] C:\Program Files\Java\jre1.8.0
List is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

List after update at index 3 is
[1, 2, 3, 50, 5, 6, 7, 8, 9, 10]
```

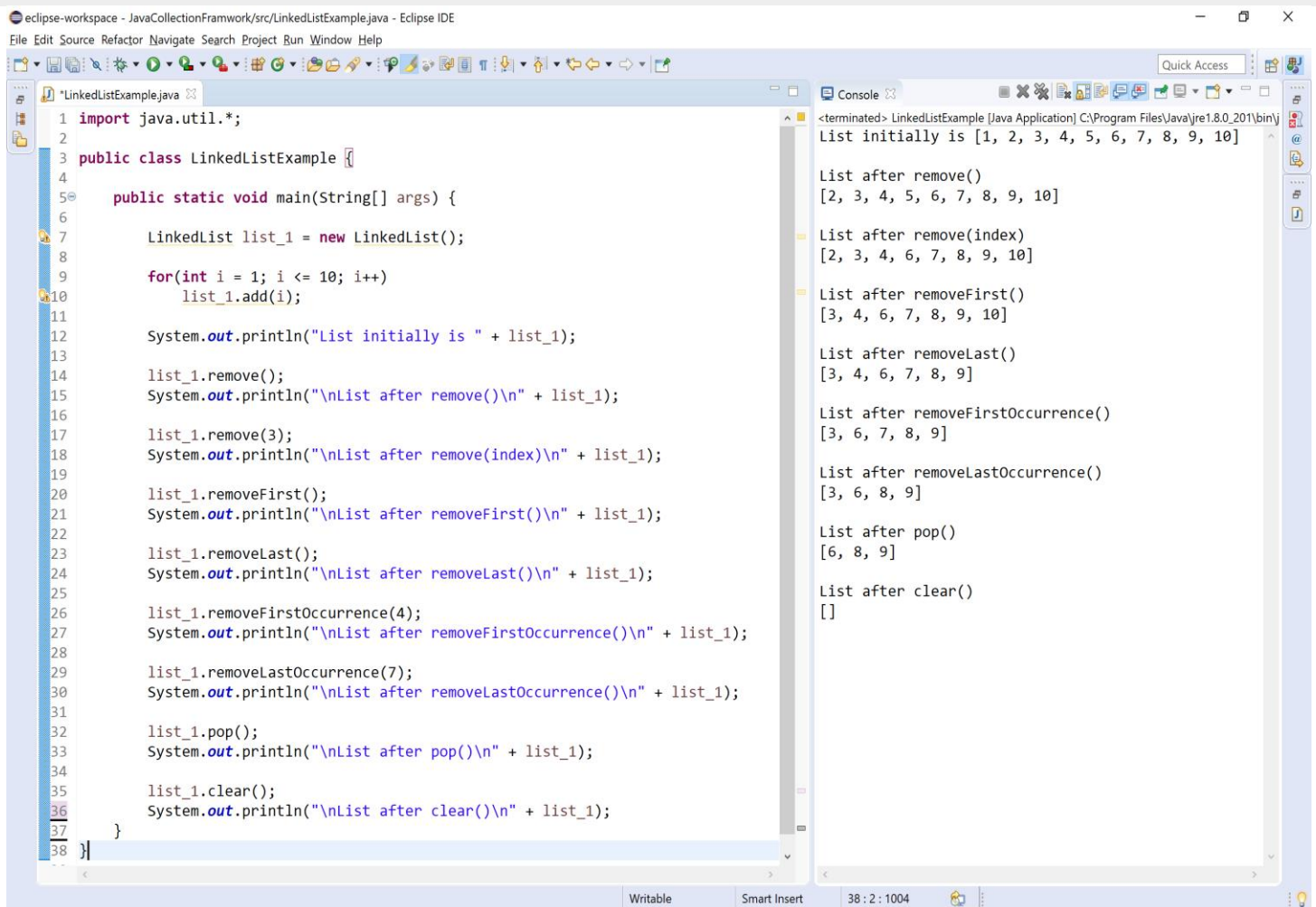
Removing Items

The LinkedList class has the following methods to remove items.

- **E remove()** - Removes the first element from the invoking LinkedList.
- **E remove(int index)** - Removes the element at specified index in the invoking LinkedList.
- **boolean remove(Object element)** - Removes the first occurrence of the given element from the invoking LinkedList.
- **E removeFirst()** - Removes the first element from the invoking LinkedList.
- **E removeLast()** - Removes the last element from the invoking LinkedList.
- **boolean removeFirstOccurrence(Object element)** - Removes from the first occurrence of the given element from the invoking LinkedList.
- **boolean removeLastOccurrence(Object element)** - Removes from the last occurrence of the given element from the invoking LinkedList.
- **E poll()** - Removes the first element from the LinkedList, and returns null if the list is empty.
- **E pollFirst()** - Removes the first element from the LinkedList, and returns null if the list is empty.
- **E pollLast()** - Removes the last element from the LinkedList, and returns null if the list is empty.
- **E pop()** - Removes the first element from the LinkedList.
- **void clear()** - Removes all the elements from the LinkedList.

Let's consider an example program to illustrate removing items from the LinkedList.

Example



```
1 import java.util.*;
2
3 public class LinkedListExample {
4
5     public static void main(String[] args) {
6
7         LinkedList list_1 = new LinkedList();
8
9         for(int i = 1; i <= 10; i++)
10             list_1.add(i);
11
12         System.out.println("List initially is " + list_1);
13
14         list_1.remove();
15         System.out.println("\nList after remove()\n" + list_1);
16
17         list_1.remove(3);
18         System.out.println("\nList after remove(index)\n" + list_1);
19
20         list_1.removeFirst();
21         System.out.println("\nList after removeFirst()\n" + list_1);
22
23         list_1.removeLast();
24         System.out.println("\nList after removeLast()\n" + list_1);
25
26         list_1.removeFirstOccurrence(4);
27         System.out.println("\nList after removeFirstOccurrence()\n" + list_1);
28
29         list_1.removeLastOccurrence(7);
30         System.out.println("\nList after removeLastOccurrence()\n" + list_1);
31
32         list_1.pop();
33         System.out.println("\nList after pop()\n" + list_1);
34
35         list_1.clear();
36         System.out.println("\nList after clear()\n" + list_1);
37     }
38 }
```

Console Output:

```
<terminated> LinkedListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\j
List initially is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

List after remove()
[2, 3, 4, 5, 6, 7, 8, 9, 10]

List after remove(index)
[2, 3, 4, 6, 7, 8, 9, 10]

List after removeFirst()
[3, 4, 6, 7, 8, 9, 10]

List after removeLast()
[3, 4, 6, 7, 8, 9]

List after removeFirstOccurrence()
[3, 6, 7, 8, 9]

List after removeLastOccurrence()
[3, 6, 8, 9]

List after pop()
[6, 8, 9]

List after clear()
[]
```

Other utility methods

The LinkedList class has the following methods to work with elements of it.

- **int size()** - Returns the total number of elements in the invoking LinkedList.
- **boolean isEmpty()** - Returns true if the list is empty otherwise returns false.
- **boolean contains(Object element)** - Returns true if the list contains given element otherwise returns false.
- **void sort(Comparator c)** - Sorts all the elements of invoking list based on the given comparator.
- **List[] subList(int startIndex, int endIndex)** - Returns list of elements starting from startIndex to endIndex-1.
- **Object clone()** - Returns a shallow copy of an LinkedList.
- **Object[] toArray()** - Returns an array of Object instances that contains all the elements from invoking LinkedList.
- **Spliterator spliterator()** - Creates spliterator over the elements in a list.
- **void trimToSize()** - Used to trim a LinkedList instance to the number of elements it contains.

Let's consider an example program to illustrate other utility methods of the LinkedList.

Example

```
1 import java.util.*;
2
3 public class LinkedListExample {
4
5     public static void main(String[] args) {
6
7         LinkedList list_1 = new LinkedList();
8
9         for(int i = 1; i <= 10; i++)
10             list_1.add(i);
11
12         System.out.println("List initially is " + list_1);
13
14         System.out.println("Size() - " + list_1.size());
15
16         System.out.println("isEmpty() - " + list_1.isEmpty());
17
18         System.out.println("contains(4) - " + list_1.contains(4));
19
20         System.out.println("subList(start, end-1) - " + list_1.subList(2, 6));
21     }
22 }
23
```

Console Output:

```
<terminated> LinkedListExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\j
List initially is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Size() - 10
isEmpty() - false
contains(4) - true
subList(start, end-1) - [3, 4, 5, 6]
```

Consolidated list of methods

The following table provides a consolidated view of all methods of LinkedList.

Method	Description
boolean add(E element)	Appends given element to the List.
boolean addAll(Collection c)	Appends given collection of elements to the List.
void add(int index, E element)	Inserts the given element at specified index.
boolean addAll(int index, Collection c)	Inserts the given collection of elements at specified index.
void addFirst(E element)	Inserts the given element at beginning of the list.
void addLast(E element)	Inserts the given element at end of the list.
boolean offer(E element)	Inserts the given element at end of the list.
boolean offerFirst(E element)	Inserts the given element at beginning of the list.
boolean offerLast(E element)	Inserts the given element at end of the list.
void push(E element)	Inserts the given element at beginning of the list.
E get(int index)	Returns element at specified index from the list.
E getFirst()	Returns the first element from the list.
E getLast()	Returns the last element from the list.
E element()	Returns the first element from the list.

Method	Description
E peek()	Returns the first element from the list.
E peekFirst()	Returns the first element from the invoking LinkedList, and returns null if list is empty.
E peekLast()	Returns the last element from the invoking LinkedList, and returns null if list is empty.
int indexOf(E element)	Returns the index value of given element first occurrence in the list.
int lastIndexOf(E element)	Returns the index value of given element last occurrence in the list.
E set(int index, E newElement)	Replace the element at specified index with newElement in the invoking list.
E remove()	Removes the first element from the invoking list.
E remove(int index)	Removes the element at specified index in the invoking list.
boolean remove(Object element)	Removes the first occurrence of the given element from the invoking list.
boolean removeAll(Collection c)	Removes the given collection of elements from the invoking LinkedList.
E removeFirst()	Removes the first element from the invoking list.
E removeLast()	Removes the first element from the invoking list.
E removeFirstOccurrence(Object element)	Removes from the first occurrence of the given element from the invoking LinkedList.
E removeLastOccurrence(Object element)	Removes from the last occurrence of the given element from the invoking list.

Method	Description
element)	LinkedList.
E poll()	Removes the first element from the LinkedList, and returns null if the list is empty.
E pollFirst()	Removes the first element from the LinkedList, and returns null if the list is empty.
E pollLast()	Removes the last element from the LinkedList, and returns null if the list is empty.
E pop()	Removes the first element from the LinkedList.
void clear()	Removes all the elements from the LinkedList.
int size()	Returns the total number of elements in the invoking LinkedList.
boolean isEmpty()	Returns true if the list is empty otherwise returns false.
boolean contains(Object element)	Returns true if the list contains given element otherwise returns false.
void sort(Comparator c)	Sorts all the elements of invoking list based on the given comparator.
List[] subList(int startIndex, int endIndex)	Returns list of elements starting from startIndex to endIndex-1.
Object clone()	Returns a shallow copy of an LinkedList.
Object[] toArray()	Returns an array of Object instances that contains all the elements from invoking LinkedList.

Method	Description
Spliterator spliterator()	Creates spliterator over the elements in a list.
void trimToSize()	Used to trim a LinkedList instance to the number of elements it contains.

Java HashSet Class

- The **HashSet** class is a part of java collection framework.
- It is available inside the **java.util** package.
- The HashSet class extends **AbstractSet** class and implements **Set** interface.
- The elements of HashSet are organized using a mechanism called hashing. The HashSet is used to create hash table for storing set of elements.
- The HashSet class is used to create a collection that uses a hash table for storing set of elements.

- ☐ The HashSet is a child class of **AbstractSet**
- ☐ The HashSet implements interfaces like **Set**, **Cloneable**, and **Serializable**.
- ☐ The HashSet does not allow to store duplicate data values, but null values are allowed.
- ☐ The HashSet does not maintain the order of insertion.
- ☐ The HashSet initial capacity is 16 elements.
- ☐ The HashSet is best suitable for search operations.

HashSet class declaration

The HashSet class has the following declaration.

Example

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable
```

HashSet class constructors

The HashSet class has the following constructors.

- **HashSet()** - Creates an empty HashSet with the default initial capacity (16).
- **HashSet(Collection c)** - Creates a HashSet with given collection of elements.
- **HashSet(int initialCapacity)** - Creates an empty HashSet with the specified initial capacity.
- **HashSet(int initialCapacity, float loadFactor)** - Creates an empty HashSet with the specified initial capacity and loadFactor.

Operations on HashSet

The HashSet class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

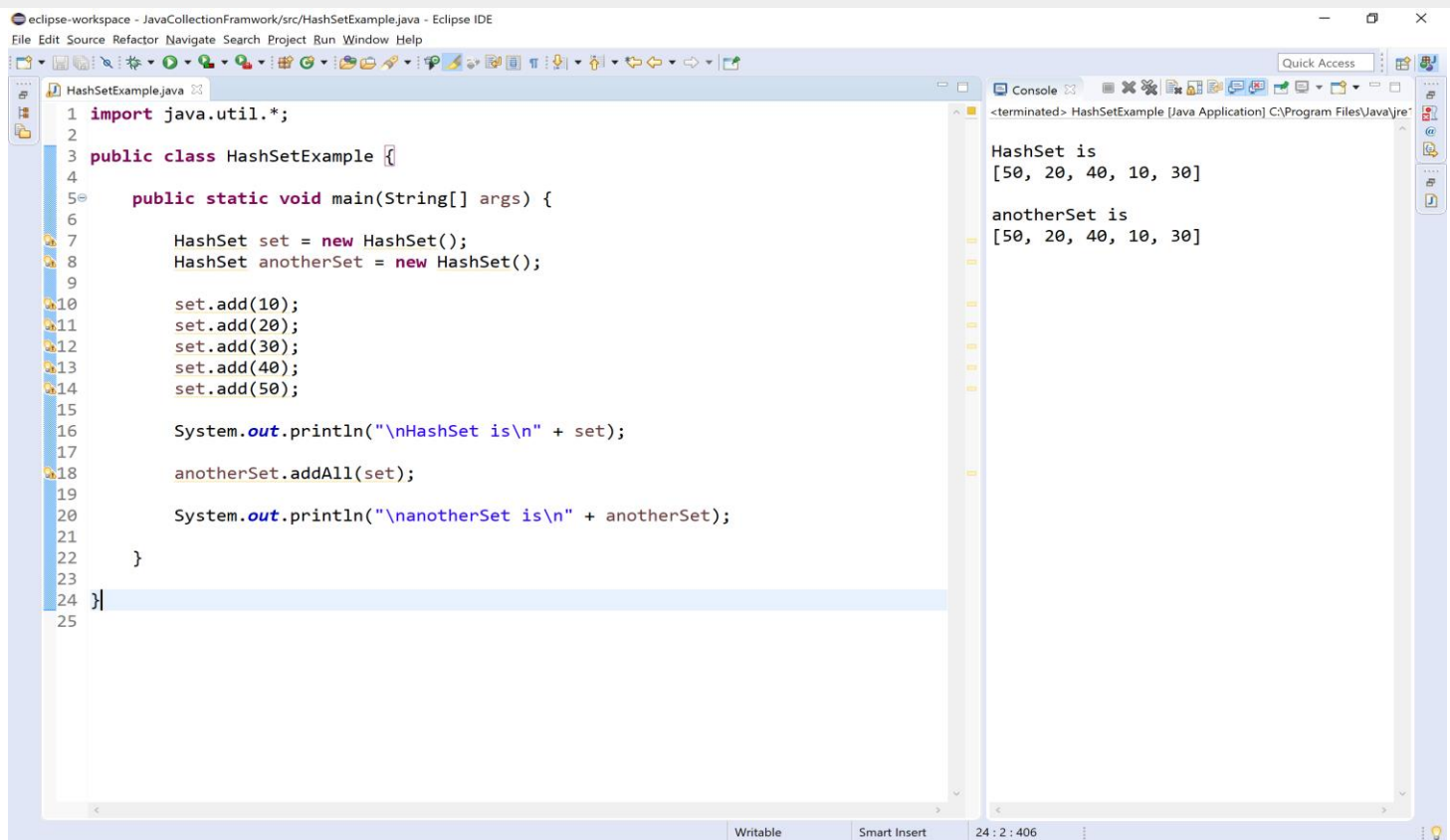
Adding Items

The HashSet class has the following methods to add items.

- **boolean add(E element)** - Inserts given element to the HashSet.
- **boolean addAll(Collection c)** - Inserts given collection of elements to the HashSet.

Let's consider an example program to illustrate adding items to the HashSet.

Example



```
eclipse-workspace - JavaCollectionFramework/src/HashSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashSetExample.java
1 import java.util.*;
2
3 public class HashSetExample {
4
5     public static void main(String[] args) {
6
7         HashSet set = new HashSet();
8         HashSet anotherSet = new HashSet();
9
10        set.add(10);
11        set.add(20);
12        set.add(30);
13        set.add(40);
14        set.add(50);
15
16        System.out.println("\nHashSet is\n" + set);
17
18        anotherSet.addAll(set);
19
20        System.out.println("\nanotherSet is\n" + anotherSet);
21
22    }
23
24 }
25

Console
<terminated> HashSetExample [Java Application] C:\Program Files\Java\jre-
HashSet is
[50, 20, 40, 10, 30]
anotherSet is
[50, 20, 40, 10, 30]
```

Accessing Items

The HashSet class has no methods to access items, we can access whole set using its name.

Updating Items

The HashSet class has no methods to update or change items.

Removing Items

The HashSet class has the following methods to remove items.

- **boolean remove(Object o)** - Removes the specified element from the invoking HashSet.
- **boolean removeAll(Collection c)** - Removes all the elements of specified collection from the invoking HashSet.
- **boolean removeIf(Predicate p)** - Removes all of the elements of HashSet collection that satisfy the given predicate.
- **boolean retainAll(Collection c)** - Removes all of the elements of HashSet collection except specified collection of elements.
- **void clear()** - Removes all the elements from the HashSet.

Let's consider an example program to illustrate removing items from the HashSet.

Example

```

eclipse-workspace - JavaCollectionFramework/src/HashSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashSetExample.java
1 import java.util.*;
2
3 public class HashSetExample {
4
5     public static void main(String[] args) {
6
7         HashSet set = new HashSet();
8         HashSet anotherSet = new HashSet();
9
10        set.add(10);
11        set.add(20);
12        set.add(30);
13        set.add(40);
14        set.add(50);
15
16        System.out.println("\nHashSet is\n" + set);
17
18        anotherSet.addAll(set);
19
20        System.out.println("\nanotherSet is\n" + anotherSet);
21
22        set.remove(20);
23        System.out.println("\nHashSet after remove(20) is\n" + set);
24
25        anotherSet.removeAll(set);
26        System.out.println("\nanotherSet after removeAll(set) is\n" + anotherSet);
27
28        set.retainAll(anotherSet);
29        System.out.println("\nset after retainAll(anotherSet) is\n" + set);
30
31        anotherSet.clear();
32        System.out.println("\nanotherSet after clear() is\n" + anotherSet);
33
34    }
35
36 }
37

Console
<terminated> HashSetExample [Java Application] C:\Program Files\Java\
HashSet is
[50, 20, 40, 10, 30]
anotherSet is
[50, 20, 40, 10, 30]
HashSet after remove(20) is
[50, 40, 10, 30]
anotherSet after removeAll(set) is
[20]
set after retainAll(anotherSet) is
[]
anotherSet after clear() is
[]

```

Other utility methods

The HashSet class has the following methods to work with elements of it.

- **int size()** - Returns the total number of elements in the invoking HashSet.
- **boolean isEmpty()** - Returns true if the HashSet is empty otherwise returns false.
- **HashSet clone()** - Returns a copy of the invoking HashSet.
- **boolean contains(Object element)** - Returns true if the HashSet contains given element otherwise returns false.
- **boolean containsAll(Collection c)** - Returns true if the HashSet contains given collection of elements otherwise returns false.
- **boolean equals(Object o)** - Compares the specified object with invoking HashSet collection for equality.
- **int hashCode()** - Returns the hash code of the invoking HashSet.

- **Object[] toArray()** - Returns an array of Object instances that contains all the elements from invoking HashSet.
- **Splitter splitter()** - Creates splitter over the elements in a HashSet.
- **Iterator iterator()** - Returns an iterator over the elements in the HashSet. The iterator does not return the elements in any particular order.

Let's consider an example program to illustrate other utility methods of the HashSet.

Example

```

eclipse-workspace - JavaCollectionFramework/src/HashSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashSetExample.java
1 import java.util.*;
2
3 public class HashSetExample {
4
5     public static void main(String[] args) {
6
7         HashSet set = new HashSet();
8
9         set.add(10);
10        set.add(20);
11        set.add(30);
12        set.add(40);
13        set.add(50);
14
15        System.out.println("\nHashSet is\n" + set);
16
17        System.out.println("\nsize() - " + set.size());
18
19        System.out.println("\nisEmpty() - " + set.isEmpty());
20
21        System.out.println("\ncontains(30) - " + set.contains(30));
22
23        System.out.println("\nequals(30) - " + set.equals(30));
24
25        System.out.println("\nhashCode() - " + set.hashCode());
26
27    }
28
29 }
30

Console
<terminated> HashSetExample [Java Application] C:\Program Files\Java\
HashSet is
[50, 20, 40, 10, 30]
size() - 5
isEmpty() - false
contains(30) - true
equals(30) - false
hashCode() - 150

Writable Smart Insert 29 : 2 : 586
  
```

Consolidated list of methods

The following table provides a consolidated view of all methods of HashSet.

Method	Description
boolean add(E element)	Appends given element to the HashSet.
boolean addAll(Collection c)	Appends given collection of elements to the HashSet.

Method	Description
boolean remove(Object element)	Removes the first occurrence of the given element from the invoking HashSet.
boolean removeAll(Collection c)	Removes all the elements those are in the specified collection from the invoking HashSet.
boolean removeIf(Predicate p)	Removes all of the elements of the HashSet collection that satisfy the given predicate.
boolean retainAll(Collection c)	Removes all the elements except those are in the specified collection from the invoking HashSet.
void clear()	Removes all the elements from the HashSet.
int size()	Returns the total number of elements in the invoking HashSet.
boolean isEmpty()	Returns true if the HashSet is empty otherwise returns false.
boolean equals()	Compares the specified object with invoking HashSet collection for equality.
boolean contains(Object element)	Returns true if the HashSet contains given element otherwise returns false.
boolean containsAll(Collection c)	Returns true if the HashSet contains all elements of given collection otherwise returns false.
int clone()	Returns a copy of the invoking HashSet.
int hashCode()	Returns the hash code of the invoking HashSet.
Object[] toArray()	Returns an array of Object instances that contains all the elements from

Method	Description
	invoking HashSet.
Splititerator spliterator()	Creates spliterator over the elements in a HashSet.
Iterator iterator()	Returns an iterator over the elements in the HashSet. The iterator does not return the elements in any particular order.

Java TreeSet Class

- The **TreeSet** class is a part of java collection framework.
- It is available inside the **java.util** package.
- The TreeSet class extends **AbstractSet** class and implements **NavigableSet**, **Cloneable**, and **Serializable** interfaces.
- The elements of TreeSet are organized using a mechanism called tree.
- The TreeSet class internally uses a TreeMap to store elements.
- The elements in a TreeSet are sorted according to their natural ordering.

- ❑ The TreeSet is a child class of **AbstractSet**
- ❑ The TreeSet implements interfaces like **NavigableSet**, **Cloneable**, and **Serializable**.
- ❑ The TreeSet does not allow to store duplicate data values, but null values are allowed.
- ❑ The elements in a TreeSet are sorted according to their natural ordering.
- ❑ The TreeSet initial capacity is 16 elements.
- ❑ The TreeSet is best suitable for search operations.

TreeSet class declaration

The TreeSet class has the following declaration.

Examplesorting order

```
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable
```

TreeSet class constructors

The TreeSet class has the following constructors.

- **TreeSet()** - Creates an empty TreeSet in which elements will get stored in default natural sorting order.
- **TreeSet(Collection c)** - Creates a TreeSet with given collection of elements.
- **TreeSet(Comparator c)** - Creates an empty TreeSet with the specified sorting order.
- **TreeSet(SortedSet s)** - This constructor is used to convert SortedSet to TreeSet.

Operations on TreeSet

The TreeSet class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The TreeSet class has the following methods to add items.

- **boolean add(E element)** - Inserts given element to the TreeSet if it does not exist.
- **boolean addAll(Collection c)** - Inserts given collection of elements to the TreeSet.

Let's consider an example program to illustrate adding items to the TreeSet.

Example

```
eclipse-workspace - JavaCollectionFramework/src/TreeSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashSetExample.java *TreeSetExample.java
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         TreeSet set = new TreeSet();
8         TreeSet anotherSet = new TreeSet();
9
10        set.add(10);
11        set.add(20);
12        set.add(15);
13        set.add(5);
14
15        System.out.println("\nset is\n" + set);
16
17        anotherSet.addAll(set);
18
19        System.out.println("\nanotherSet is\n" + anotherSet);
20    }
21 }
22
23
24

Console
<terminated> TreeSetExample [Java Application] C:\Program Files\Java
set is
[5, 10, 15, 20]
anotherSet is
[5, 10, 15, 20]

Writable Smart Insert 24 : 1 : 391
```

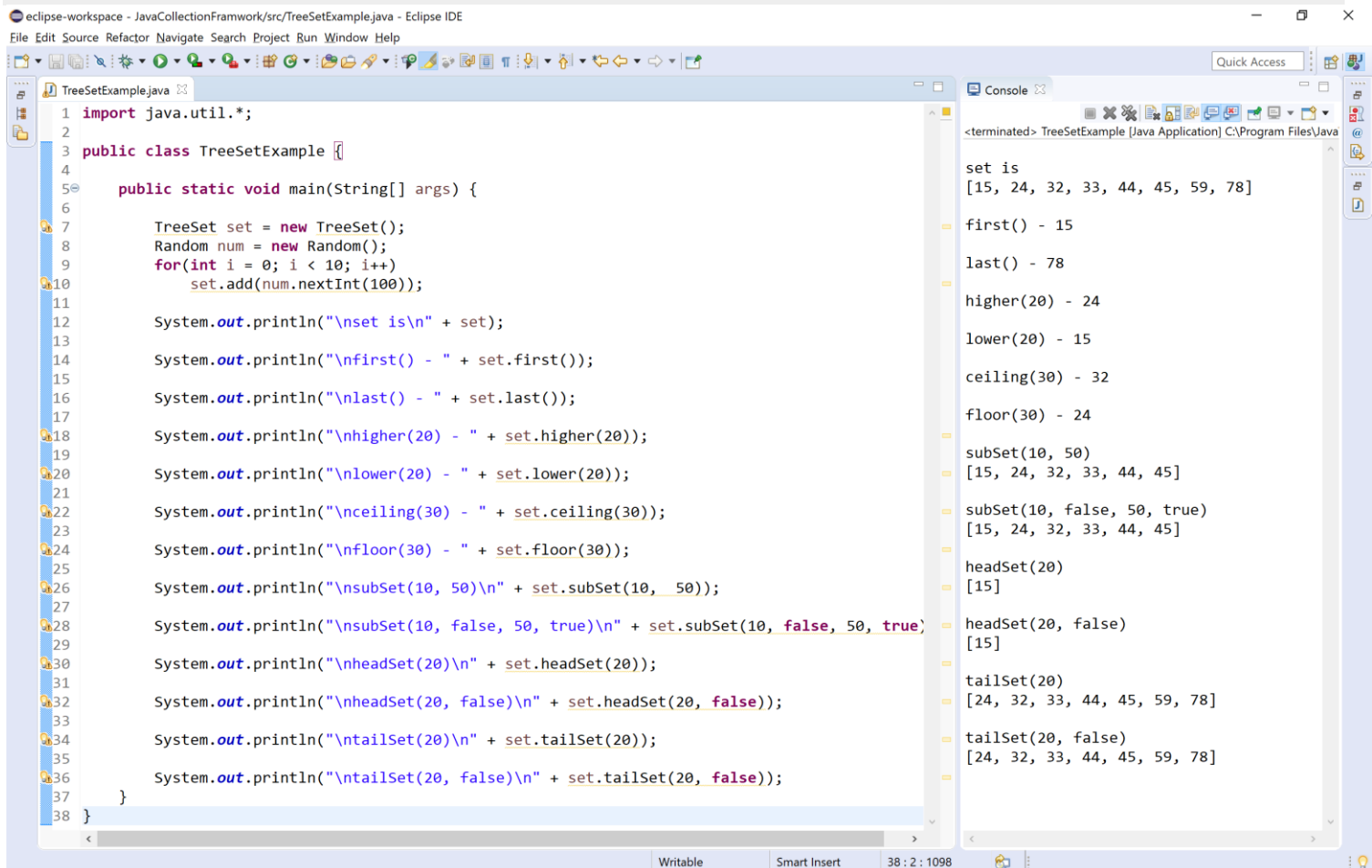

Accessing Items

The TreeSet class has provides the following methods to access items.

- **E First()** - Returns the first (smallest) element from the invoking TreeSet.
- **E last()** - Returns the last (largest) element from the invoking TreeSet.
- **E higher(E obj)** - Returns the largest element e such that $e > \text{obj}$. If it does not found returns null.
- **E lower(E obj)** - Returns the largest element e such that $e < \text{obj}$. If it does not found returns null.
- **E ceiling(E obj)** - Returns the smallest element e such that $e \geq \text{obj}$. If it does not found returns null.
- **E floor(E obj)** - Returns the largest element e such that $e \leq \text{obj}$. If it does not found returns null.
- **SortedSet subSet(E fromElement, E toElement)** - Returns a set of elements that lie between the given range which includes fromElement and excludes toElement.
- **NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)** - Returns a set of elements that lie between the given range from the invoking TreeSet.
- **SortedSet tailSet(E fromElement)** - Returns a set of elements that are greater than or equal to the specified fromElement from the invoking TreeSet.
- **NavigableSet tailSet(E fromElement, boolean inclusive)** - Returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element from the invoking TreeSet.
- **SortedSet headSet(E fromElement)** - Returns a set of elements that are smaller than or equal to the specified fromElement from the invoking TreeSet.
- **NavigableSet headSet(E fromElement, boolean inclusive)** - Returns a set of elements that are smaller than or equal to (if, inclusive is true) the specified element from the invoking TreeSet.

Let's consider an example program to illustrate accessing items from a TreeSet.

Example



```
eclipse-workspace - JavaCollectionFramework/src/TreeSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

TreeSetExample.java
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         TreeSet set = new TreeSet();
8         Random num = new Random();
9         for(int i = 0; i < 10; i++)
10             set.add(num.nextInt(100));
11
12         System.out.println("\nset is\n" + set);
13
14         System.out.println("\nfirst() - " + set.first());
15
16         System.out.println("\nlast() - " + set.last());
17
18         System.out.println("\nhigher(20) - " + set.higher(20));
19
20         System.out.println("\nlower(20) - " + set.lower(20));
21
22         System.out.println("\nceiling(30) - " + set.ceiling(30));
23
24         System.out.println("\nfloor(30) - " + set.floor(30));
25
26         System.out.println("\nsubSet(10, 50)\n" + set.subSet(10, 50));
27
28         System.out.println("\nsubSet(10, false, 50, true)\n" + set.subSet(10, false, 50, true));
29
30         System.out.println("\nheadSet(20)\n" + set.headSet(20));
31
32         System.out.println("\nheadSet(20, false)\n" + set.headSet(20, false));
33
34         System.out.println("\ntailSet(20)\n" + set.tailSet(20));
35
36         System.out.println("\ntailSet(20, false)\n" + set.tailSet(20, false));
37     }
38 }
```

```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java
set is
[15, 24, 32, 33, 44, 45, 59, 78]
first() - 15
last() - 78
higher(20) - 24
lower(20) - 15
ceiling(30) - 32
floor(30) - 24
subSet(10, 50)
[15, 24, 32, 33, 44, 45]
subSet(10, false, 50, true)
[15, 24, 32, 33, 44, 45]
headSet(20)
[15]
headSet(20, false)
[15]
tailSet(20)
[24, 32, 33, 44, 45, 59, 78]
tailSet(20, false)
[24, 32, 33, 44, 45, 59, 78]
```

Writable Smart Insert 38 : 2 : 1098

Updating Items

The TreeSet class has no methods to update or change items.

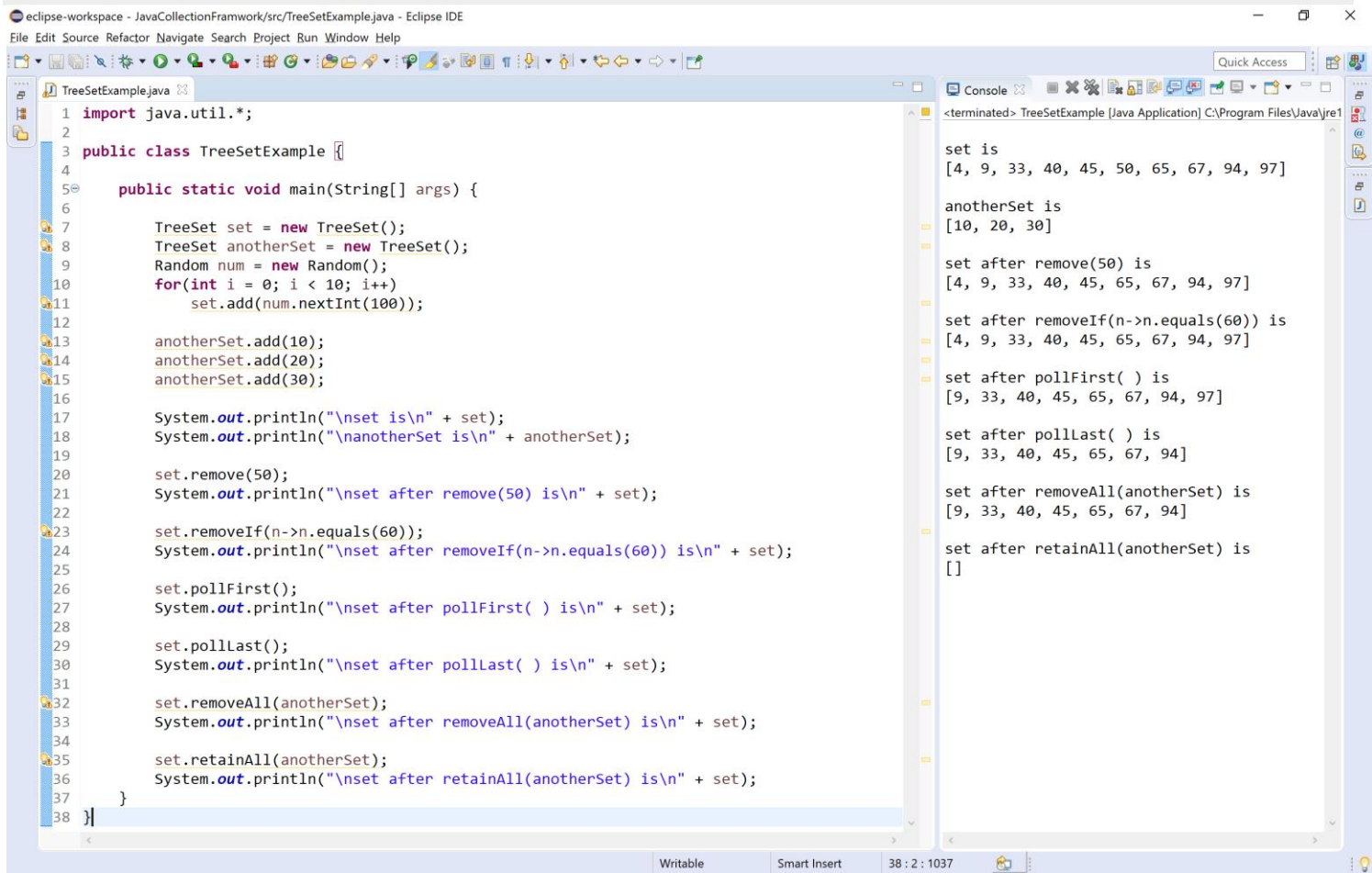
Removing Items

The TreeSet class has the following methods to remove items.

- **boolean remove(Object o)** - Removes the specified element from the invoking TreeSet.
- **boolean removeAll(Collection c)** - Removes all the elements those are in the specified collection from the invoking TreeSet.
- **boolean removeIf(Predicate p)** - Removes all of the elements of the TreeSet collection that satisfy the given predicate.
- **boolean retainAll(Collection c)** - Removes all the elements except those are in the specified collection from the invoking TreeSet.
- **E pollFirst()** - Removes the first (smallest) element from the invoking TreeSet, and returns the same.
- **E pollLast()** - Removes the last (largest) element from the invoking TreeSet, and returns the same.
- **void clear()** - Removes all the elements from the TreeSet.

Let's consider an example program to illustrate removing items from the TreeSet.

Example



```
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         TreeSet set = new TreeSet();
8         TreeSet anotherSet = new TreeSet();
9         Random num = new Random();
10        for(int i = 0; i < 10; i++)
11            set.add(num.nextInt(100));
12
13        anotherSet.add(10);
14        anotherSet.add(20);
15        anotherSet.add(30);
16
17        System.out.println("\nset is\n" + set);
18        System.out.println("\nanotherSet is\n" + anotherSet);
19
20        set.remove(50);
21        System.out.println("\nset after remove(50) is\n" + set);
22
23        set.removeIf(n->n.equals(60));
24        System.out.println("\nset after removeIf(n->n.equals(60)) is\n" + set);
25
26        set.pollFirst();
27        System.out.println("\nset after pollFirst( ) is\n" + set);
28
29        set.pollLast();
30        System.out.println("\nset after pollLast( ) is\n" + set);
31
32        set.removeAll(anotherSet);
33        System.out.println("\nset after removeAll(anotherSet) is\n" + set);
34
35        set.retainAll(anotherSet);
36        System.out.println("\nset after retainAll(anotherSet) is\n" + set);
37    }
38 }
```

Console Output:

```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jre1
set is
[4, 9, 33, 40, 45, 50, 65, 67, 94, 97]
anotherSet is
[10, 20, 30]
set after remove(50) is
[4, 9, 33, 40, 45, 65, 67, 94, 97]
set after removeIf(n->n.equals(60)) is
[4, 9, 33, 40, 45, 65, 67, 94, 97]
set after pollFirst( ) is
[9, 33, 40, 45, 65, 67, 94, 97]
set after pollLast( ) is
[9, 33, 40, 45, 65, 67, 94]
set after removeAll(anotherSet) is
[9, 33, 40, 45, 65, 67, 94]
set after retainAll(anotherSet) is
[]
```

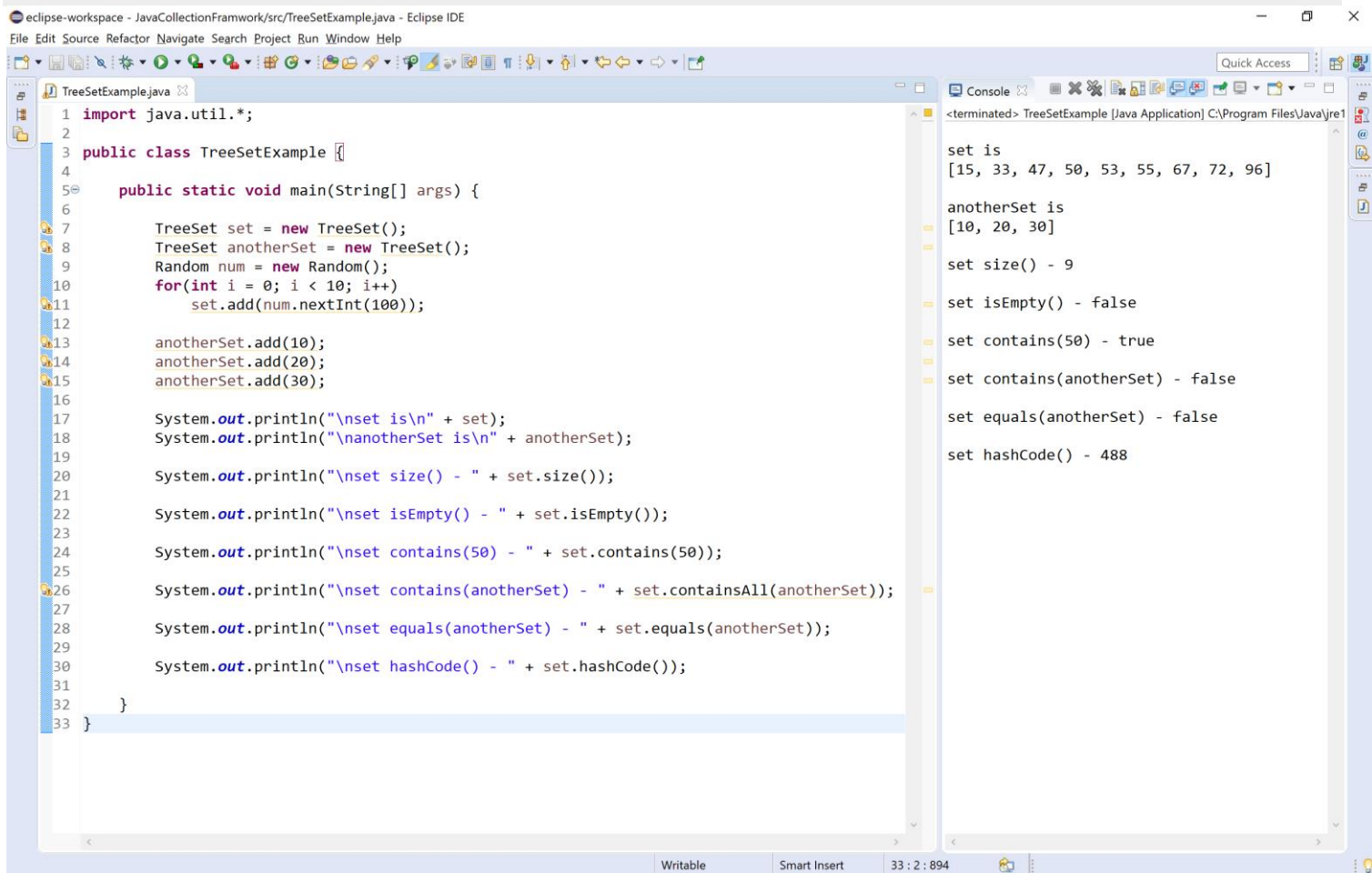

Other utility methods

The TreeSet class has the following methods to work with elements of it.

- **int size()** - Returns the total number of elements in the invoking TreeSet.
- **boolean isEmpty()** - Returns true if the TreeSet is empty otherwise returns false.
- **TreeSet clone()** - Returns a copy of the invoking TreeSet.
- **boolean contains(Object element)** - Returns true if the TreeSet contains given element otherwise returns false.
- **boolean containsAll(Collection c)** - Returns true if the TreeSet contains given collection of elements otherwise returns false.
- **boolean equals(Object o)** - Compares the specified object with invoking TreeSet collection for equality.
- **int hashCode()** - Returns the hash code of the invoking TreeSet.
- **Object clone()** - Returns a shallow copy of invoking TreeSet instance.
- **Spliterator spliterator()** - Creates spliterator over the elements in a TreeSet.
- **Iterator iterator()** - Returns an iterator over the elements in the TreeSet. The iterator does not return the elements in any particular order.

Let's consider an example program to illustrate other utility methods of the TreeSet.

Example



```
eclipse-workspace - JavaCollectionFramework/src/TreeSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

TreeSetExample.java
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         TreeSet set = new TreeSet();
8         TreeSet anotherSet = new TreeSet();
9         Random num = new Random();
10        for(int i = 0; i < 10; i++)
11            set.add(num.nextInt(100));
12
13        anotherSet.add(10);
14        anotherSet.add(20);
15        anotherSet.add(30);
16
17        System.out.println("\nset is\n" + set);
18        System.out.println("\nanotherSet is\n" + anotherSet);
19
20        System.out.println("\nset size() - " + set.size());
21
22        System.out.println("\nset isEmpty() - " + set.isEmpty());
23
24        System.out.println("\nset contains(50) - " + set.contains(50));
25
26        System.out.println("\nset contains(anotherSet) - " + set.containsAll(anotherSet));
27
28        System.out.println("\nset equals(anotherSet) - " + set.equals(anotherSet));
29
30        System.out.println("\nset hashCode() - " + set.hashCode());
31
32    }
33 }
```

```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jre1
set is
[15, 33, 47, 50, 53, 55, 67, 72, 96]
anotherSet is
[10, 20, 30]
set size() - 9
set isEmpty() - false
set contains(50) - true
set contains(anotherSet) - false
set equals(anotherSet) - false
set hashCode() - 488
```

Writable Smart Insert 33 : 2 : 894

Consolidated list of methods

The following table provides a consolidated view of all methods of TreeSet.

Method	Description
boolean add(E element)	Appends given element to the TreeSet.
boolean addAll(Collection c)	Appends given collection of elements to the TreeSet.
E First()	Returns the first (smallest) element from the invoking TreeSet.
E last()	Returns the last (largest) element from the invoking TreeSet.
E higher(E obj)	Returns the largest element e such that $e > \text{obj}$. If it does not found returns null.
E lower(E obj)	Returns the largest element e such that $e < \text{obj}$. If it does not found returns null.
E ceiling(E obj)	Returns the smallest element e such that $e \geq \text{obj}$. If it does not found returns null.
E floor(E obj)	Returns the largest element e such that $e \leq \text{obj}$. If it does not found returns null.
SortedSet subSet(E fromElement, E toElement)	Returns a set of elements that lie between the given range which includes fromElement and excludes toElement.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	Returns a set of elements that lie between the given range from the invoking TreeSet.
SortedSet tailSet(E fromElement)	Returns a set of elements that are greater than or equal to the specified fromElement from the

Method	Description
	invoking TreeSet.
NavigableSet tailSet(E fromElement, boolean inclusive)	Returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element from the invoking TreeSet.
SortedSet headSet(E fromElement)	Returns a set of elements that are smaller than or equal to the specified fromElement from the invoking TreeSet.
NavigableSet headSet(E fromElement, boolean inclusive)	Returns a set of elements that are smaller than or equal to (if, inclusive is true) the specified element from the invoking TreeSet.
boolean remove(Object element)	Removes the first occurrence of the given element from the invoking TreeSet.
boolean removeAll(Collection c)	Removes all the elements those are in the specified collection from the invoking TreeSet.
boolean removeIf(Predicate p)	Removes all of the elements of the TreeSet collection that satisfy the given predicate.
boolean retainAll(Collection c)	Removes all the elements except those are in the specified collection from the invoking TreeSet.
void clear()	Removes all the elements from the TreeSet.
int size()	Returns the total number of elements in the invoking TreeSet.
boolean isEmpty()	Returns true if the TreeSet is empty otherwise returns false.

Method	Description
boolean equals()	Compares the specified object with invoking TreeSet collection for equality.
boolean contains(Object element)	Returns true if the HashSet contains given element otherwise returns false.
boolean containsAll(Collection c)	Returns true if the TreeSet contains all elements of given collection otherwise returns false.
int clone()	Returns a copy of the invoking TreeSet.
int hashCode()	Returns the hash code of the invoking TreeSet.
Spliterator spliterator()	Creates spliterator over the elements in a TreeSet.
Iterator iterator()	Returns an iterator over the elements in the TreeSet. The iterator does not return the elements in any particular order.

Java PriorityQueue Class

- The **PriorityQueue** class is a part of java collection framework. It is available inside the **java.util** package.
- The PriorityQueue class extends **AbstractQueue** class and implements **Serializable** interface.
- The elements of PriorityQueue are organized as the elements of queue data structure, but it does not follow FIFO principle.
- The PriorityQueue elements are organized based on the priority heap. The PriorityQueue class is used to create a dynamic queue of elements that can grow or shrunk as needed.

- ❑ The PriorityQueue is a child class of **AbstractQueue**
- ❑ The PriorityQueue implements interface **Serializable**.
- ❑ The PriorityQueue allows to store duplicate data values, but not null values.
- ❑ The PriorityQueue maintains the order of insertion.
- ❑ The PriorityQueue used priority heap to organize its elements.

PriorityQueue class declaration

The PriorityQueue class has the following declaration.

Example

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

PriorityQueue class constructors

The PriorityQueue class has the following constructors.

- **PriorityQueue()** - Creates an empty PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
- **PriorityQueue(Collection c)** - Creates a PriorityQueue with given collection of elements.
- **PriorityQueue(int initialCapacity)** - Creates an empty PriorityQueue with the specified initial capacity.
- **PriorityQueue(int initialCapacity, Comparator comparator)** - Creates an empty PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
- **PriorityQueue(PriorityQueue pq)** - Creates a PriorityQueue with the elements in the specified priority queue.
- **PriorityQueue(SortedSet ss)** - Creates a PriorityQueue with the elements in the specified SortedSet.

Operations on PriorityQueue

The PriorityQueue class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

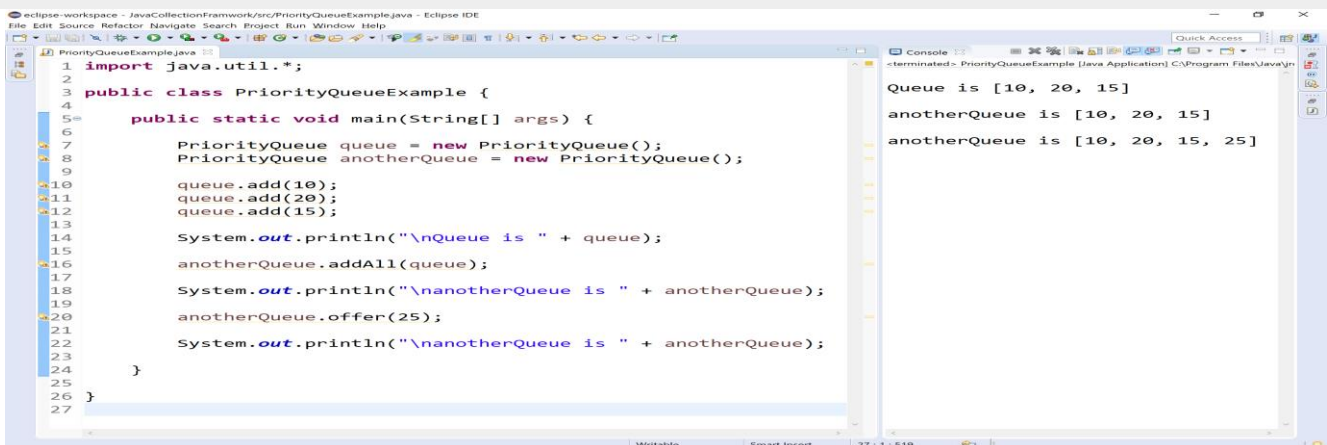
Adding Items

The PriorityQueue class has the following methods to add items.

- **boolean add(E element)** - Appends given element to the PriorityQueue.
- **boolean addAll(Collection c)** - Appends given collection of elements to the PriorityQueue.
- **boolean offer(E element)** - Appends given element to the PriorityQueue.

Let's consider an example program to illustrate adding items to the PriorityQueue.

Example



```
1 import java.util.*;
2
3 public class PriorityQueueExample {
4
5     public static void main(String[] args) {
6
7         PriorityQueue queue = new PriorityQueue();
8         PriorityQueue anotherQueue = new PriorityQueue();
9
10        queue.add(10);
11        queue.add(20);
12        queue.add(15);
13
14        System.out.println("\nQueue is " + queue);
15
16        anotherQueue.addAll(queue);
17
18        System.out.println("\nanotherQueue is " + anotherQueue);
19
20        anotherQueue.offer(25);
21        System.out.println("\nanotherQueue is " + anotherQueue);
22
23    }
24
25 }
26
27
```

Queue is [10, 20, 15]
anotherQueue is [10, 20, 15]
anotherQueue is [10, 20, 15, 25]

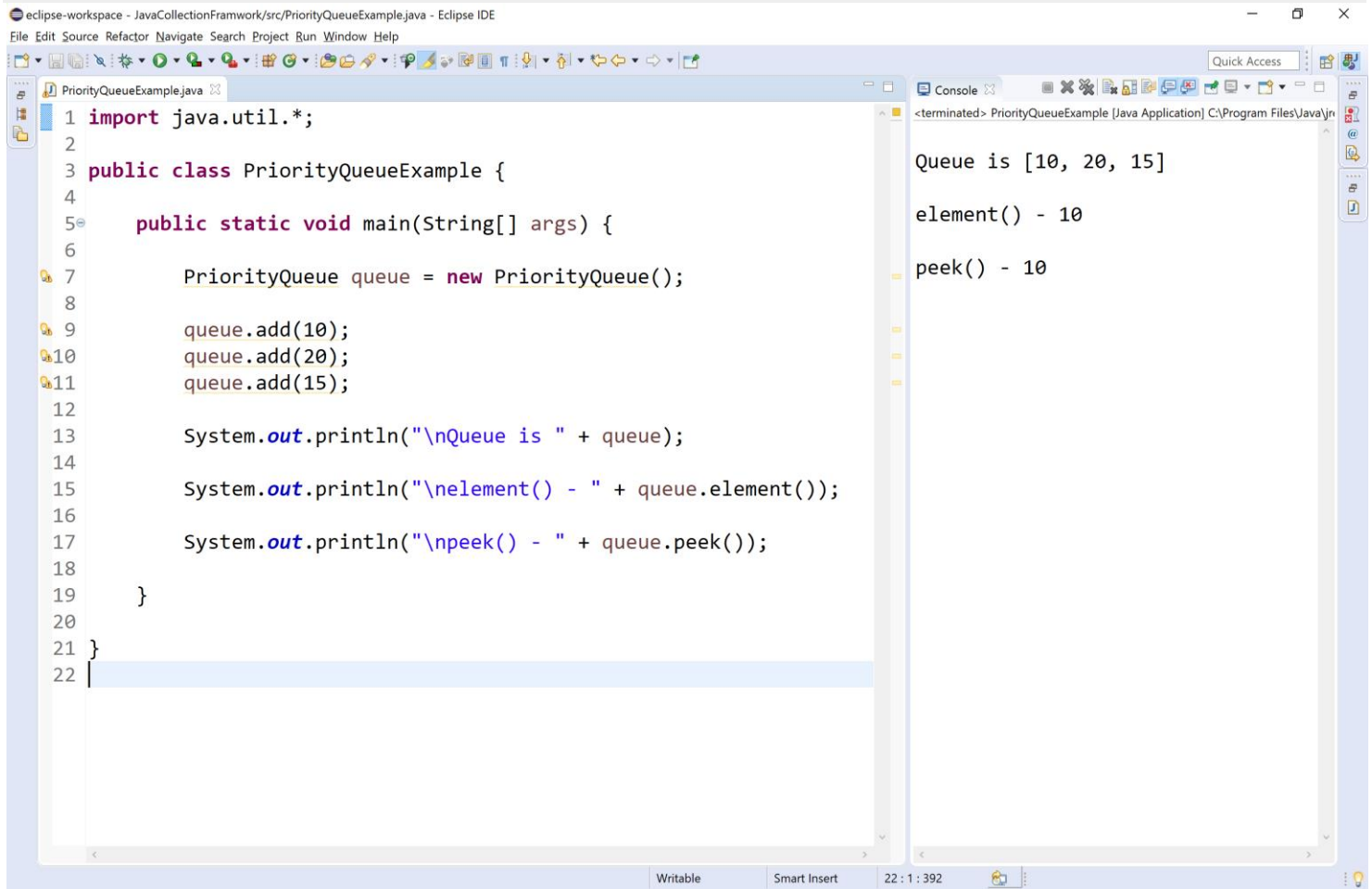
Accessing Items

The PriorityQueue class has the following methods to access items.

- **E element()** - Returns the first element from the invoking PriorityQueue.
- **E peek()** - Returns the first element from the invoking PriorityQueue, returns null if this queue is empty.

Let's consider an example program to illustrate accessing items from the PriorityQueue.

Example



```
eclipse-workspace - JavaCollectionFramework/src/PriorityQueueExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

PriorityQueueExample.java
1 import java.util.*;
2
3 public class PriorityQueueExample {
4
5     public static void main(String[] args) {
6
7         PriorityQueue queue = new PriorityQueue();
8
9         queue.add(10);
10        queue.add(20);
11        queue.add(15);
12
13        System.out.println("\nQueue is " + queue);
14
15        System.out.println("\nelement() - " + queue.element());
16
17        System.out.println("\npeek() - " + queue.peek());
18
19    }
20
21 }
22
```

```
<terminated> PriorityQueueExample [Java Application] C:\Program Files\Java\j
Queue is [10, 20, 15]
element() - 10
peek() - 10
```

Updating Items

The PriorityQueue class has no methods to update or change items.

Removing Items

The PriorityQueue class has the following methods to remove items.

- **E remove()** - Removes the first element from the invoking PriorityQueue.
- **boolean remove(Object element)** - Removes the first occurrence of the given element from the invoking PriorityQueue.
- **boolean removeAll(Collection c)** - Removes all the elements of specified collection from the invoking PriorityQueue.

- **boolean removeIf(Predicate p)** - Removes all of the elements of this collection that satisfy the given predicate.
- **boolean retainAll(Collection c)** - Removes all the elements except those are in the specified collection from the invoking PriorityQueue.
- **E poll()** - Removes the first element from the PriorityQueue, and returns null if the list is empty.
- **void clear()** - Removes all the elements from the PriorityQueue.

Let's consider an example program to illustrate removing items from the PriorityQueue.

Example

```

1 import java.util.*;
2
3 public class PriorityQueueExample {
4
5     public static void main(String[] args) {
6
7         PriorityQueue queue = new PriorityQueue();
8         PriorityQueue anotherQueue = new PriorityQueue();
9
10        for(int i = 1; i <= 10; i++)
11            queue.add(i);
12        anotherQueue.add(5);
13        anotherQueue.add(7);
14        anotherQueue.add(8);
15        System.out.println("\nQueue initially is\n" + queue);
16
17        queue.remove();
18        System.out.println("\nQueue after remove() is\n" + queue);
19
20        queue.remove(8);
21        System.out.println("\nQueue after remove(8) is\n" + queue);
22
23        queue.poll();
24        System.out.println("\nQueue after poll() is\n" + queue);
25
26        queue.removeIf(n->n.equals(6));
27        System.out.println("\nQueue after removeIf(n->n.equals(6) is\n" + queue);
28
29        queue.retainAll(anotherQueue);
30        System.out.println("\nQueue after retainAll(anotherQueue) is\n" + queue);
31
32        queue.removeAll(anotherQueue);
33        System.out.println("\nQueue after removeAll(anotherQueue) is\n" + queue);
34
35        queue.clear();
36        System.out.println("\nQueue after clear() is\n" + queue);
37    }
38 }
  
```

Console Output:

```

Queue initially is
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Queue after remove() is
[2, 4, 3, 8, 5, 6, 7, 10, 9]

Queue after remove(8) is
[2, 4, 3, 9, 5, 6, 7, 10]

Queue after poll() is
[3, 4, 6, 9, 5, 10, 7]

Queue after removeIf(n->n.equals(6) is
[3, 4, 7, 9, 5, 10]

Queue after retainAll(anotherQueue) is
[5, 7]

Queue after removeAll(anotherQueue) is
[]

Queue after clear() is
[]
  
```

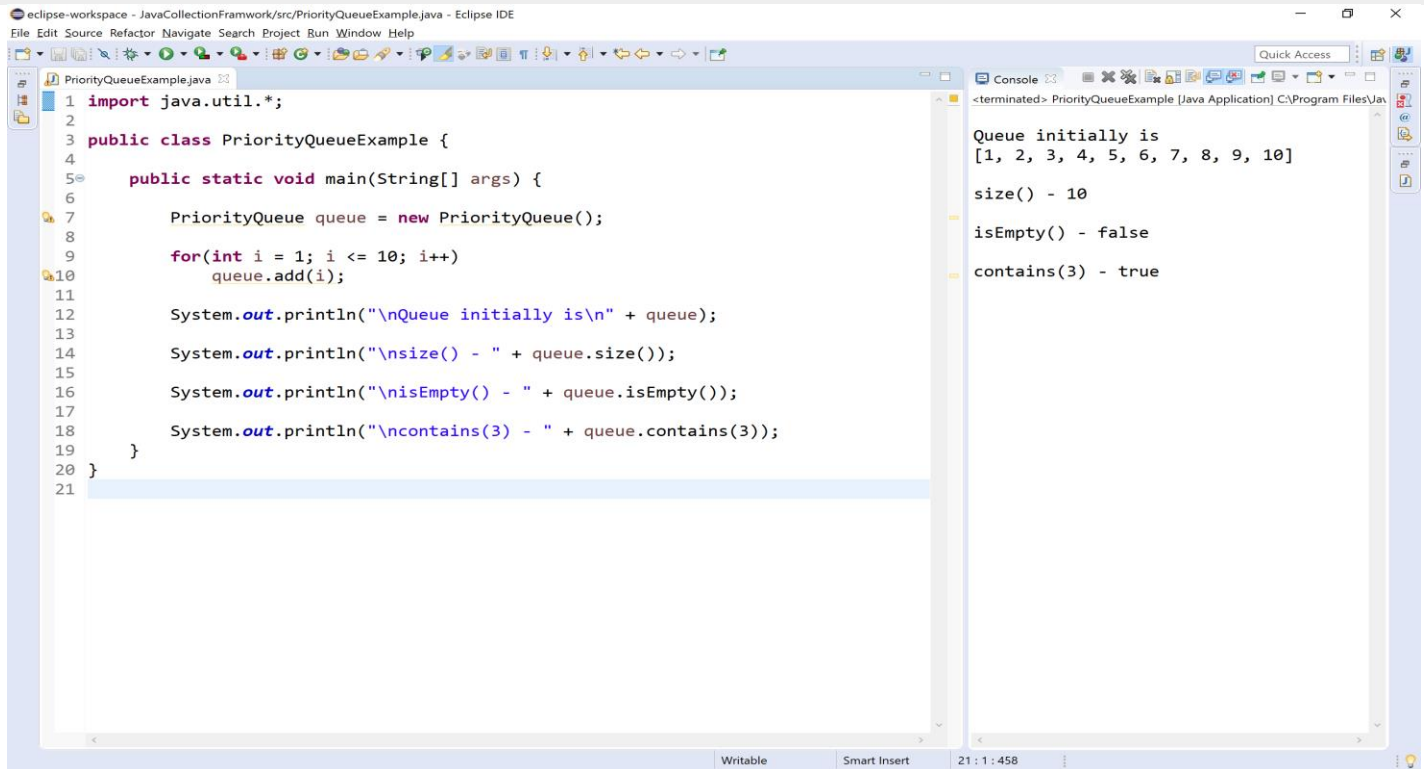
Other utility methods

The PriorityQueue class has the following methods to work with elements of it.

- **int size()** - Returns the total number of elements in the invoking PriorityQueue.
- **boolean isEmpty()** - Returns true if the list is empty otherwise returns false.
- **boolean contains(Object element)** - Returns true if the list contains given element otherwise returns false.
- **Object[] toArray()** - Returns an array of Object instances that contains all the elements from invoking LinkedList.
- **Spliterator spliterator()** - Creates spliterator over the elements in a list.
- **void trimToSize()** - Used to trim a LinkedList instance to the number of elements it contains.
- **Iterator iterator()** - Returns an iterator over the elements in the PriorityQueue. The iterator does not return the elements in any particular order.

Let's consider an example program to illustrate other utility methods of the PriorityQueue.

Example



```
1 import java.util.*;
2
3 public class PriorityQueueExample {
4
5     public static void main(String[] args) {
6
7         PriorityQueue queue = new PriorityQueue();
8
9         for(int i = 1; i <= 10; i++)
10             queue.add(i);
11
12         System.out.println("\nQueue initially is\n" + queue);
13
14         System.out.println("\nsize() - " + queue.size());
15
16         System.out.println("\nisEmpty() - " + queue.isEmpty());
17
18         System.out.println("\ncontains(3) - " + queue.contains(3));
19     }
20 }
21
```

Console Output:

```
<terminated> PriorityQueueExample [Java Application] C:\Program Files\Java
Queue initially is
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
size() - 10
isEmpty() - false
contains(3) - true
```

Consolidated list of methods

The following table provides a consolidated view of all methods of PriorityQueue.

Method	Description
boolean add(E element)	Appends given element to the PriorityQueue.
boolean addAll(Collection c)	Appends given collection of elements to the PriorityQueue.
boolean offer(E element)	Inserts the given element at end of the PriorityQueue.
E element()	Returns the first element from the PriorityQueue.
E peek()	Returns the first element from the PriorityQueue.
E remove()	Removes the first element from the invoking PriorityQueue.

Method	Description
boolean remove(Object element)	Removes the first occurrence of the given element from the invoking PriorityQueue.
boolean removeAll(Collection c)	Removes the given collection of elements from the invoking PriorityQueue.
boolean removeIf(Predicate p)	Removes all of the elements of this collection that satisfy the given predicate.
boolean retainAll(Collection c)	Removes all the elements except those are in the specified collection from the invoking PriorityQueue.
E poll()	Removes the first element from the PriorityQueue, and returns null if the PriorityQueue is empty.
void clear()	Removes all the elements from the PriorityQueue.
int size()	Returns the total number of elements in the invoking PriorityQueue.
boolean isEmpty()	Returns true if the PriorityQueue is empty otherwise returns false.
boolean contains(Object element)	Returns true if the PriorityQueue contains given element otherwise returns false.
Object[] toArray()	Returns an array of Object instances that contains all the elements from invoking PriorityQueue.
Spliterator spliterator()	Creates spliterator over the elements in a PriorityQueue.
void trimToSize()	Used to trim a PriorityQueue instance to the number of elements it contains.

Java ArrayDeque Class

- The **ArrayDeque** class is a part of java collection framework.
- It is available inside the **java.util** package.
- The **ArrayDeque** class extends **AbstractCollection** class and implements **Deque**, **Cloneable**, and **Serializable** interfaces.
- The elements of **ArrayDeque** are organized as the elements of double ended queue data structure.
- The **ArrayDeque** is a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.
- The **ArrayDeque** class is used to create a dynamic double ended queue of elements that can grow or shrunk as needed.

- ❑ The **ArrayDeque** is a child class of **AbstractCollection**.
- ❑ The **ArrayDeque** implements interfaces like **Deque**, **Cloneable**, and **Serializable**.
- ❑ The **ArrayDeque** allows to store duplicate data values, but not null values.
- ❑ The **ArrayDeque** maintains the order of insertion.
- ❑ The **ArrayDeque** allows to add and remove elements at both the ends.
- ❑ The **ArrayDeque** is faster than **LinkedList** and **Stack**.

ArrayDeque class declaration

The **ArrayDeque** class has the following declaration.

Example

```
public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable
```

ArrayDeque class constructors

The **PriorityQueue** class has the following constructors.

- **ArrayDeque()** - Creates an empty **ArrayDeque** with the default initial capacity (16).
- **ArrayDeque(Collection c)** - Creates a **ArrayDeque** with given collection of elements.
- **ArrayDeque(int initialCapacity)** - Creates an empty **ArrayDeque** with the specified initial capacity.

Operations on ArrayDeque

The **ArrayDeque** class allow us to perform several operations like adding, accesing, deleting, updating, looping, etc. Let's look at each operation with examples.

Adding Items

The **ArrayDeque** class has the following methods to add items.

- **boolean add(E element)** - Appends given element to the **ArrayDeque**.

- **boolean addAll(Collection c)** - Appends given collection of elements to the ArrayDeque.
- **void addFirst(E element)** - Adds given element at front of the ArrayDeque.
- **void addLast(E element)** - Adds given element at end of the ArrayDeque.
- **boolean offer(E element)** - Adds given element at end of the ArrayDeque.
- **boolean offerFirst(E element)** - Adds given element at front of the ArrayDeque.
- **boolean offerLast(E element)** - Adds given element at end of the ArrayDeque.
- **void push(E element)** - Adds given element at front of the ArrayDeque.

Let's consider an example program to illustrate adding items to the ArrayDeque.

Example

```

eclipse-workspace - JavaCollectionFramework/src/ArrayDequeExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

ArrayDequeExample.java
1  import java.util.*;
2
3  public class ArrayDequeExample {
4
5      public static void main(String[] args) {
6
7          ArrayDeque deque = new ArrayDeque();
8          ArrayDeque anotherDeque = new ArrayDeque();
9
10         deque.add(10);
11         deque.addFirst(5);
12         deque.addLast(15);
13         deque.offer(20);
14         deque.offerFirst(10);
15         deque.offerLast(30);
16
17         System.out.println("\ndeque is\n" + deque);
18
19         anotherDeque.addAll(deque);
20
21         System.out.println("\nanotherDeque is\n" + anotherDeque);
22
23         anotherDeque.push(40);
24
25         System.out.println("\nanotherDeque after push(40) is\n" + anotherDeque);
26     }
27 }
28
29

```

```

<terminated> ArrayDequeExample [Java Application] C:\Program Files\Java\
Deque is
[10, 5, 10, 15, 20, 30]

anotherDeque is
[10, 5, 10, 15, 20, 30]

anotherDeque after push(40) is
[40, 10, 5, 10, 15, 20, 30]

```

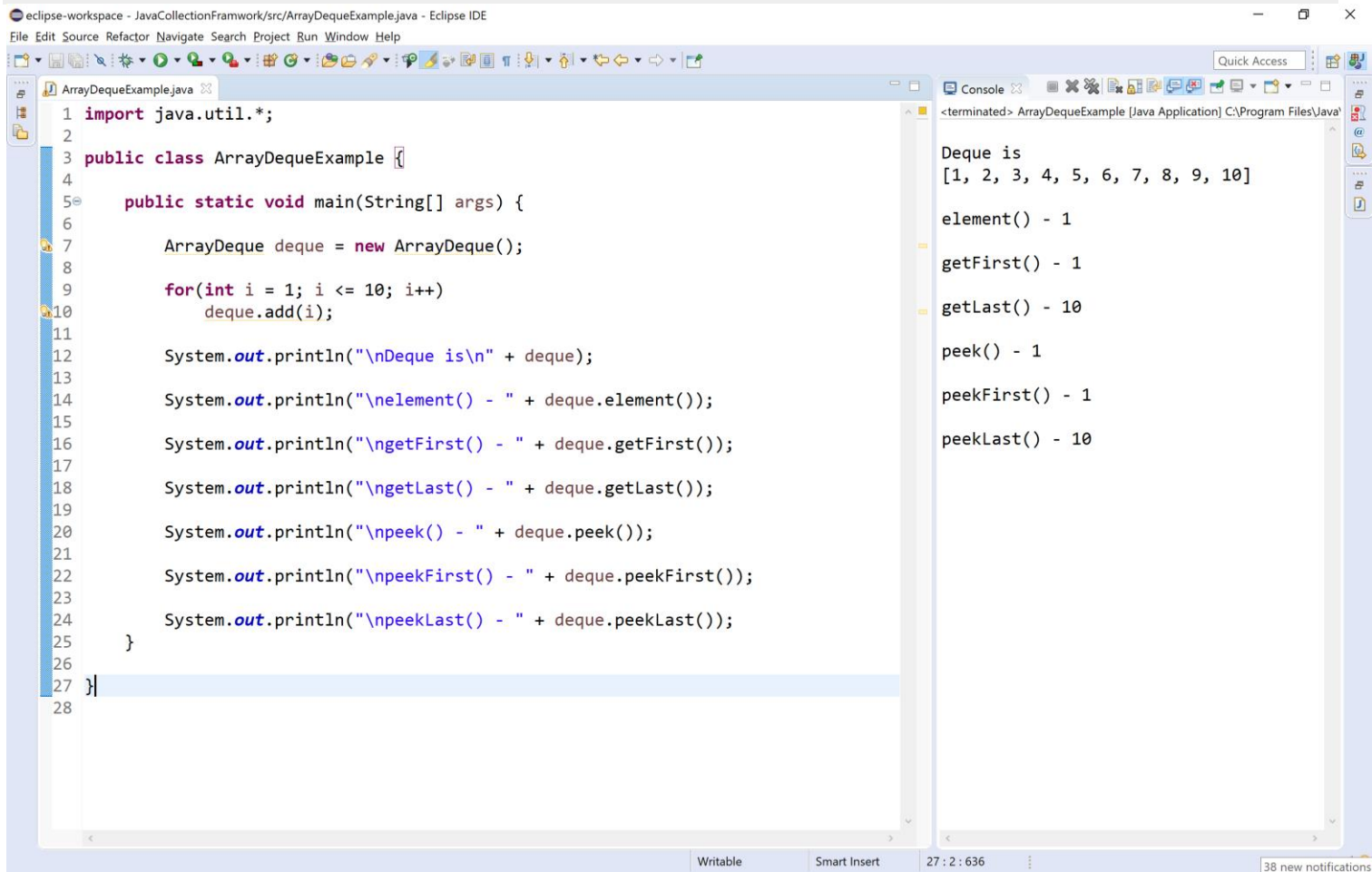
Accessing Items

The ArrayDeque class has the following methods to access items.

- **E element()** - Returns the first element from the invoking ArrayDeque.
- **E getFirst()** - Returns the first element from the invoking ArrayDeque.
- **E getLast()** - Returns the last element from the invoking ArrayDeque.
- **E peek()** - Returns the first element from the invoking ArrayDeque, returns null if this queue is empty.
- **E peekFirst()** - Returns the first element from the invoking ArrayDeque, returns null if this queue is empty.
- **E peekLast()** - Returns the last element from the invoking ArrayDeque, returns null if this queue is empty.

Let's consider an example program to illustrate accessing items from the ArrayDeque.

Example



```
eclipse-workspace - JavaCollectionFramework/src/ArrayDequeExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

ArrayDequeExample.java
1 import java.util.*;
2
3 public class ArrayDequeExample {
4
5     public static void main(String[] args) {
6
7         ArrayDeque deque = new ArrayDeque();
8
9         for(int i = 1; i <= 10; i++)
10             deque.add(i);
11
12         System.out.println("\nDeque is\n" + deque);
13
14         System.out.println("\nelement() - " + deque.element());
15
16         System.out.println("\ngetFirst() - " + deque.getFirst());
17
18         System.out.println("\ngetLast() - " + deque.getLast());
19
20         System.out.println("\npeek() - " + deque.peek());
21
22         System.out.println("\npeekFirst() - " + deque.peekFirst());
23
24         System.out.println("\npeekLast() - " + deque.peekLast());
25     }
26 }
27
28

Console
<terminated> ArrayDequeExample [Java Application] C:\Program Files\Java\
Deque is
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
element() - 1
getFirst() - 1
getLast() - 10
peek() - 1
peekFirst() - 1
peekLast() - 10

Writable Smart Insert 27 : 2 : 636 38 new notifications
```

Updating Items

The ArrayDeque class has no methods to update or change items.

Removing Items

The ArrayDeque class has the following methods to remove items.

- **E remove()** - Removes the first element from the invoking ArrayDeque.
- **E removeFirst()** - Removes the first element from the invoking ArrayDeque.
- **E removeLast()** - Removes the last element from the invoking ArrayDeque.
- **boolean remove(Object o)** - Removes the specified element from the invoking ArrayDeque.
- **boolean removeFirstOccurrence(Object o)** - Removes the first occurrence of the specified element in this ArrayDeque.
- **boolean removeLastOccurrence(Object o)** - Removes the last occurrence of the specified element in this ArrayDeque.
- **boolean removeAll(Predicate p)** - Removes all of the elements of ArrayDeque collection that satisfy the given predicate.
- **boolean retainAll(Collection c)** - Removes all of the elements of ArrayDeque collection except specified collection of elements.

- **E poll()** - Removes the first element from the ArrayDeque, and returns null if the list is empty.
- **E pollFirst()** - Removes the first element from the ArrayDeque, and returns null if the list is empty.
- **E pollLast()** - Removes the last element from the ArrayDeque, and returns null if the list is empty.
- **E pop()** - Removes the first element from the ArrayDeque.
- **void clear()** - Removes all the elements from the PriorityQueue.

Let's consider an example program to illustrate removing items from the ArrayDeque.

Example

```

1 import java.util.*;
2 public class ArrayDequeExample {
3
4     public static void main(String[] args) {
5
6         ArrayDeque deque = new ArrayDeque();
7         for(int i = 1; i <= 10; i++)
8             deque.add(i);
9         System.out.println("\nDeque is\n" + deque);
10
11         deque.remove();
12         System.out.println("\nDeque after remove()\n" + deque);
13
14         deque.removeFirst();
15         System.out.println("\nDeque after removeFirst()\n" + deque);
16
17         deque.removeLast();
18         System.out.println("\nDeque after removeLast()\n" + deque);
19
20         deque.remove(5);
21         System.out.println("\nDeque after remove(5)\n" + deque);
22
23         deque.poll();
24         System.out.println("\nDeque after poll()\n" + deque);
25
26         deque.pollFirst();
27         System.out.println("\nDeque after pollFirst()\n" + deque);
28
29         deque.pollLast();
30         System.out.println("\nDeque after pollLast()\n" + deque);
31
32         deque.pop();
33         System.out.println("\nDeque after pop()\n" + deque);
34
35         deque.clear();
36         System.out.println("\nDeque after clear()\n" + deque);
37     }
38 }

```

Console Output:

```

<terminated> ArrayDequeExample [Java Application] C:\Program Files\Java\
Deque is
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Deque after remove()
[2, 3, 4, 5, 6, 7, 8, 9, 10]
Deque after removeFirst()
[3, 4, 5, 6, 7, 8, 9, 10]
Deque after removeLast()
[3, 4, 5, 6, 7, 8, 9]
Deque after remove(5)
[3, 4, 6, 7, 8, 9]
Deque after poll()
[4, 6, 7, 8, 9]
Deque after pollFirst()
[6, 7, 8, 9]
Deque after pollLast()
[6, 7, 8]
Deque after pop()
[7, 8]
Deque after clear()
[]

```

Other utility methods

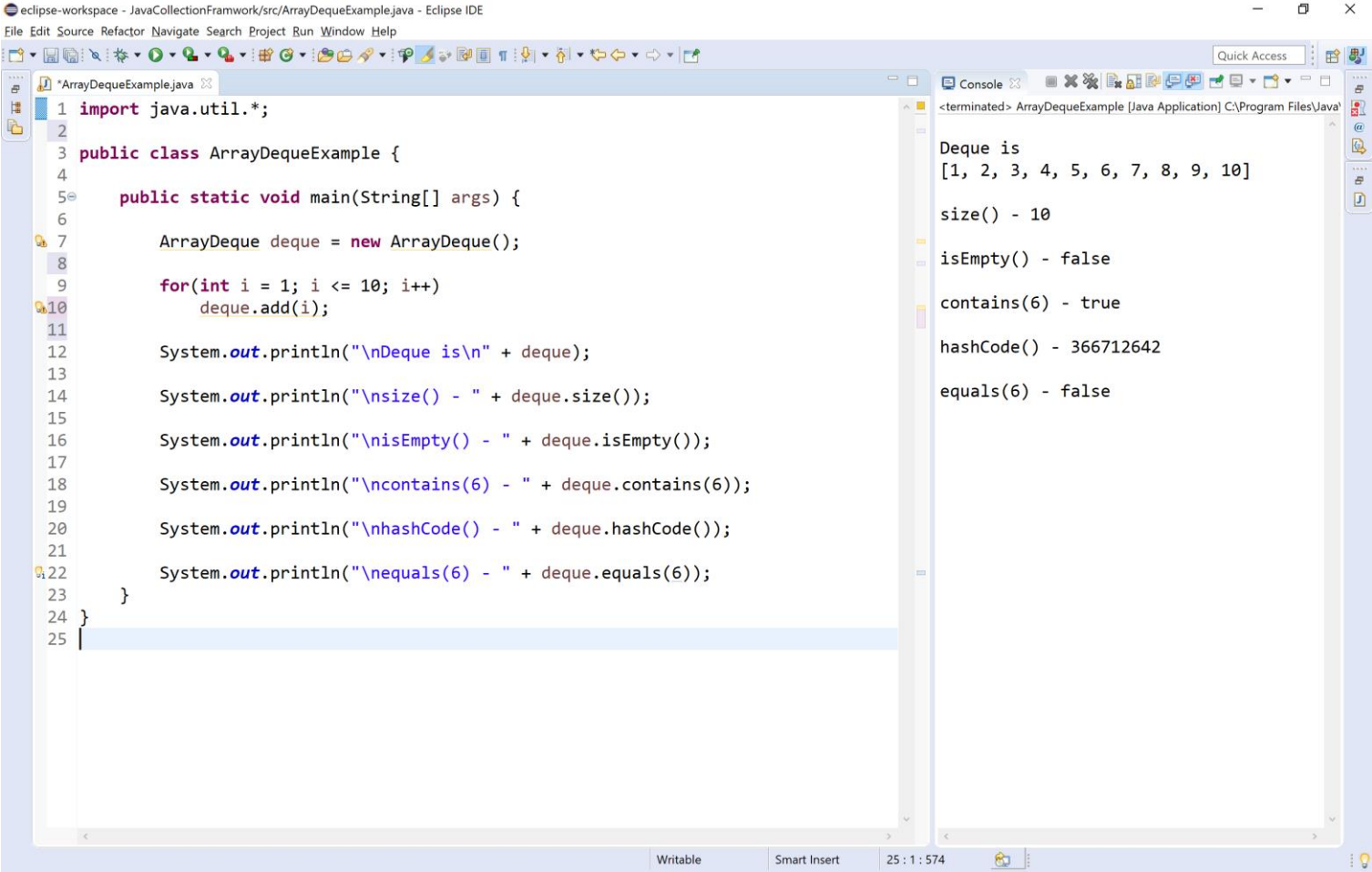
The ArrayDeque class has the following methods to work with elements of it.

- **int size()** - Returns the total number of elements in the invoking ArrayDeque.
- **boolean isEmpty()** - Returns true if the ArrayDeque is empty otherwise returns false.
- **ArrayDeque clone()** - Returns a copy of the invoking ArrayDeque.
- **boolean contains(Object element)** - Returns true if the ArrayDeque contains given element otherwise returns false.
- **boolean containsAll(Collection c)** - Returns true if the ArrayDeque contains given collection of elements otherwise returns false.
- **boolean equals(Object o)** - Compares the specified object with invoking ArrayDeque collection for equality.
- **int hashCode()** - Returns the hash code of the invoking ArrayDeque.

- **Object[] toArray()** - Returns an array of Object instances that contains all the elements from invoking ArrayDeque.
- **Spliterator spliterator()** - Creates spliterator over the elements in a ArrayDeque.
- **Iterator iterator()** - Returns an iterator over the elements in the ArrayDeque. The iterator does not return the elements in any particular order.

Let's consider an example program to illustrate other utility methods of the ArrayDeque.

Example



Consolidated list of methods

The following table providess a consolidated view of all methods of ArrayDeque.

Method	Description
boolean add(E element)	Appends given element to the ArrayDeque.
boolean addAll(Collection c)	Appends given collection of elements to the ArrayDeque.

Method	Description
void addFirst(E element)	Inserts the given element at front of the ArrayDeque.
void addLast(E element)	Inserts the given element at end of the ArrayDeque.
boolean offer(E element)	Inserts the given element at end of the ArrayDeque.
boolean offerFirst(E element)	Inserts the given element at front of the ArrayDeque.
boolean offerLast(E element)	Inserts the given element at end of the ArrayDeque.
void push(E element)	Inserts the given element at front of the ArrayDeque.
E element()	Returns the first element from the ArrayDeque.
E getFirst()	Returns the first element from the ArrayDeque.
E getLast()	Returns the last element from the ArrayDeque.
E peek()	Returns the first element from the invoking ArrayDeque, returns null if this queue is empty.
E peekFirst()	Returns the first element from the invoking ArrayDeque, returns null if this queue is empty.
E peekLast()	Returns the last element from the invoking ArrayDeque, returns null if this queue is empty.
E remove()	Removes the first element from the invoking ArrayDeque.
E removeFirst()	Removes the first element from the invoking ArrayDeque.

Method	Description
E removeLast()	Removes the last element from the invoking ArrayDeque.
boolean remove(Object element)	Removes the first occurrence of the given element from the invoking ArrayDeque.
boolean removeFirstOccurrence(Object element)	Removes the first occurrence of the specified element in this ArrayDeque.
boolean removeLastOccurrence(Object element)	Removes the last occurrence of the specified element in this ArrayDeque.
boolean removeIf(Predicate p)	Removes all of the elements of the ArrayDeque collection that satisfy the given predicate.
boolean retainAll(Collection c)	Removes all the elements except those are in the specified collection from the invoking ArrayDeque.
E poll()	Removes the first element from the ArrayDeque, and returns null if the ArrayDeque is empty.
E pollFirst()	Removes the first element from the ArrayDeque, and returns null if the ArrayDeque is empty.
E pollLast()	Removes the last element from the ArrayDeque, and returns null if the ArrayDeque is empty.
E pop()	Removes the first element from the invoking ArrayDeque.
void clear()	Removes all the elements from the ArrayDeque.
int size()	Returns the total number of elements in the invoking ArrayDeque.

Method	Description
boolean isEmpty()	Returns true if the ArrayDeque is empty otherwise returns false.
boolean equals()	Compares the specified object with invoking ArrayDeque collection for equality.
boolean contains(Object element)	Returns true if the ArrayDeque contains given element otherwise returns false.
boolean containsAll(Collection c)	Returns true if the ArrayDeque contains all elements of given collection otherwise returns false.
int hashCode()	Returns the hash code of the invoking ArrayDeque.
Object[] toArray()	Returns an array of Object instances that contains all the elements from invoking ArrayDeque.
Spliterator spliterator()	Creates spliterator over the elements in a ArrayDeque.
Iterator iterator()	Returns an iterator over the elements in the ArrayDeque. The iterator does not return the elements in any particular order.

Accessing a Java Collection via a Iterator

The java collection framework often we want to cycle through the elements. For example, we might want to display each element of a collection. The java provides an interface **Iterator** that is available inside the **java.util** package to cycle through each element of a collection.

- ❑ The **Iterator** allows us to move only forward direction.
- ❑ The **Iterator** does not support the replacement and addition of new elements.

We use the following steps to access a collection of elements using the Iterator.

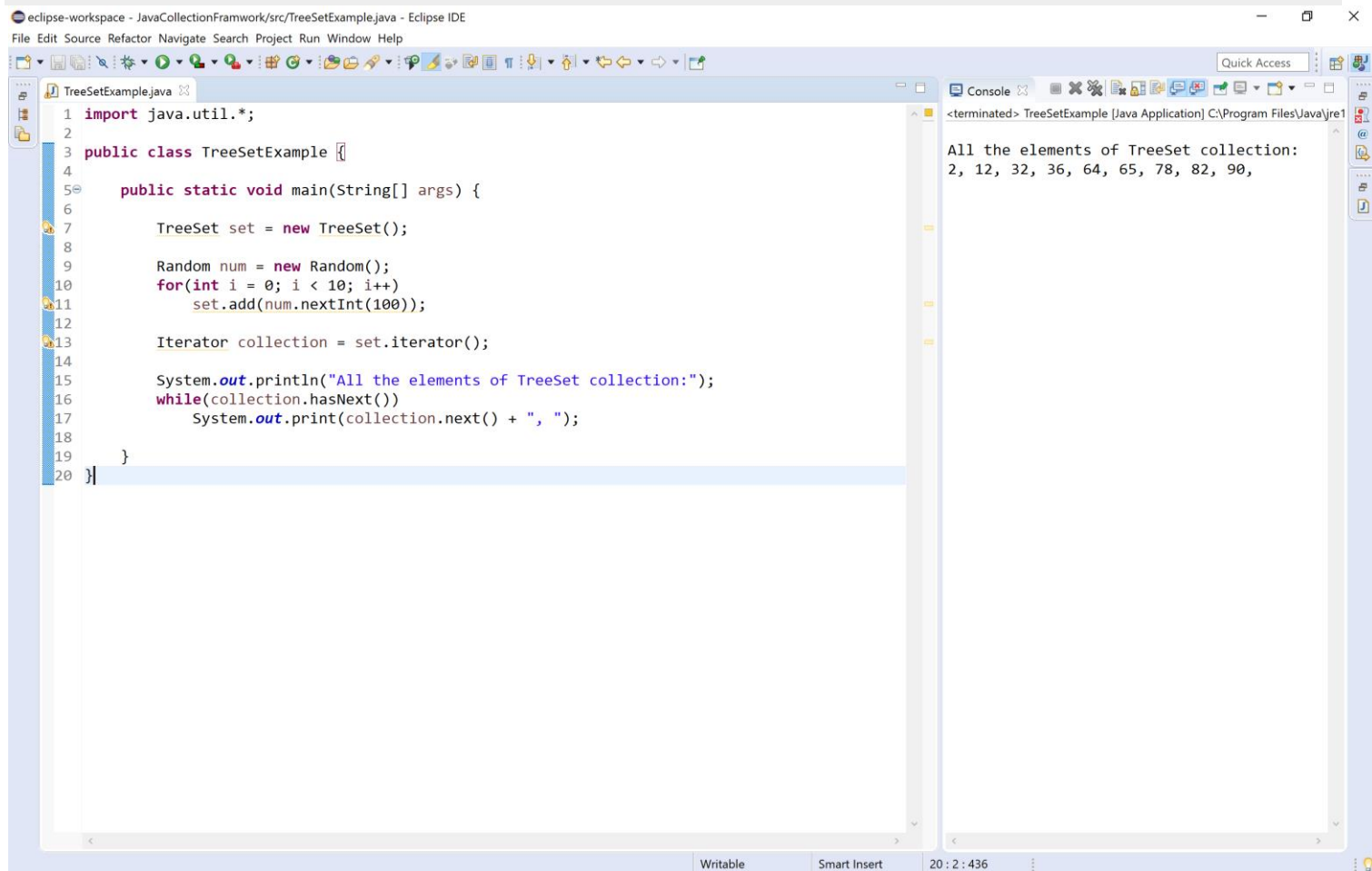
- **Step - 1:** Create an object of the Iterator by calling **collection.iterator()** method.
- **Step - 2:** Use the method **hasNext()** to access to check does the collection has the next element. (Use a loop).
- **Step - 3:** Use the method **next()** to access each element from the collection. (use inside the loop).

The Iterator has the following methods.

Method	Description
Iterator iterator()	Used to obtain an iterator to the start of the collection.
boolean hasNext()	Returns true if the collection has the next element, otherwise, it returns false.
E next()	Returns the next element available in the collection.

Let's consider an example program to illustrate accessing elements of a collection via the Iterator.

Example



The screenshot shows the Eclipse IDE with a Java project named 'JavaCollectionFramework'. The source file 'TreeSetExample.java' is open, displaying the following code:

```
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         TreeSet set = new TreeSet();
8
9         Random num = new Random();
10        for(int i = 0; i < 10; i++)
11            set.add(num.nextInt(100));
12
13        Iterator collection = set.iterator();
14
15        System.out.println("All the elements of TreeSet collection:");
16        while(collection.hasNext())
17            System.out.print(collection.next() + ", ");
18
19    }
20 }
```

The console output on the right shows the result of the program execution:

```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jre1
All the elements of TreeSet collection:
2, 12, 32, 36, 64, 65, 78, 82, 90,
```

Accessing a collection using for-each

We can use the for-each statement to access elements of a collection.

Let's consider an example program to illustrate accessing items from a collection using a for-each statement.

Example

The screenshot shows the Eclipse IDE with a Java project named 'JavaCollectionFramework'. The file 'TreeSetExample.java' is open in the editor. The code defines a public class 'TreeSetExample' with a 'main' method. It initializes a 'TreeSet', an 'ArrayList', and a 'PriorityQueue'. A 'Random' object is used to generate 100 random integers, which are added to each collection. The 'main' method then prints the elements of each collection using 'for' loops. The console on the right shows the output of the program, displaying the elements of the 'TreeSet', 'ArrayList', and 'PriorityQueue' collections.

```
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         TreeSet set = new TreeSet();
8         ArrayList list = new ArrayList();
9         PriorityQueue queue = new PriorityQueue();
10
11         Random num = new Random();
12         for(int i = 0; i < 100; i++) {
13             set.add(num.nextInt(100));
14             list.add(num.nextInt(100));
15             queue.add(num.nextInt(100));
16         }
17
18         System.out.println("\nAll the elements of TreeSet collection:");
19         for(Object element:set) {
20             System.out.print(element + " ");
21         }
22
23         System.out.println("\n\nAll the elements of ArrayList collection:");
24         for(Object element:list) {
25             System.out.print(element + " ");
26         }
27
28         System.out.println("\n\nAll the elements of PriorityQueue collection:");
29         for(Object element:queue) {
30             System.out.print(element + " ");
31         }
32     }
33 }
34
35 }
```

Console Output:

```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
All the elements of TreeSet collection:
25, 44, 53, 65, 67, 73, 78, 88, 97,
All the elements of ArrayList collection:
38, 49, 54, 53, 44, 28, 45, 69, 53, 33,
All the elements of PriorityQueue collection:
12, 14, 19, 33, 15, 85, 57, 98, 49, 66,
```

The For-Each alternative

- Using for-each, we can access the elements of a collection. But for-each can only be used if we don't want to modify the contents of a collection, and we don't want any reverse access.
- Alternatively, we can use the Iterator to access or cycle through a collection of elements.

Let's consider an example program to illustrate The for-each alternative.

Example

The screenshot shows the Eclipse IDE with the same 'JavaCollectionFramework' project. The file 'TreeSetExample.java' is open, but the code is modified to use 'for-each' and 'Iterator'. It still initializes the 'ArrayList' and 'PriorityQueue' with random numbers. However, it uses 'for(Object element:list)' to iterate over the 'ArrayList' and 'collection.iterator()' to iterate over the 'PriorityQueue'. The console on the right shows the output of the program, displaying the elements of the 'ArrayList' and 'PriorityQueue' collections.

```
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         ArrayList list = new ArrayList();
8         PriorityQueue queue = new PriorityQueue();
9
10        Random num = new Random();
11        for(int i = 0; i < 100; i++) {
12            list.add(num.nextInt(100));
13            queue.add(num.nextInt(100));
14        }
15
16        // Accessing using for-each statement
17        System.out.println("\n\nAll the elements of ArrayList collection:");
18        for(Object element:list) {
19            System.out.print(element + " ");
20        }
21
22        // Accessing using Iterator
23        Iterator collection = queue.iterator();
24        System.out.println("\n\nAll the elements of PriorityQueue collection:");
25        while(collection.hasNext()) {
26            System.out.print(collection.next() + " ");
27        }
28    }
29 }
30 }
```

Console Output:

```
<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
All the elements of ArrayList collection:
76, 2, 3, 25, 37, 6, 46, 89, 51, 63,
All the elements of PriorityQueue collection:
2, 3, 17, 50, 24, 79, 79, 92, 63, 90,
```

Accessing elements using ListIterator

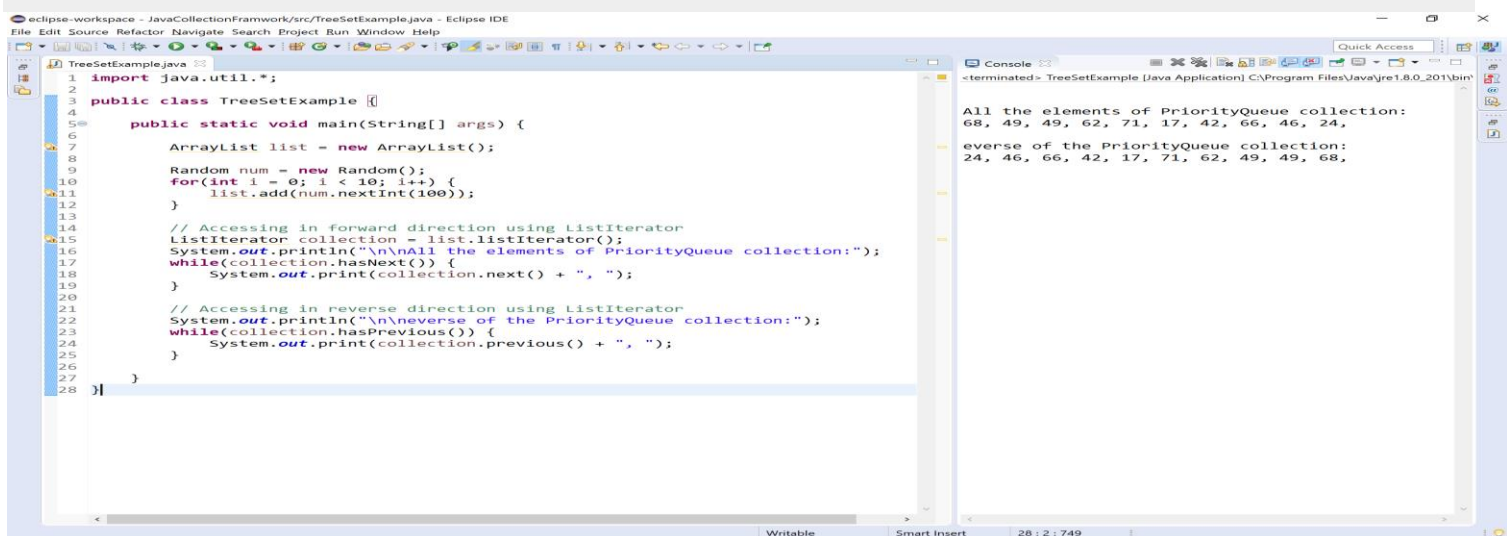
The ListIterator interface is used to traverse through a list in both forward and backward directions. It does not support all types of collections. It supports only the collection which implements the List interface.

The ListIterator provides the following methods to traverse through a list of elements.

Method	Description
ListIterator listIterator()	Used obtain an iterator of the list collection.
boolean hasNext()	Returns true if the list collection has next element, otherwise it returns false.
E next()	Returns the next element available in the list collection.
boolean hasPrevious()	Returns true if the list collection has previous element, otherwise it returns false.
E previous()	Returns the previous element available in the list collection.
int nextIndex()	Returns the index of the next element. If there is no next element, returns the size of the list.
E previousIndex()	Returns the index of the previous element. If there is no previous element, returns -1.

Let's consider an example program to illustrate ListIterator to access elements of a list.

Example



```
eclipse.workspace - JavaCollectionFramework/src/TreeSetExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

TreeSetExample.java
1 import java.util.*;
2
3 public class TreeSetExample {
4
5     public static void main(String[] args) {
6
7         ArrayList list = new ArrayList();
8
9         Random num = new Random();
10        for(int i = 0; i < 10; i++) {
11            list.add(num.nextInt(100));
12        }
13
14        // Accessing in forward direction using ListIterator
15        ListIterator collection = list.listIterator();
16        System.out.println("\n\nAll the elements of PriorityQueue collection:");
17        while(collection.hasNext()) {
18            System.out.print(collection.next() + ", ");
19        }
20
21        // Accessing in reverse direction using ListIterator
22        System.out.println("\n\nreverse of the PriorityQueue collection:");
23        while(collection.hasPrevious()) {
24            System.out.print(collection.previous() + ", ");
25        }
26    }
27
28 }
```

```
Console
terminated: TreeSetExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\
All the elements of PriorityQueue collection:
68, 49, 49, 62, 71, 17, 42, 66, 46, 24,
reverse of the PriorityQueue collection:
24, 46, 66, 42, 17, 71, 62, 49, 49, 68,
```

Map Interface in java

- The java collection framework has an interface **Map** that is available inside the **java.util** package.
- The Map interface is not a subtype of Collection interface.

- ❑ The **Map** stores the elements as a pair of key and value.
- ❑ The **Map** does not allow duplicate keys, but allows duplicate values.
- ❑ In a **Map**, each key can map to at most one value only.
- ❑ In a **Map**, the order of elements depends on specific implementations, e.g. TreeMap and LinkedHashMap have predictable order, while HashMap does not.

The **Map** interface has the following child interfaces.

Interface	Description
Map	Maps unique key to value.
Map.Entry	Describe an element in key and value pair in a map. Entry is sub interface of Map.
SortedMap	It is a child of Map so that key are maintained in an ascending order.
NavigableMap	It is a child of SortedMap to handle the retrieval of entries based on closest match searches.

The **Map** interface has the following three classes.

Class	Description
HashMap	It implements the Map interface, but it doesn't maintain any order.
LinkedHashMap	It implements the Map interface, it also extends HashMap class. It maintains the insertion order.
TreeMap	It implements the Map and SortedMap interfaces. It maintains the ascending order.

Map Interface methods

The Map interface contains methods for handling elements of a map. It has the following methods.

The insertion of a key, value pair is said to be an entry in the map terminology.

Method	Description
V put(Object key, Object value)	Inserts an entry in the map.
void putAll(Map map)	Inserts the specified map into the invoking map.
V putIfAbsent(K key, V value)	Inserts the specified value with the specified key in the map only if that key does not exist.
Set keySet()	Returns a Set that contains all the keys of invoking Map.
Collection values()	Returns a collection that contains all the values of invoking Map.
Set<Map.Entry<K,V>> entrySet()	Returns a Set that contains all the keys and values of invoking Map.
V get(Object key)	Returns the value associated with the specified key.
V getOrDefault(Object key, V defaultValue)	Returns the value associated with the specified key, or defaultValue if the map does not contain the key.
boolean containsValue(Object value)	Returns true if specified value found in the map, else return false.
boolean containsKey(Object key)	Returns true if specified key found in the map, else return false.
V replace(K key, V value)	Used to replace the specified value for the specified key.
boolean replace(K key, V oldValue, V newValue)	Used to replaces the oldValue with the newValue for a specified key.
void replaceAll(BiFunction function)	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
V merge(K key, V value, BiFunction	If the specified key is not already associated with a value or is

Method	Description
remappingFunction)	associated with null, associates it with the given non-null value.
V compute(K key, BiFunction remappingFunction)	Used to compute a mapping for the specified key and its current mapped value.
V computeIfAbsent(K key, Function mappingFunction)	Used to compute its value using the given mapping function, if the specified key is not already associated with a value, and enters it into this map unless null.
V computeIfPresent(K key, BiFunction remappingFunction)	Used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
void forEach(BiConsumer action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V remove(Object key)	Removes an entry for the specified key.
boolean remove(Object key, Object value)	Removes the specified values with the associated specified keys from the map.
void clear()	Removes all the entries from the map.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
int hashCode()	Returns the hash code for invoking the Map.
boolean isEmpty()	Returns true if the map is empty; otherwise returns false.
int size()	Returns total number of entries in the invoking Map.

Map Interface Classes in java

- The java collection framework has an interface **Map** that is available inside the **java.util** package.
- The Map interface is not a subtype of Collection interface.

The **Map** interface has the following three classes.

Class	Description
HashMap	It implements the Map interface, but it doesn't maintain any order.
LinkedHashMap	It implements the Map interface, it also extends HashMap class. It maintains the insertion order.
TreeMap	It implements the Map and SortedMap interfaces. It maintains the ascending order.

Commonly used methods defined by Map interface

Method	Description
Object put(Object k, Object v)	It performs an entry into the Map.
Object putAll(Map m)	It inserts all the entries of m into invoking Map.
Object get(Object k)	It returns the value associated with given key.
boolean containsKey(Object k)	It returns true if map contain k as key. Otherwise false.
Set keySet()	It returns a set that contains all the keys from the invoking Map.
Set valueSet()	It returns a set that contains all the values from the invoking Map.
Set entrySet()	It returns a set that contains all the entries from the invoking Map.

Now, let's look at each class in detail with example programs.

HashMap Class

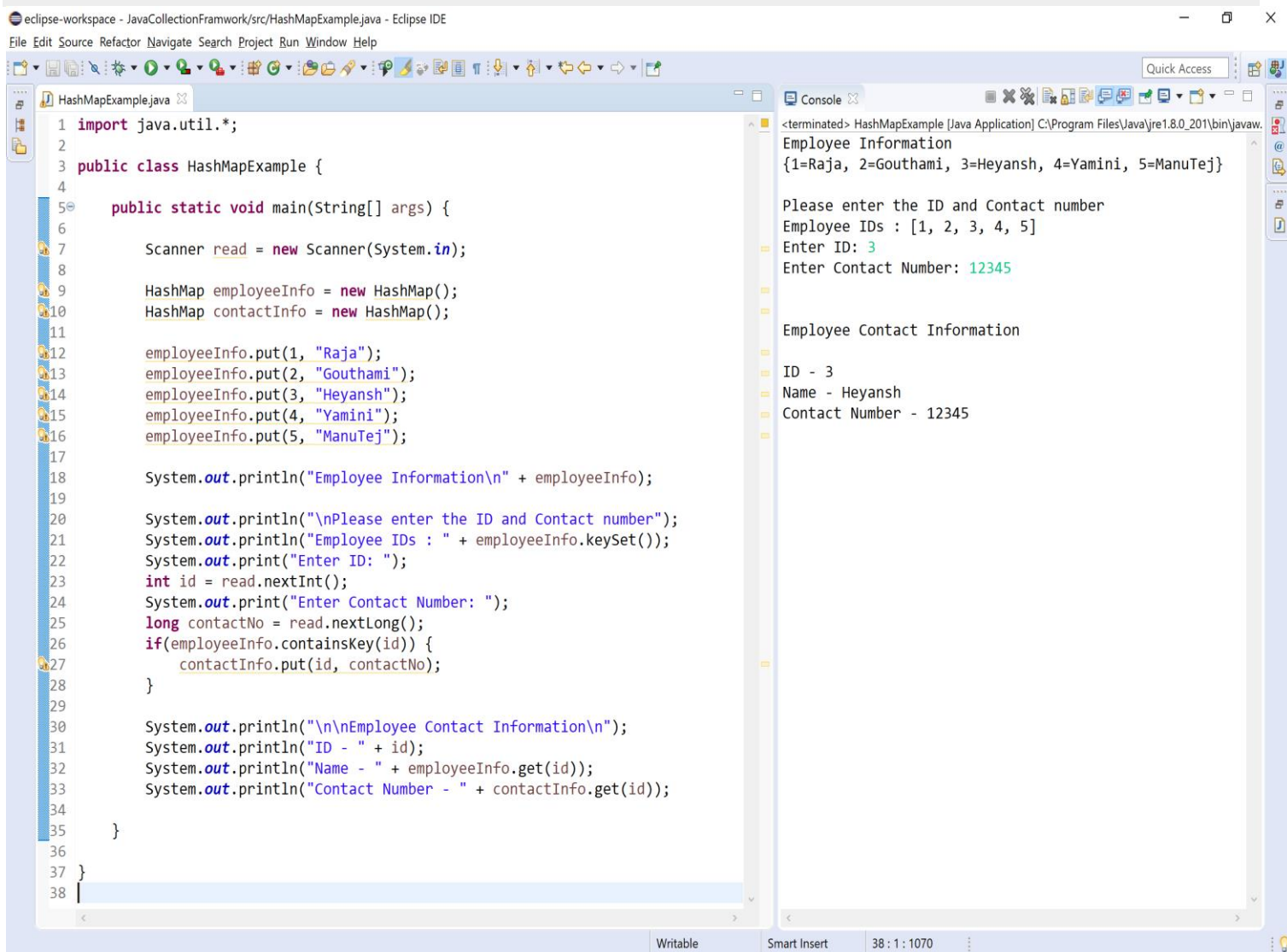
- The HashMap class is a child class of AbstractMap, and it implements the Map interface.
- The HashMap is used to store the data in the form of key, value pair using hash table concept.

Key Properties of HashMap

- HashMap is a child class of AbstractMap class.
- HashMap implements the interfaces Map, Cloneable, and Serializable.
- HashMap stores data as a pair of key and value.
- HashMap uses Hash table concept to store the data.
- HashMap does not allow duplicate keys, but values may be repeated.
- HashMap allows only one null key and multiple null values.
- HashMap does not follow any order.
- HashMap has the default capacity 16 entries.

Let's consider an example program to illustrate HashMap.

Example



```
eclipse-workspace - JavaCollectionFramework/src/HashMapExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashMapExample.java
1 import java.util.*;
2
3 public class HashMapExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8
9         HashMap employeeInfo = new HashMap();
10        HashMap contactInfo = new HashMap();
11
12        employeeInfo.put(1, "Raja");
13        employeeInfo.put(2, "Gouthami");
14        employeeInfo.put(3, "Heyansh");
15        employeeInfo.put(4, "Yamini");
16        employeeInfo.put(5, "ManuTej");
17
18        System.out.println("Employee Information\n" + employeeInfo);
19
20        System.out.println("\nPlease enter the ID and Contact number");
21        System.out.println("Employee IDs : " + employeeInfo.keySet());
22        System.out.print("Enter ID: ");
23        int id = read.nextInt();
24        System.out.print("Enter Contact Number: ");
25        long contactNo = read.nextLong();
26        if(employeeInfo.containsKey(id)) {
27            contactInfo.put(id, contactNo);
28        }
29
30        System.out.println("\n\nEmployee Contact Information\n");
31        System.out.println("ID - " + id);
32        System.out.println("Name - " + employeeInfo.get(id));
33        System.out.println("Contact Number - " + contactInfo.get(id));
34
35    }
36
37 }
38 |

Console
<terminated> HashMapExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.
Employee Information
{1=Raja, 2=Gouthami, 3=Heyansh, 4=Yamini, 5=ManuTej}

Please enter the ID and Contact number
Employee IDs : [1, 2, 3, 4, 5]
Enter ID: 3
Enter Contact Number: 12345

Employee Contact Information
ID - 3
Name - Heyansh
Contact Number - 12345

Writable Smart Insert 38 : 1 : 1070
```

LinkedHashMap Class

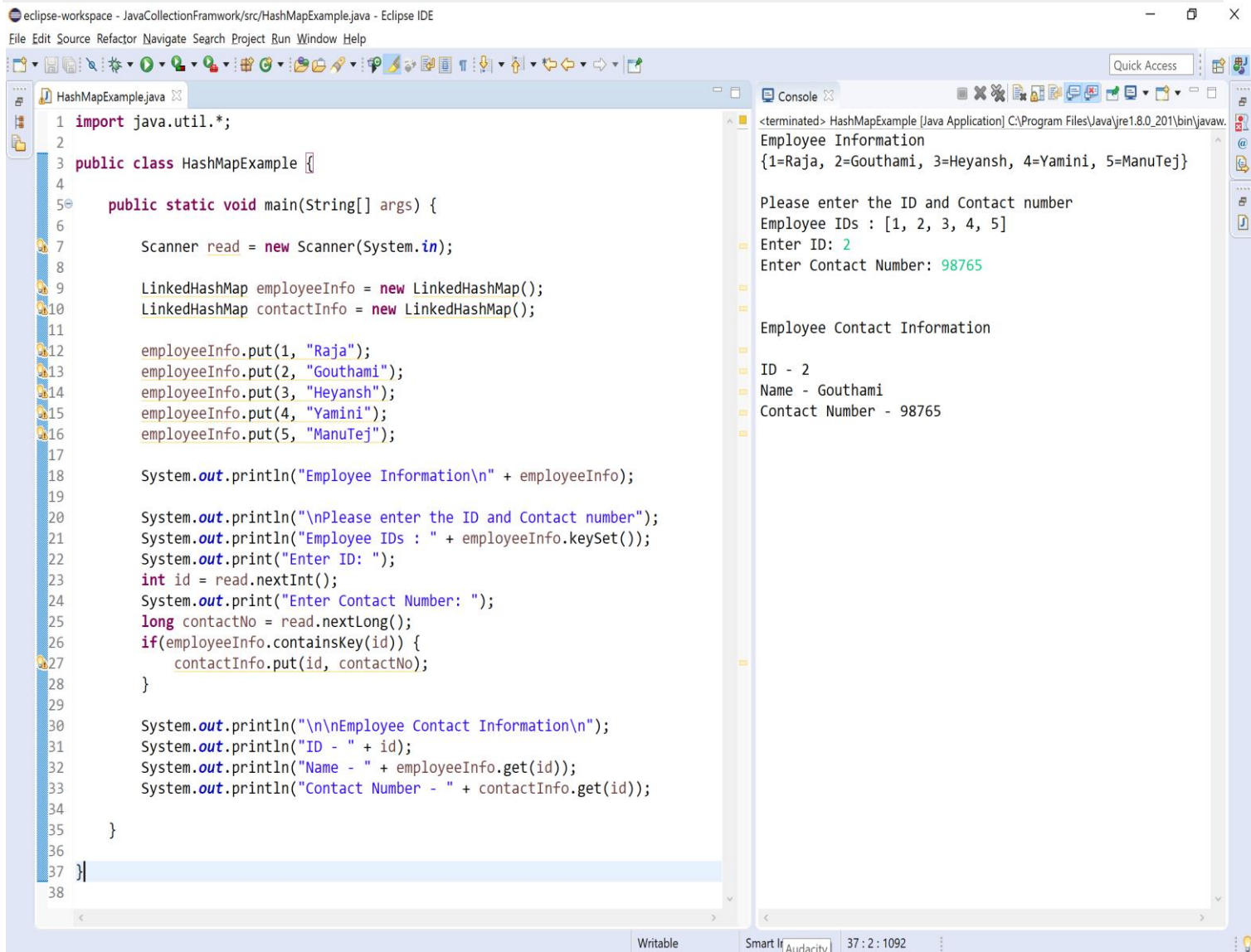
The LinkedHashMap class is a child class of HashMap, and it implements the Map interface. The LinkedHashMap is used to store the data in the form of key, value pair using hash table and linked list concepts.

Key Properties of LinkedHashMap

- LinkedHashMap is a child class of HashMap class.
- LinkedHashMap implements the Map interface.
- LinkedHashMap stores data as a pair of key and value.
- LinkedHashMap uses Hash table concept to store the data.
- LinkedHashMap does not allow duplicate keys, but values may be repeated.
- LinkedHashMap allows only one null key and multiple null values.
- LinkedHashMap follows the insertion order.
- LinkedHashMap has the default capacity 16 entries.

Let's consider an example program to illustrate LinkedHashMap.

Example



```
eclipse-workspace - JavaCollectionFramework/src/HashMapExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashMapExample.java
1 import java.util.*;
2
3 public class HashMapExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8
9         LinkedHashMap employeeInfo = new LinkedHashMap();
10        LinkedHashMap contactInfo = new LinkedHashMap();
11
12        employeeInfo.put(1, "Raja");
13        employeeInfo.put(2, "Gouthami");
14        employeeInfo.put(3, "Heyansh");
15        employeeInfo.put(4, "Yamini");
16        employeeInfo.put(5, "ManuTej");
17
18        System.out.println("Employee Information\n" + employeeInfo);
19
20        System.out.println("\nPlease enter the ID and Contact number");
21        System.out.println("Employee IDs : " + employeeInfo.keySet());
22        System.out.print("Enter ID: ");
23        int id = read.nextInt();
24        System.out.print("Enter Contact Number: ");
25        long contactNo = read.nextLong();
26        if(employeeInfo.containsKey(id)) {
27            contactInfo.put(id, contactNo);
28        }
29
30        System.out.println("\n\nEmployee Contact Information\n");
31        System.out.println("ID - " + id);
32        System.out.println("Name - " + employeeInfo.get(id));
33        System.out.println("Contact Number - " + contactInfo.get(id));
34
35    }
36
37 }
38 }
```

Console

```
<terminated> HashMapExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.
Employee Information
{1=Raja, 2=Gouthami, 3=Heyansh, 4=Yamini, 5=ManuTej}

Please enter the ID and Contact number
Employee IDs : [1, 2, 3, 4, 5]
Enter ID: 2
Enter Contact Number: 98765

Employee Contact Information
ID - 2
Name - Gouthami
Contact Number - 98765
```

Writable Smart If Audacity 37 : 2 : 1092

TreeMap Class

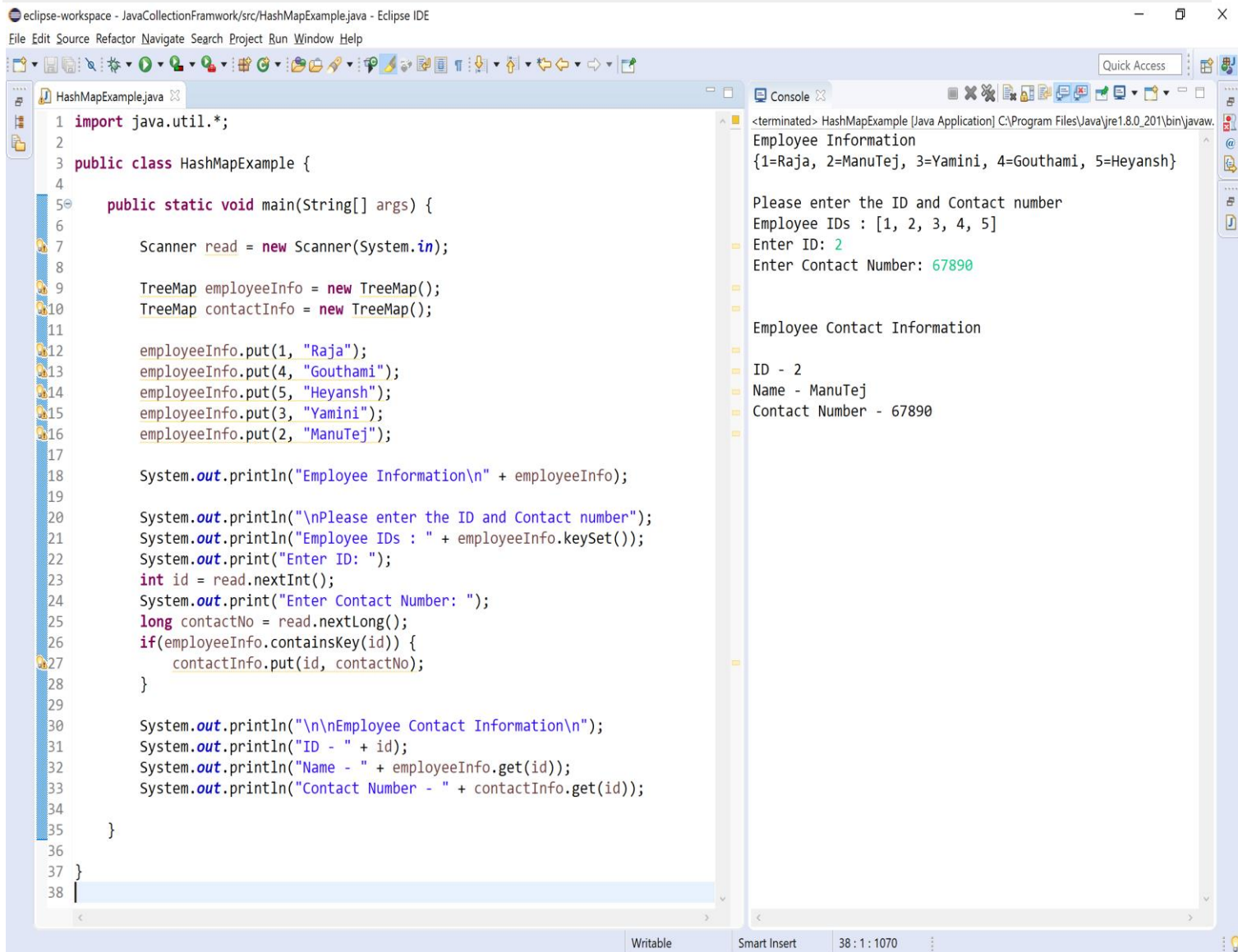
The TreeMap class is a child class of AbstractMap, and it implements the NavigableMap interface which is a child interface of SortedMap. The TreeMap is used to store the data in the form of key, value pair using a red-black tree concepts.

Key Properties of TreeMap

- TreeMap is a child class of AbstractMap class.
- TreeMap implements the NavigableMap interface which is a child interface of SortedMap interface.
- TreeMap stores data as a pair of key and value.
- TreeMap uses red-black tree concept to store the data.
- TreeMap does not allow duplicate keys, but values may be repeated.
- TreeMap does not allow null key, but allows null values.
- TreeMap follows the ascending order based on keys.

Let's consider an example program to illustrate TreeMap.

Example



```
eclipse-workspace - JavaCollectionFramework/src/HashMapExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

HashMapExample.java
1 import java.util.*;
2
3 public class HashMapExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8
9         TreeMap employeeInfo = new TreeMap();
10        TreeMap contactInfo = new TreeMap();
11
12        employeeInfo.put(1, "Raja");
13        employeeInfo.put(4, "Gouthami");
14        employeeInfo.put(5, "Heyansh");
15        employeeInfo.put(3, "Yamini");
16        employeeInfo.put(2, "ManuTej");
17
18        System.out.println("Employee Information\n" + employeeInfo);
19
20        System.out.println("\nPlease enter the ID and Contact number");
21        System.out.println("Employee IDs : " + employeeInfo.keySet());
22        System.out.print("Enter ID: ");
23        int id = read.nextInt();
24        System.out.print("Enter Contact Number: ");
25        long contactNo = read.nextLong();
26        if(employeeInfo.containsKey(id)) {
27            contactInfo.put(id, contactNo);
28        }
29
30        System.out.println("\n\nEmployee Contact Information\n");
31        System.out.println("ID - " + id);
32        System.out.println("Name - " + employeeInfo.get(id));
33        System.out.println("Contact Number - " + contactInfo.get(id));
34    }
35 }
36
37
38
```

```
<terminated> HashMapExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.
Employee Information
{1=Raja, 2=ManuTej, 3=Yamini, 4=Gouthami, 5=Heyansh}

Please enter the ID and Contact number
Employee IDs : [1, 2, 3, 4, 5]
Enter ID: 2
Enter Contact Number: 67890

Employee Contact Information
ID - 2
Name - ManuTej
Contact Number - 67890
```

Writable Smart Insert 38 : 1 : 1070

Comparators in java

- The Comparator is an interface available in the **java.util** package.
- The java **Comparator** is used to order the objects of user-defined classes.
- The java Comparator can compare two objects from two different classes.
- Using the java Comparator, we can sort the elements based on data members of a class.
- For example, we can sort based on rollNo, age, salary, marks, etc.

The Comparator interface has the following methods.

Method	Description
int compare(Object obj1, Object obj2)	It is used to compares the obj1 with o bj2 .
boolean equals(Object obj)	It is used to check the equity between current object and argueded object.

The Comparator can be used in the following three ways.

- Using a seperate class that implements Comparator interface.
- Using anonymous class.
- Using lamda expression.

Using a seperate class

We use the following steps to use Comparator with a seperate class.

- **Step - 1:** Create the user-defined class.
- **Step - 2:** Create a class that implements Comparator interface.
- **Step - 3:** Implement the comapare() method of Comparator interface inside the above defined class(step - 2).
- **Step - 4:** Create the actual class where we use the Compatator object with sort method of Collections class.
- **Step - 5:** Create the object of Compatator interface using the class created in step - 2.
- **Step - 6:** Call the sort method of Collections class by passing the object created in step - 6.
- **Step - 7:** Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

Example

The screenshot shows the Eclipse IDE with the `StudentCompare.java` file open in the editor. The code defines a `Student` class, a `PercentageComparator` implementing `Comparator<Student>`, and a `StudentCompare` class with a `main` method. The `main` method creates a list of students, sorts them using the comparator, and prints the results. The console on the right shows the output of the program, which is a list of average percentages and names sorted in descending order of percentage.

```
1 import java.util.*;
2
3 class Student{
4     String name;
5     float percentage;
6
7     Student(String name, float percentage){
8         this.name = name;
9         this.percentage = percentage;
10    }
11 }
12
13 class PercentageComparator implements Comparator<Student>{
14     public int compare(Student stud1, Student stud2) {
15         if(stud1.percentage < stud2.percentage)
16             return 1;
17         return -1;
18     }
19 }
20
21 public class StudentCompare{
22     public static void main(String args[]) {
23
24         ArrayList<Student> studList = new ArrayList<Student>();
25
26         studList.add(new Student("Gouthami", 90.61f));
27         studList.add(new Student("Raja", 83.55f));
28         studList.add(new Student("Honey", 85.55f));
29         studList.add(new Student("Teja", 77.56f));
30         studList.add(new Student("Varshith", 80.89f));
31
32         Comparator<Student> com = new PercentageComparator();
33
34         Collections.sort(studList, com);
35
36         System.out.println("Avg % --> Name");
37         System.out.println("-----");
38         for(Student stud:studList) {
39             System.out.println(stud.percentage + " --> " + stud.name);
40         }
41     }
42 }
```

Console Output:

```
<terminated> StudentCompare [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
Avg % --> Name
-----
90.61 --> Gouthami
85.55 --> Honey
83.55 --> Raja
80.89 --> Varshith
77.56 --> Teja
```


Using anonymous class

We use the following steps to use Comparator with anonymous class.

- **Step - 1:** Create the user-defined class.
- **Step - 2:** Create the actual class where we use the Comparator object with sort method of Collections class.
- **Step - 3:** Create the object of Comparator interface using anonymous class and implement compare method of Comparator interface.
- **Step - 4:** Call the sort method of Collections class by passing the object created in step - 3.
- **Step - 5:** Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

Example

The screenshot shows the Eclipse IDE with a Java project named 'JavaCollectionFramework'. The main editor displays the file 'StudentCompare.java' with the following code:

```
1 import java.util.*;
2
3 class Student{
4     String name;
5     float percentage;
6
7     Student(String name, float percentage){
8         this.name = name;
9         this.percentage = percentage;
10    }
11 }
12
13 public class StudentCompare{
14     public static void main(String args[]) {
15
16         ArrayList<Student> studList = new ArrayList<Student>();
17
18         studList.add(new Student("Gouthami", 90.61f));
19         studList.add(new Student("Raja", 83.55f));
20         studList.add(new Student("Honey", 85.55f));
21         studList.add(new Student("Teja", 77.56f));
22         studList.add(new Student("Varshith", 80.89f));
23
24         Comparator<Student> com = new Comparator<Student>() {
25             public int compare(Student stud1, Student stud2) {
26                 if(stud1.percentage < stud2.percentage)
27                     return 1;
28                 return -1;
29             }
30         };
31
32         Collections.sort(studList, com);
33
34         System.out.println("Avg % --> Name");
35         System.out.println("-----");
36         for(Student stud:studList) {
37             System.out.println(stud.percentage + " --> " + stud.name);
38         }
39     }
40 }
```

The console output on the right shows the sorted results:

```
<terminated> StudentCompare [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
Avg % --> Name
-----
90.61 --> Gouthami
85.55 --> Honey
83.55 --> Raja
80.89 --> Varshith
77.56 --> Teja
```

The status bar at the bottom indicates 'Writable', 'Smart Insert', and '30 : 11 [197]'.

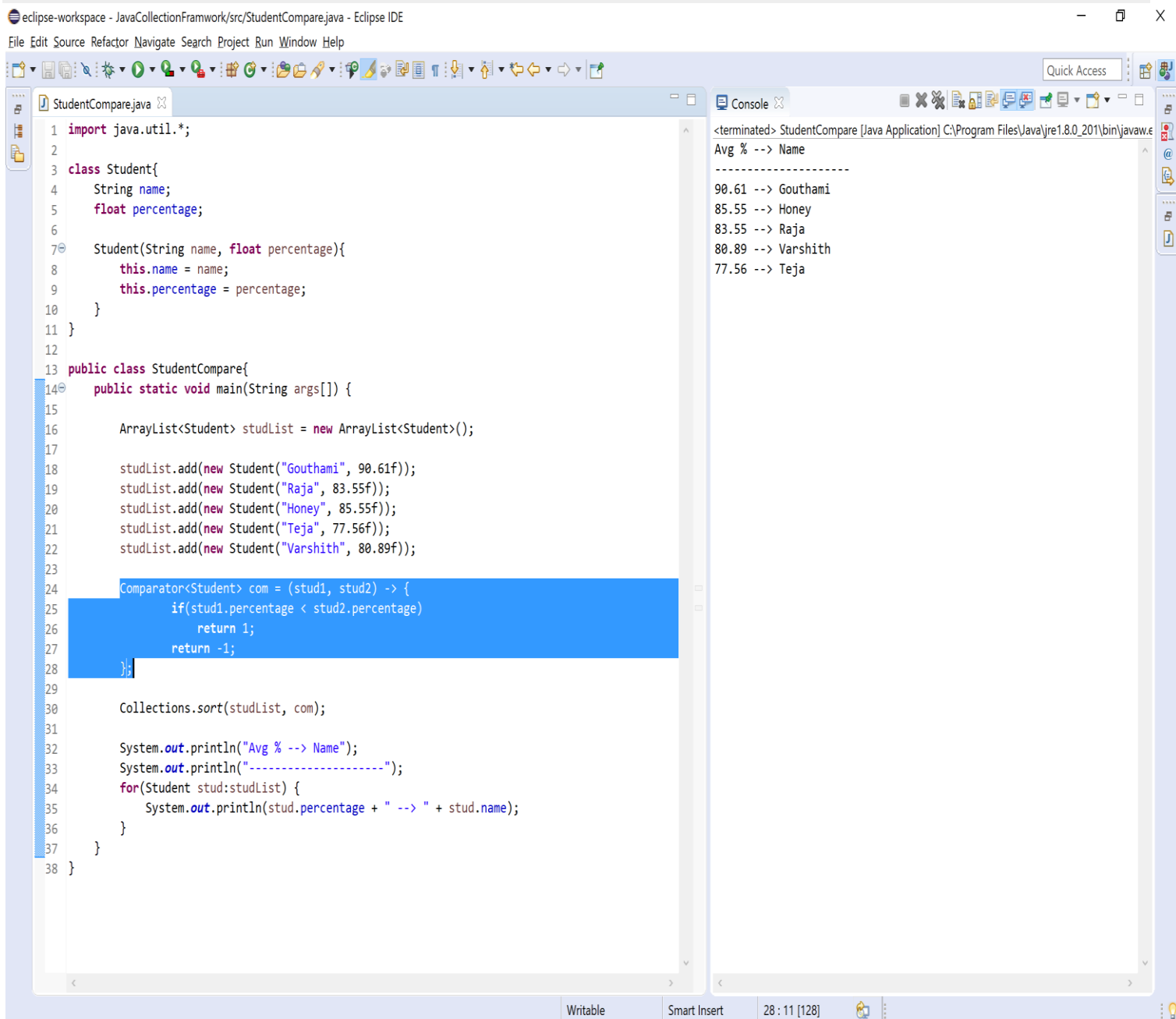
Using lamda expression

We use the following steps to use Comparator with lamda expression.

- **Step - 1:** Create the user-defined class.
- **Step - 2:** Create the actual class where we use the Comparator object with sort method of Collections class.
- **Step - 3:** Create the object of Comparator interface using lamda expression and implement the code for compare method of Comparator interface.
- **Step - 4:** Call the sort method of Collections class by passing the object created in step - 6.
- **Step - 5:** Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

Example



```
eclipse-workspace - JavaCollectionFramework/src/StudentCompare.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

StudentCompare.java
1 import java.util.*;
2
3 class Student{
4     String name;
5     float percentage;
6
7     Student(String name, float percentage){
8         this.name = name;
9         this.percentage = percentage;
10    }
11 }
12
13 public class StudentCompare{
14     public static void main(String args[]) {
15
16         ArrayList<Student> studList = new ArrayList<Student>();
17
18         studList.add(new Student("Gouthami", 90.61f));
19         studList.add(new Student("Raja", 83.55f));
20         studList.add(new Student("Honey", 85.55f));
21         studList.add(new Student("Teja", 77.56f));
22         studList.add(new Student("Varshith", 80.89f));
23
24         Comparator<Student> com = (stud1, stud2) -> {
25             if(stud1.percentage < stud2.percentage)
26                 return 1;
27             return -1;
28         };
29
30         Collections.sort(studList, com);
31
32         System.out.println("Avg % --> Name");
33         System.out.println("-----");
34         for(Student stud:studList) {
35             System.out.println(stud.percentage + " --> " + stud.name);
36         }
37     }
38 }
```

```
<terminated> StudentCompare [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.e
Avg % --> Name
-----
90.61 --> Gouthami
85.55 --> Honey
83.55 --> Raja
80.89 --> Varshith
77.56 --> Teja
```

Writable Smart Insert 28 : 11 [128]

Collection algorithms in java

- The java collection framework defines several algorithms as static methods that can be used with collections and map objects.
- All the collection algorithms in the java are defined in a class called **Collections** which defined in the **java.util** package.
- All these algorithms are highly efficient and make coding very easy. It is better to use them than trying to re-implement them.

The collection framework has the following methods as algorithms.

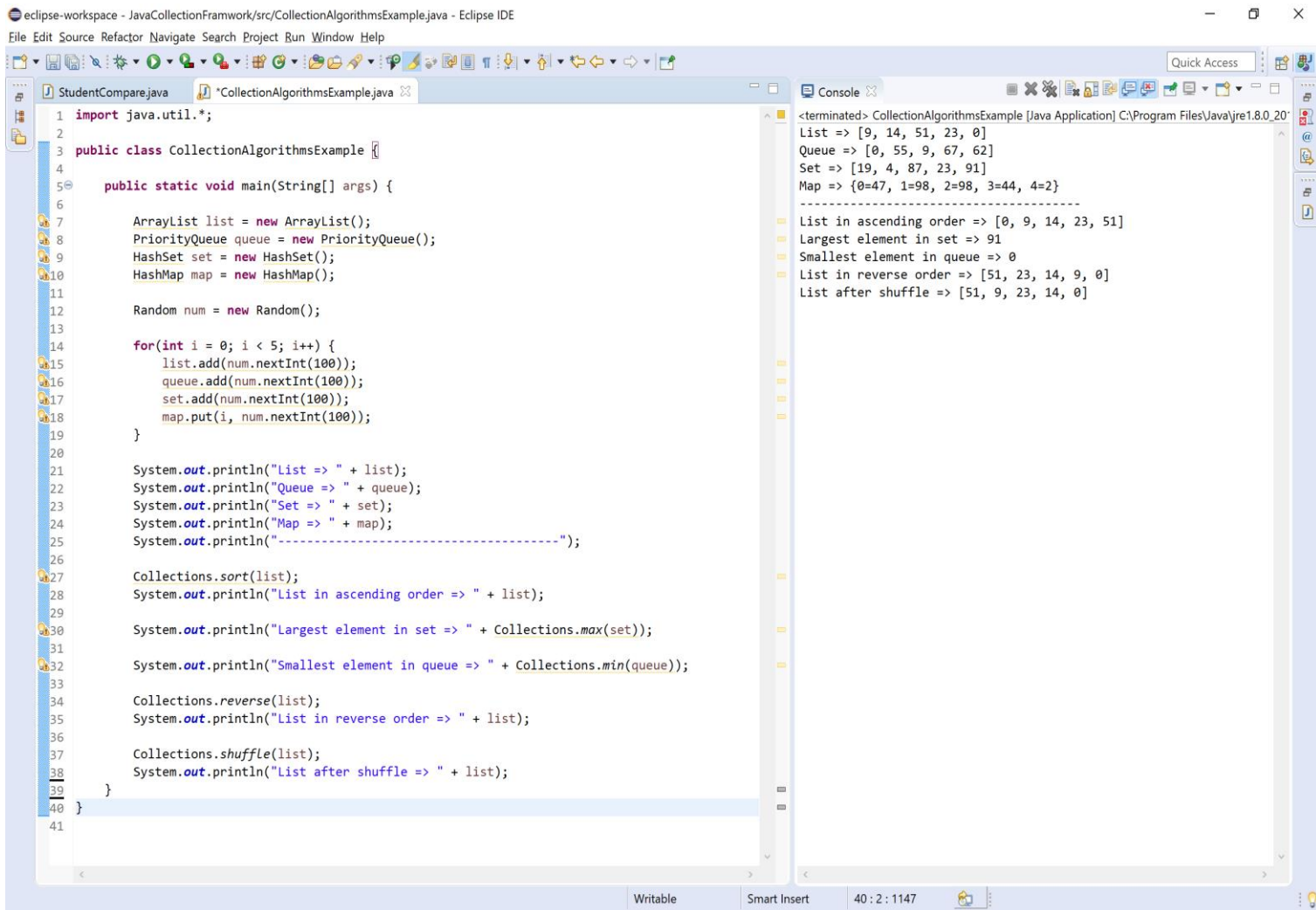
Method	Description
void sort(List list)	Sorts the elements of the list as determined by their natural ordering.
void sort(List list, Comparator comp)	Sorts the elements of the list as determined by Comparator comp.
void reverse(List list)	Reverses all the elements sequence in list.
void rotate(List list, int n)	Rotates list by n places to the right. To rotate left, use a negative value for n.
void shuffle(List list)	Shuffles the elements in list.
void shuffle(List list, Random r)	Shuffles the elements in the list by using r as a source of random numbers.
void copy(List list1, List list2)	Copies the elements of list2 to list1.
List nCopies(int num, Object obj)	Returns num copies of obj contained in an immutable list. num can not be zero or negative.
void swap(List list, int idx1, int idx2)	Exchanges the elements in the list at the indices specified by idx1 and idx2.
int binarySearch(List list, Object value)	Returns the position of value in the list (must be in the sorted order), or -1 if value is not found.

Method	Description
int binarySearch(List list, Object value, Comparator c)	Returns the position of value in the list ordered according to c, or -1 if value is not found.
int indexOfSubList(List list, List subList)	Returns the index of the first match of subList in the list, or -1 if no match is found.
int lastIndexOfSubList(List list, List subList)	Returns the index of the last match of subList in the list, or -1 if no match is found.
Object max(Collection c)	Returns the largest element from the collection c as determined by natural ordering.
Object max(Collection c, Comparator comp)	Returns the largest element from the collection c as determined by Comparator comp.
Object min(Collection c)	Returns the smallest element from the collection c as determined by natural ordering.
Object min(Collection c, Comparator comp)	Returns the smallest element from the collection c as determined by Comparator comp.
void fill(List list, Object obj)	Assigns obj to each element of the list.
boolean replaceAll(List list, Object old, Object new)	Replaces all occurrences of old with new in the list.
Enumeration enumeration(Collection c)	Returns an enumeration over Collection c.
ArrayList list(Enumeration enum)	Returns an ArrayList that contains the elements of enum.
Set singleton(Object obj)	Returns obj as an immutable set.
List singletonList(Object obj)	Returns obj as an immutable list.

Method	Description
Map singletonMap(Object k, Object v)	Returns the key(k)/value(v) pair as an immutable map.
Collection synchronizedCollection(Collection c)	Returns a thread-safe collection backed by c.
List synchronizedList(List list)	Returns a thread-safe list backed by list.
Map synchronizedMap(Map m)	Returns a thread-safe map backed by m.
SortedMap synchronizedSortedMap(SortedMap sm)	Returns a thread-safe SortedMap backed by sm.
Set synchronizedSet(Set s)	Returns a thread-safe set backed by s.
SortedSet synchronizedSortedSet(SortedSet ss)	Returns a thread-safe set backed by ss.
Collection unmodifiableCollection(Collection c)	Returns an unmodifiable collection backed by c.
List unmodifiableList(List list)	Returns an unmodifiable list backed by list.
Set unmodifiableSet(Set s)	Returns an unmodifiable thread-safe set backed by s.
SortedSet unmodifiableSortedSet(SortedSet ss)	Returns an unmodifiable set backed by ss.
Map unmodifiableMap(Map m)	Returns an unmodifiable map backed by m.
SortedMap unmodifiableSortedMap(SortedMap sm)	Returns an unmodifiable SortedMap backed by sm.

Let's consider an example program to illustrate Collections algorithms.

Example



Arrays class in java

- The java collection framework has a class Arrays that provides methods for creating dynamic array and perform various operations like search, asList, compare, etc.
- The Arrays class in java is defined in the **java.util** package. All the methods defined by Arrays class are static methods.

The Arrays class in java has the following methods.

Method	Description
List<T> asList(T[] arr)	It returns a fixed-size list backed by the specified Arrays.
int binarySearch(T[] arr, element)	It searches for the specified element in the array with the help of Binary Search algorithm, and returns the position.
int binarySearch(T[] arr, int fromIndex, int	It searches a range of the specified array for the specified object

Method	Description
toIndex, T key, Comparator c)	using the binary search algorithm.
T[] copyOf(T[] originalArr, int newLength)	It copies the specified array, truncating or padding with the default value (if necessary) so the copy has the specified length.
T[] copyOfRange(T[] originalArr, int fromIndex, int endIndex)	It copies the specified range of the specified array into a new Arrays.
boolean equals(T[] arr1, T[] arr2)	It returns true if the two specified arrays of booleans are equal to one another, otherwise retruns false.
boolean deepEquals(T[] arr1, T[] arr2)	It returns true if the two specified arrays of booleans are deeply equal to one another, otherwise retruns false (it compares including nested arrays).
int hashCode(T[] arr)	It returns the hash code for the specified array.
int deepHashCode(T[] arr)	It returns the hash code for the specified array including nested arrays.
String toString(T[] arr)	It Returns a string representation of the contents of the specified array.
String deepToString(T[] arr)	It Returns a string representation of the contents of the specified array including nested arrays.
void fill(T[] arr, T value)	It assigns the specified value to each element of the specified array.
void fill(T[] arr, int fromIndex, int toIndex, T value)	It assigns the specified value to each element of the specified range of the specified array. The range to be filled extends from fromIndex, inclusive, to toIndex, exclusive.
void parallelPrefix(T[] arr, BinaryOperator o)	It Cumulates, in parallel, each element of the given array in place, using the supplied function.

Method	Description
<code>void setAll(T[] arr, FunctionGenerator)</code>	It sets all elements of the specified array, using the provided generator function to compute each element.
<code>void parallelSetAll(T[] arr, FunctionGenerator)</code>	It Sets all elements of the specified array, in parallel, using the provided generator function to compute each element.
<code>void sort(T[] arr)</code>	It sorts the specified array into ascending order.
<code>void parallelSort(T[] arr)</code>	It sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
<code>Of<T> spliterator(T[] arr)</code>	It returns a <code>Spliterator.Of<T></code> covering all of the specified array.
<code>Stream<T> stream(T[] arr)</code>	It returns a sequential <code>Stream</code> with the specified array as its source.

Let's consider an example program to illustrate methods of Arrays class.

Example

```

eclipse-workspace - JavaCollectionFramework/src/ArraysClassExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

1 import java.util.*;
2
3 public class ArraysClassExample {
4
5     public static void main(String[] args) {
6
7         int[] arr1 = {10, 3, 50, 7, 30, 66, 28, 54, 42};
8
9         int[] arr2 = {67, 2, 54, 67, 13, 56, 98};
10
11         System.out.print("Array1 => ");
12         for(int i:arr1)
13             System.out.print(i + ", ");
14         System.out.print("\nArray2 => ");
15         for(int i:arr2)
16             System.out.print(i + ", ");
17         System.out.println("\n-----");
18
19         System.out.println("Array1 as List => " + Arrays.asList(arr1));
20
21         System.out.println("Position of 30 in Array1 => " + Arrays.binarySearch(arr1, 30));
22
23         System.out.println("equity of array1 and array2 => " + Arrays.equals(arr1, arr2));
24
25         System.out.println("Hash code of Array1 => " + Arrays.hashCode(arr1));
26
27         Arrays.fill(arr1, 15);
28         System.out.print("fill Array1 with 15 => ");
29         for(int i:arr1)
30             System.out.print(i + ", ");
31
32         Arrays.sort(arr2);
33         System.out.print("\nArray2 in sorted order => ");
34         for(int i:arr2)
35             System.out.print(i + ", ");
36     }
37 }
38

```

```

<terminated> ArraysClassExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\jav
Array1 => 10, 3, 50, 7, 30, 66, 28, 54, 42,
Array2 => 67, 2, 54, 67, 13, 56, 98,
-----
Array1 as List => [[I@15db9742]
Position of 30 in Array1 => 4
equity of array1 and array2 => false
Hash code of Array1 => 1497890365
fill Array1 with 15 => 15, 15, 15, 15, 15, 15, 15, 15, 15,
Array2 in sorted order => 2, 13, 54, 56, 67, 67, 98,

```

Writable Smart Insert 38 : 1 : 1048

Dictionary class in java

- In java, the package **java.util** contains a class called **Dictionary** which works like a Map.
- The Dictionary is an abstract class used to store and manage elements in the form of a pair of key and value.
- The Dictionary stores data as a pair of key and value. In the dictionary, each key associates with a value.
- We can use the key to retrieve the value back when needed.

❑ The Dictionary class is no longer in use, it is obsolete.

❑ As Dictionary is an abstract class we can not create its object. It needs a child class like Hashtable.

The Dictionary class in java has the following methods.

S. No.	Methods with Description
1	Dictionary() It's a constructor.
2	Object put(Object key, Object value) Inserts a key and its value into the dictionary. Returns null on success; returns the previous value associated with the key if the key is already exist.
3	Object remove(Object key) It returns the value associated with given key and removes the same; Returns null if the key does not exist.
4	Object get(Object key) It returns the value associated with given key; Returns null if the key does not exist.
5	Enumeration keys() Returns an enumeration of the keys contained in the dictionary.
6	Enumeration elements() Returns an enumeration of the values contained in the dictionary.
7	boolean isEmpty() It returns true if dictionary has no elements; otherwise returns false.
8	int size() It returns the total number of elements in the dictionary.

Let's consider an example program to illustrate methods of Dictionary class.

Example

The screenshot shows the Eclipse IDE with a Java project named 'DictionaryExample.java'. The code defines a `Dictionary` class that uses `Hashtable` internally. It includes methods for adding elements, iterating over keys and values, retrieving values by key, and checking the size and emptiness of the dictionary. The console output shows the execution results, including the keys and values stored in the dictionary.

```
1 import java.util.*;
2 public class DictionaryExample {
3     public static void main(String args[]) {
4
5         Dictionary dict = new Hashtable();
6
7         dict.put(1, "Rama");
8         dict.put(2, "Seetha");
9         dict.put(3, "Heyansh");
10        dict.put(4, "Varshith");
11        dict.put(5, "Manutej");
12        System.out.println("Dictionary\n=> " + dict);
13
14        // keys()
15        System.out.println("\nKeys in Dictionary\n=> ");
16        for (Enumeration i = dict.keys(); i.hasMoreElements(); )
17        {
18            System.out.print(" " + i.nextElement());
19        }
20
21        // elements()
22        System.out.println("\n\nValues in Dictionary\n=> ");
23        for (Enumeration i = dict.elements(); i.hasMoreElements(); )
24        {
25            System.out.print(" " + i.nextElement());
26        }
27
28        //get()
29        System.out.println("\n\nValue associated with key 3 => " + dict.get(3));
30        System.out.println("Value associated with key 30 => " + dict.get(30));
31
32        //size()
33        System.out.println("\nDictionary has " + dict.size() + " elements");
34
35        //isEmpty()
36        System.out.println("\nIs Dictionary empty? " + dict.isEmpty());
37    }
38 }
```

Console Output:

```
<terminated> DictionaryExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw
Dictionary
=> {5=Manutej, 4=Varshith, 3=Heyansh, 2=Seetha, 1=Rama}

Keys in Dictionary
=> 5 4 3 2 1

Values in Dictionary
=> Manutej Varshith Heyansh Seetha Rama

Value associated with key 3 => Heyansh
Value associated with key 30 => null

Dictionary has 5 elements

Is Dictionary empty? false
```

Hashtable class in java

- In java, the package `java.util` contains a class called `Hashtable` which works like a `HashMap` but it is synchronized.
- The `Hashtable` is a concrete class of `Dictionary`.
- It is used to store and manage elements in the form of a pair of key and value.
- The `Hashtable` stores data as a pair of key and value. In the `Hashtable`, each key associates with a value.
- Any non-null object can be used as a key or as a value.
- We can use the key to retrieve the value back when needed.

- ☐ The `Hashtable` class is no longer in use, it is obsolete. The alternate class is `HashMap`.
- ☐ The `Hashtable` class is a concrete class of `Dictionary`.
- ☐ The `Hashtable` class is synchronized.
- ☐ The `Hashtable` does not allow null key or value.
- ☐ The `Hashtable` has the initial default capacity 11.

The `Hashtable` class in java has the following constructors.

S. No.	Constructor with Description
1	Hashtable() It creates an empty hashtable with the default initial capacity 11.
2	Hashtable(int capacity) It creates an empty hashtable with the specified initial capacity.
3	Hashtable(int capacity, float loadFactor) It creates an empty hashtable with the specified initial capacity and loading factor.
4	Hashtable(Map m) It creates a hashtable containing elements of Map m.

The Hashtable class in java has the following methods.

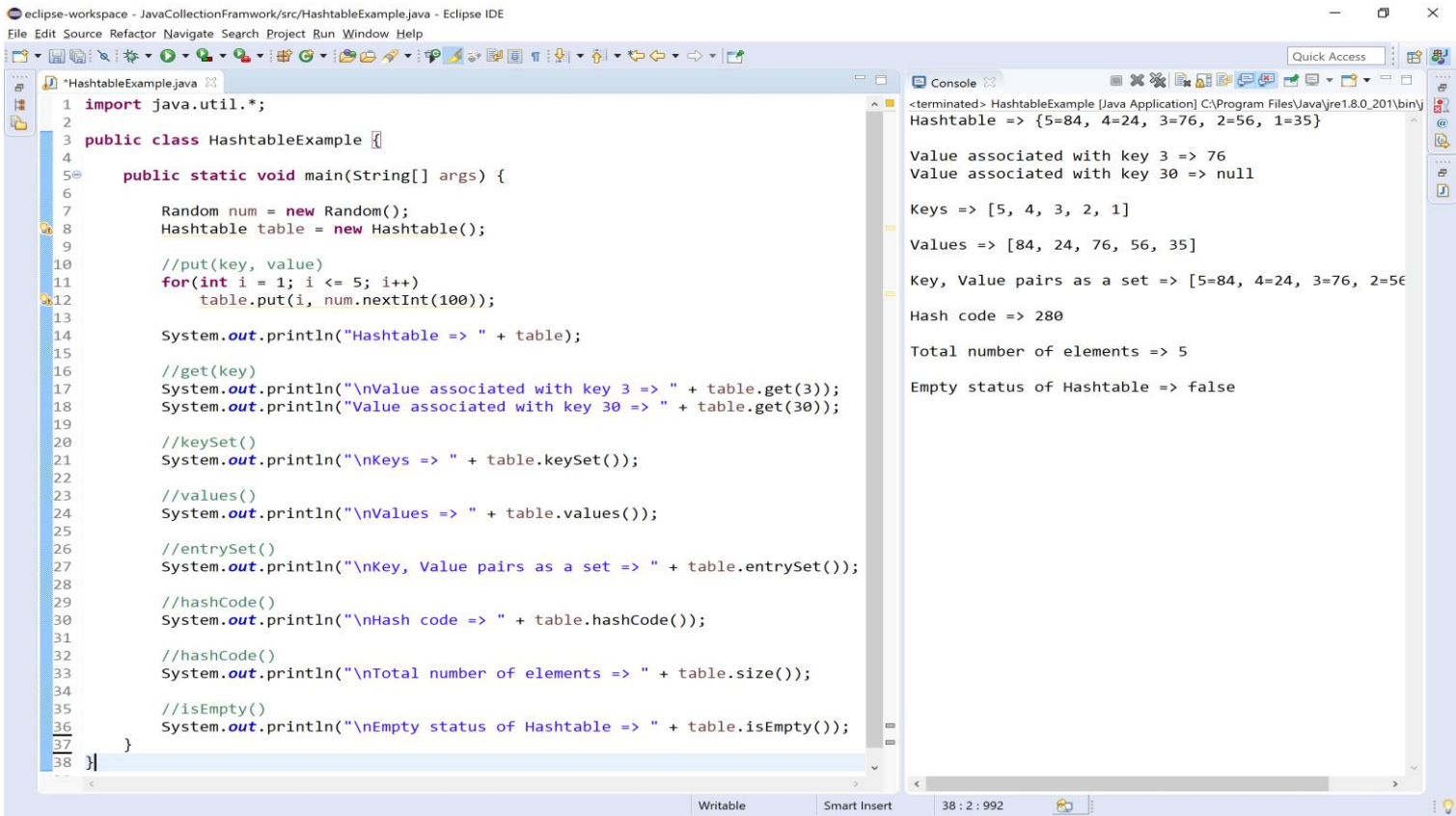
S. No.	Methods with Description
1	V put(K key, V value) It inserts the specified key and value into the hash table.
2	void putAll(Map m) It inserts all the elements of Map m into the invoking Hashtable.
3	V putIfAbsent(K key, V value) If the specified key is not already associated with a value associates it with the given value and returns null, else returns the current value.
4	V getOrDefault(Object key, V defaultValue) It returns the value associated with given key; or defaultValue if the hashtable contains no mapping for the key.
5	V get(Object key) It returns the value associated with the given key.
6	Enumeration keys() Returns an enumeration of the keys of the hashtable.
7	Set keySet() Returns a set view of the keys of the hashtable.

S. No.	Methods with Description
8	Collection values() It returns a collection view of the values contained in the Hashtable.
9	Enumeration elements() Returns an enumeration of the values of the hashtable.
10	Set entrySet() It returns a set view of the mappings contained in the hashtable.
11	int hashCode() It returns the hash code of the hashtable.
12	Object clone() It returns a shallow copy of the Hashtable.
13	V remove(Object key) It returns the value associated with given key and removes the same.
14	boolean remove(Object key, Object value) It removes the specified values with the associated specified keys from the hashtable.
15	boolean contains(Object value) It returns true if the specified value found within the hash table, else return false.
16	boolean containsValue(Object value) It returns true if the specified value found within the hash table, else return false.
17	boolean containsKey(Object key) It returns true if the specified key found within the hash table, else return false.
18	V replace(K key, V value) It replaces the specified value for a specified key.
19	boolean replace(K key, V oldValue, V newValue) It replaces the old value with the new value for a specified key.
20	void replaceAll(BiFunction function) It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
21	void rehash()

S. No.	Methods with Description
	It is used to increase the size of the hash table and rehashes all of its keys.
22	String toString() It returns a string representation of the Hashtable object.
23	V merge(K key, V value, BiFunction remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
24	void forEach(BiConsumer action) It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
25	boolean isEmpty() It returns true if Hashtable has no elements; otherwise returns false.
26	int size() It returns the total number of elements in the Hashtable.
27	void clear() It is used to remove all the lements of a Hashtable.
28	boolean equals(Object o) It is used to compare the specified Object with the Hashtable.

Let's consider an example program to illustrate methods of Hashtable class.

Example



```
1 import java.util.*;
2
3 public class HashtableExample {
4
5     public static void main(String[] args) {
6
7         Random num = new Random();
8         Hashtable table = new Hashtable();
9
10        //put(key, value)
11        for(int i = 1; i <= 5; i++)
12            table.put(i, num.nextInt(100));
13
14        System.out.println("Hashtable => " + table);
15
16        //get(key)
17        System.out.println("\nValue associated with key 3 => " + table.get(3));
18        System.out.println("Value associated with key 30 => " + table.get(30));
19
20        //keySet()
21        System.out.println("\nKeys => " + table.keySet());
22
23        //values()
24        System.out.println("\nValues => " + table.values());
25
26        //entrySet()
27        System.out.println("\nKey, Value pairs as a set => " + table.entrySet());
28
29        //hashCode()
30        System.out.println("\nHash code => " + table.hashCode());
31
32        //hashCode()
33        System.out.println("\nTotal number of elements => " + table.size());
34
35        //isEmpty()
36        System.out.println("\nEmpty status of Hashtable => " + table.isEmpty());
37    }
38 }
```

Console Output:

```
<terminated> HashtableExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\
Hashtable => {5=84, 4=24, 3=76, 2=56, 1=35}

Value associated with key 3 => 76
Value associated with key 30 => null

Keys => [5, 4, 3, 2, 1]

Values => [84, 24, 76, 56, 35]

Key, Value pairs as a set => [5=84, 4=24, 3=76, 2=56, 1=35]

Hash code => 280

Total number of elements => 5

Empty status of Hashtable => false
```

Properties class in java

- In java, the package **java.util** contains a class called **Properties** which is a child class of Hashtable class.
- It implements interfaces like Map, Cloneable, and Serializable.
- Java has this built-in class Properties which allow us to save and load multiple values from a file.
- This makes the class extremely useful for accessing data related to configuration.
- The Properties class used to store configuration values managed as key, value pairs.
- In each pair, both key and value are String values. We can use the key to retrieve the value back when needed.
- The Properties class provides methods to get data from the properties file and store data into the properties file.
- It can also be used to get the properties of a system.

- ❑ The Properties class is child class of Hashtable class.
- ❑ The Properties class implements Map, Cloneable, and Serializable interfaces.
- ❑ The Properties class used to store configuration values.
- ❑ The Properties class stores the data as key, value pairs.
- ❑ In Properties class both key and value are String data type.
- ❑ Using Properties class, we can load key, value pairs into a Properties object from a stream.
- ❑ Using Properties class, we can save the Properties object to a stream.

The Properties class in java has the following constructors.

S. No.	Constructor with Description
1	Properties() It creates an empty property list with no default values.
2	Properties(Properties defaults) It creates an empty property list with the specified defaults.

The Properties class in java has the following methods.

S.No.	Methods with Description
1	void load(Reader r) It loads data from the Reader object.
2	void load(InputStream is) It loads data from the InputStream object.
3	void store(Writer w, String comment) It writes the properties in the writer object.
4	void store(OutputStream os, String comment) It writes the properties in the OutputStream object.
5	String getProperty(String key) It returns value associated with the specified key.
6	String getProperty(String key, String defaultValue) It returns the value associated with given key; or defaultValue if the Properties contains no mapping for the key.
7	void setProperty(String key, String value) It calls the put method of Hashtable.
8	Enumeration propertyNames() It returns an enumeration of all the keys from the property list.
9	Set stringPropertyNames() Returns a set view of the keys of the Properties.
10	void list(PrintStream out)

S.No. Methods with Description

It is used to print the property list out to the specified output stream.

11 **void loadFromXML(InputStream in)**

It is used to load all of the properties represented by the XML document on the specified input stream into this properties table.

12 **void storeToXML(OutputStream os, String comment)**

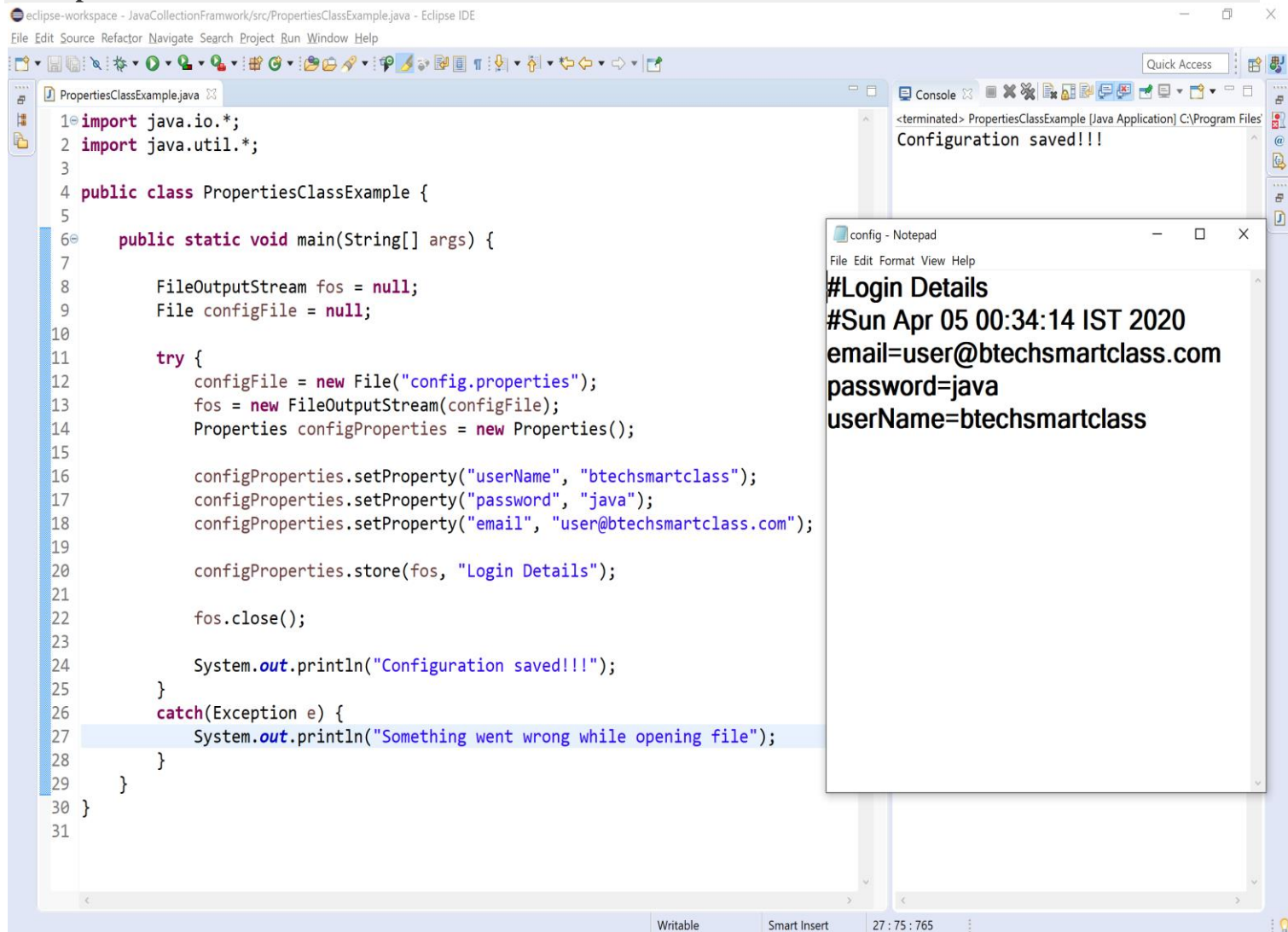
It writes the properties in the writer object for generating XML document.

13 **void storeToXML(Writer w, String comment, String encoding)**

It writes the properties in the writer object for generating XML document with the specified encoding.

Let's consider an example program to illustrate methods of Properties class to store a user configuration details to a properties file.

Example



```
eclipse-workspace - JavaCollectionFramework/src/PropertiesClassExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

PropertiesClassExample.java
1 import java.io.*;
2 import java.util.*;
3
4 public class PropertiesClassExample {
5
6     public static void main(String[] args) {
7
8         FileOutputStream fos = null;
9         File configFile = null;
10
11         try {
12             configFile = new File("config.properties");
13             fos = new FileOutputStream(configFile);
14             Properties configProperties = new Properties();
15
16             configProperties.setProperty("userName", "btechsmartclass");
17             configProperties.setProperty("password", "java");
18             configProperties.setProperty("email", "user@btechsmartclass.com");
19
20             configProperties.store(fos, "Login Details");
21
22             fos.close();
23
24             System.out.println("Configuration saved!!!");
25         }
26         catch(Exception e) {
27             System.out.println("Something went wrong while opening file");
28         }
29     }
30 }
31
```

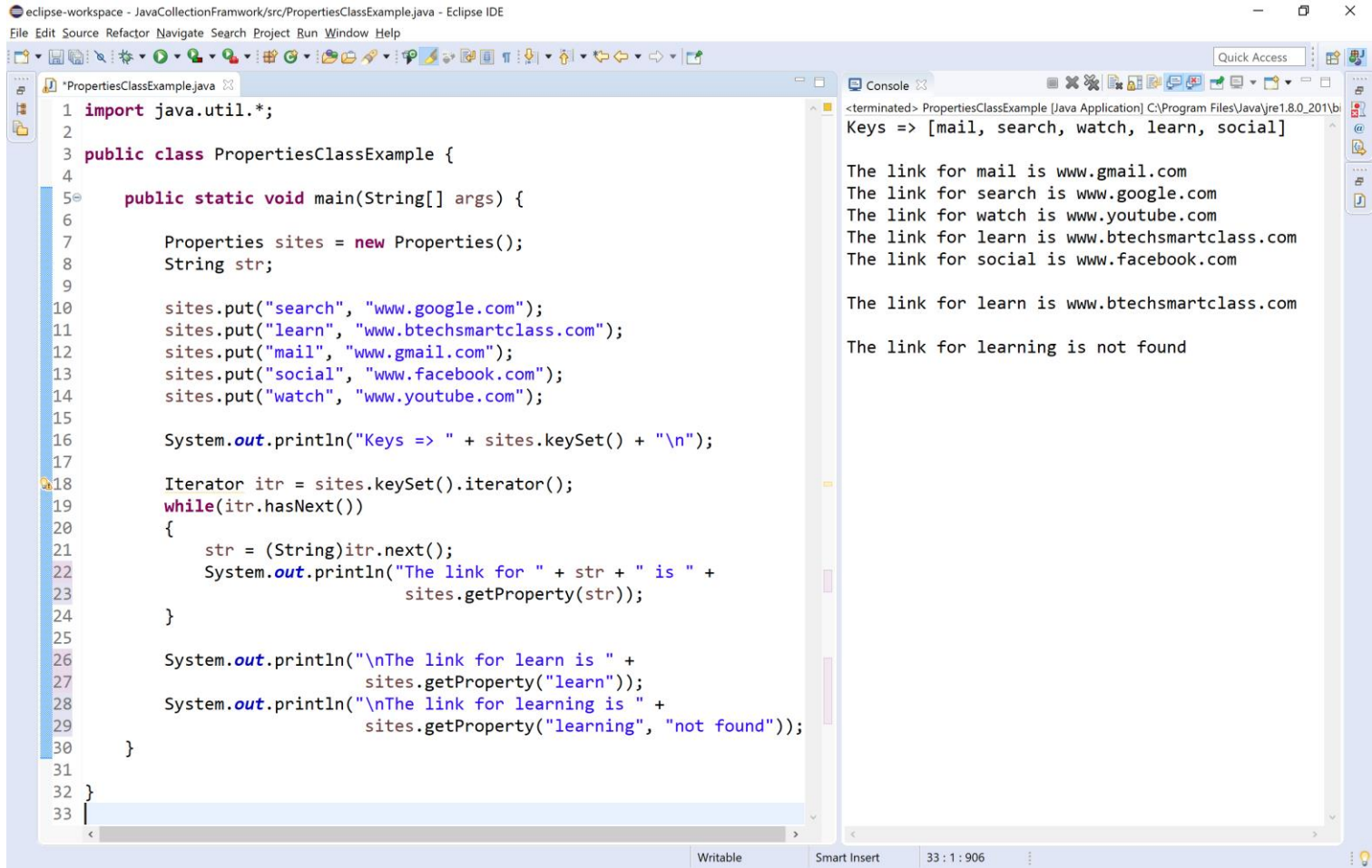
Console: <terminated> PropertiesClassExample [Java Application] C:\Program Files\... Configuration saved!!!

config - Notepad
File Edit Format View Help
#Login Details
#Sun Apr 05 00:34:14 IST 2020
email=user@btechsmartclass.com
password=java
userName=btechsmartclass

Writable Smart Insert 27 : 75 : 765

Let's consider another example program to illustrate methods of Properties class using console.

Example



The screenshot shows the Eclipse IDE with a Java file named `PropertiesClassExample.java`. The code defines a `PropertiesClassExample` class with a `main` method. It uses the `Properties` class to store key-value pairs for different links. The console output shows the keys, the links for 'mail', 'search', 'watch', and 'social', and a message indicating that the link for 'learning' is not found.

```
1 import java.util.*;
2
3 public class PropertiesClassExample {
4
5     public static void main(String[] args) {
6
7         Properties sites = new Properties();
8         String str;
9
10        sites.put("search", "www.google.com");
11        sites.put("learn", "www.btechsmartclass.com");
12        sites.put("mail", "www.gmail.com");
13        sites.put("social", "www.facebook.com");
14        sites.put("watch", "www.youtube.com");
15
16        System.out.println("Keys => " + sites.keySet() + "\n");
17
18        Iterator itr = sites.keySet().iterator();
19        while(itr.hasNext())
20        {
21            str = (String)itr.next();
22            System.out.println("The link for " + str + " is " +
23                               sites.getProperty(str));
24        }
25
26        System.out.println("\nThe link for learn is " +
27                             sites.getProperty("learn"));
28        System.out.println("\nThe link for learning is " +
29                             sites.getProperty("learning", "not found"));
30    }
31
32 }
33
```

Console Output:

```
<terminated> PropertiesClassExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin
Keys => [mail, search, watch, learn, social]

The link for mail is www.gmail.com
The link for search is www.google.com
The link for watch is www.youtube.com
The link for learn is www.btechsmartclass.com
The link for social is www.facebook.com

The link for learn is www.btechsmartclass.com

The link for learning is not found
```

Stack class in java

- In java, the package **java.util** contains a class called **Stack** which is a child class of Vector class.
- It implements the standard principle Last-In-First-Out of stack data structure.
- The Stack has push method for insertion and pop method for deletion. It also has other utility methods.

□ In Stack, the elements are added to the top of the stack and removed from the top of the stack.

The Stack class in java has the following constructor.

S. No.	Constructor with Description
1	Stack() It creates an empty Stack.

The Stack class in java has the following methods.

S.No.	Methods with Description
1	Object push(Object element) It pushes the element onto the stack and returns the same.
2	Object pop() It returns the element on the top of the stack and removes the same.
3	int search(Object element) If element found, it returns offset from the top. Otherwise, -1 is returned.
4	Object peek() It returns the element on the top of the stack.
5	boolean empty() It returns true if the stack is empty, otherwise returns false.

Let's consider an example program to illustrate methods of Stack class.

Example

```

eclipse-workspace - JavaCollectionFramework/src/StackClassExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

1 import java.util.*;
2
3 public class StackClassExample {
4
5     public static void main(String[] args) {
6
7         Stack stack = new Stack();
8
9         Random num = new Random();
10
11         for(int i = 0; i < 5; i++)
12             stack.push(num.nextInt(100));
13
14         System.out.println("Stack elements => " + stack);
15
16         System.out.println("Top element is " + stack.peek());
17
18         System.out.println("Removed element is " + stack.pop());
19
20         System.out.println("Element 50 availability => " + stack.search(50));
21
22         System.out.println("Stack is empty? - " + stack.isEmpty());
23
24     }
25
26 }
27

```

```

<terminated> StackClassExample [Java Application] C:\Program Files\Java\jre1.8.0_
Stack elements => [35, 54, 77, 22, 64]
Top element is 64
Removed element is 64
Element 50 availability => -1
Stack is empty? - false

```

Task View Writable Smart Insert 26 : 2 : 571

Vector class in java

- In java, the package **java.util** contains a class called **Vector** which implements the **List** interface.
- The Vector is similar to an ArrayList. Like ArrayList Vector also maintains the insertion order.
- But Vector is synchronized, due to this reason, it is rarely used in the non-thread application. It also leads to poor performance.

☐ The Vector is a class in the java.util package.

☐ The Vector implements List interface.

☐ The Vector is a legacy class.

☐ The Vector is synchronized.

The Vector class in java has the following constructor.

S. No.	Constructor with Description
1	Vector() It creates an empty Vector with default initial capacity of 10.
2	Vector(int initialSize) It creates an empty Vector with specified initial capacity.
3	Vector(int initialSize, int incr) It creates a vector whose initial capacity is specified by size and whose increment is specified by incr.
4	Vector(Collection c) It creates a vector that contains the elements of collection c.

The Vector class in java has the following methods.

S.No.	Methods with Description
1	boolean add(Object o) It appends the specified element to the end of this Vector.
2	void add(int index, Object element) It inserts the specified element at the specified position in this Vector.
3	void addElement(Object obj) Adds the specified object to the end of the vector, increasing its size by one.
4	boolean addAll(Collection c)

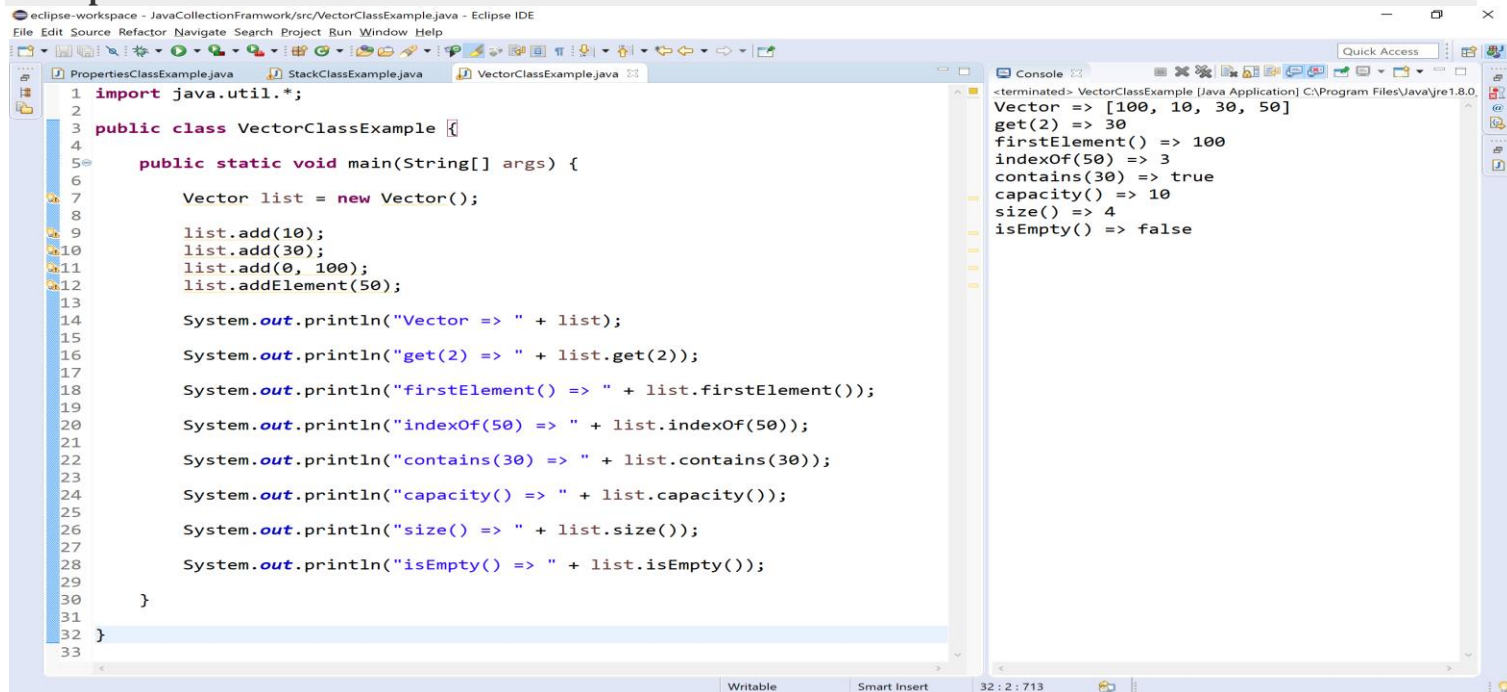
S.No.	Methods with Description
	It appends all of the elements in the specified Collection to the end of the Vector.
5	boolean addAll(int index, Collection c) It inserts all of the elements in in the specified Collection into the Vector at the specified position.
6	Object set(int index, Object element) It replaces the element at the specified position in the vector with the specified element.
7	void setElementAt(Object obj, int index) It sets the element at the specified index of the vector to be the specified object.
8	Object remove(int index) It removes the element at the specified position in the vector.
9	boolean remove(Object o) It removes the first occurrence of the specified element in the vector.
10	boolean removeElement(Object obj) It removes the first occurrence of the specified element in the vector.
11	void removeElementAt(int index) It removes the element at specified index in the vector.
12	void removeRange(int fromIndex, int toIndex) It removes from the Vector all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
13	boolean removeAll(Collection c) It removes from the vector all of its elements that are contained in the specified Collection.
14	void removeAllElements() It removes all the elements from the vector.
15	boolean retainAll(Collection c) It removes all the elements from the vector except elements those are in the given collection.
16	Object elementAt(int index) It returns the element at specified index in the Vector.
17	Object get(int index) It returns the element at specified index in the Vector.

S.No.	Methods with Description
18	Enumeration elements() It returns the Enumeration of all the elements of the Vector.
19	Object firstElement() It returns the first element of the Vector.
20	Object lastElement() It returns the last element of the Vector.
21	int indexOf(Object element) It returns the index value of the first occurrence of the given element in the Vector.
22	int indexOf(Object elem, int index) It returns the index value of the first occurrence of the given element, search beginning at specified index in the Vector.
23	int lastIndexOf(Object element) It returns the index value of the last occurrence of the given element, search beginning at specified index in the Vector.
24	List subList(int fromIndex, int toIndex) It returns a list containing elements fromIndex to toIndex in the Vector.
25	int capacity() It returns the current capacity of the Vector.
26	void clear() It removes all the elements from the Vector.
27	Object clone() It returns a clone of the Vector.
28	boolean contains(Object element) It returns true if element found in the Vector, otherwise returns false.
29	boolean containsAll(Collection c) It returns true if all the elements of given collection found in the Vector, otherwise returns false.
30	void ensureCapacity(int minCapacity) It increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

S.No.	Methods with Description
31	boolean equals(Object o) It compares the specified Object with this vector for equality.
32	int hashCode() It returns the hash code of the Vector.
33	boolean isEmpty() It returns true if Vector has no elements, otherwise returns false.
34	void setSize(int newSize) It sets the size of the vector.
35	int size() It returns total number of elements in the vector.
36	Object[] toArray() It returns an array containing all the elements of the Vector.
37	String toString() It returns a string representation of the Vector.
38	void trimToSize() It trims the capacity of the vector to be the vector's current size.

Let's consider an example program to illustrate methods of Vector class.

Example



```

1 import java.util.*;
2
3 public class VectorClassExample {
4
5     public static void main(String[] args) {
6
7         Vector list = new Vector();
8
9         list.add(10);
10        list.add(30);
11        list.add(0, 100);
12        list.addElement(50);
13
14        System.out.println("Vector => " + list);
15
16        System.out.println("get(2) => " + list.get(2));
17
18        System.out.println("firstElement() => " + list.firstElement());
19
20        System.out.println("indexOf(50) => " + list.indexOf(50));
21
22        System.out.println("contains(30) => " + list.contains(30));
23
24        System.out.println("capacity() => " + list.capacity());
25
26        System.out.println("size() => " + list.size());
27
28        System.out.println("isEmpty() => " + list.isEmpty());
29
30    }
31 }
32
33

```

```

<terminated> VectorClassExample [Java Application] C:\Program Files\Java\jre1.8.0_
Vector => [100, 10, 30, 50]
get(2) => 30
firstElement() => 100
indexOf(50) => 3
contains(30) => true
capacity() => 10
size() => 4
isEmpty() => false

```

StringTokenizer class in java

- The StringTokenizer is a built-in class in java used to break a string into tokens.
- The StringTokenizer class is available inside the java.util package.
- The StringTokenizer class object internally maintains a current position within the string to be tokenized.

□ A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

The StringTokenizer class in java has the following constructor.

S. No.	Constructor with Description
1	StringTokenizer(String str) It creates StringTokenizer object for the specified string str with default delimiter.
2	StringTokenizer(String str, String delimiter) It creates StringTokenizer object for the specified string str with specified delimiter.
3	StringTokenizer(String str, String delimiter, boolean returnValue) It creates StringTokenizer object with specified string, delimiter and returnValue.

The StringTokenizer class in java has the following methods.

S.No.	Methods with Description
1	String nextToken() It returns the next token from the StringTokenizer object.
2	String nextToken(String delimiter) It returns the next token from the StringTokenizer object based on the delimiter.
3	Object nextElement() It returns the next token from the StringTokenizer object.
4	boolean hasMoreTokens() It returns true if there are more tokens in the StringTokenizer object. otherwise returns false.
5	boolean hasMoreElements() It returns true if there are more tokens in the StringTokenizer object. otherwise returns false.
6	int countTokens() It returns total number of tokens in the StringTokenizer object.

Let's consider an example program to illustrate methods of StringTokenizer class.

Example

The screenshot shows the Eclipse IDE with a Java file named `StringTokenizerExample.java`. The code uses `StringTokenizer` to process a title and a URL. The console output shows the results of the tokenization.

```
1 import java.util.StringTokenizer;
2
3 public class StringTokenizerExample {
4
5     public static void main(String[] args) {
6
7         String url = "www.btechsmartclass.com";
8         String title = "BTech Smart Class";
9
10
11         StringTokenizer tokens = new StringTokenizer(title);
12         StringTokenizer anotherTokens = new StringTokenizer(url, ".");
13
14         System.out.println("\nTotal tokens in title is " + tokens.countTokens());
15
16         System.out.print("Tokens in the title => ");
17         while(tokens.hasMoreTokens()) {
18             System.out.print(tokens.nextToken() + ", ");
19         }
20
21         System.out.println("\n\nTotal tokens in url is " + anotherTokens.countTokens());
22
23         System.out.println("Tokens in the url with delimiter (.) => ");
24         while(anotherTokens.hasMoreElements()) {
25             System.out.print(anotherTokens.nextElement() + ", ");
26         }
27
28     }
29 }
30
31
```

Console Output:

```
<terminated> StringTokenizerExample [Java Application] C:\Program Files\Java\
Total tokens in title is 3
Tokens in the title => BTech, Smart, Class,
Total tokens in url is 3
Tokens in the url with delimiter (.) =>
www, btechsmartclass, com,
Total tokens in title is 0
```

BitSet class in java

- The BitSet is a built-in class in java used to create a dynamic array of bits represented by boolean values.
- The BitSet class is available inside the java.util package.
- The BitSet array can increase in size as needed. This feature makes the BitSet similar to a Vector of bits.

- ❑ The bit values can be accessed by non-negative integers as an index.
- ❑ The size of the array is flexible and can grow to accommodate additional bit as needed.
- ❑ The default value of the BitSet is boolean false with a representation as 0 (off).
- ❑ BitSet uses 1 bit of memory per each boolean value.

The BitSet class in java has the following constructor.

S. No.	Constructor with Description
1	BitSet() It creates a default BitSet object.

S. No.	Constructor with Description
2	BitSet(int noOfBits) It creates a BitSet object with number of bits that it can hold. All bits are initialized to zero.

The BitSet class in java has the following methods.

S.No.	Methods with Description
1	void and(BitSet bitSet) It performs AND operation on the contents of the invoking BitSet object with those specified by bitSet.
2	void andNot(BitSet bitSet) For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	void flip(int index) It reverses the bit at specified index.
4	void flip(int startIndex, int endIndex) It reverses all the bits from specified startIndex to endIndex.
5	void or(BitSet bitSet) It performs OR operation on the contents of the invoking BitSet object with those specified by bitSet.
6	void xor(BitSet bitSet) It performs XOR operation on the contents of the invoking BitSet object with those specified by bitSet.
7	int cardinality() It returns the number of bits set to true in the invoking BitSet.
8	void clear() It sets all the bits to zeros of the invoking BitSet.
9	void clear(int index) It set the bit specified by given index to zero.
10	void clear(int startIndex, int endIndex) It sets all the bits from specified startIndex to endIndex to zero.
11	Object clone() It duplicates the invoking BitSet object.

S.No.	Methods with Description
12	boolean equals(Object bitSet) It retruns true if both invoking and argumented BitSets are equal, otherwise returns false.
13	boolean get(int index) It retruns the present state of the bit at given index in the invoking BitSet.
14	BitSet get(int startIndex, int endIndex) It returns a BitSet object that consists all the bits from startIndex to endIndex.
15	int hashCode() It returns the hash code of the invoking BitSet.
16	boolean intersects(BitSet bitSet) It returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
17	boolean isEmpty() It returns true if all bits in the invoking object are zero, otherwise returns false.
17	int length() It returns the total number of bits in the invoking BitSet.
18	int nextClearBit(int startIndex) It returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex.
19	int nextSetBit(int startIndex) It returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned.
20	void set(int index) It sets the bit specified by index.
21	void set(int index, boolean value) It sets the bit specified by index to the value passed in value.
22	void set(int startIndex, int endIndex) It sets all the bits from startIndex to endIndex.
23	void set(int startIndex, int endIndex, boolean value) It sets all the bits to specified value from startIndex to endIndex.
24	int size()

S.No. Methods with Description

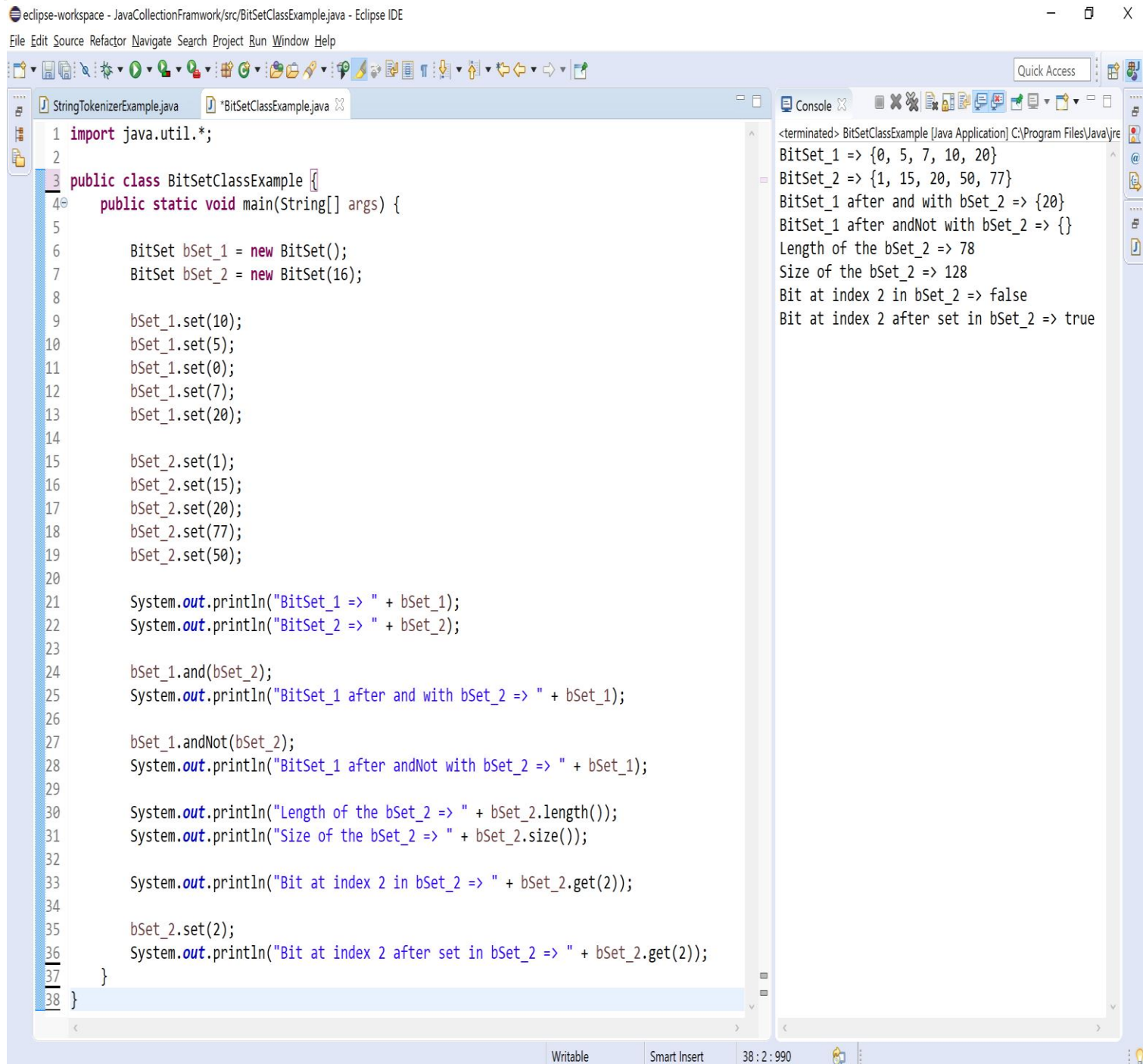
It returns the total number of bits in the invoking BitSet.

25 String toString()

It returns the string equivalent of the invoking BitSet object.

Let's consider an example program to illustrate methods of BitSet class.

Example



```
eclipse-workspace - JavaCollectionFramework/src/BitSetClassExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

1 import java.util.*;
2
3 public class BitSetClassExample {
4     public static void main(String[] args) {
5
6         BitSet bSet_1 = new BitSet();
7         BitSet bSet_2 = new BitSet(16);
8
9         bSet_1.set(10);
10        bSet_1.set(5);
11        bSet_1.set(0);
12        bSet_1.set(7);
13        bSet_1.set(20);
14
15        bSet_2.set(1);
16        bSet_2.set(15);
17        bSet_2.set(20);
18        bSet_2.set(77);
19        bSet_2.set(50);
20
21        System.out.println("BitSet_1 => " + bSet_1);
22        System.out.println("BitSet_2 => " + bSet_2);
23
24        bSet_1.and(bSet_2);
25        System.out.println("BitSet_1 after and with bSet_2 => " + bSet_1);
26
27        bSet_1.andNot(bSet_2);
28        System.out.println("BitSet_1 after andNot with bSet_2 => " + bSet_1);
29
30        System.out.println("Length of the bSet_2 => " + bSet_2.length());
31        System.out.println("Size of the bSet_2 => " + bSet_2.size());
32
33        System.out.println("Bit at index 2 in bSet_2 => " + bSet_2.get(2));
34
35        bSet_2.set(2);
36        System.out.println("Bit at index 2 after set in bSet_2 => " + bSet_2.get(2));
37    }
38 }
```

Console

```
<terminated> BitSetClassExample [Java Application] C:\Program Files\Java\jre
BitSet_1 => {0, 5, 7, 10, 20}
BitSet_2 => {1, 15, 20, 50, 77}
BitSet_1 after and with bSet_2 => {20}
BitSet_1 after andNot with bSet_2 => {}
Length of the bSet_2 => 78
Size of the bSet_2 => 128
Bit at index 2 in bSet_2 => false
Bit at index 2 after set in bSet_2 => true
```

Writable Smart Insert 38 : 2 : 990

Date class in java

- The **Date** is a built-in class in java used to work with date and time in java.
- The Date class is available inside the **java.util** package.
- The Date class represents the date and time with millisecond precision.
- The Date class implements **Serializable**, **Cloneable** and **Comparable** interface.

□ Most of the constructors and methods of Date class has been deprecated after Calendar class introduced.

The Date class in java has the following constructor.

S. No.	Constructor with Description
1	Date() It creates a Date object that represents current date and time.
2	Date(long milliseconds) It creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
3	Date(int year, int month, int date) - Deprecated It creates a date object with the specified year, month, and date.
4	Date(int year, int month, int date, int hrs, int min) - Deprecated It creates a date object with the specified year, month, date, hours, and minutes.
5	Date(int year, int month, int date, int hrs, int min, int sec) - Deprecated It creates a date object with the specified year, month, date, hours, minutes and seconds.
5	Date(String s) - Deprecated It creates a Date object and initializes it so that it represents the date and time indicated by the string s, which is interpreted as if by the parse(java.lang.String) method.

The Date class in java has the following methods.

S.No.	Methods with Description
1	long getTime() It returns the time represented by this date object.
2	boolean after(Date date) It returns true, if the invoking date is after the argumented date.

S.No.	Methods with Description
3	boolean before(Date date) It returns true, if the invoking date is before the argued date.
4	Date from(Instant instant) It returns an instance of Date object from Instant date.
5	void setTime(long time) It changes the current date and time to given time.
6	Object clone() It duplicates the invoking Date object.
7	int compareTo(Date date) It compares current date with given date.
8	boolean equals(Date date) It compares current date with given date for equality.
9	int hashCode() It returns the hash code value of the invoking date object.
10	Instant toInstant() It converts current date into Instant object.
11	String toString() It converts this date into Instant object.

Let's consider an example program to illustrate methods of Date class.

Example

```

eclipse-workspace - JavaCollectionFramework/src/DateClassExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

DateClassExample.java
1 import java.time.Instant;
2 import java.util.Date;
3
4 public class DateClassExample {
5
6     public static void main(String[] args) {
7
8         Date time = new Date();
9
10        System.out.println("Current date => " + time);
11
12        System.out.println("Date => " + time.getTime() + " milliseconds");
13
14        System.out.println("after() => " + time.after(time) + " milliseconds");
15
16        System.out.println("before() => " + time.before(time) + " milliseconds");
17
18        System.out.println("hashCode() => " + time.hashCode());
19
20    }
21
22 }
23

Console
<terminated> DateClassExample [Java Application] C:\Program Files\Java\jre1.8.0_201\
Current date => Tue Apr 07 15:04:16 IST 2020
Date => 1586252056507 milliseconds
after() => false milliseconds
before() => false milliseconds
hashCode() => 1409124042
  
```

Calendar class in java

- The **Calendar** is a built-in abstract class in java used to convert date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc.
- The Calendar class is available inside the **java.util** package.
- The Calendar class implements **Serializable**, **Cloneable** and **Comparable** interface.

❑ As the Calendar class is an abstract class, we can not create an object using it.

❑ We will use the static method **Calendar.getInstance()** to instantiate and implement a sub-class.

The Calendar class in java has the following methods.

S.No.	Methods with Description
1	Calendar getInstance() It returns a calendar using the default time zone and locale.
2	Date getTime() It returns a Date object representing the invoking Calendar's time value.
3	TimeZone getTimeZone() It returns the time zone object associated with the invoking calendar.
4	String getCalendarType() It returns an instance of Date object from Instant date.
5	int get(int field) It returns the value for the given calendar field.
6	int getFirstDayOfWeek() It returns the day of the week in integer form.
7	int getWeeksInWeekYear() It returns the total weeks in week year.
8	int getWeekYear() It returns the week year represented by current Calendar.
9	void add(int field, int amount) It adds the specified amount of time to the given calendar field.
10	boolean after (Object when) It returns true if the time represented by the Calendar is after the time represented by when Object.

S.No.	Methods with Description
11	boolean before(Object when) It returns true if the time represented by the Calendar is before the time represented by when Object.
12	void clear(int field) It sets the given calendar field value and the time value of this Calendar undefined.
13	Object clone() It retruns the copy of the current object.
14	int compareTo(Calendar anotherCalendar) It compares and retruns the time values (millisecond offsets) between two calendar object.
15	void complete() It sets any unset fields in the calendar fields.
16	void computeFields() It converts the current millisecond time value time to calendar field values in fields[].
17	void computeTime() It converts the current calendar field values in fields[] to the millisecond time value time.
18	boolean equals(Object object) It returns true if both invoking object and argumented object are equal.
19	int getActualMaximum(int field) It returns the Maximum possible value of the specified calendar field.
20	int getActualMinimum(int field) It returns the Minimum possible value of the specified calendar field.
21	Set getAvailableCalendarTypes() It returns a string set of all available calendar type supported by Java Runtime Environment.
22	Locale[] getAvailableLocales() It returns an array of all locales available in java runtime environment.
23	String getDisplayName(int field, int style, Locale locale) It returns the String representation of the specified calendar field value in a given style, and local.
24	Map getDisplayNames(int field, int style, Locale locale) It returns Map representation of the given calendar field value in a given style and local.

S.No.	Methods with Description
25	int getGreatestMinimum(int field) It returns the highest minimum value of the specified Calendar field.
26	int getLeastMaximum(int field) It returns the highest maximum value of the specified Calendar field.
27	int getMaximum(int field) It returns the maximum value of the specified calendar field.
28	int getMinimalDaysInFirstWeek() It returns required minimum days in integer form.
29	int getMinimum(int field) It returns the minimum value of specified calendar field.
30	long getTimeInMillis() It returns the current time in millisecond.
31	int hashCode() It returns the hash code of the invoking object.
32	int internalGet(int field) It returns the value of the given calendar field.
33	boolean isLenient() It returns true if the interpretation mode of this calendar is lenient; false otherwise.
34	boolean isSet(int field) If not set then it returns false otherwise true.
35	boolean isWeekDateSupported() It returns true if the calendar supports week date. The default value is false.
36	void roll(int field, boolean up) It increase or decrease the specified calendar field by one unit without affecting the other field
37	void set(int field, int value) It sets the specified calendar field by the specified value.
38	void setFirstDayOfWeek(int value) It sets the first day of the week.

S.No.	Methods with Description
39	void setMinimalDaysInFirstWeek(int value) It sets the minimal days required in the first week.
40	void setTime(Date date) It sets the Time of current calendar object.
41	void setTimeInMillis(long millis) It sets the current time in millisecond.
42	void setTimeZone(TimeZone value) It sets the TimeZone with passed TimeZone value.
43	void setWeekDate(int weekYear, int weekOfYear, int dayOfWeek) It sets the current date with specified integer value.
44	Instant toInstant() It converts the current object to an instant.
45	String toString() It returns a string representation of the current object.

Let's consider an example program to illustrate methods of Calendar class.

Example

```
1 import java.util.*;
2
3 public class CalendarClassExample {
4
5     public static void main(String[] args) {
6
7         Calendar cal = Calendar.getInstance();
8
9         System.out.println("Current date and time : \n=>" + cal);
10        System.out.println("Current Calendar type : " + cal.getCalendarType());
11        System.out.println("Current date and time : \n=>" + cal.getTime());
12        System.out.println("Current date time zone : \n=>" + cal.getTimeZone());
13        System.out.println("Calendar filed 1 (year): " + cal.get(1));
14        System.out.println("Calendar day in integer form: " + cal.getFirstDayOfWeek());
15        System.out.println("Calendar weeks in a year: " + cal.getWeeksInWeekYear());
16        System.out.println("Time in milliseconds: " + cal.getTimeInMillis());
17        System.out.println("Available Calendar types: " + cal.getAvailableCalendarTypes());
18        System.out.println("Calendar hash code: " + cal.hashCode());
19        System.out.println("Is calendar supports week date? " + cal.isWeekDateSupported());
20        System.out.println("Calendar string representation: " + cal.toString());
21    }
22 }
```

Console

<terminated> CalendarClassExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (7 Apr 2020, 16:26:43)

Current date and time :

=>java.util.GregorianCalendar[time=1586257003299,areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Asia/Calcutta",offset=19800000,dstSavings=0,useDaylight=false,transitions=7,lastRule=null]]

Current Calendar type : gregory

Current date and time :

=>Tue Apr 07 16:26:43 IST 2020

Current date time zone :

=>sun.util.calendar.ZoneInfo[id="Asia/Calcutta",offset=19800000,dstSavings=0,useDaylight=false,transitions=7,lastRule=null]

Calendar filed 1 (year): 2020

Calendar day in integer form: 2

Calendar weeks in a year: 53

Time in milliseconds: 1586257003299

Available Calendar types: [gregory, buddhist, japanese]

Random class in java

- The **Random** is a built-in class in java used to generate a stream of pseudo-random numbers in java programming.
- The Random class is available inside the **java.util** package.
- The Random class implements **Serializable**, **Cloneable** and **Comparable** interface.

☐ The **Random** class is a part of java.util package.

☐ The **Random** class provides several methods to generate random numbers of type integer, double, long, float etc.

☐ The **Random** class is thread-safe.

☐ Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.

The Random class in java has the following constructors.

S.No.	Constructor with Description
1	Random() It creates a new random number generator.
2	Random(long seedValue) It creates a new random number generator using a single long seedValue.

The Random class in java has the following methods.

S.No.	Methods with Description
1	int next(int bits) It generates the next pseudo-random number.
2	Boolean nextBoolean() It generates the next uniformly distributed pseudo-random boolean value.
3	double nextDouble() It generates the next pseudo-random double number between 0.0 and 1.0.
4	void nextBytes(byte[] bytes) It places the generated random bytes into an user-supplied byte array.
5	float nextFloat() It generates the next pseudo-random float number between 0.0 and 1.0..

S.No.	Methods with Description
6	int nextInt() It generates the next pseudo-random int number.
7	int nextInt(int n) It generates the next pseudo-random integer value between zero and n.
8	long nextLong() It generates the next pseudo-random, uniformly distributed long value.
9	double nextGaussian() It generates the next pseudo-random Gaussian distributed double number with mean 0.0 and standard deviation 1.0.
10	void setSeed(long seedValue) It sets the seed of the random number generator using a single long seedValue.
11	DoubleStream doubles() It returns a stream of pseudo-random double values, each conforming between 0.0 and 1.0.
12	DoubleStream doubles(double start, double end) It retruns an unlimited stream of pseudo-random double values, each conforming to the given start and end.
13	DoubleStream doubles(long streamSize) It returns a stream producing the pseudo-random double values for the given streamSize number, each between 0.0 and 1.0.
14	DoubleStream doubles(long streamSize, double start, double end) It returns a stream producing the given streamSizenumber of pseudo-random double values, each conforming to the given start and end.
15	IntStream ints() It returns a stream of pseudo-random integer values.
16	IntStream ints(int start, int end) It retruns an unlimited stream of pseudo-random integer values, each conforming to the given start and end.
17	IntStream ints(long streamSize) It returns a stream producing the pseudo-random integer values for the given streamSize number.
18	IntStream ints(long streamSize, int start, int end) It returns a stream producing the given streamSizenumber of pseudo-random integer values, each conforming to the given start and end.

S.No. Methods with Description

- | | |
|----|--|
| 19 | LongStream longs()
It returns a stream of pseudo-random long values. |
| 20 | LongStream longs(long start, long end)
It retruns an unlimited stream of pseudo-random long values, each conforming to the given start and end. |
| 21 | LongStream longs(long streamSize)
It returns a stream producing the pseudo-random long values for the given streamSize number. |
| 22 | LongStream longs(long streamSize, long start, long end)
It returns a stream producing the given streamSizenumber of pseudo-random long values, each conforming to the given start and end. |

Let's consider an example program to illustrate methods of Random class.

Example

The screenshot shows the Eclipse IDE with a Java project named 'JavaCollectionFramework'. The source file 'RandomClassExample.java' is open in the editor. The code imports 'java.util.Random' and defines a 'RandomClassExample' class with a 'main' method. Inside the 'main' method, a 'Random' object is created, and several random values are generated and printed to the console: an integer, an integer from 0 to 100, a boolean, a double, a float, a long, and a Gaussian number. The console output on the right shows the results of these operations.

```
1 import java.util.Random;
2
3 public class RandomClassExample {
4
5     public static void main(String[] args) {
6
7         Random rand = new Random();
8
9         System.out.println("Integer random number - " + rand.nextInt());
10        System.out.println("Integer random number from 0 to 100 - " + rand.nextInt(100));
11        System.out.println("Boolean random value - " + rand.nextBoolean());
12        System.out.println("Double random number - " + rand.nextDouble());
13        System.out.println("Float random number - " + rand.nextFloat());
14        System.out.println("Long random number - " + rand.nextLong());
15        System.out.println("Gaussian random number - " + rand.nextGaussian());
16
17    }
18 }
19
20
```

Console Output:

```
<terminated> RandomClassExample [Java Application] C:\Program Files\Java\
Integer random number - -1530677352
Integer random number from 0 to 100 - 77
Boolean random value - true
Double random number - 0.5830584766781006
Float random number - 0.31534636
Long random number - 414211629640411364
Gaussian random number - -0.265361191344894:
```

Formatter class in java

- The **Formatter** is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming.
- The Formatter class is defined as final class inside the **java.util** package.
- The Formatter class implements **Cloneable** and **Flushable** interface.

The Formatter class in java has the following constructors.

S.No.	Constructor with Description
1	Formatter() It creates a new formatter.
2	Formatter(Appendable a) It creates a new formatter with the specified destination.
3	Formatter(Appendable a, Locale l) It creates a new formatter with the specified destination and locale.
4	Formatter(File file) It creates a new formatter with the specified file.
5	Formatter(File file, String charset) It creates a new formatter with the specified file and charset.
6	Formatter(File file, String charset, Locale l) It creates a new formatter with the specified file, charset, and locale.
7	Formatter(Locale l) It creates a new formatter with the specified locale.
8	Formatter(OutputStream os) It creates a new formatter with the specified output stream.
9	Formatter(OutputStream os, String charset) It creates a new formatter with the specified output stream and charset.
10	Formatter(OutputStream os, String charset, Locale l) It creates a new formatter with the specified output stream, charset, and locale.
11	Formatter(PrintStream ps) It creates a new formatter with the specified print stream.

S.No.	Constructor with Description
12	Formatter(String fileName) It creates a new formatter with the specified file name.
13	Formatter(String fileName, String charset) It creates a new formatter with the specified file name and charset.
14	Formatter(String fileName, String charset, Locale l) It creates a new formatter with the specified file name, charset, and locale.

The Formatter class in java has the following methods.

S.No.	Methods with Description
1	Formatter format(Locale l, String format, Object... args) It writes a formatted string to the invoking object's destination using the specified locale, format string, and arguments.
2	Formatter format(String format, Object... args) It writes a formatted string to the invoking object's destination using the specified format string and arguments.
3	void flush() It flushes the invoking formatter.
4	Appendable out() It returns the destination for the output.
5	Locale locale() It returns the locale set by the construction of the invoking formatter.
6	String toString() It converts the invoking object to string.
7	IOException ioException() It returns the IOException last thrown by the invoking formatter's Appendable.
8	void close() It closes the invoking formatter.

Let's consider an example program to illustrate methods of Formatter class.

Example

```
1 import java.util.*;
2
3 public class FormatterClassExample {
4
5     public static void main(String[] args) {
6
7         Formatter formatter=new Formatter();
8         formatter.format("%2$s %1$s %3$s", "Smart", "BTech", "Class");
9         System.out.println(formatter);
10
11         formatter = new Formatter();
12         formatter.format(Locale.FRANCE, "%.5f", -1325.789);
13         System.out.println(formatter);
14
15         String name = "Java";
16         formatter = new Formatter();
17         formatter.format(Locale.US, "Hello %s !", name);
18         System.out.println(" " + formatter + " " + formatter.locale());
19
20         formatter = new Formatter();
21         formatter.format("%.4f", 123.1234567);
22         System.out.println("Decimal floating-point notation to 4 places: " + formatter);
23
24         formatter = new Formatter();
25         formatter.format("%010d", 88);
26         System.out.println("value in 10 digits: " + formatter);
27     }
28 }
29 }
```

Console

<terminated> FormatterClassExample [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (7 Apr 2020, 22:09:42)

BTech Smart Class

-1325,78900

Hello Java ! en_GB

Decimal floating-point notation to 4 places: 123.1235

value in 10 digits: 0000000088

Scanner class in java

- The **Scanner** is a built-in class in java used for read the input from the user in java programming. The Scanner class is defined inside the **java.util** package.
- The Scanner class implements **Iterator** interface.
- The Scanner class provides the easiest way to read input in a Java program.

□ The Scanner object breaks its input into tokens using a delimiter pattern, the default delimiter is whitespace.

The Scanner class in java has the following constructors.

S.No.	Constructor with Description
1	Scanner(InputStream source) It creates a new Scanner that produces values read from the specified input stream.
2	Scanner(InputStream source, String charsetName) It creates a new Scanner that produces values read from the specified input stream.
3	Scanner(File source) It creates a new Scanner that produces values scanned from the specified file.
4	Scanner(File source, String charsetName) It creates a new Scanner that produces values scanned from the specified file.
5	Scanner(String source) It creates a new Scanner that produces values scanned from the specified string.
6	Scanner(Readable source) It creates a new Scanner that produces values scanned from the specified source.
7	Scanner(ReadableByteChannel source) It creates a new Scanner that produces values scanned from the specified channel.
8	Scanner(ReadableByteChannel source, String charsetName) It creates a new Scanner that produces values scanned from the specified channel.

The Scanner class in java has the following methods.

S.No.	Methods with Description
1	String next() It reads the next complete token from the invoking scanner.

S.No.	Methods with Description
2	String next(Pattern pattern) It reads the next token if it matches the specified pattern.
3	String next(String pattern) It reads the next token if it matches the pattern constructed from the specified string.
4	boolean nextBoolean() It reads a boolean value from the user.
5	byte nextByte() It reads a byte value from the user.
6	double nextDouble() It reads a double value from the user.
7	float nextFloat() It reads a floating-point value from the user.
8	int nextInt() It reads an integer value from the user.
9	long nextLong() It reads a long value from the user.
10	short nextShort() It reads a short value from the user.
11	String nextLine() It reads a string value from the user.
12	boolean hasNext() It returns true if the invoking scanner has another token in its input.
13	void remove() It is used when remove operation is not supported by this implementation of Iterator.
14	void close() It closes the invoking scanner.

Let's consider an example program to illustrate methods of Scanner class.

Example

The screenshot displays the Eclipse IDE interface. The main editor window shows the source code for `ScannerClassExample.java`. The code imports `java.util.Scanner` and defines a `ScannerClassExample` class with a `main` method. The `main` method uses a `Scanner` object to read input from `System.in`. It prompts the user for their name, age, salary, and a message. The input values are `Raja`, `32`, `30000`, and `Good luck` respectively. The program then prints a summary of the input data, including a dashed line separator.

```
1 import java.util.Scanner;
2
3 public class ScannerClassExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in); // Input stream is used
8
9         System.out.print("Enter any name: ");
10        String name = read.next();
11
12        System.out.print("Enter your age in years: ");
13        int age = read.nextInt();
14
15        System.out.print("Enter your salary: ");
16        double salary = read.nextDouble();
17
18        System.out.print("Enter any message: ");
19        read = new Scanner(System.in);
20        String msg = read.nextLine();
21
22        System.out.println("\n-----");
23        System.out.println("Hello, " + name);
24        System.out.println("You are " + age + " years old.");
25        System.out.println("You are earning Rs." + salary + " per month.");
26        System.out.println("Words from " + name + " - " + msg);
27    }
28 }
29
```

The Console window on the right shows the output of the program. It displays the prompts and the user's input, followed by the formatted output messages.

```
<terminated> ScannerClassExample [Java Application] C:\Program Files\Java\jre1.8.0
Enter any name: Raja
Enter your age in years: 32
Enter your salary: 30000
Enter any message: Good luck

-----
Hello, Raja
You are 32 years old.
You are earning Rs.30000.0 per month.
Words from Raja - Good luck
```

The status bar at the bottom indicates the current line is 28, column 2, and the time is 8:45. The 'Writable' and 'Smart Insert' modes are also shown.

Java Programming (R20CSE2204)

UNIT - V

GUI Programming with Swing – Introduction, limitations of AWT, MVC architecture, components, containers. Understanding Layout Managers, Flow Layout, Border Layout, Grid Layout, Card Layout, Grid Bag Layout.

Event Handling- The Delegation event model- Events, Event sources, Event Listeners, Event classes, Handling mouse and keyboard events, Adapter classes, Inner classes, Anonymous Inner classes. A Simple Swing Application, Applets – Applets and HTML, Security Issues, Applets and Applications, passing parameters to applets. Creating a Swing Applet, Painting in Swing, A Paint example, Exploring Swing Controls- JLabel and Image Icon, JText Field, The Swing Buttons- JButton, JToggleButton, JCheckBox, JRadioButton, JTabbedPane, JScrollPane, JList, JComboBox, Swing Menus, Dialogs.

Topics Covered:

- A Simple Swing Application, Applets
- Applets and HTML,
- Security Issues,
- Applets and Applications,
- passing parameters to applets.
- Creating a Swing Applet,
- Painting in Swing,
- A Paint example,
- Exploring Swing Controls- JLabel and Image Icon, JText Field,

Applet Definition:

- Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as apart of a web document.
- After a user receives an applet, the applet can produce a graphical user interface. It has limited access to resources so that it can run complex computations without introducing the risk of viruses or breaching data integrity.
- Any applet in Java is a class that extends the java.applet.Applet class.
- An Applet class does not have any main() method. It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.
- JVM creates an instance of the applet class and invokes **init()** method to initialize an Applet.

An applet is a Java program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

The HTML document contains tags, which specify the name of the Java applet and its Uniform Resource Locator (URL). The URL is the location at which the applet bytecodes reside on the Internet.

When an HTML document containing a Java applet tag is displayed, a Java-enabled Web browser downloads the Java bytecodes from the Internet and uses the Java virtual machine to process the code from within the Web document. These Java applets are what enable Web pages to contain animated graphics or interactive content.

Applications of Java Applet

The **Applets** are used to provide interactive features to web **applications** that cannot be provided by HTML alone. They can capture mouse input and also have controls like buttons or check boxes. In response to user actions, an **applet** can change the provided graphic content.

Difference between Applet and Application

AWT	Swing
AWT components are heavyweight components	Swing components are lightweight components
AWT doesn't support pluggable look and feel	Swing supports pluggable look and feel
AWT programs are not portable	Swing programs are portable
AWT is old framework for creating GUIs	Swing is new framework for creating GUIs
AWT components require java.awt package	Swing components require javax.swing package
AWT supports limited number of GUI controls	Swing provides advanced GUI controls like Jtable, JTabbedPane etc
More code is needed to implement AWT controls functionality	Less code is needed to implement swing controls functionality
AWT doesn't follow MVC	Swing follows MVC

AWT Event Listener Interfaces

Following is the list of commonly used event listeners:

Sr. No.	Control & Description
1	ActionListener This interface is used for receiving the action events.
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.
4	KeyListener This interface is used for receiving the key events.
5	MouseListener This interface is used for receiving the mouse events.

6	<u>TextListener</u> This interface is used for receiving the text events.
7	<u>WindowListener</u> This interface is used for receiving the window events.
8	<u>AdjustmentListener</u> This interface is used for receiving the adjusmtent events.
9	<u>ContainerListener</u> This interface is used for receiving the container events.
10	<u>MouseMotionListener</u> This interface is used for receiving the mouse motion events.
11	<u>FocusListener</u> This interface is used for receiying the focus events.

Advantages of Applets

1. It takes very less response time as it works on the client side.
2. It can be run on any browser which has JVM running in it.

Applet class

Applet class provides all necessary support for applet execution, such as initializing and destroying of applet. It also provide methods that load and display images and methods that load and play audio clips

Lifecycle/ An Applet Skeleton

Most applets override these four methods. These four methods forms Applet lifecycle.

- **init()** : init() is the first method to be called. This is where variable are initialized. This method is called only once during the runtime of applet.
- **start()** : start() method is called after init(). This method is called to restart an applet after it has been stopped.
- **stop()** : stop() method is called to suspend thread that does not need to run when applet is not visible.
- **destroy()** : destroy() method is called when your applet needs to be removed completely from memory.

Note: The stop() method is always called before destroy() method.

Applet and AWT:

To create an applet first write the program and Save the file with name FirstApplet.java

```
import java.applet.Applet;
import java.awt.Graphics;
public class FirstApplet extends Applet{

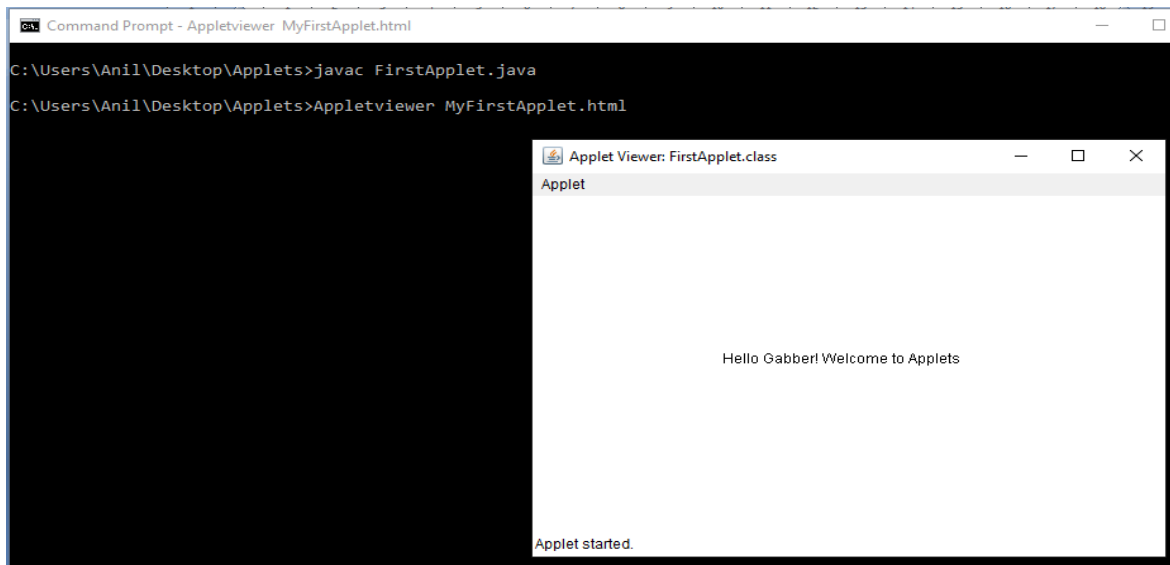
    public void paint(Graphics g){
        g.drawString("Hello Gabber! Welcome to Applets",150,150);
    }
}
```

Next:

Write the following program and save the file say with MyFirstApplet.html

```
<html>
<body>
<applet code="FirstApplet.class" width="500" height="300">
</applet>
</body>
</html>
```

To run applet programs



Create a Swing Applet

The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: **A Swing applet extends JApplet rather than Applet. JApplet** is derived from **Applet**. Thus, **JApplet** includes all of the functionality found in **Applet** and adds support for Swing. **JApplet** is a top-level Swing container, which means that it is *not* derived from **JComponent**. Because **JApplet** is a top-level container, it includes the various panes. This means that all components are added to **JApplet**'s content pane in the same way that components are added to **JFrame**'s content pane.

Swing applets use the same four life-cycle methods i.e **init()**, **start()**, **stop()**, and **destroy()**. Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the **paint()** method.

All interaction with components in a Swing applet must take place on the event dispatching thread, This threading issue applies to all Swing programs.

//A simple Swing-based applet

```
package com.myPack;
```

```
//A simple Swing-based applet
```

```
import javax.swing.*; import
java.awt.*; import java.awt.event.*;
/*
```

This HTML can be used to launch the applet:

```
<applet code="MySwingApplet" width=400 height=250> </applet>
*/
```

```
public class MySwingApplet extends JApplet { JButton jbBvritn;
JButton jbBvrith;JLabel jl;
```

```
// Initialize the applet.
```

```
public void init() {
```

```
try {
```

```
SwingUtilities.invokeLater(new Runnable () {
```

```
    public void run() {
```

```
        displayGUI(); // initialize the GUI
```

```
    }
```

```
});
```

```
} catch(Exception exc) {
```

```
    System.out.println("Not Possible to create: becoz "+ exc);
```

```
}
```

```
}
```

```
//This applet does not need to override start(), stop(),
```

```
//or destroy().
```

```
//Set up and initialize the GUI.
```

```
private void displayGUI() {
```

```
    //Set the applet to use flow layout.setLayout(new
    FlowLayout());
```

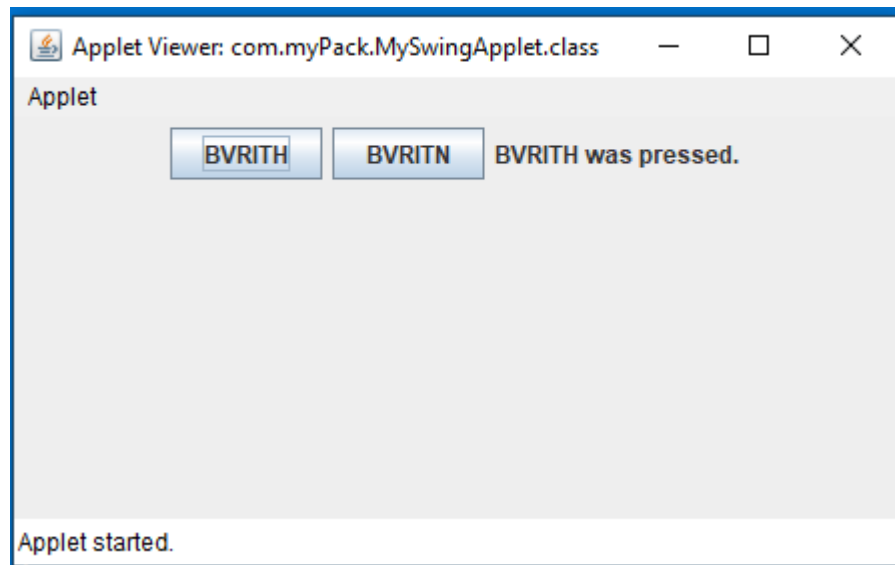
```

//Make two buttons.
jbBvritn = new JButton("BVRITH");

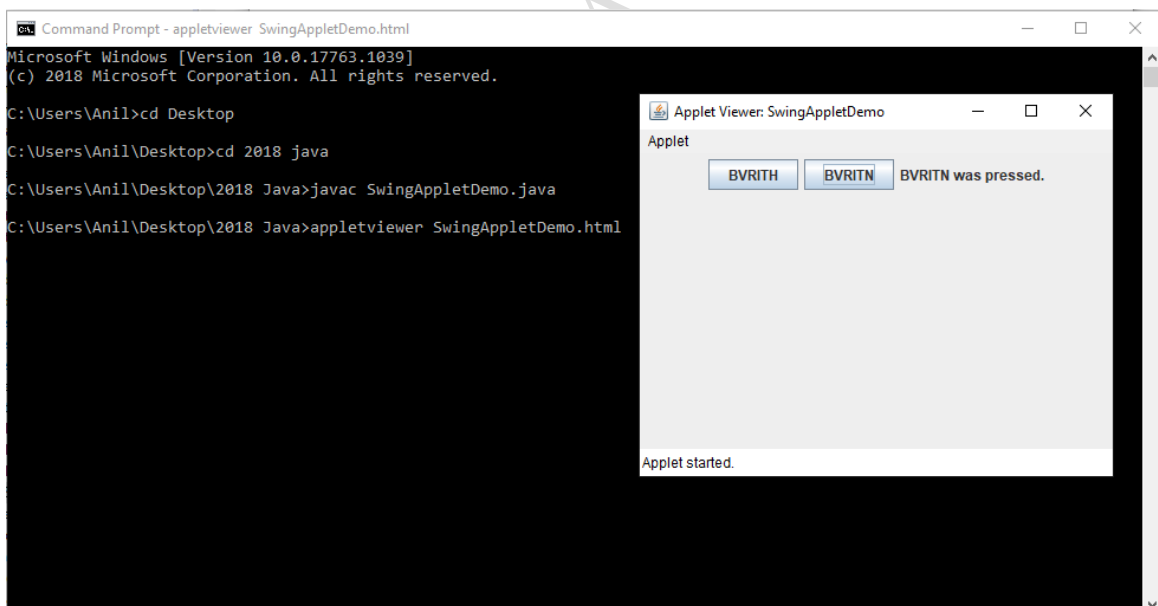
jbBvrith = new JButton("BVRITN");
//      Add action listener for Alpha.

jbBvritn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le)
    {
        jl.setText("BVRITH was pressed.");
    }
});
//Add action listener for Beta. jbBvrith.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {jl.setText("BVRITN was pressed.");
    }
});
});

```



To run in command prompt



There are two important things to notice about this applet. First, **MySwingApplet** extends **JApplet**. As explained, all Swing-based applets extend **JApplet** rather than **Applet**. Second, the **init()** method initializes the Swing components on the event dispatching thread by setting up a call to **displayGUI()**. Notice that this is accomplished through the use of **invokeAndWait()** rather than **invokeLater()**. Applets must use **invokeAndWait()** because the **init()** method must not return until the entire initialization process has been completed. In essence, the **start()** method cannot be called until after initialization, which means that the GUI must be fully constructed.

Inside **displayGUI()**, the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane. Although this example is quite simple, this same general approach must be used when building any Swing GUI that will be used by an applet.

GUI Programming with Swing – Introduction

GUI (Graphical User Interface) , Here the user interact with any application by clicking on some images or graphics.

Example: if the user wants to print a file, he can click on the printer images and the rest of the things will be taken care of by the application.

Like magnifying glass symbol for searching, a briefcase symbol for a directory etc.

Hence, the environment where the user can interact with the application through graphics or images is called GUI (Graphical User Interface)

- GUI is user friendly
- It gives attraction and beauty to any application by adding pictures, colors, menus, animation etc.
- It is possible to simulate a real life objects using GUI
- GUI helps to create graphical components like push buttons, radio buttons, check boxes etc.

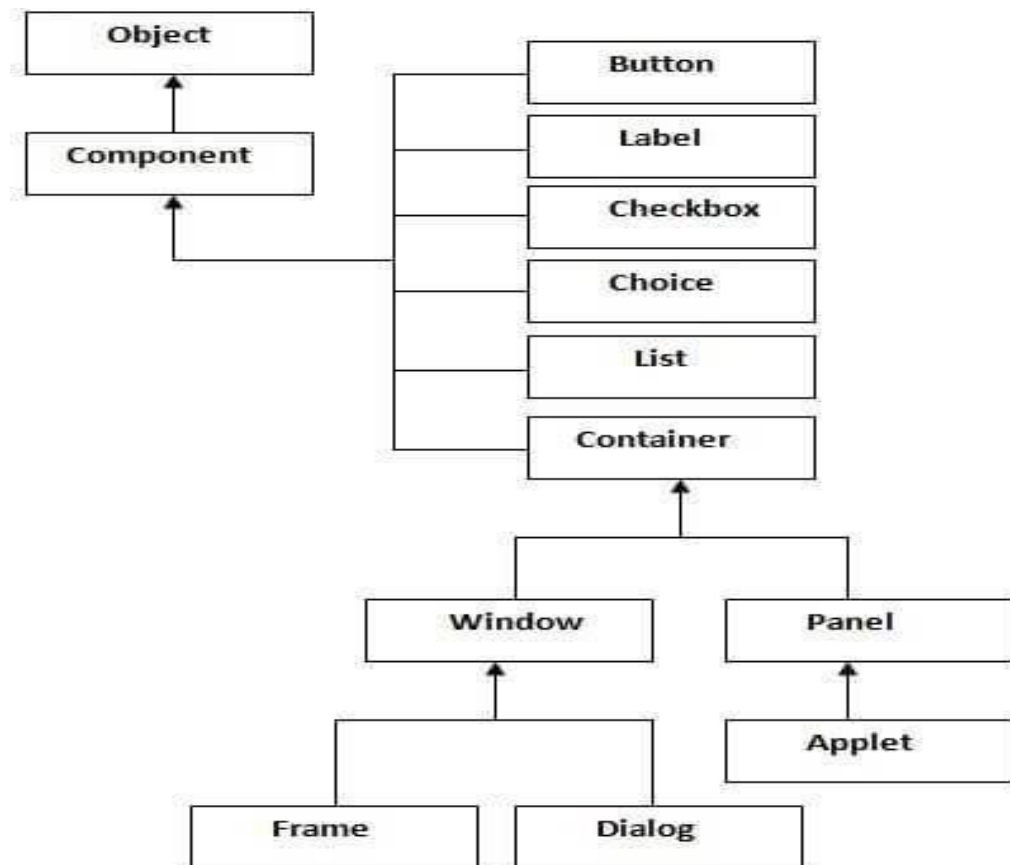
Java AWT (Abstract Window Toolkit)

- The java.awt package provides classes for AWT API such as
 - TextField,
 - Label,
 - TextArea,
 - RadioButton,
 - CheckBox,
 - Choice,
 - List
 - Button

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.

Components:



A component represents an object which is displayed pictorially on the screen. For example, we create an object of Button class as:

Button b = new Button();

Now, b is object of Button class, If we display this b on the screen, it displays a push button. Therefore, the object b, on going to the screen is becoming a component called 'Push Button'.

In the same way any component is a graphical representation of an object. Push Buttons, radio buttons, check boxes etc are all components

Container

The Container is a component in AWT that can contain another components like Buttons, Textfields, labels etc. The classes that extend Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component Class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize (int width, int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

AWT Example by Inheritance

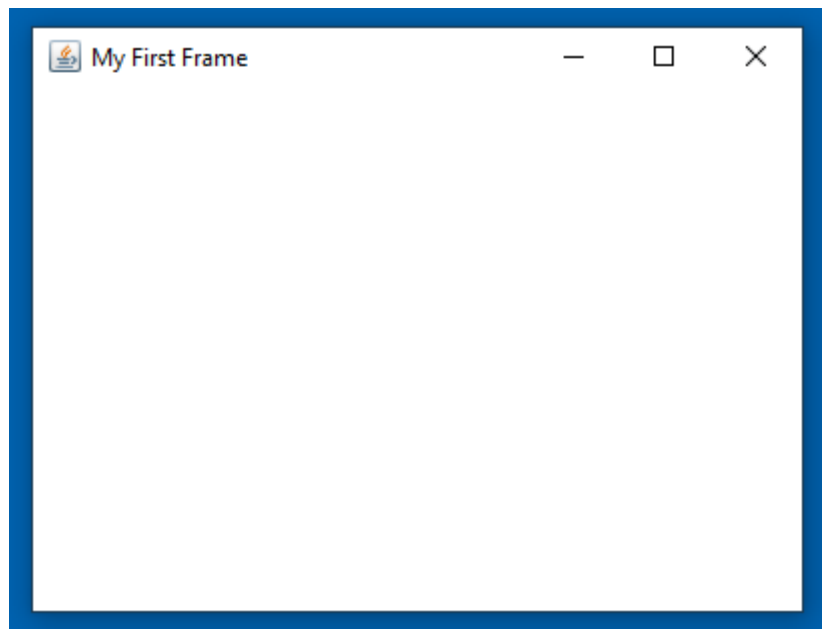
Program1: Example program to create a frame

```
import java.awt.Frame;

public class FirstFrame {

    public static void main(String[] args) {
        Frame f = new Frame ("My First Frame");
        f.setSize(400,300);
        f.setVisible(true);
    }
}
```

Output:



AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are showing Button component on the Frame.

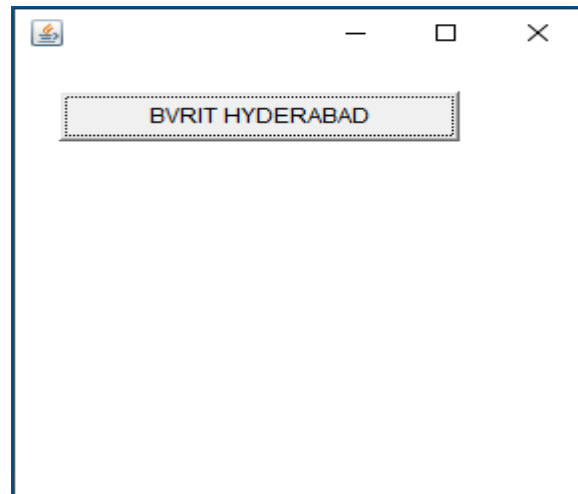
```
import java.awt.*;

class First2{
    First2(){
        Frame f=new Frame();
        Button b=new Button("BVRIT HYDERABAD");
        b.setBounds(30,50,80,30);
        f.add(b);
    }
}
```

```

f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}
}

```



After executing this program we observe that the frame can be minimized, maximized and resized but cannot be closed. Even if we click on the close button of the frame, it will not perform any closing action.

Then how to close the frame?

Closing a frame means attaching action to the component. To attach action to the frame we need “EVENT DELIGATION MODEL”.

EVENT DELIGATION MODEL

An event represents a **specific action** done on the component

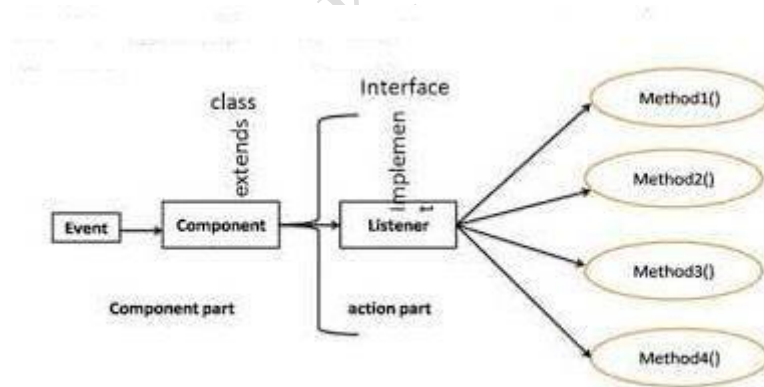
- Clicking
- Double Clicking
- Typing data inside the component
- Mouse over etc

When an event is generated on the component, the component will not know about it because it cannot listen to the event. To let the component understand that an event occurred on it, we should **add some listeners** to the components.

A **listener is an interface** which listens to an event coming from a component.

A **listener will have some abstract methods** which need to be implemented by the programmer.

When an even is generated by the user on the component, the event is not handled by the component; on the other hand, the component sends (delegate) the event to the listener attached to it. The listener will not handle the event. It hands over (Delegates) the event to an appropriate method. Finally, the method is executed and the event is handled. This is called 'Event Delegation Model'



Steps involved in the Event Delegation Model are:

- We should attach an appropriate listener to a component. This is done using `addxxxListener()` method. Similarly, to remove a listener from a component, we can use `removexxxListener()` method
- Implement the methods of the listener, especially the method which handles the event.
- When an event is generated on the component, then the method in step2 will be executed and the event is handled.

Closing the Frame:

We know frame is also a component. We want to close the frame by clicking on its close button. Let us follow these steps to see how to use event delegation model to do this.

We should attach a listener to the frame component. Remember, all listeners are available in `java.awt.event` package.

The most suitable listener to the frame is 'window listener' it can be attached using `addWindowListener()` method as;

`f.addWindowListener(WindowListener obj);`

Note that, the `addWindowListener()` method has a parameter that is expecting object of `WindowListener` interface. Since it is not possible to create an object to an interface, we should create an object to the implemented class of the interface and pass it to the method.

Implement all the methods of the `WindowListener` interface. The following methods are found in `WindowListener` interface.

```

public void windowActivated(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowClosing(WindowEvent e)
public void windowDeactivated(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowOpened(WindowEvent e)
  
```

In all the preceding methods, WindowListener interface calls the public void WindowClosing() method when the frame is being closed. So, implementing this method alone is enough as:

```
Public void windowClosing(WindowEvent e) {
    System.exit(0); // closing the application
}
```

For the remaining methods, we can provide empty body.

So, when the frame is closed, the body of this method is executed and the application gets closed. **In this way we can handle the frame closing event.**

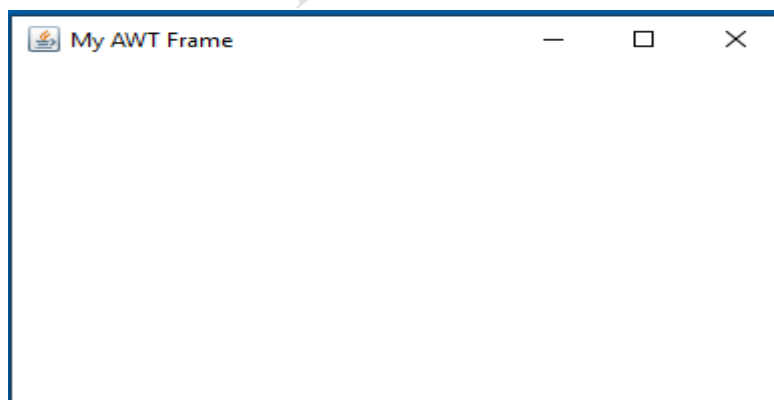
Write a program to create a frame and then close it on clicking the close button

```
import java.awt.*;
import java.awt.event.*;

class MyFrame extends Frame{
    public static void main(String args[]) {
        MyFrame f = new MyFrame();
        f.setTitle("My AWT Frame");
        f.setSize(400, 250);
        f.setVisible(true);
        f.addWindowListener(new MyClass());
    }
}

class MyClass implements WindowListener{
    public void windowActivated(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowOpened(WindowEvent e) { }
}
```

Output:



Now this window is able to close

In the above program we not only create a frame but also close the frame when the user clicks on the close button.

In the above example, we had to mention all the methods of WindowListener interface, just for the sake of one method.

There is another way to do this.

- There is a class WindowAdapter in java.awt.event package. That contains all the methods of the WindowListener interface with an empty implementation.
- If we extend MyClass from this WindowAdapter class, then we need not write all the methods with empty implementation.
- We can write only that method which we need.

// Program: Frame closing with WindowAdapter class

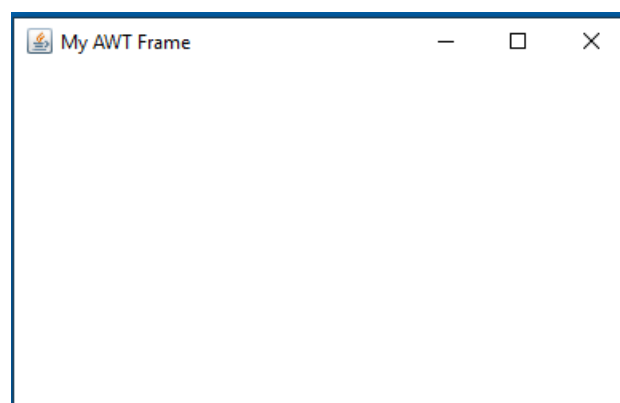
```
import java.awt.*;
import java.awt.event.*;

class MyFrame1 extends Frame {
    public static void main(String args[]){

        MyFrame1 f = new MyFrame1();
        f.setTitle("My AWT Frame");
        f.setSize(400,250);
        f.setVisible(true);
        f.addWindowListener(new MyClass());
    }
}

class MyClass extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
}
```

Output:



What is an Adapter Class?

An adapter class is an implementation class of a listener interface which contains all methods implemented with empty body. For example, WindowAdapter is an adapter class of WindowListener interface.

In the above program, the code of MyClass can be copied directly into addWindowListener() method as

```
f.addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});
```

In this code, we cannot find the name of MyClass anywhere in the code. It means the name of MyClass is hidden in MyFrame class and hence MyClass is an inner class in MyFrame class whose name is not mentioned. Such inner class is called 'anonymous inner class';

What is anonymous inner class?

Anonymous inner class is an inner class whose name is not mentioned, and for which only one object is created.

Program to close the frame using an Anonymous inner class.

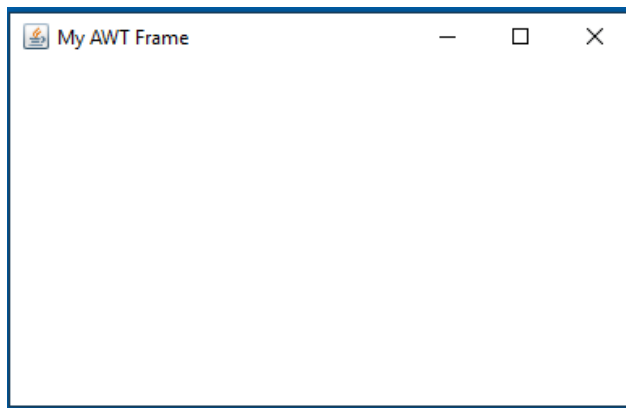
// Frame closing with WindowAdapter class

```
import java.awt.*;
import java.awt.event.*;

class MyFrame2 extends Frame {
    public static void main(String args[]){

        MyFrame2 f = new MyFrame2();
        f.setTitle("My AWT Frame");
        f.setSize(400,250);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
}
```

Output:



Program: Displaying text message in the frame using drawstring() Method

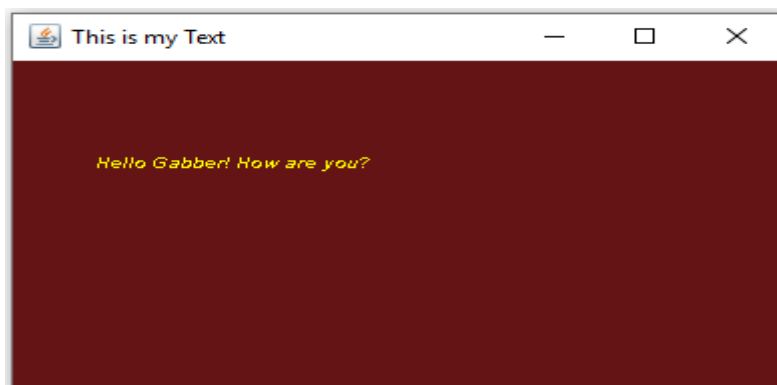
```
import java.awt.*;
import java.awt.event.*;

class Message extends Frame {Message(){
    addWindowListener(new WindowAdapter());
    {
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    }
}

public static void main(String args[]){
    Message m = new Message();

    m.setSize(400,250); m.setTitle("This is my
    Text");m.setVisible(true);
}
}
```

Output:



Program2: to create frame and adding a button

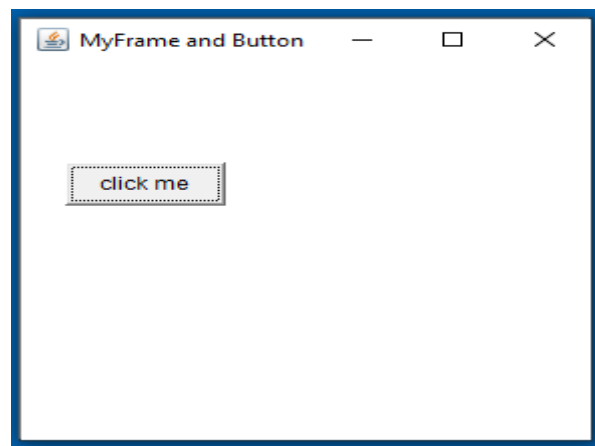
Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.Button;
import java.awt.Frame;

class FirstFrameAndButton extends Frame{
    public static void main(String args[]){
        Frame f = new Frame();

        Button b=new Button("click me");
        b.setBounds(30,100,80,30);// setting button position
        f.add(b);//adding button into frame
        f.setSize(300,300);//frame size 300 width and 300 height
        f.setLayout(null);//no layout manager
        f.setVisible(true);//now frame will be visible, by default not
        visible
    }
}
```

Output



The **setBounds(int xaxis, int yaxis, int width, int height)** method is used in the above example that sets the position of the awt button.

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
 - `public void addActionListener(ActionListener a){ }`
- **MenuItem**
 - `public void addActionListener(ActionListener a){ }`

- **TextField**
 - public void addActionListener(ActionListener a){ }
 - public void addTextListener(TextListener a){ }
- **TextArea**
 - public void addTextListener(TextListener a){ }
- **Checkbox**
 - public void addItemListener(ItemListener a){ }
- **Choice**
 - public void addItemListener(ItemListener a){ }
- **List**
 - public void addActionListener(ActionListener a){ }
 - public void addItemListener(ItemListener a){ }

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;

class AEvent extends Frame implements ActionListener{
    TextField tf;
    AEvent(){
```

```
//create components
```

```
tf=new TextField();
```

```
tf.setBounds(60,50,170,20);
```

```
Button b=new Button("click me");
```

```
b.setBounds(100,120,80,30);
```

```
//register listener
```

```
b.addActionListener(this);//passing current instance
```

```
//add components and set size, layout and visibility
```

```
add(b);add(tf);
```

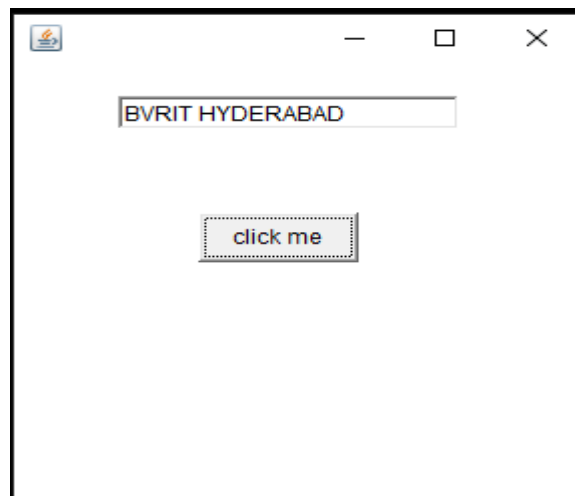
```
setSize(300,300);
```

```
setLayout(null);
```

```
setVisible(true);
```

```
addWindowListener(new WindowAdapter());  
{  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(0);  
    }  
}  
}  
public void actionPerformed(ActionEvent e)  
{  
    tf.setText("BVRIT HYDERABAD");  
}  
public static void main(String args[]){  
    new AEvent();  
}  
}
```

Output:



public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

Java AWT Button

The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

AWT Button Class declaration

public class Button **extends** Component **implements** Accessible

Java AWT Button Example

//AWT Button Class declaration

```
import java.awt.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        Frame f=new Frame("Button Example");
        Button b=new Button("Goto BVRITH");
        b.setBounds(100,100,80,30);
        f.add(b);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        }
    }
}
```

Output:



Java AWT Label

The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

AWT Label Class Declaration

public class Label **extends** Component **implements** Accessible

Java Label Example

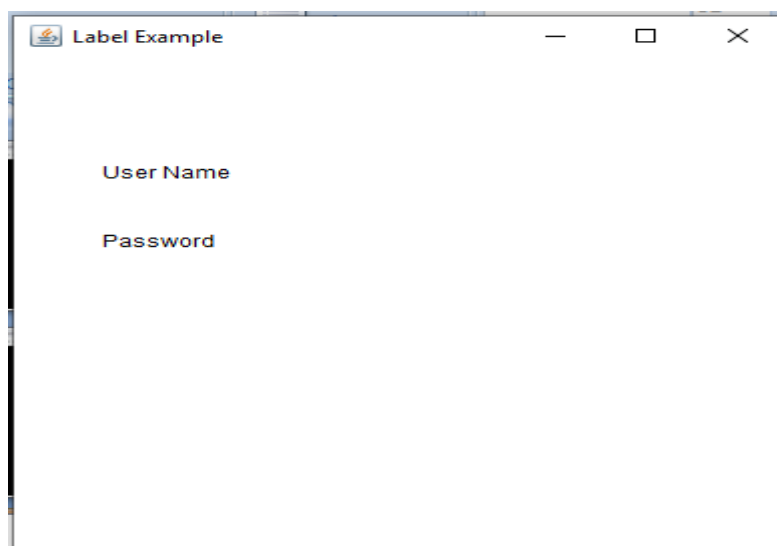
//AWT Label Class declaration

```
import java.awt.*;
import java.awt.event.*;

class LabelExample{
public static void main(String args[]){
    Frame f= new Frame("Label Example");
    Label l1,l2;
    l1=new Label("User Name");
    l1.setBounds(50,100, 100,30);
    l2=new Label("Password");
    l2.setBounds(50,150, 100,30);
    f.add(l1); f.add(l2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);

    f.addWindowListener(new
    WindowAdapter());
    { public void windowClosing(WindowEvent
    e){System.exit(0);
    }
    } } }
```

Output:



Java AWT TextField

The object of a TextField class is a text component that allows the editing of a single line text. It inherits TextComponent class.

AWT TextField Class Declaration

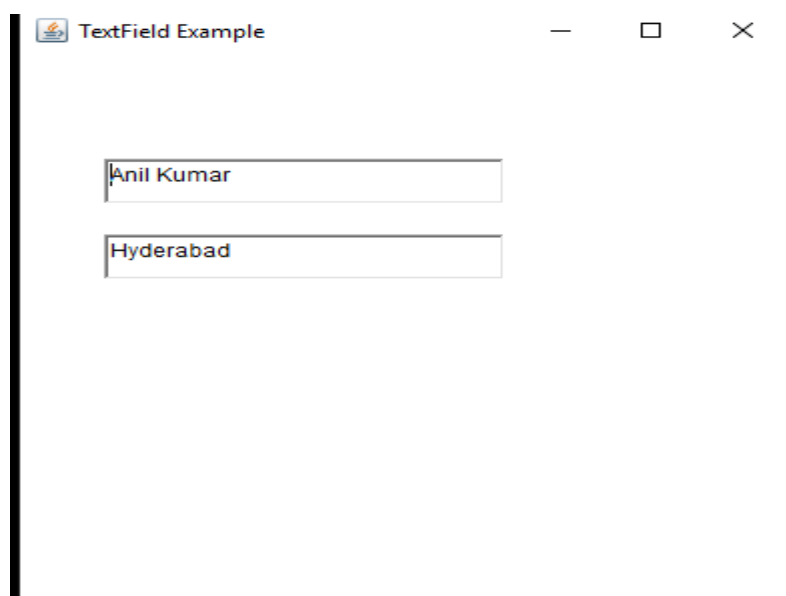
public class TextField **extends** TextComponent

Java AWT TextField Example

```
import java.awt.*;
import java.awt.event.*;
class TextFieldExample{
public static void main(String args[]){
    Frame f= new Frame("TextField Example");
    TextField t1,t2;
    t1=new TextField("Anil Kumar");
    t1.setBounds(50,100, 200,30);
    t2=new TextField("Hyderabad");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);

    f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
}
}
```

Output:



Java AWT TextArea

The object of a TextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

AWT TextArea Class Declaration

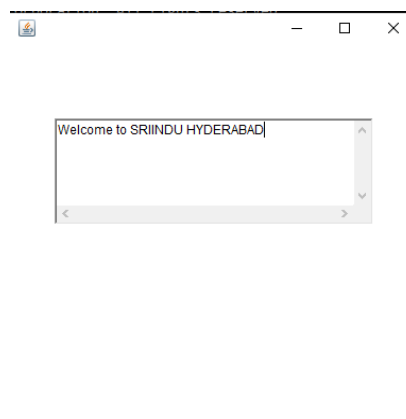
public class TextArea **extends** TextComponent

Java AWT TextArea Example

```
import java.awt.*;
import java.awt.event.*;

public class TextAreaExample
{
    TextAreaExample(){
        Frame f= new Frame();
        TextArea area=new TextArea("Welcome to SRIINDU HYDERABAD");
        area.setBounds(50,100, 300,100);
        f.add(area);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
    public static void main(String args[])
    {
        new TextAreaExample();
    }
}
```

Output:



Java AWT Checkbox

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

AWT Checkbox Class Declaration

public class Checkbox **extends** Component **implements** ItemSelectable, Accessible

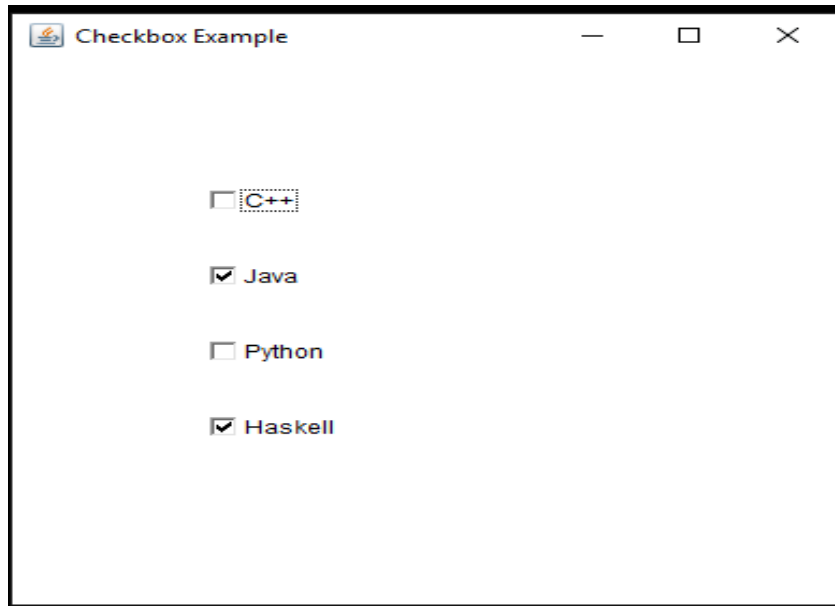
Java AWT Checkbox Example

```
import java.awt.*;
import java.awt.event.*;
public class CheckboxExample
{
    CheckboxExample(){
        Frame f= new Frame("Checkbox Example");
        Checkbox checkbox1 = new Checkbox("C++");
        checkbox1.setBounds(100,100, 50,50);
        Checkbox checkbox2 = new Checkbox("Java", true);
        checkbox2.setBounds(100,150, 50,50);
        Checkbox checkbox3 = new Checkbox("Python");
        checkbox3.setBounds(100,200, 70,50);
        Checkbox checkbox4 = new Checkbox("Haskell", true);
        checkbox4.setBounds(100,250,70,50);
        f.add(checkbox1);
        f.add(checkbox2);
        f.add(checkbox3);
        f.add(checkbox4);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }

    public static void main(String args[])
    {
        new CheckboxExample();
    }
}
```

Output:



Java AWT CheckboxGroup

The object of CheckboxGroup class is used to group together a set of **Checkbox**. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the **object class**.

Note: CheckboxGroup enables you to create radio buttons in AWT. There is no special control for creating radio buttons in AWT.

AWT CheckboxGroup Class Declaration

public class CheckboxGroup **extends** Object **implements** Serializable

Java AWT CheckboxGroup Example

```
import java.awt.*;
import java.awt.event.*;

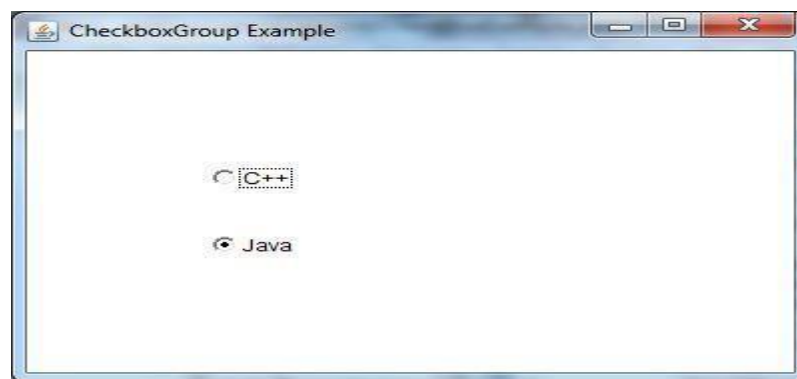
public class CheckboxGroupExample
{
    CheckboxGroupExample(){
        Frame f= new Frame("CheckboxGroup Example");
        CheckboxGroup cbg = new CheckboxGroup();
        Checkbox checkBox1 = new Checkbox("C++", cbg, false);
        checkBox1.setBounds(100,100, 50,50);
        Checkbox checkBox2 = new Checkbox("Java", cbg, true);
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1);
        f.add(checkBox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

```

        f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
        System.exit(0);
        }
        });
    }
    public static void main(String args[])
    {
        new CheckboxGroupExample();
    }
}

```

Output:



Java AWT Choice

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

AWT Choice Class Declaration

public class Choice **extends** Component **implements** ItemSelectable, Accessible

Java AWT Choice Example

```

import java.awt.*;
import java.awt.event.*;

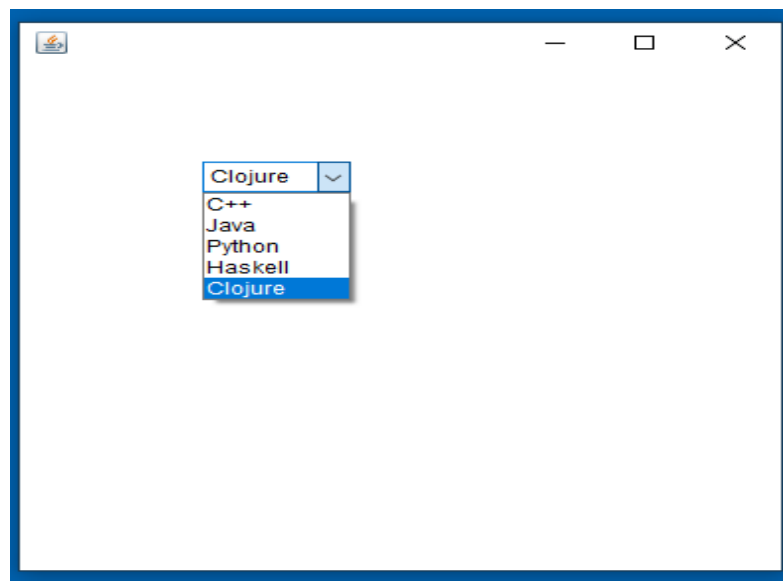
public class ChoiceExample
{
    ChoiceExample(){
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(100,100, 75,75);
        c.add("C++");
        c.add("Java");
        c.add("Python");
        c.add("Haskell");
        c.add("Clojure");
    }
}

```

```
f.add(c);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);

    f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
}
public static void main(String args[])
{
    new ChoiceExample();
}
}
```

Output:



Java AWT List

The object of List class represents a list of text items. By the help of list, user can choose either one item or multiple items. It inherits Component class.

AWT List class Declaration

public class List **extends** Component **implements** ItemSelectable, Accessible

Java AWT List Example

```
import java.awt.*;
import java.awt.event.*;

public class ListExample
{
    ListExample(){
        Frame f= new Frame();
        List l1=new List(5);
        l1.setBounds(100,100, 100,75);
        l1.add("Anil");
        l1.add("Gabber");
        l1.add("Akshara");
        l1.add("Vikram");
        l1.add("Vijay");
        f.add(l1);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }

    public static void main(String args[])
    {
        new ListExample();
    }
}
```

Output:



Java AWT Canvas

The Canvas control represents a blank rectangular area where the application can draw or trap input events from the user. It inherits the Component class.

AWT Canvas class Declaration

public class Canvas **extends** Component **implements** Accessible

Java AWT Canvas Example

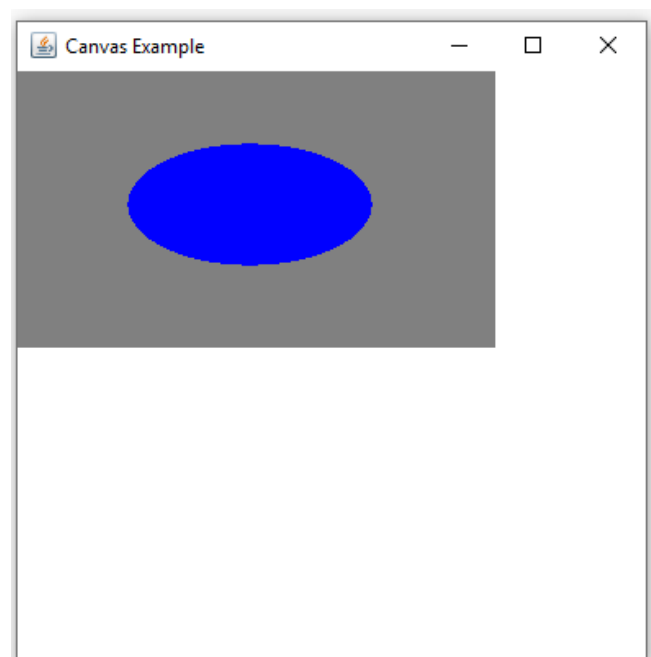
```
import java.awt.*;
import java.awt.event.*;

public class CanvasExample
{
    public CanvasExample()
    {
        Frame f= new Frame("Canvas Example");
        f.add(new MyCanvas());
        f.setLayout(null);
        f.setSize(400, 400);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
    public static void main(String args[])
    {
        new CanvasExample();
    }
}

class MyCanvas extends Canvas
{
    public MyCanvas() {
        setBackground (Color.GRAY);
        setSize(300, 200);
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.fillOval(75, 75, 150, 75);
    }
}
```

Output:



Java AWT Scrollbar

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

AWT Scrollbar class declaration

public class Scrollbar **extends** Component **implements** Adjustable, Accessible

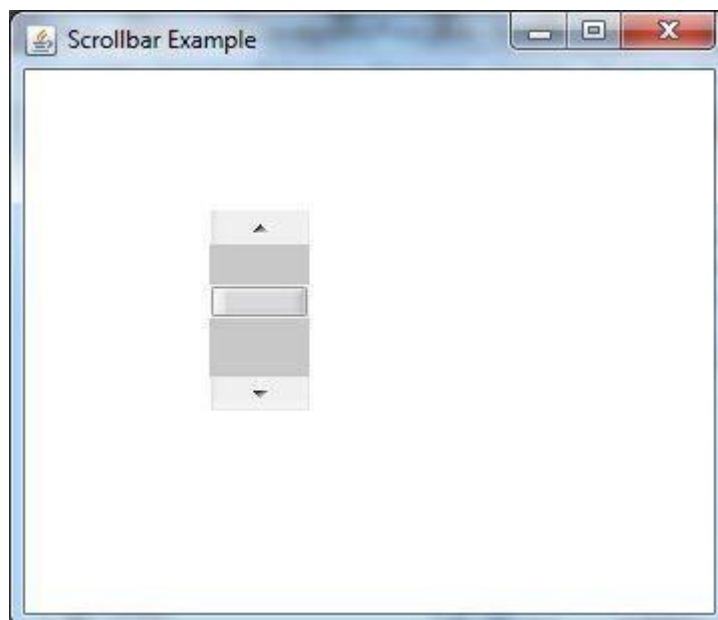
Java AWT Scrollbar Example

```
import java.awt.*;
import java.awt.event.*;

class ScrollbarExample{
ScrollbarExample(){
    Frame f= new Frame("Scrollbar Example");
    Scrollbar s=new Scrollbar();
    s.setBounds(100,100, 50,100);
    f.add(s);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);

    f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
}
public static void main(String args[])
{
    new ScrollbarExample();
}
}
```

Output:



Java AWT MenuItem and Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

public class MenuItem **extends** MenuComponent **implements** Accessible

AWT Menu class declaration

public class Menu **extends** MenuItem **implements** MenuContainer, Accessible

Java AWT MenuItem and Menu Example

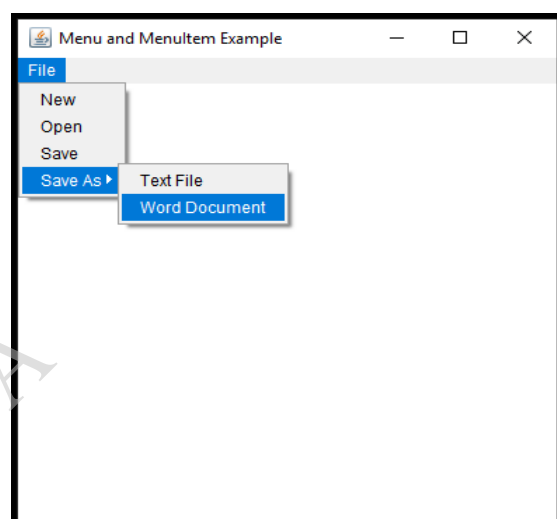
```
import java.awt.*;
import java.awt.event.*;

class MenuExample
{
    MenuExample(){
        Frame f= new Frame("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("File");

        Menu submenu=new Menu("Save As");
        MenuItem i1=new MenuItem("New");
        MenuItem i2=new MenuItem("Open");
        MenuItem i3=new MenuItem("Save");
        MenuItem i4=new MenuItem("Text File");
        MenuItem i5=new MenuItem("Word Document");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }

    public static void main(String args[])
    {
        new MenuExample();
    }
}
```



Java AWT PopupMenu

PopupMenu can be dynamically popped up at specific position within a component. It inherits the [Menu class](#).

AWT PopupMenu class declaration

public class PopupMenu **extends** Menu **implements** MenuContainer, Accessible

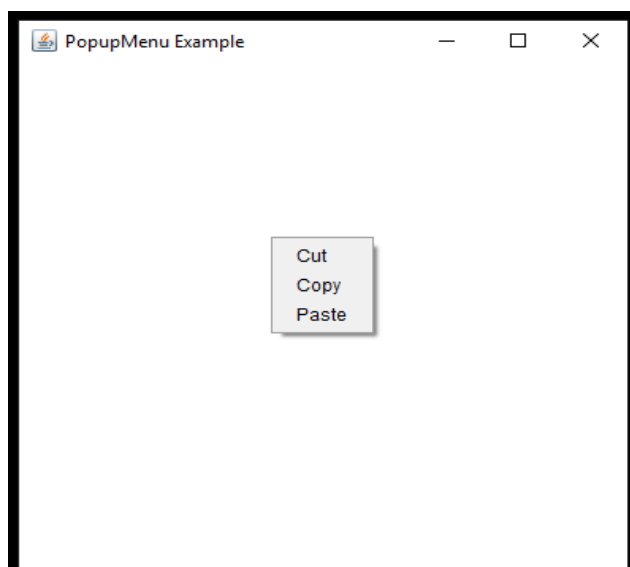
Java AWT PopupMenu Example

```
import java.awt.*;
import java.awt.event.*;

class PopupMenuExample
{
    PopupMenuExample(){
        final Frame f= new Frame("PopupMenu Example");
        final PopupMenu popupmenu = new PopupMenu("Edit");
        MenuItem cut = new MenuItem("Cut");
        cut.setActionCommand("Cut");
        MenuItem copy = new MenuItem("Copy");
        copy.setActionCommand("Copy");
        MenuItem paste = new MenuItem("Paste");
        paste.setActionCommand("Paste");
        popupmenu.add(cut);
        popupmenu.add(copy);
        popupmenu.add(paste);
        f.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                popupmenu.show(f, e.getX(), e.getY());
            }
        });
        f.add(popupmenu);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
}

public static void main(String args[])
{
    new PopupMenuExample();
}
```



Java AWT Panel

The Panel is a simplest container class. It provides space in which an application can attach any other component. It inherits the Container class.

It doesn't have title bar.

AWT Panel class declaration

public class Panel **extends** Container **implements** Accessible

Java AWT Panel Example

```
import java.awt.*;
import java.awt.event.*;

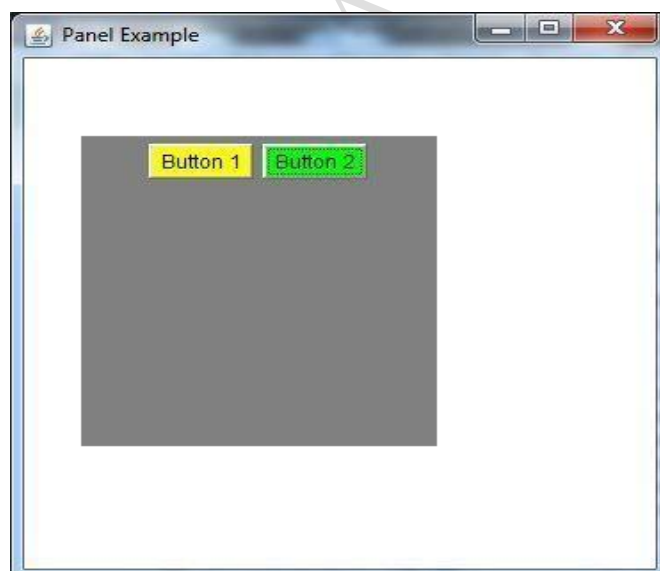
public class PanelExample {
    PanelExample()
    {
        Frame f= new Frame("Panel Example");
        Panel panel=new Panel();
        panel.setBounds(40,80,200,200);
        panel.setBackground(Color.gray);
        Button b1=new Button("Button 1");
        b1.setBounds(50,100,80,30);
        b1.setBackground(Color.yellow);
        Button b2=new Button("Button 2");
        b2.setBounds(100,100,80,30);

        b2.setBackground(Color.green);
        panel.add(b1); panel.add(b2);
        f.add(panel);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }

    public static void main(String args[])
    {
        new PanelExample();
    }
}
```

Output:



Java AWT Dialog

The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.

Unlike Frame, it doesn't have maximize and minimize [buttons](#).

Frame vs Dialog

Frame and Dialog both inherits Window class. Frame has maximize and minimize buttons but Dialog doesn't have.

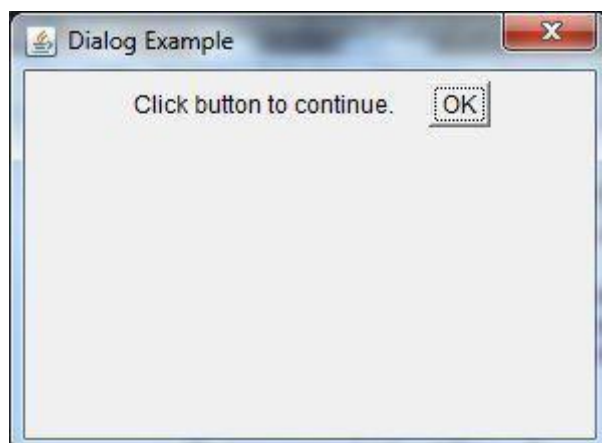
AWT Dialog class declaration

public class Dialog **extends** Window

Java AWT Dialog Example

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
    private static Dialog d;
    DialogExample() {
        Frame f= new Frame();
        d = new Dialog(f , "Dialog Example", true);
        d.setLayout( new FlowLayout() );
        Button b = new Button ("OK");
        b.addActionListener ( new ActionListener()
        {
            public void actionPerformed( ActionEvent e )
            {
                DialogExample.d.setVisible(false);
            }
        });
        d.add( new Label ("Click button to continue."));
        d.add(b);
        d.setSize(300,300);
        d.setVisible(true);
    }
    public static void main(String args[])
    {
        new DialogExample();
    }
}
```

Output:



Handling Keyboard Events

A user interacts with the application by pressing either keys on the keyboard or by using mouse. A programmer should know which key the user has pressed on the keyboard or whether the mouse is moved, pressed, or released. These are also called '**events**'. Knowing these events will enable the programmer to write his code according to the key pressed or mouse event.

KeyListener interface of **java.awt.event** package helps to know which key is pressed or released by the user. It has 3 methods

1. **Public void keyPressed(KeyEvent ke):** This method is called when a key is pressed on the keyboard. This include any key on the keyboard along with special keys like function keys, shift, alter, caps lock, home, end etc.
2. **Public void keyTyped(KeyEvent ke) :** This method is called when a key is typed on the keyboard. This is same as keyPressed() method but this method is called when general keys like A to Z or 1 to 9 etc are typed. It cannot work with special keys.
3. **Public void keyReleased(KeyEvent ke):** this method is called when a key is release.

KeyEvent class has the following methods to know which key is typed by the user.

1. Char getKeyChar(): this method returns the key name (or character) related to the key pressed or released.
2. Int getKeyCode(): this method returns an integer number which is the value of the key presed by the user.

The follwing are the key codes for the keys on the keyboard. They are defined as constants in KeyEvent class. **Remember VK represents Virtual Key.**

- To represent keys from a to z : VK_A to VK_Z.
- To represent keys from 1 to 9: VK_0 to VK_9.
- To represent keys from F1 to F12: VK_F1 to VK_F12.
- To represent home, end : VK_HOME, VK_END.
- To represent PageUp, PageDown: VK_PAGE_UP, VK_PAGE_DOWN
- To represent Insert, Delete: VK_INSERT, VK_DELETE
- To represent caps lock: VK_CAPS_LOCK
- To represent alter key: VK_ALT
- To represent Control Key: VK_CONTROL
- To represent shift: VK_SHIFT
- To represent tab key: VK_TAB
- To represent arrow keys: VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN
- To represent Escape key: VK_ESCAPE
- Static String getKeyText(int keyCode); this method returns a string describing the keyCode such as HOME, F1 or A.

Program: to trap a key which is pressed on the keyboard and display its name in the text area.

```
package com.myPack;
```

```
import java.awt.*; import
```

```
java.awt.event.*;import
```

```
javax.swing.*;
```

```
class KeyBoardEvents extends JFrame implements KeyListener
```

```
{
```

```
    private static final Font Font = null; Container c;
```

```
    JTextArea a;
```

```
String str = " ";
```

```
KeyBoardEvents()
```

```
{
```

```
    c=getContentPane();
```

```
    a = new JTextArea("Press a Key");
```

```
    a.setFont(new Font ("Times New Roman", Font.BOLD,30));
```

```
    c.add(a); a.addKeyListener(this);
```

```
}
```

```
public void keyPressed(KeyEvent ke)
```

```
{
```

```
    int keycode = ke.getKeyCode();
```

```
    if(keycode == KeyEvent.VK_F1) str += "F1 Key"; if(keycode ==
```

```
    KeyEvent.VK_F2) str += "F2 Key"; if(keycode == KeyEvent.VK_F3) str
```

```
    += "F3 Key"; if(keycode == KeyEvent.VK_PAGE_UP) str += "Page UP";
```

```
    if(keycode == KeyEvent.VK_PAGE_DOWN) str += "Page Down";
```

```
    a.setText(str);
```

```
    str = " " ;
```

```
}
```

```
public void keyRelease(KeyEvent ke)
```

```
{
```

```
}
```

```
public void keyTyped(KeyEvent ke)
```

```
{
```

```
}
```

```
public static void main(String args[])
```

```
{
```

```
    KeyBoardEvents kbe = new KeyBoardEvents();
```

```
    kbe.setSize(400,400); kbe.setVisible(true);
```

```
    kbe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
}
```

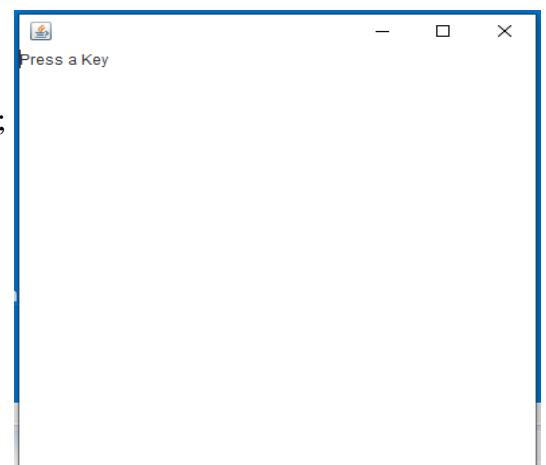
```
@Override
```

```
    public void keyReleased(KeyEvent arg0) {
```

```
        // TODO Auto-generated method stub
```

```
}
```

Output:



Handling Mouse Events

The user may click, release, drag, or move a mouse while interacting with the application. If the programmer knows what the user has done, he can write the code according to the mouse events. To trap the mouse events, **MouseListener** and **MouseMotionListener** interfaces of **java.awt.event** package are used.

MouseListener interface has the following methods.

1. **void mouseClicked(MouseEvent e):** void MouseClicked this method is invoked when the mouse button has been clicked (pressed and released) on a component.
2. **void mouseEntered(MouseEvent e):** this method is invoked when the mouse enters a component.
3. **void mouseExited(MouseEvent e):** this method is invoked when the mouse exits a component.
4. **void mousePressed(MouseEvent e):** this method is invoked when a mouse button has been pressed on a component.
5. **void mouseReleased(MouseEvent e):** this method is invoked when a mouse button has been released on a component.

MouseMotionListener interface has the following methods.

1. **void mouseDragged(MouseEvent e):** this method is invoked when a mouse button is pressed on a component and then dragged.
2. **void mouseMoved(MouseEvent e):** this method is invoked when a mouse cursor has been moved onto a component and then dragged.

The **MouseEvent** class has the following methods

1. **int getButton():** this method returns a value representing a mouse button, when it is clicked it returns 1 if left button is clicked, 2 if middle button, and 3 if right button is clicked.
2. **int getX():** this method returns the horizontal x position of the event relative to the source component.
3. **int getY():** this method returns the vertical y position of the event relative to the source component.

Program to create a text area and display the mouse event when the button on the mouse is clicked, when mouse is moved, etc. is done by user.

```
package com.myPack;
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
import javax.swing.*;
```

```
class MouseEvents extends JFrame implements MouseListener, MouseMotionListener  
{
```

```
    String str = " ";
```

```
    JTextArea ta;
```

```
    Container c;
```

```
    int x, y;
```

```
    MouseEvents()  
{
```

```

c=getContentPane();
c.setLayout(new FlowLayout());

ta = new JTextArea("Click the mouse or move it", 5, 20);
ta.setFont(new Font("Arial", Font.BOLD, 30));
c.add(ta);

ta.addMouseListener(this);
ta.addMouseMotionListener(this);
}

public void mouseClicked(MouseEvent me)
{
    int i = me.getButton();
    if(i==1)
        str+= "Clicked Button : Left";
    else if(i==2)
        str+= "Clicked Button : Middle";
    else if(i==3)
        str+= "Clicked Button : Right";
    this.display();
}

public void mouseEntered(MouseEvent me)
{
    str += "Mouse Entered";
    this.display();
}

public void mouseExited(MouseEvent me)
{
    str += "Mouse Exited";
    this.display();
}

public void mousePressed(MouseEvent me)
{
    x = me.getX();
    y= me.getY();
    str += "Mouse Pressed at :" +x + "\t" + y;
    this.display();
}

public void mouseReleased(MouseEvent me)
{
    x = me.getX();
    y= me.getY();
    str += "Mouse Released at :" +x + "\t" + y;
    this.display();
}

public void mouseDragged(MouseEvent me)
{
    x = me.getX();
    y= me.getY();

    str += "Mouse Dragged at :" +x + "\t" + y;
    this.display();
}

```

```

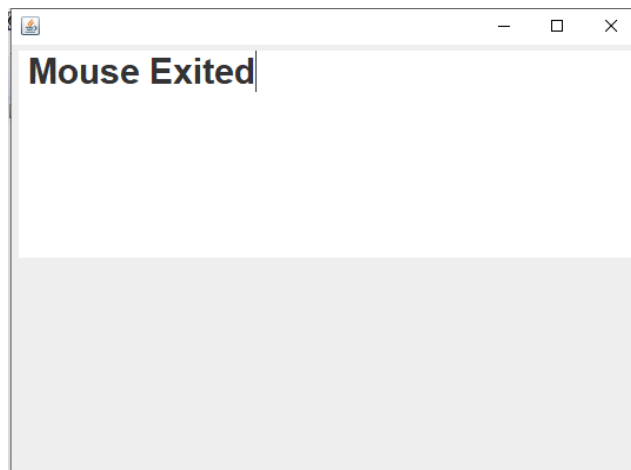
}
public void mouseMoved(MouseEvent me)
{
    x = me.getX();
    y= me.getY();
    str += "Mouse Moved at : " +x + "\t" + y;
    this.display();
}

public void display()
{
    ta.setText(str);
    str=" ";
}

public static void main(String args[])
{
    MouseEvents mes = new MouseEvents();
    mes.setSize(400,400);
    mes.setVisible(true);
    mes.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}

```

Output:



RK JAVA

Layout Managers

We Create several components like push buttons, checkboxes, radio buttons etc. in GUI. After creating these components, they should be placed in the frame (in AWT) or container (in Swing). While arranging them in the frame or container, they can be arranged in a particular manner by using layout managers. We have LayoutManger interface in java.awt package which is implemented in various classes which provides various layouts to arrange the components.

The following classes represents the layout managers in java

1. FlowLayout
2. BorderLayout
3. CardLayout
4. GridLayout
5. GridBagLayout
6. BoxLayout

To set a particular layout, we should first create an object to the layout class and pass the object to setLayout() method. For example to set FlowLayout to the container that holds the components, we can write:

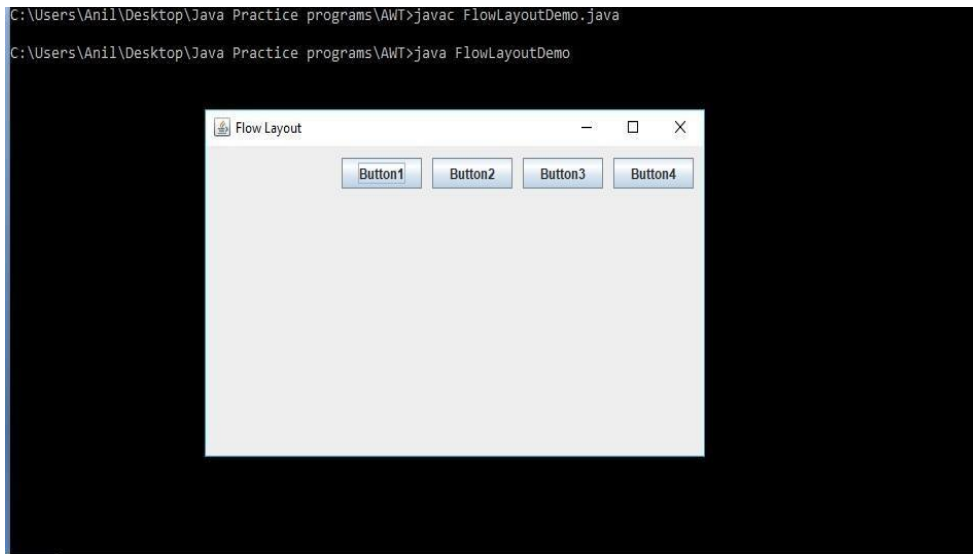
```
FlowLayout obj = new FlowLayout();
c.setLayout(obj);
```

FlowLayout

Program:

```
import java.awt.*;
import javax.swing.*;
class FlowLayoutDemo extends JFrame
{
    FlowLayoutDemo()
    {
        Container c = getContentPane();
        FlowLayout obj = new FlowLayout(FlowLayout.RIGHT, 10,10);
        c.setLayout(obj);
        JButton b1,b2,b3,b4;
        b1 = new JButton("Button1");
        b2 = new JButton("Button2");
        b3 = new JButton("Button3");
        b4 = new JButton("Button4");
        c.add(b1);
        c.add(b2);
        c.add(b3);
        c.add(b4);
    }
    public static void main (String args[])
    {
        FlowLayoutDemo demo = new FlowLayoutDemo();
        demo.setSize(500,300);
        demo.setTitle("Flow Layout");
        demo.setVisible(true);
        demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

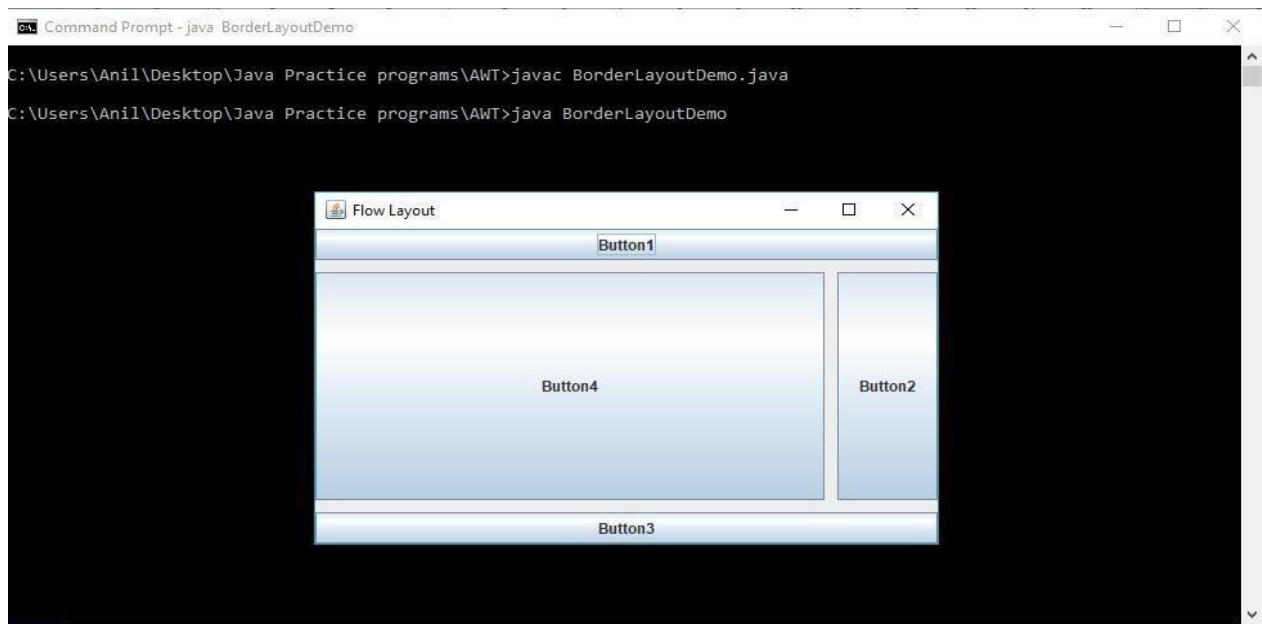
Output:



Border Layout

```
import java.awt.*;
import javax.swing.*;
class BorderLayoutDemo extends JFrame
{
    BorderLayoutDemo()
    {
        Container c = getContentPane();
        BorderLayout obj = new BorderLayout(10,10);
        c.setLayout(obj);
        JButton b1,b2,b3,b4;
        b1 = new JButton("Button1");
        b2 = new JButton("Button2");
        b3 = new JButton("Button3");
        b4 = new JButton("Button4");
        c.add("North" , b1);
        c.add("East" , b2);
        c.add("South" , b3);
        c.add("Center" , b4);
        c.add(b1, BorderLayout.NORTH);
        c.add(b2, BorderLayout.EAST);
        c.add(b3, BorderLayout.SOUTH);
        c.add(b4, BorderLayout.CENTER);
    }
    public static void main (String args[])
    {
        BorderLayoutDemo demo = new BorderLayoutDemo();
        demo.setSize(500,300);
        demo.setTitle("Flow Layout");
        demo.setVisible(true);
        demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

output



Card Layout

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

class CardLayoutDemo extends JFrame implements ActionListener

```
{
    Container c;
    CardLayout card;
    JButton b1,b2, b3,b4;

    CardLayoutDemo()
    {
        c = getContentPane();
        card = new CardLayout(50,10);
        c.setLayout(card);

        b1 = new JButton("Button1");
        b2 = new JButton("Button2");
        b3 = new JButton("Button3");
        b4 = new JButton("Button4");

        c.add("First Card" , b1);
        c.add("Second Card" , b2);
        c.add("Third Card" , b3);
        c.add("Fourth Card" , b4);

        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);
    }
}
```

```

public void actionPerformed(ActionEvent ae)
{
    card.next(c);
    //card.show(c,"Third Card")
}

public static void main (String args[])
{
    CardLayoutDemo demo = new CardLayoutDemo();
    demo.setSize(500,300);
    demo.setTitle("Card Layout");
    demo.setVisible(true);
    demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}

```

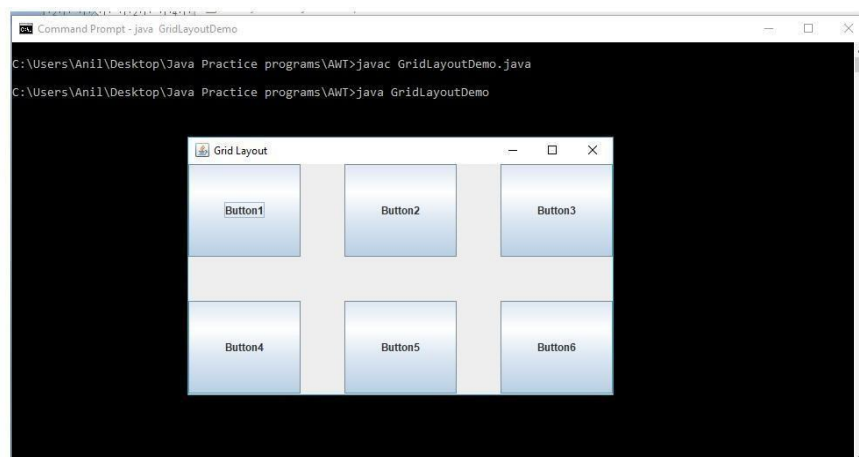
Output



Grid Layout

```
import java.awt.*;
import javax.swing.*;
class GridLayoutDemo extends JFrame
{
    GridLayoutDemo()
    {
        Container c = getContentPane();
        GridLayout grid = new GridLayout(2,3,50,50);
        c.setLayout(grid);
        //JButton b1,b2,b3,b4, b5, b6;
        JButton b1 = new JButton("Button1");
        JButton b2 = new JButton("Button2");
        JButton b3 = new JButton("Button3");
        JButton b4 = new JButton("Button4");
        JButton b5 = new JButton("Button5");
        JButton b6 = new JButton("Button6");
        c.add(b1);
        c.add(b2);
        c.add(b3);
        c.add(b4);
        c.add(b5);
        c.add(b6);
    }
    public static void main (String args[])
    {
        GridLayoutDemo demo = new GridLayoutDemo();
        demo.setSize(500,300);
        demo.setTitle("Grid Layout");
        demo.setVisible(true);
        demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output



GridBag Layout

```
import java.awt.*;
import javax.swing.*;
class GridBagLayoutDemo extends JFrame
{
    GridBagLayout gbag;
    GridBagConstraints cons;

    GridBagLayoutDemo()
    {
        Container c = getContentPane();
        gbag = new GridBagLayout();
        c.setLayout(gbag);

        cons = new GridBagConstraints();

        JButton b1 = new JButton("Button1");
        JButton b2 = new JButton("Button2");
        JButton b3 = new JButton("Button3");
        JButton b4 = new JButton("Button4");
        JButton b5 = new JButton("Button5");
        //JButton b6 = new JButton("Button6");

        cons.fill = GridBagConstraints.HORIZONTAL;
        cons.gridx =0;
        cons.gridy =0;

        cons.weightx = 0.7;
        cons.weighty = 0.7;
        gbag.setConstraints(b1,cons);
        c.add(b1);

        cons.gridx =1;
        cons.gridy =0;
        gbag.setConstraints(b2,cons);
        c.add(b2);

        cons.gridx =2;
        cons.gridy =0;
        gbag.setConstraints(b3,cons);
        c.add(b3);

        cons.gridx =0;
        cons.gridy =1;
        cons.ipady = 100;
        cons.gridwidth = 3;
```

```
gbag.setConstraints(b4,cons);
c.add(b4);
```

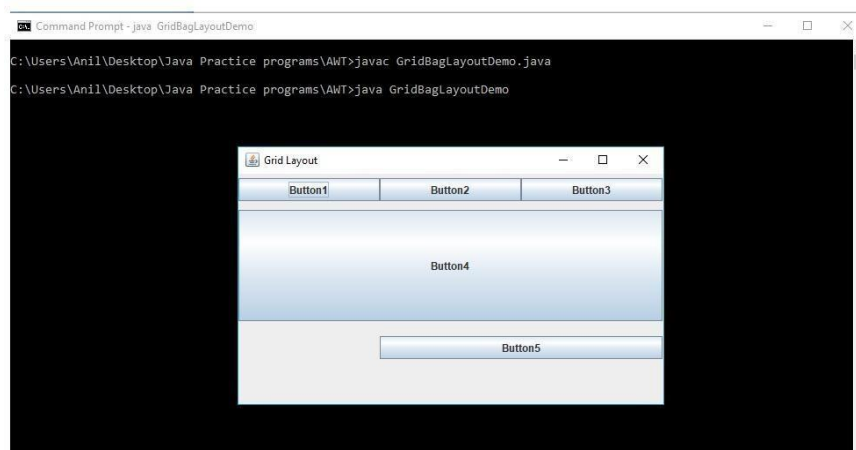
```
cons.gridx = 1;
cons.gridy = 2;
cons.ipady = 0;
cons.weighty = 0.8;
```

```
cons.anchor = GridBagConstraints.PAGE_END;
cons.insets = new Insets (0,0, 50,0);
cons.gridwidth =2;
```

```
gbag.setConstraints(b5,cons);
c.add(b5);
```

```
    }
    public static void main (String args[])
    {
        GridBagLayoutDemo demo = new GridBagLayoutDemo();
        demo.setSize(500,300);
        demo.setTitle("Grid Layout");
        demo.setVisible(true);
        demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output:



BoxLayout

```
import java.awt.*;
import javax.swing.*;

class BoxLayoutDemo extends JFrame
{
    BoxLayoutDemo()
    {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        MyPanel1 mp1 = new MyPanel1();
        c.add(mp1);

        MyPanel2 mp2 = new MyPanel2();
        c.add(mp2);
    }

    public static void main(String args[])
    {
        BoxLayoutDemo demo = new BoxLayoutDemo();
        demo.setSize(500,300);
        demo.setTitle("Box Layout");
        demo.setVisible(true);
        demo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

class MyPanel1 extends JPanel
{
    MyPanel1()
    {
        BoxLayout box1 = new BoxLayout(this, BoxLayout.X_AXIS);
        setLayout(box1);

        JButton b1, b2, b3;
        b1 = new JButton("Button1");
        b2 = new JButton("Button2");
        b3 = new JButton("Button3");

        add(b1);
        add(b2);
        add(b3);
    }
}

class MyPanel2 extends JPanel
{
    MyPanel2()
```



```

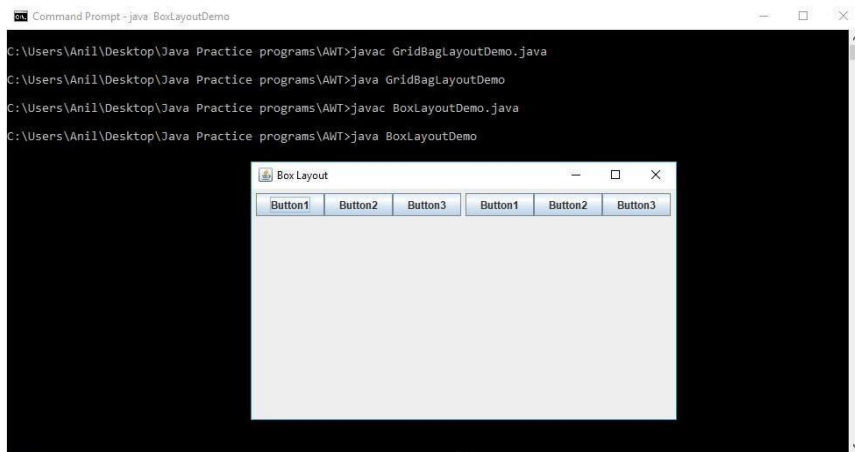
{
BoxLayout box2 = new BoxLayout(this, BoxLayout.X_AXIS);
setLayout(box2);

JButton b1, b2, b3;
b1 = new JButton("Button1");
b2 = new JButton("Button2");
b3 = new JButton("Button3");

add(b1);
add(b2);
add(b3);
}
}

```

Output:



Swing Controls

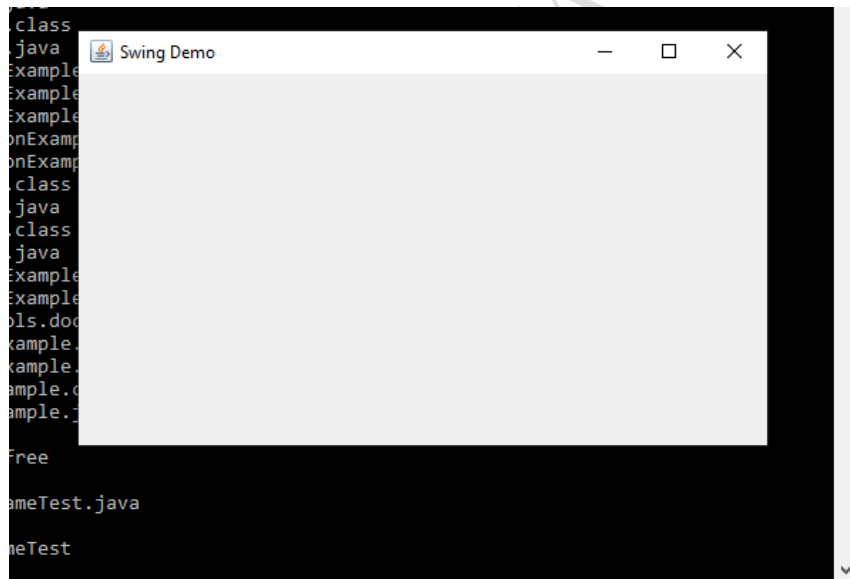
Jframe

//Frame creation in Swing

```
import javax.swing.JFrame;
```

```
public class JFrameTest extends JFrame
{
    public JFrameTest()
    {
        setSize(500,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Swing Demo");
        setVisible(true);
    }
    public static void main(String args[])
    {
        JFrameTest f = new JFrameTest();
    }
}
```

Output:



JLabel

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class JLabelTest1 extends JFrame
{
    public JLabelTest1()
    {
        setTitle("Label Demo");
        setSize(500,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

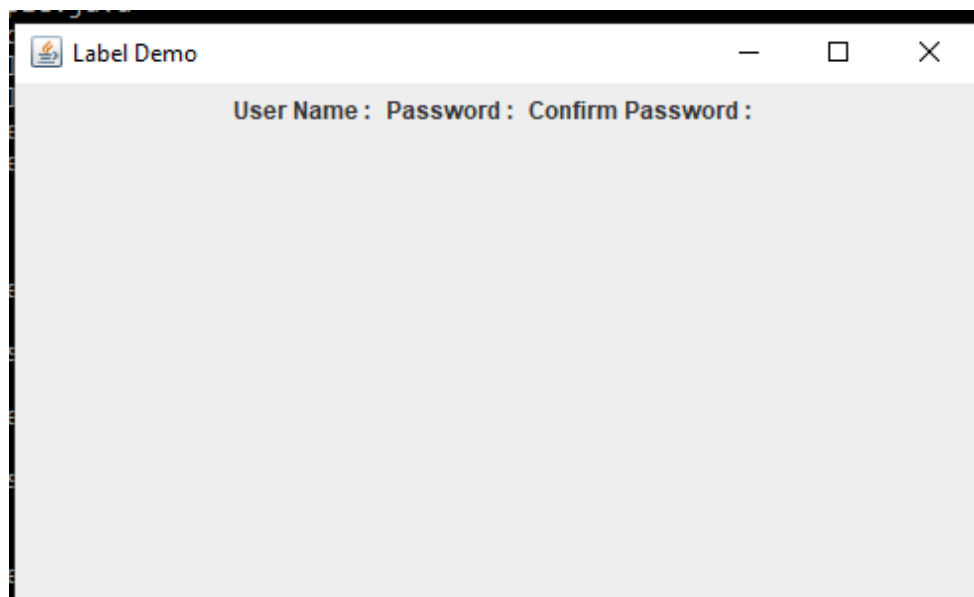
        JLabel lb1 = new JLabel("User Name : " , JLabel.RIGHT);
        JLabel lb2 = new JLabel("Password : " , JLabel.RIGHT);
        JLabel lb3 = new JLabel("Confirm Password : " , JLabel.RIGHT);

        lb1.setVerticalAlignment(JLabel.TOP);
        lb1.setToolTipText("Enter UserName");

        getContentPane().add(lb1);
        getContentPane().add(lb2);
        getContentPane().add(lb3);
    }

    public static void main(String args[])
    {
        new JLabelTest1().setVisible(true);
    }
}
```

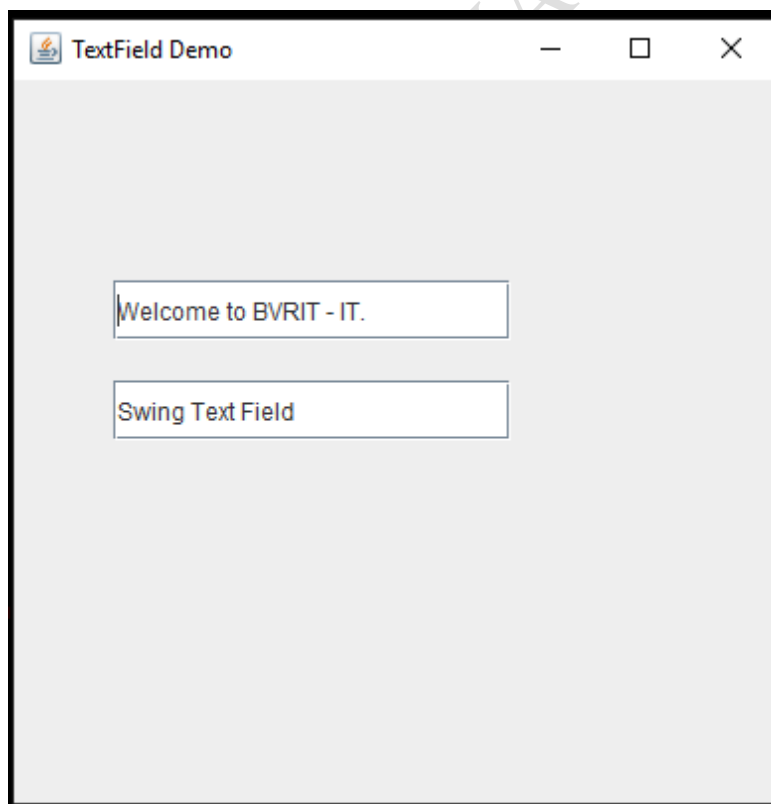
Output:



JTextField

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
{
    JFrame f= new JFrame("TextField Demo");
    JTextField t1,t2;
    t1=new JTextField("Welcome to BVRIT - IT.");
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("Swing Text Field");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}
```

Output:



Jbutton

//Program to use JButton swing component with event

```
import javax.swing.*;
import java.awt.event.*;

public class ActionEventTest implements ActionListener
{
    ActionEventTest()
    {
        JFrame f = new JFrame("BVRIT");
        JButton b = new JButton("Message");
        f.add(b);
        b.addActionListener(this);

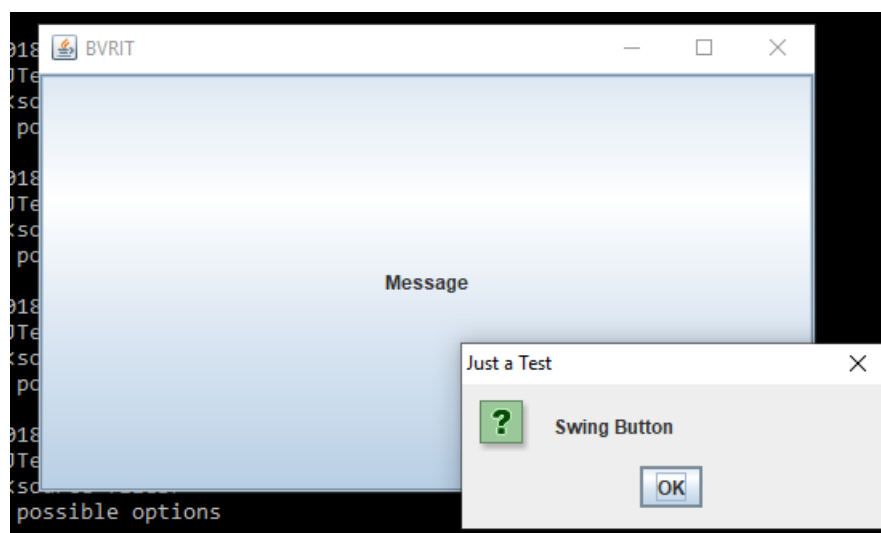
        f.setSize(500,300);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        f.setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showConfirmDialog(null,"Swing Button", "Just a Test",
        JOptionPane.PLAIN_MESSAGE);
    }

    public static void main(String args[])
    {
        new ActionEventTest();
    }
}
```

Output:



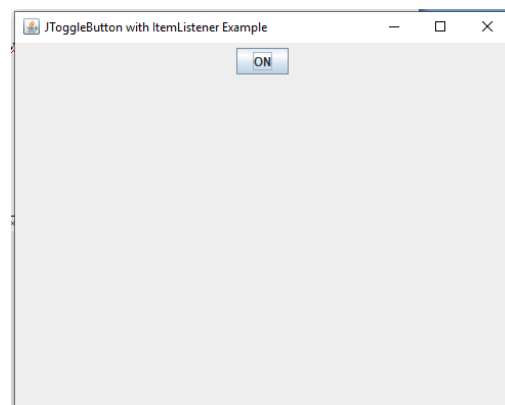
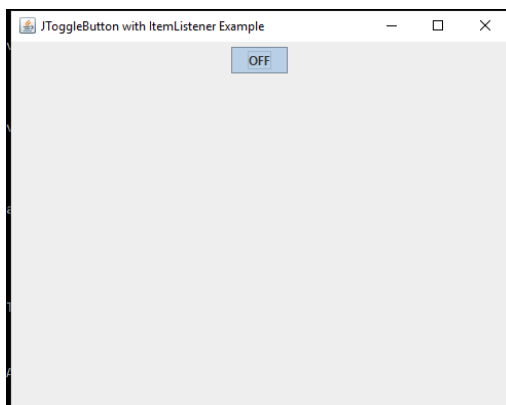
JToggleButtton

JToggleButton

```
import java.awt.FlowLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.JFrame;
import javax.swing.JToggleButton;

public class JToggleButtonExample extends JFrame implements ItemListener {
    public static void main(String[] args) {
        new JToggleButtonExample();
    }
    private JToggleButton button;
    JToggleButtonExample() {
        setTitle("JToggleButton with ItemListener Example");
        setLayout(new FlowLayout());
        setJToggleButton();
        setAction();
        setSize(500, 400);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    private void setJToggleButton() {
        button = new JToggleButton("ON");
        add(button);
    }
    private void setAction() {
        button.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent eve) {
        if (button.isSelected())
            button.setText("OFF");
        else
            button.setText("ON");
    }
}
```

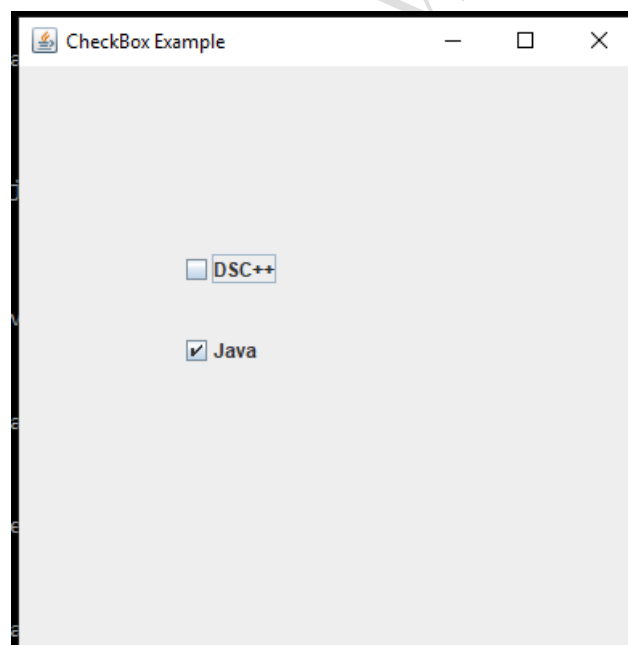
Output:



JCheckBox

```
import javax.swing.*;
public class CheckBoxExample
{
    CheckBoxExample(){
        JFrame f= new JFrame("CheckBox Example");
        JCheckBox checkBox1 = new JCheckBox("DSC++");
        checkBox1.setBounds(100,100, 100,50);
        JCheckBox checkBox2 = new JCheckBox("Java", true);
        checkBox2.setBounds(100,150, 100,50);
        f.add(checkBox1);
        f.add(checkBox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String args[])
    {
        new CheckBoxExample();
    }
}
```

Output:

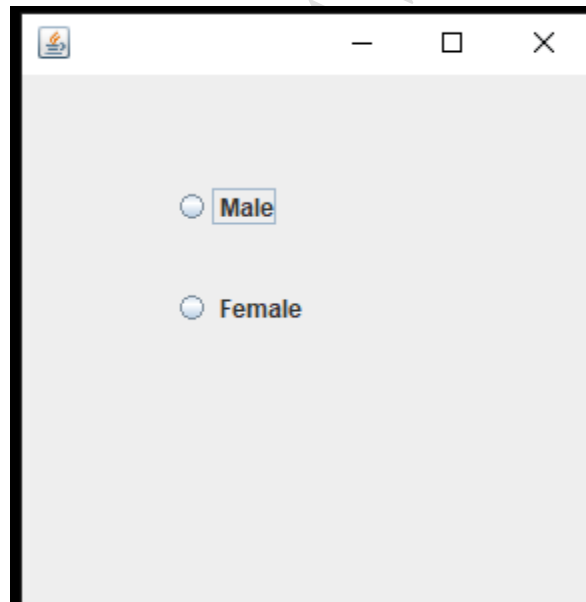


JRadioButton

JRadioButton

```
import javax.swing.*;
public class RadioButtonExample {
    JFrame f;
    RadioButtonExample(){
        f=new JFrame();
        JRadioButton r1=new JRadioButton(" Male");
        JRadioButton r2=new JRadioButton(" Female");
        r1.setBounds(75,50,100,30);
        r2.setBounds(75,100,100,30);
        ButtonGroup bg=new ButtonGroup();
        bg.add(r1);bg.add(r2);
        f.add(r1);f.add(r2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new RadioButtonExample();
    }
}
```

Output:

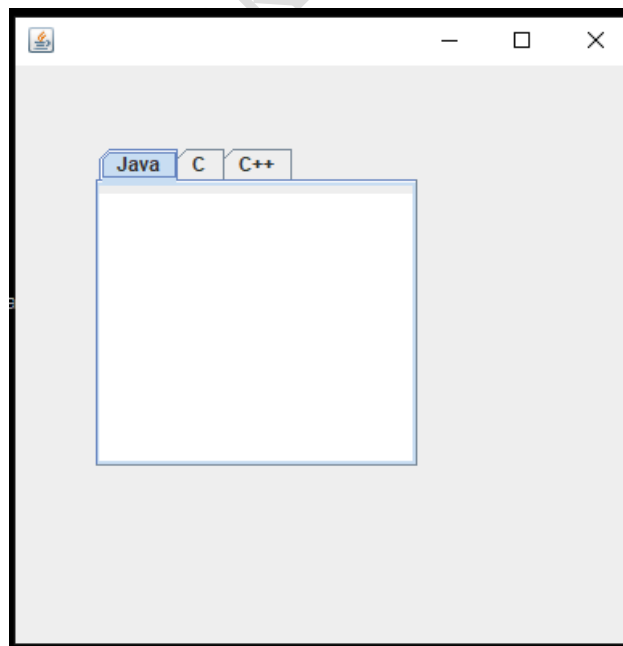


JTabbedPane

JTabbedPane

```
import javax.swing.*;
public class TabbedPaneExample {
    JFrame f;
    TabbedPaneExample(){
        f=new JFrame();
        JTextArea ta=new JTextArea(200,200);
        JPanel p1=new JPanel();
        p1.add(ta);
        JPanel p2=new JPanel();
        JPanel p3=new JPanel();
        JTabbedPane tp=new JTabbedPane();
        tp.setBounds(50,50,200,200);
        tp.add("Java",p1);
        tp.add("C",p2);
        tp.add("C++",p3);
        f.add(tp);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new TabbedPaneExample();
    }
}
```

Output:



JsScrollPane

JsScrollPane

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.*;

public class JScrollPaneExample {
    private static final long serialVersionUID = 1L;

    private static void createAndShowGUI() {

        // Create and set up the window.
        final JFrame frame = new JFrame("Scroll Pane Example");

        // Display the window.
        frame.setSize(500, 500);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // set flow layout for the frame
        frame.getContentPane().setLayout(new FlowLayout());

        JTextArea textArea = new JTextArea(10, 20);
        JScrollPane scrollableTextArea = new JScrollPane(textArea);

        scrollableTextArea.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

        scrollableTextArea.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

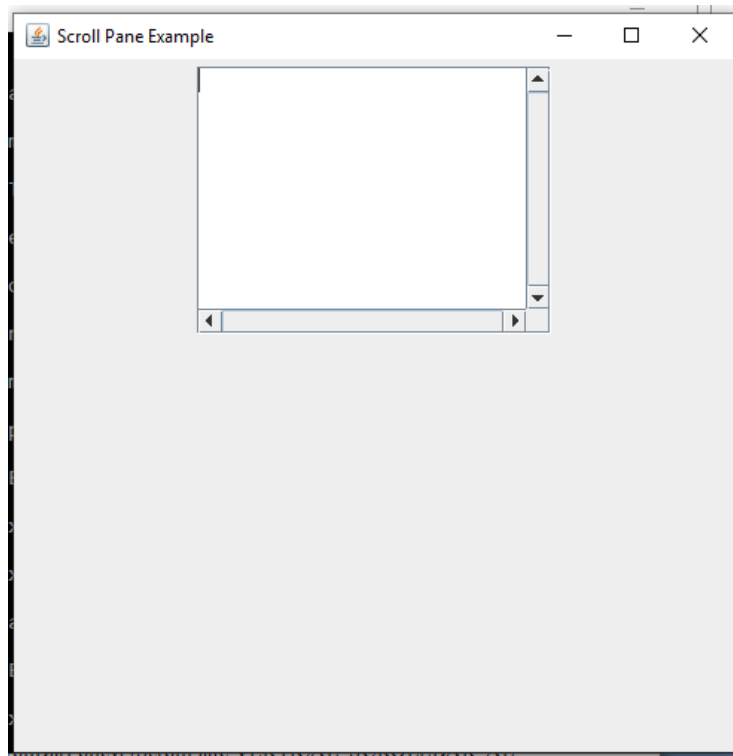
        frame.getContentPane().add(scrollableTextArea);
    }

    public static void main(String[] args) {

        javax.swing.SwingUtilities.invokeLater(new Runnable() {

            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Output:

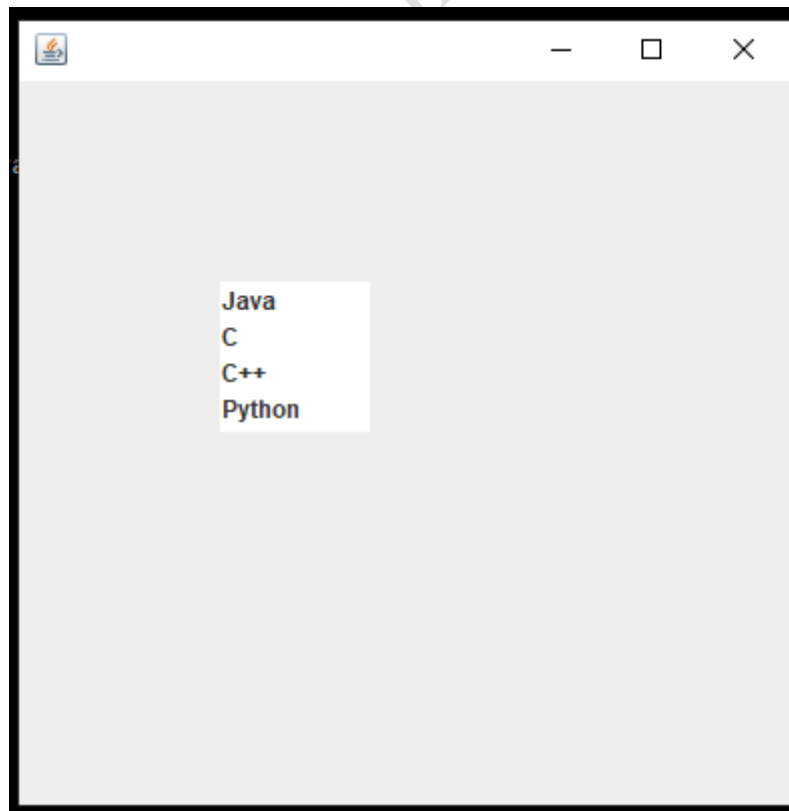


RK JAVA

JList

```
import javax.swing.*;
public class ListExample
{
    ListExample(){
        JFrame f= new JFrame();
        DefaultListModel<String> l1 = new DefaultListModel<>();
        l1.addElement("Java");
        l1.addElement("C");
        l1.addElement("C++");
        l1.addElement("Python");
        JList<String> list = new JList<>(l1);
        list.setBounds(100,100, 75,75);
        f.add(list);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String args[])
    {
        new ListExample();
    }
}
```

Output:

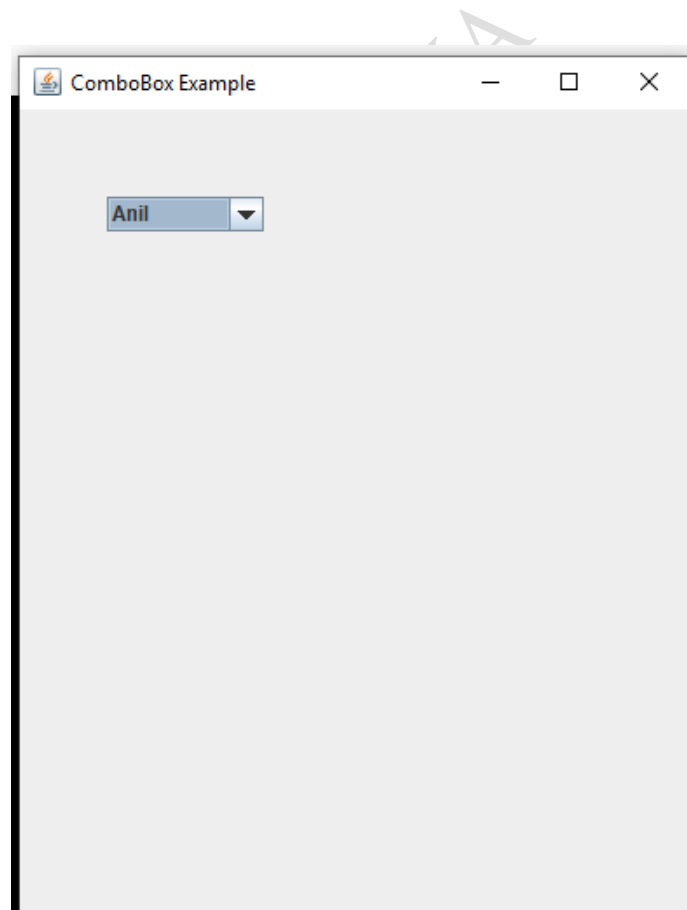


JComboBox

JComboBox

```
import javax.swing.*;
public class ComboBoxExample {
    JFrame f;
    ComboBoxExample(){
        f=new JFrame("ComboBox Example");
        String name[]={ "Anil","Akshara","Gabber","Geetha","Govind"};
        JComboBox cb=new JComboBox(name);
        cb.setBounds(50, 50,90,20);
        f.add(cb);
        f.setLayout(null);
        f.setSize(400,500);
        f.setVisible(true);
        //f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new ComboBoxExample();
    }
}
```

Output:



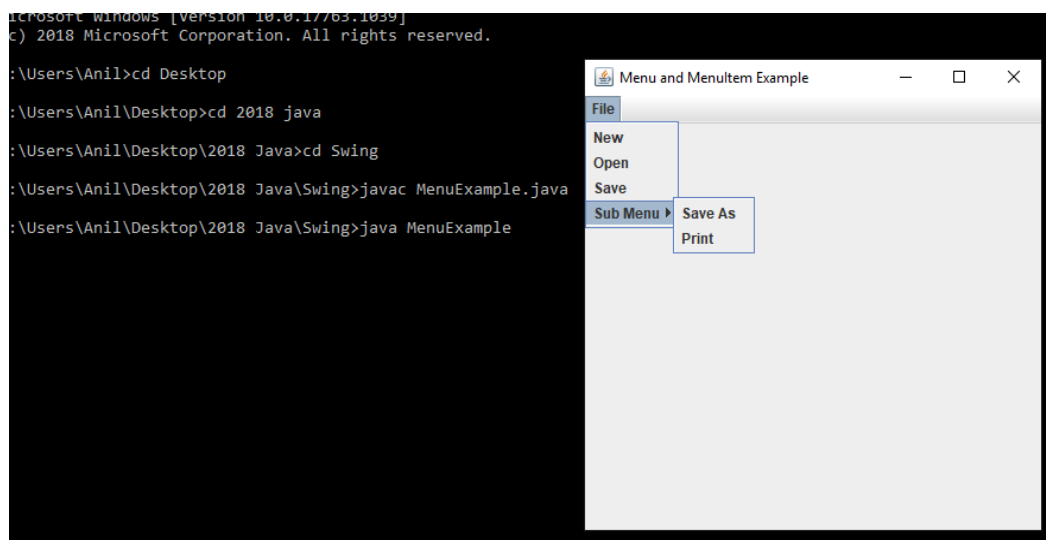
Swing Menus

Swing Menus

```
import javax.swing.*;
class MenuExample
{
    JMenu menu, submenu;
    JMenuItem i1, i2, i3, i4, i5;
    MenuExample(){
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar();
        menu=new JMenu("File");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("New");
        i2=new JMenuItem("Open");
        i3=new JMenuItem("Save");
        i4=new JMenuItem("Save As");
        i5=new JMenuItem("Print");
        menu.add(i1); menu.add(i2); menu.add(i3);
        submenu.add(i4); submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);

        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String args[])
    {
        new MenuExample();
    }
}
```

Output:



Dialogs

Dialogs

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
    private static JDialog d;
    DialogExample() {
        JFrame f= new JFrame();
        d = new JDialog(f , "Dialog Example", true);
        d.setLayout( new FlowLayout() );
        JButton b = new JButton ("OK");
        b.addActionListener ( new ActionListener()
        {
            public void actionPerformed((ActionEvent e )
            {
                DialogExample.d.setVisible(false);
            }
        });
        d.add( new JLabel ("Click button to continue."));
        d.add(b);
        d.setSize(300,300);
        d.setVisible(true);
    }
    public static void main(String args[])
    {
        new DialogExample();
    }
}
```

Output:

