# DEPARTMENT OF INFORMATION TECHNOLOGY

# Java Programming (R22CSI2215)

## UNIT-I

**Foundations of Java:** History of Java, Java Features, Variables, Data Types, Operators, Expressions, Control Statements. Elements of Java - Class, Object, Methods, Constructors and Access Modifiers, Generics, Inner classes, String class and Annotations.

**OOP Principles:** Encapsulation – concept, setter and getter method usage, this keyword. Inheritance - concept, Inheritance Types, super keyword. Polymorphism – concept, Method Overriding usage and Type Casting. Abstraction – concept, abstract keyword and Interface.

## Foundations of Java

- Java is a **Programming Language** and a **Platform Independent**. 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.

## 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".**

### 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 desk top 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 Java FX. 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 smart phones. 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 irctc.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.

## Java OOPs Concepts

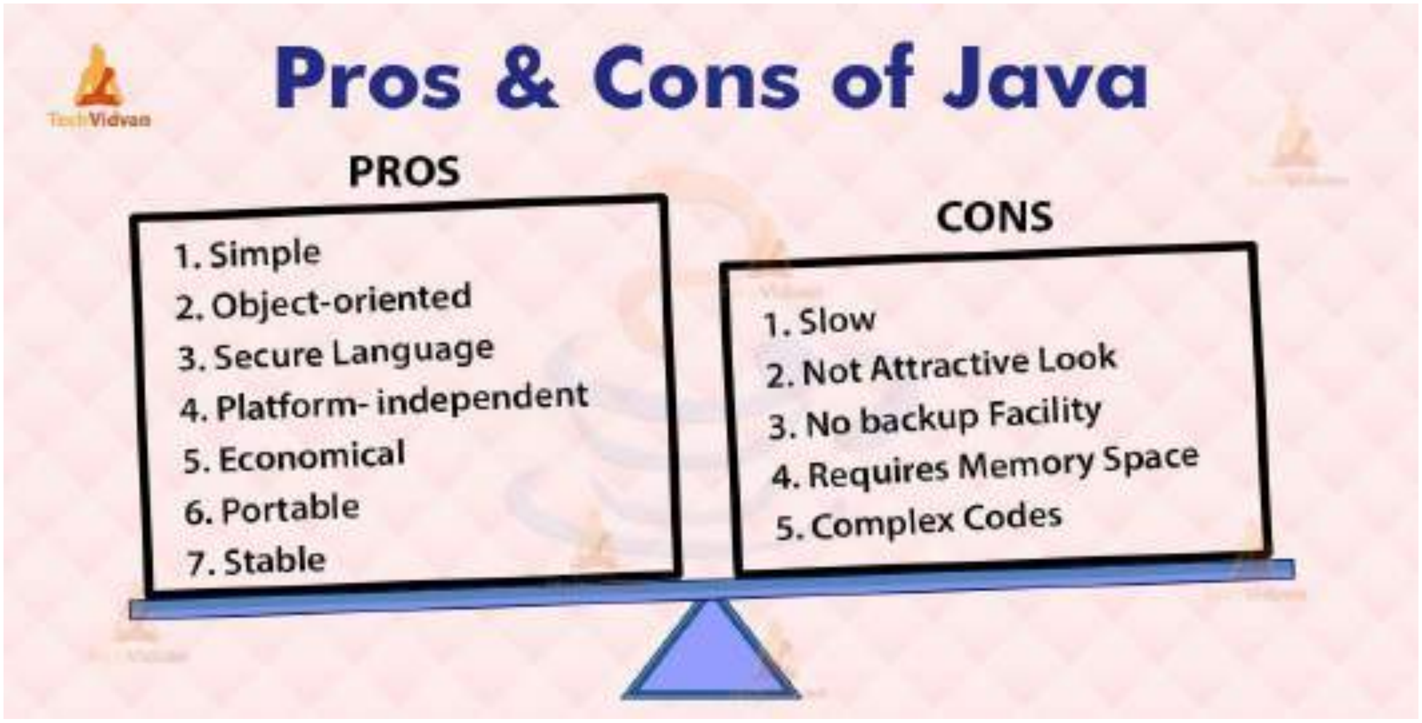
- Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.
- **Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.
- **Small talk** 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.

## 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.
- We will explain 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 straight forward, 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 in to smaller ones so they are easy to understand.

**3. Java is a secure language**

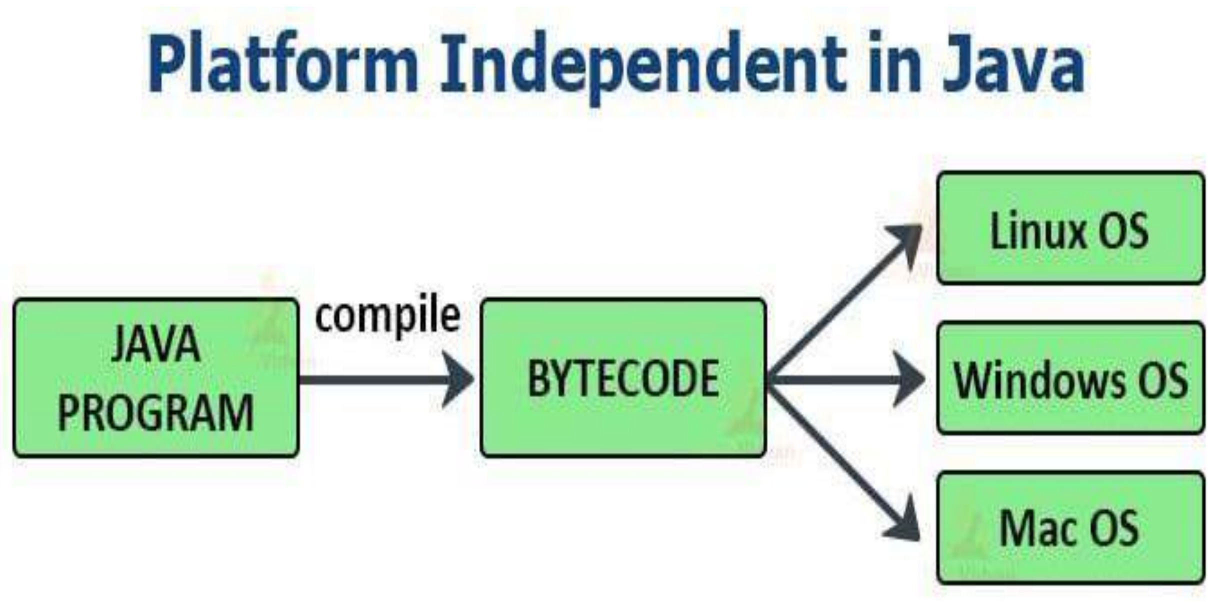
- Java reduces security threats and risks by avoiding the use of explicit pointers. A pointer stores the memory address of another value that can cause unauthorized access to memory.
- This issue is resolved by removing the concept of pointers. Also, there is a Security manager in Java for each application 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 Any where(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**

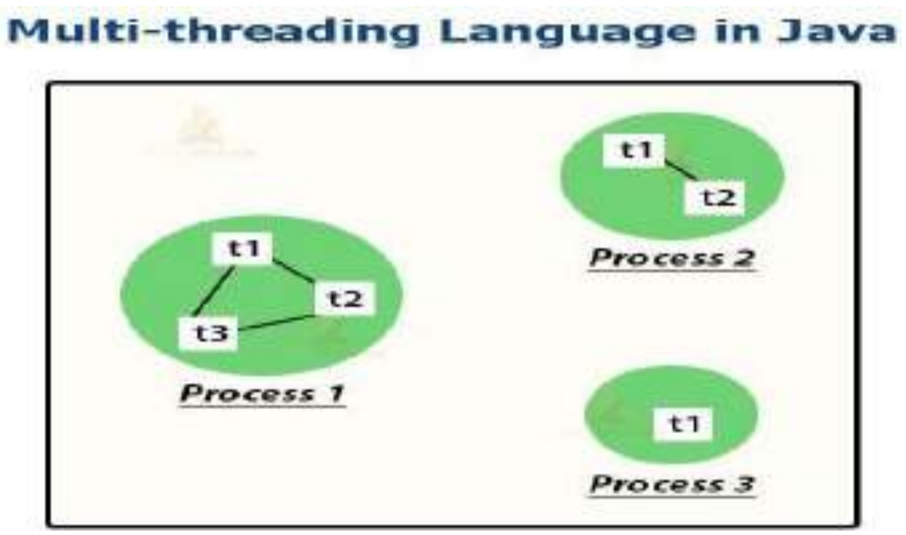
- There is automatic memory management in Java that is managed by the Java Virtual Machine(JVM).
- Whenever the objects are not used by programs anymore and they do not refer to anything that they do not need to be dereferenced or removed by the explicit programming.

- 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.



**10. Java is stable**

- Java programs are more stable as compared to programs of other languages. More over, 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 a void 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.

**2. 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, some times the garbage collector leads in the poor performance of Java as it consumes more CPU time.

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

- Though there are many GUI builders in Java for creating the graphical interfaces till they are not suitable for creating complicated UI. There are many inconsistencies while using them.
- There are many popular frameworks such as Swing, SWT, Java FX,  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.

**4. 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.

**5. 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.

**6. Verbose and Complex codes**

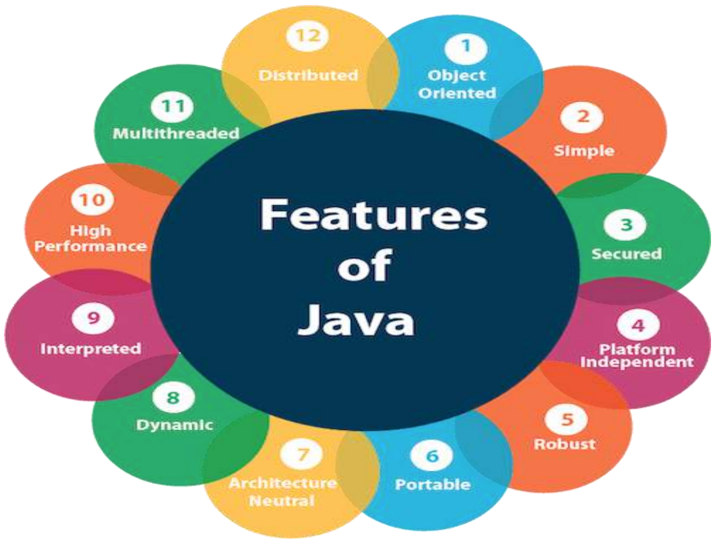
- Java codes are verbose, meaning that there are many words in it and there are many long and complex sentences that are difficult to read and understand. This can reduce the readability of the code.

- 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.

## Features of Java / 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 byte code. 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++, SmallTalk, 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 13employees of Sun Micro systems, Inc. which is lead by James Gosling, in 1991.
- The team includes persons like Patrick Naughton, Chris Warth, EdFrank, 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.

## 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 intx=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;//Instancevariables
        static int z;// Static variable

        StaticVariablesExample(intx,inty)
        {
                 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 it's 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

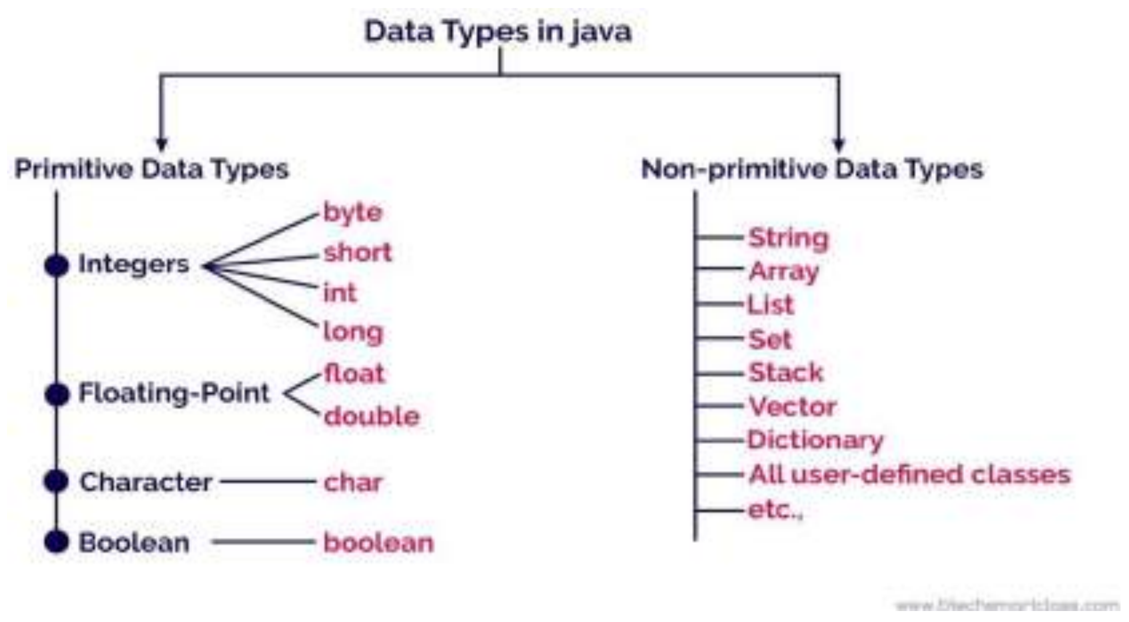
```
public class FinalVariableExample

{
        final int a = 10;

            void show()
            {
                        System.out.println("a= "+a);
                        a=20;       //Error due to final variable cann't be modified
            }

            public static void main(String[] args)
            {
                        FinalVariableExample obj = new FinalVariableExample();
                        obj.show();
            }
}
```

## 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.

    **1. Primitive Data Types**

    **2. Non-primitive Data Types**



## Primitive Data Types

- The primitive data types are built-in data types and they specify the type of values to red 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 | -128to +127 | 0 |
| short | Whole numbers | 2 bytes | -32768to +32767 | 0 |
| int | Whole numbers | 4 bytes | -2,147,483,648to +2,147,483,647 | 0 |
| long | Whole numbers | 8 bytes | -9,223,372,036,854,775,808to +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 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.

1. **Arithmetic Operators**
2. **Relational (or) Comparison Operators**
3. **Logical  Operators**
4. **Assignment Operators**
5. **Bitwise Operators**
6. **Conditional Operators**

### 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 expression and when it is used as post, the value is get modified after 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 < 5is FALSE |
| > | Returns TRUE if the first value is larger than second value otherwise returns FALSE | 10> 5isTRUE |
| <= | Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE | 10 <= 5is FALSE |
| >= | Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE | 10>= 5isTRUE |
| == | Returns TRUE if both values are equal otherwise returns FALSE | 10 == 5is FALSE |
| != | Returns TRUE if both values are not equal otherwise returns FALSE | 10!= 5 isTRUE |

Look at the following example program.

### Example

```
publicclassRelationalOperators {
        publicstaticvoidmain(String[]args){
                boolean a;
                a=10<5;
                System.out.println("10<5is"+a); a = 10>5;
                System.out.println("10>5is"+a); a = 10<=5;
                System.out.println("10<=5is"+a); a = 10>=5;
                System.out.println("10>=5is"+a); a = 10==5;
                System.out.println("10==5is"+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 in to 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 FLASE 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-circuitOR**-SimilartoLogicalOR(\|),butonceadecisionisfinalizedit 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 andShort-circuitoperatorsbotharesimilar,butincaseofshort-circuitoperatorsoncethe decision is finalized it does not evaluate remaining expressions.

Look at the following example program.

### Example

```
publicclass 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+",andz="+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 (R value) to the left-hand side variable (L value). 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

```
publicclassAssignmentOperators{
        publicstaticvoidmain(String[]args){
                int a = 10, b = 20, c;
                booleanx=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 are1otherwise it is 0 | A&B ⇒16(10000) |
| \| | The result of Bitwise OR is 0 if all the bits are 0 otherwise it is 1 | A\|B⇒29(11101) |
| ^ | The result of Bitwise XOR is 0 if all the bits are same otherwise it is 1 | A^ B ⇒13(01101) |
| ~ | The result of Bitwise once complement is negation of the bit (Flipping) | ~A⇒6 (00110) |
| << | The Bitwise left shift operator shifts all the bits to the left by the specified number of positions | A<<2⇒100(1100100) |
| >> | The Bitwise right shift operator shifts all the bits to the right by the specified number of positions | A>>2⇒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.

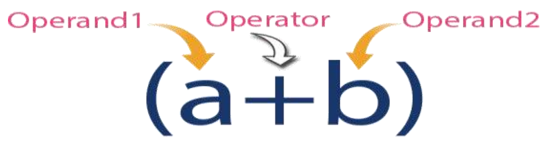
- **Infix Expression**
- **Postfix Expression**
- **Prefix Expression**

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.

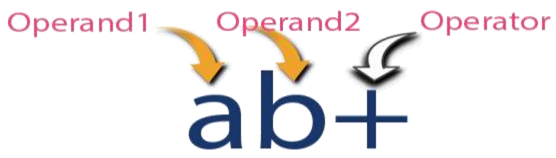
### Example



## Postfix Expression

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

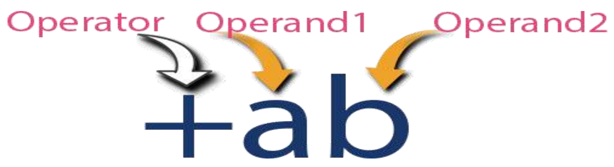
### Example



## Prefix Expression

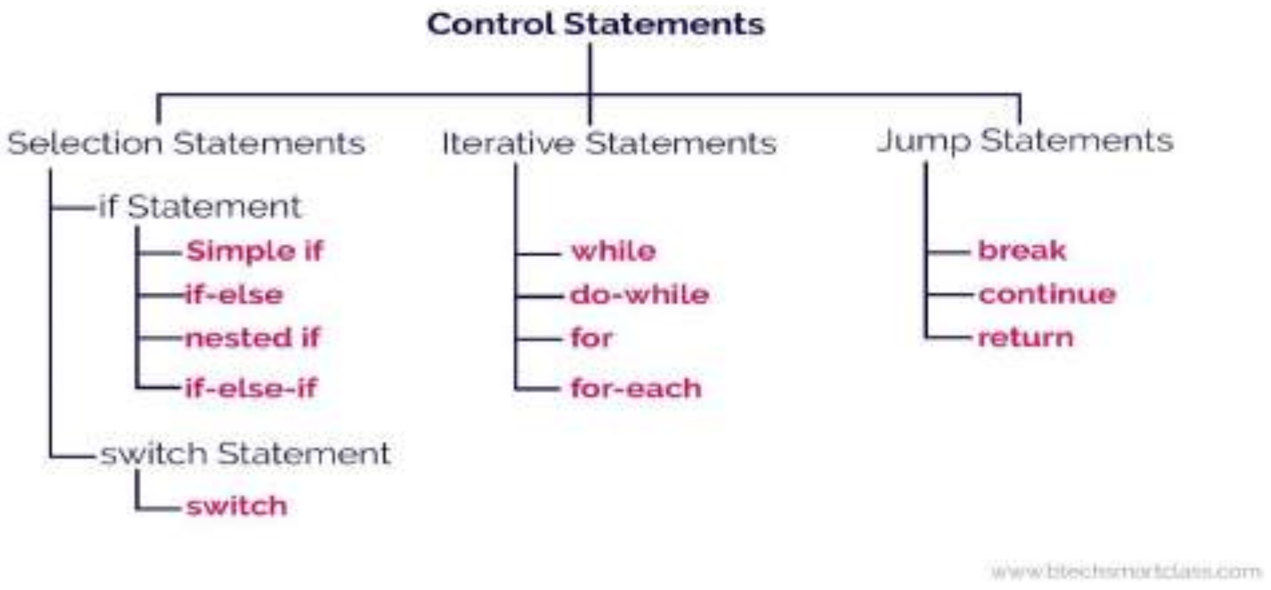
The expression in which the operator is used before operands is called a prefix expression. The prefix expression has the following general structure.

**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.
- 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

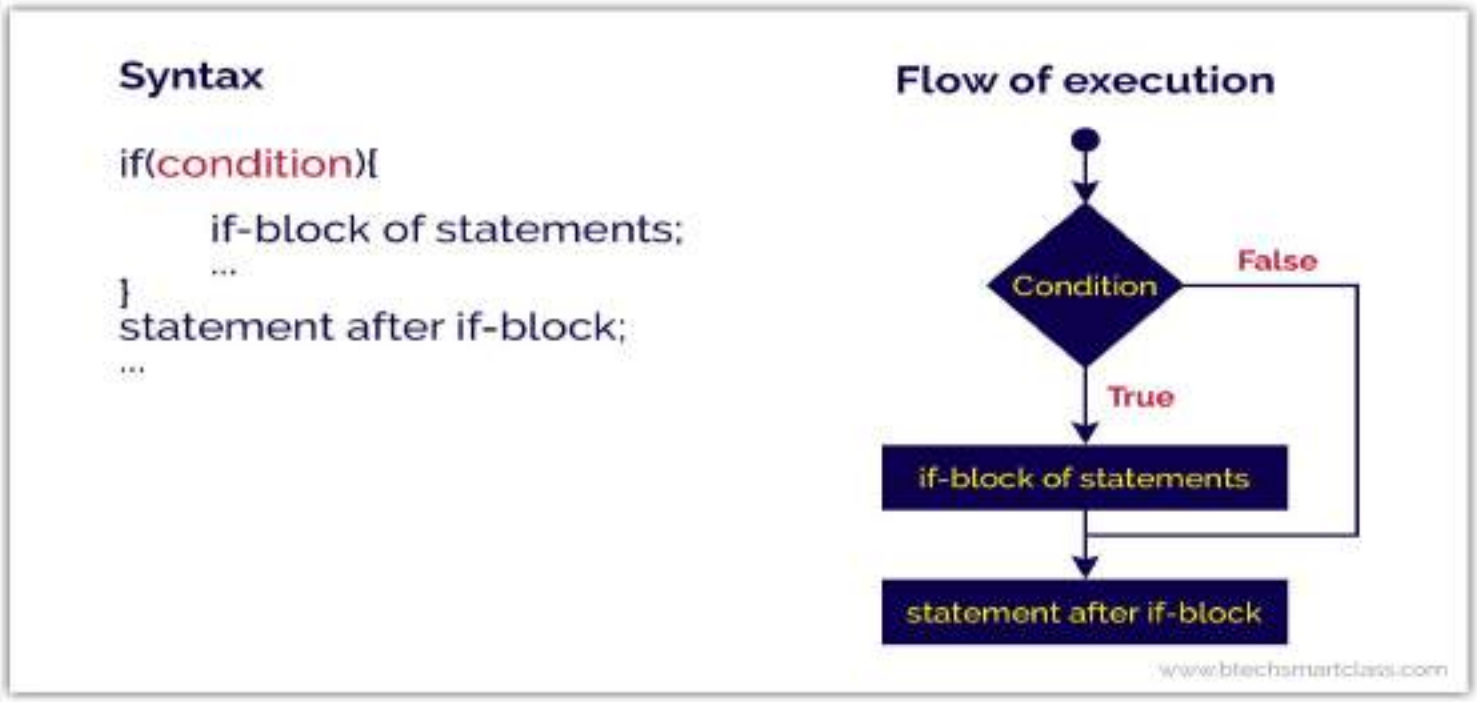
In java, the jump statements are used to terminate a block or take the execution control to the next iteration.Java provides the following jump statements.

- break
- continue
- return

### 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;
...
```

## Java Program

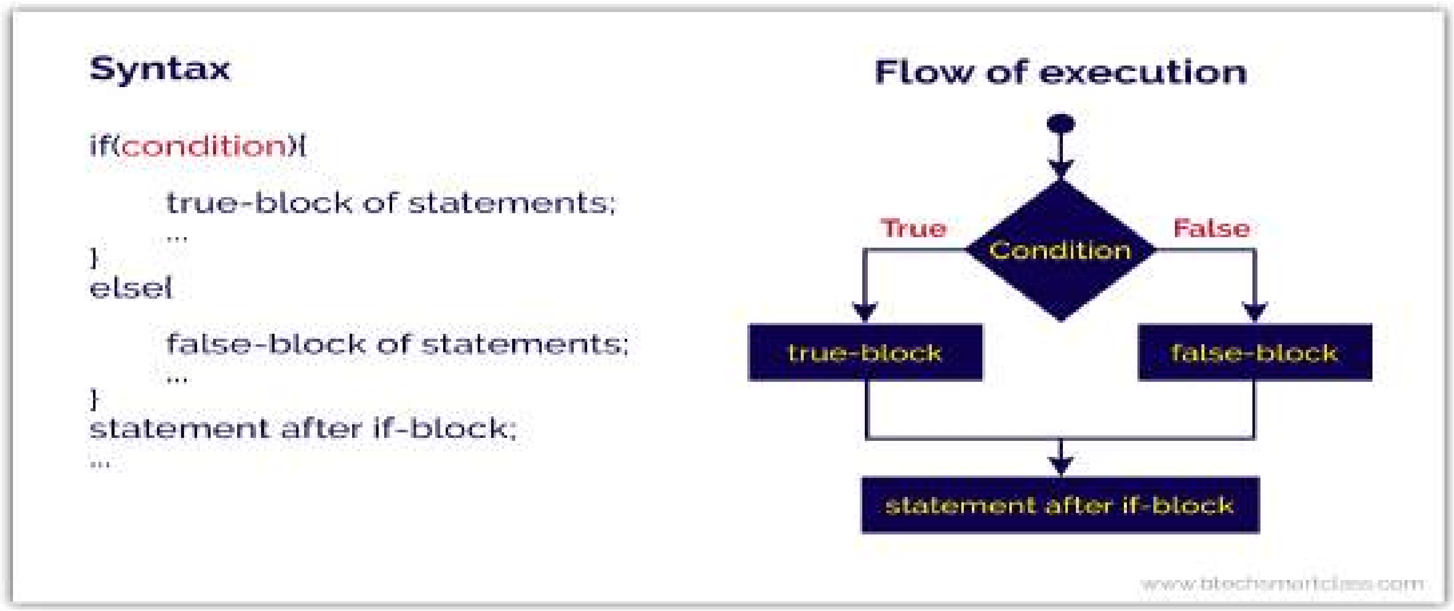
```
import java.util.Scanner;

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 by5!!");
                }
                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.

## 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.



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

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("\n Given 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-elseif 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_1true-
    block;
    ...
}
else if(condition_2){
    condition_2 true-block;
    condition_1false-
    blocktoo;
```

Let's look at the following example in java code.

### Java Program

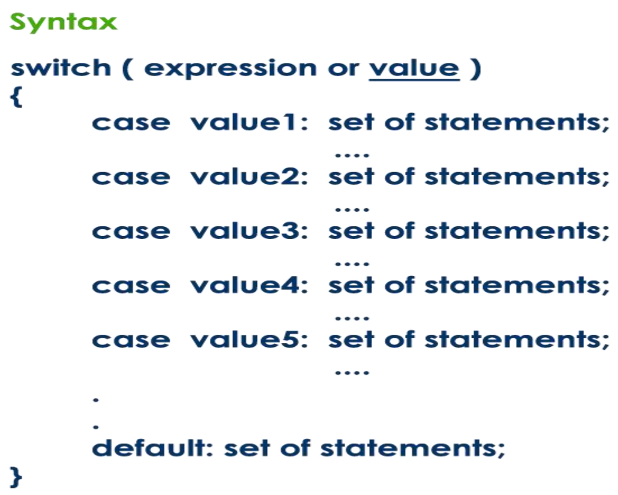
```
import  java.util.Scanner;
public class IfElseIfStatementTest{
                public static void main(String[] args){
                                int num1, num2,num3;
                                Scanner read = new Scanner(System.in);
                                System.out.print("Enter 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.

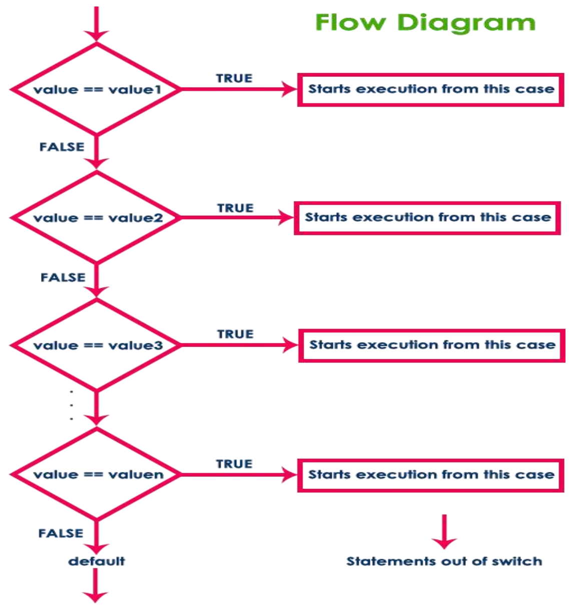
Whenever the given value matches the value associated with an option, the execution starts from that option. In the switch statement, every option is defined as a **case**.

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;
}
```



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 ;
                        case3: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 ;
                        case7: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

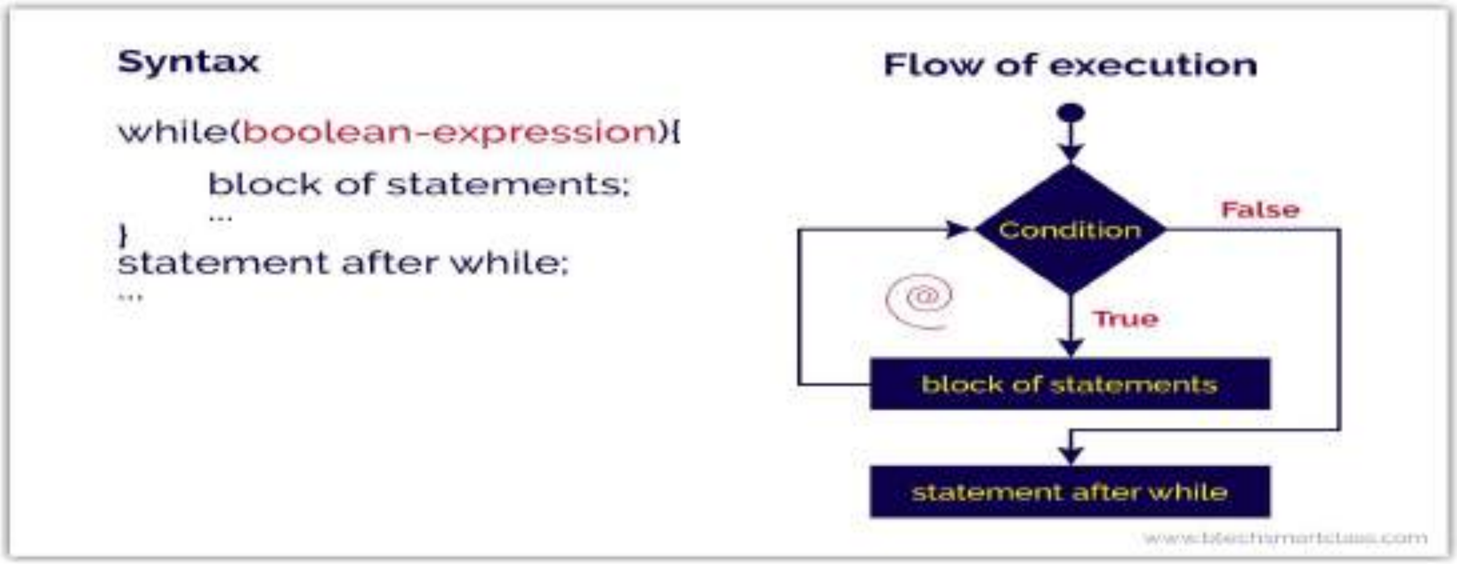
The java programming language provides a set of iterative statements that are used to execute a statement or a block of statements repeatedly as long as the given condition is true. The iterative statements are also known as looping statements or repetitive statements. Java provides the following iterative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

Let's look at the following example in java code.

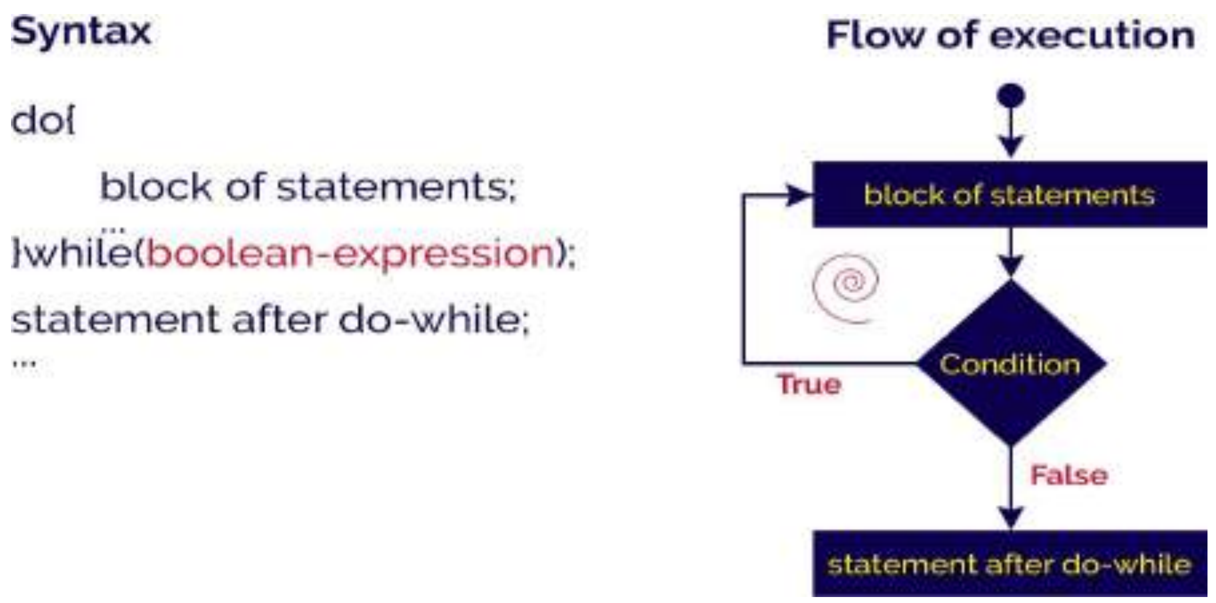
### Java Program

```
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;
...
```

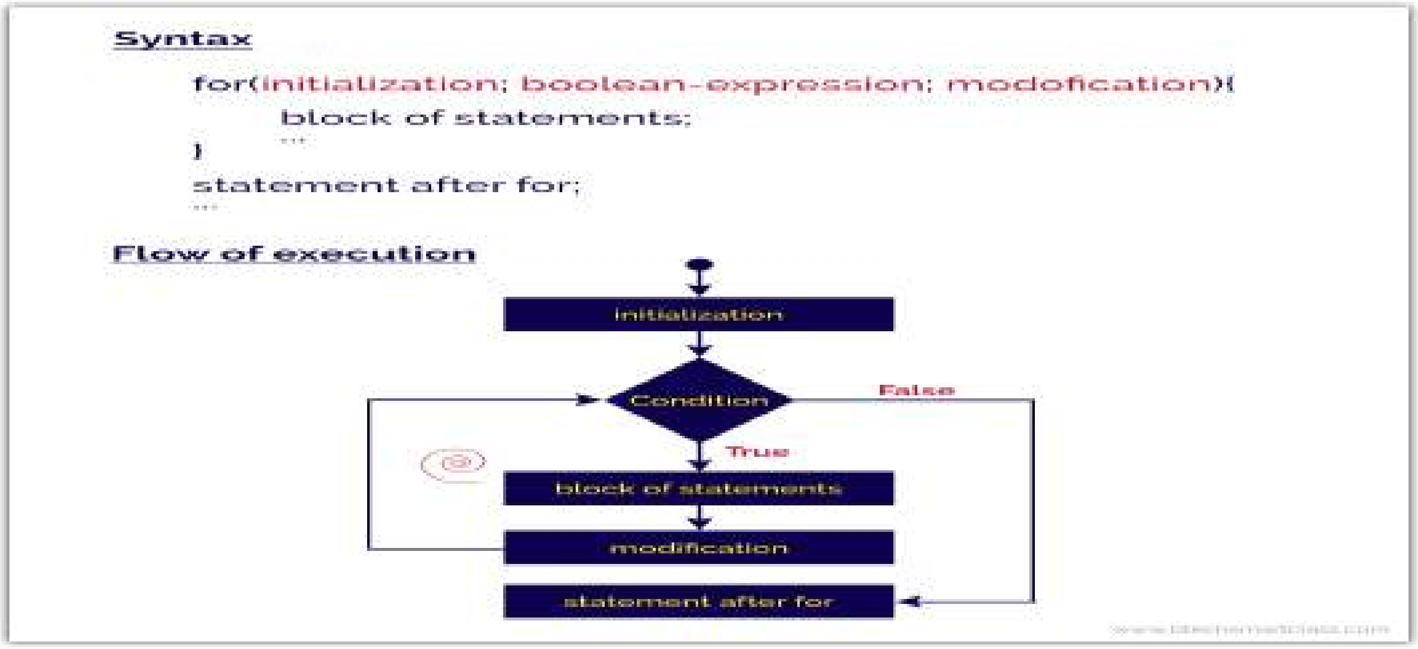
Let's look at the following example in java code.

### 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.



Syntax

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

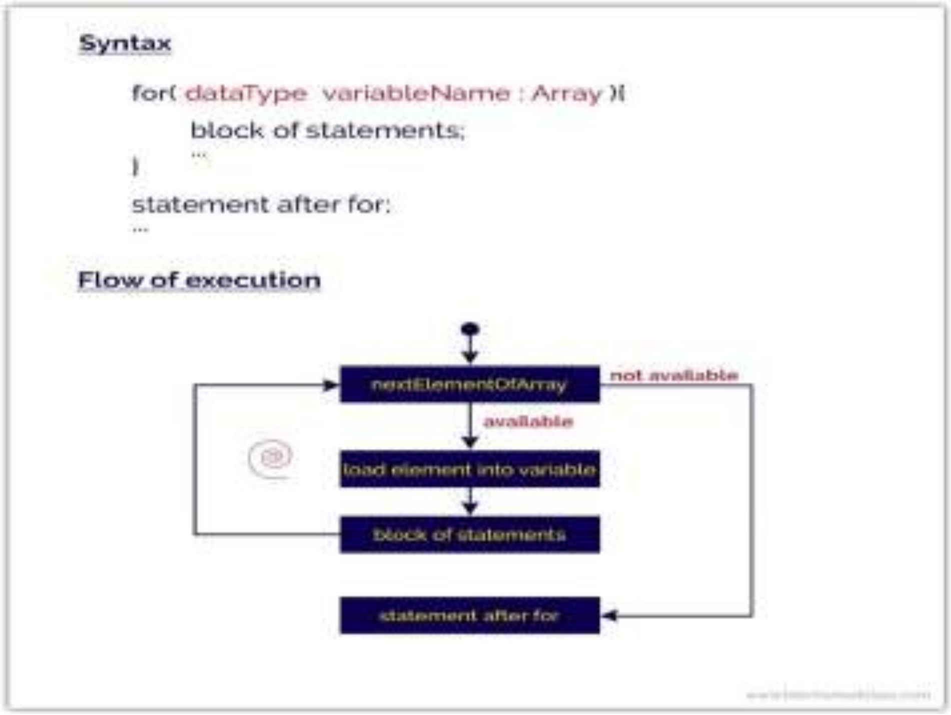
Let's look at the following example in java code.

### 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 cannot 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**

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!");
        }
}
```

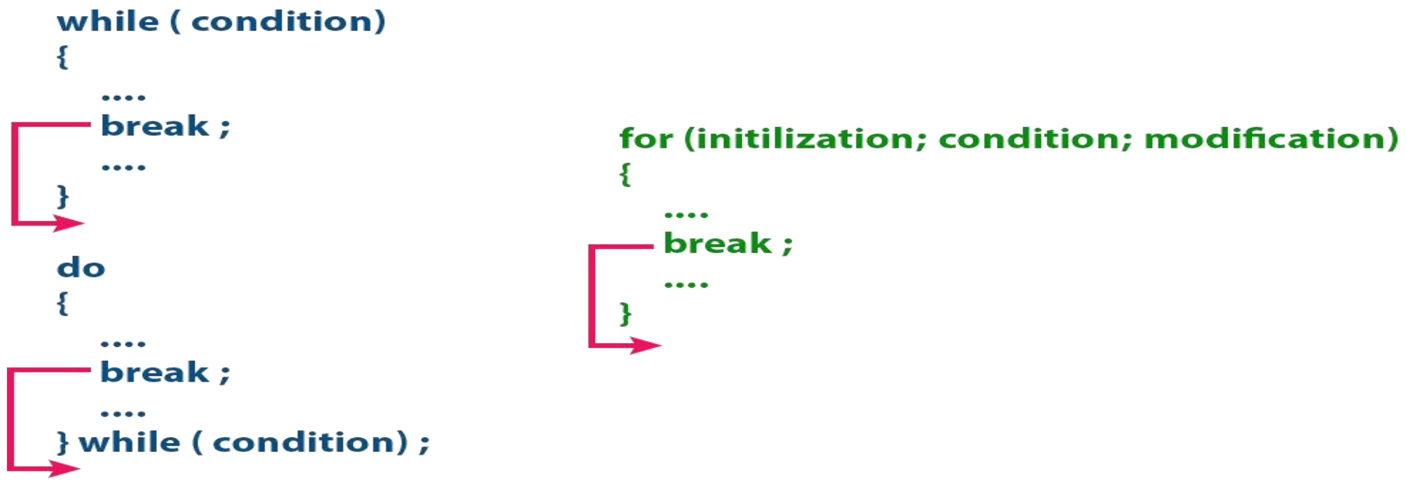
## Java Jump Statements

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
- Labeled 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.



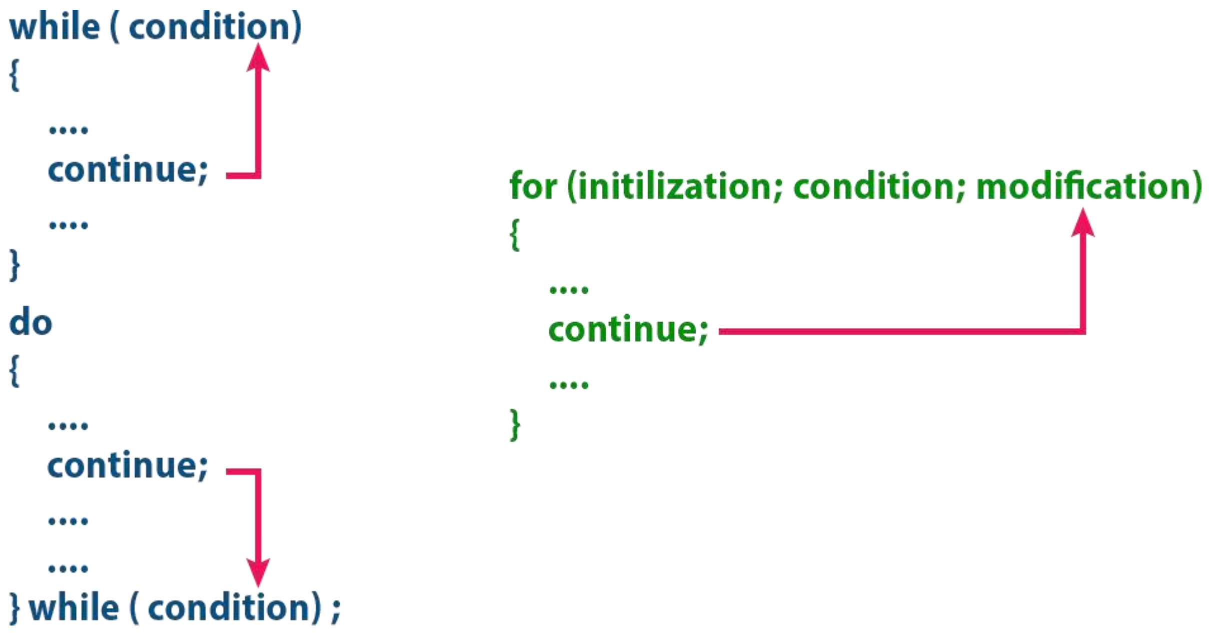
Let's look at the following example in java code.

### Java Program

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

## Labeled break and continue statement in java

- The java programming language doesnot 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 labbeled contonue 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(Stringargs[])
        {
                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 scoreis"+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=newScanner(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());
          }
}|
```

## 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

Object
Data Abstraction
Class
OOps concept in Java
Encapsulation
Polymorphism

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.

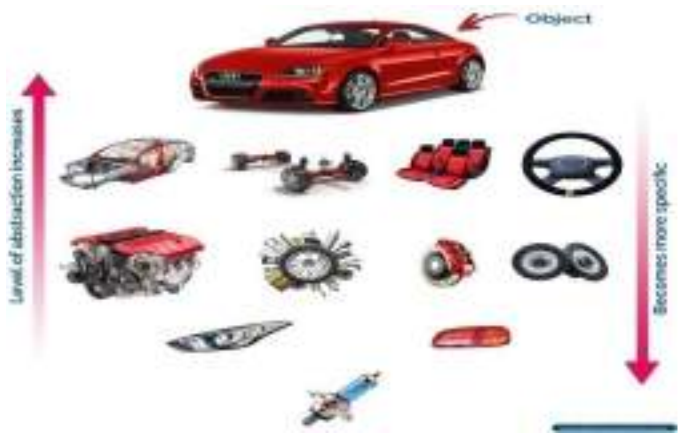
### Polymorphism

- 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 over loading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Person
Same method with different implementations
Programmer
with coding languages
Dancer
Singer
with tunes, music, songs

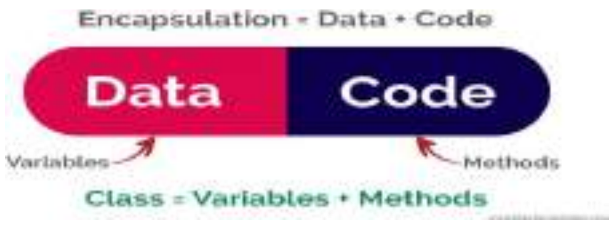
### Abstraction

- *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 unidirectional 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**

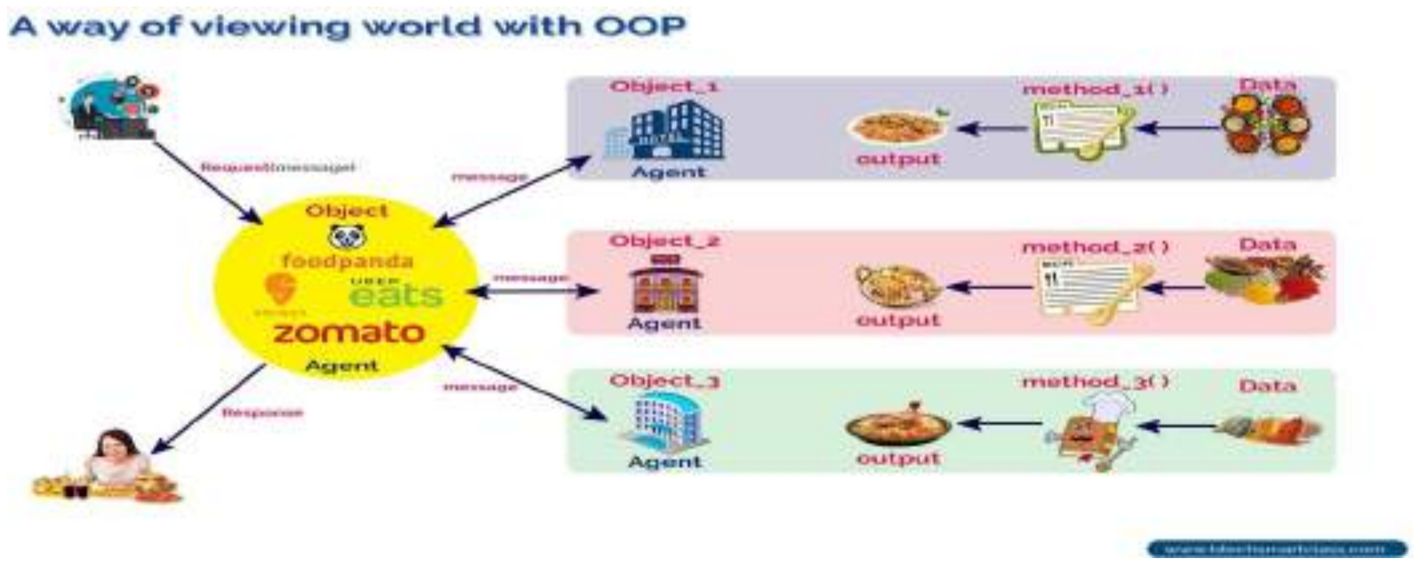
The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have 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 delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my family members. Look at the following image.



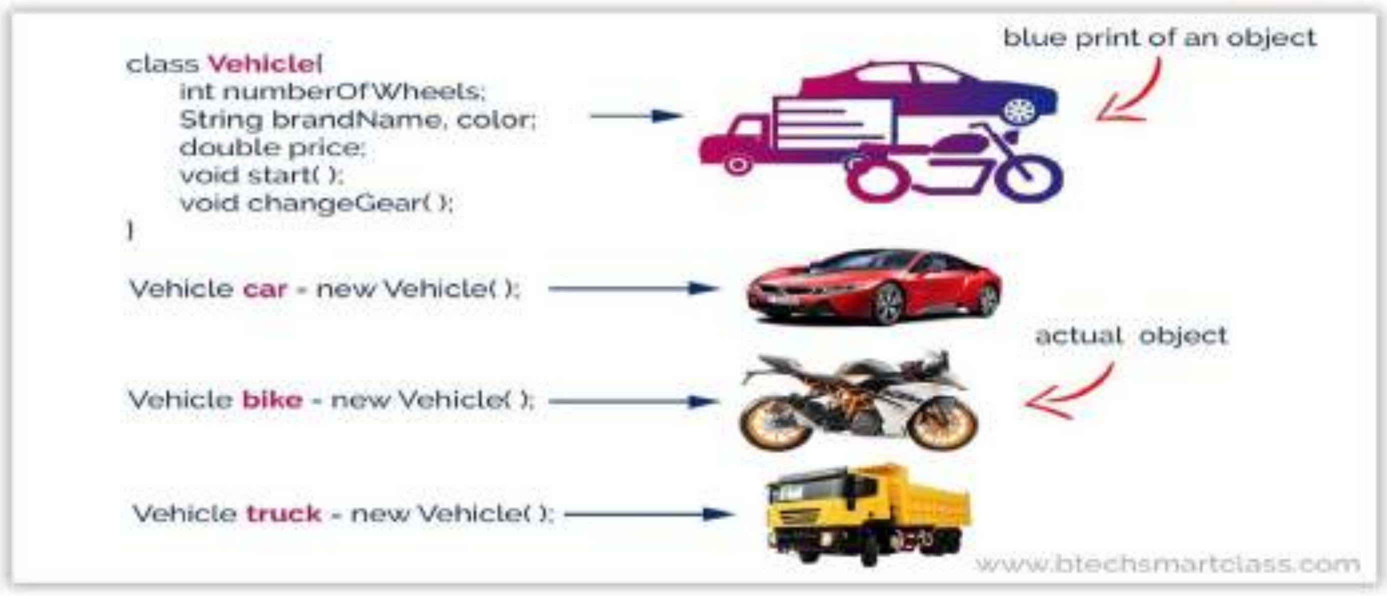
## Elements of Java

### 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.



-  The Class Name must begin with an alphabet, and the Upper-case letter is preferred.
-  The Class Name 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 object Name must begin with an alphabet, and a Lower-case letter is preferred.
-  The object Name 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 method Name must begin with an alphabet, and the Lower-case letter is preferred.
-  The method Name 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();//methodcallusingobject
                   showData(obj.sNo,obj.name);//methodcallwithoutusingobject
                }
    }
```

- The object Name must begin with an alphabet, and a Lower-case letter is preferred.
- The object Name 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(inti :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 executes 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.
- A constructor cannot have return value
- Let's look at the following example java code.

## Example

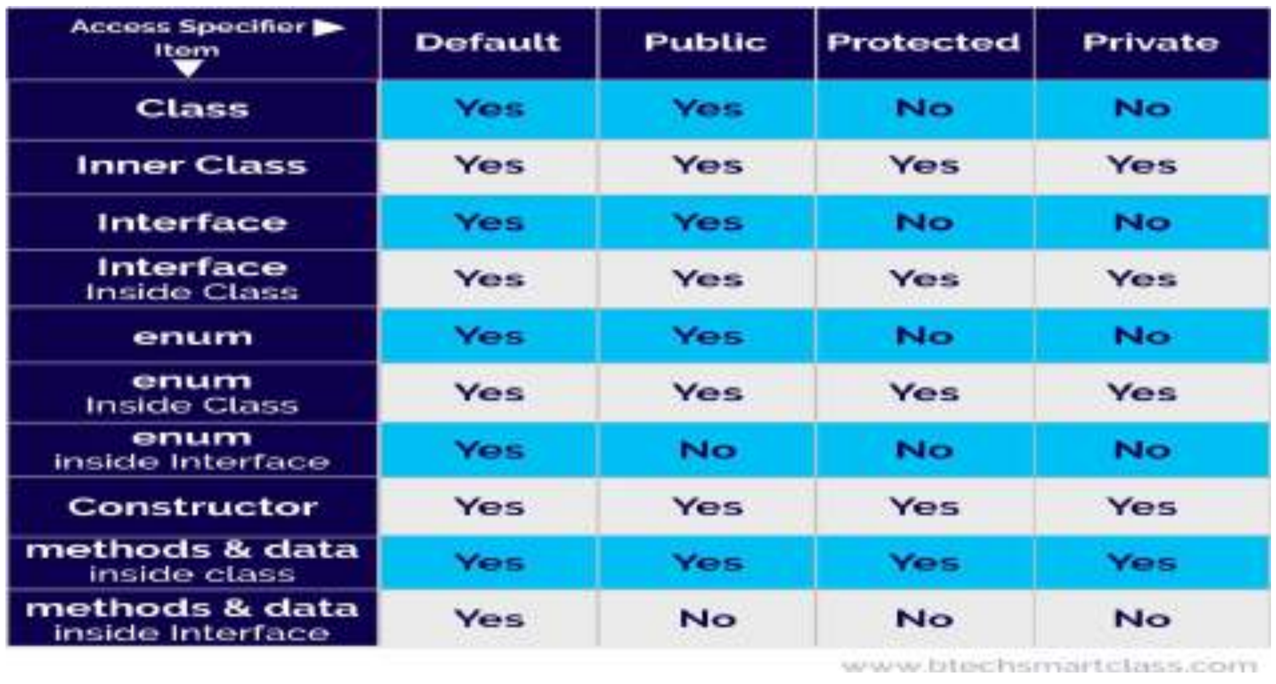
```
Public class ConstructorExample{


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

## Java Access Modifiers / MemberAccess

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 |

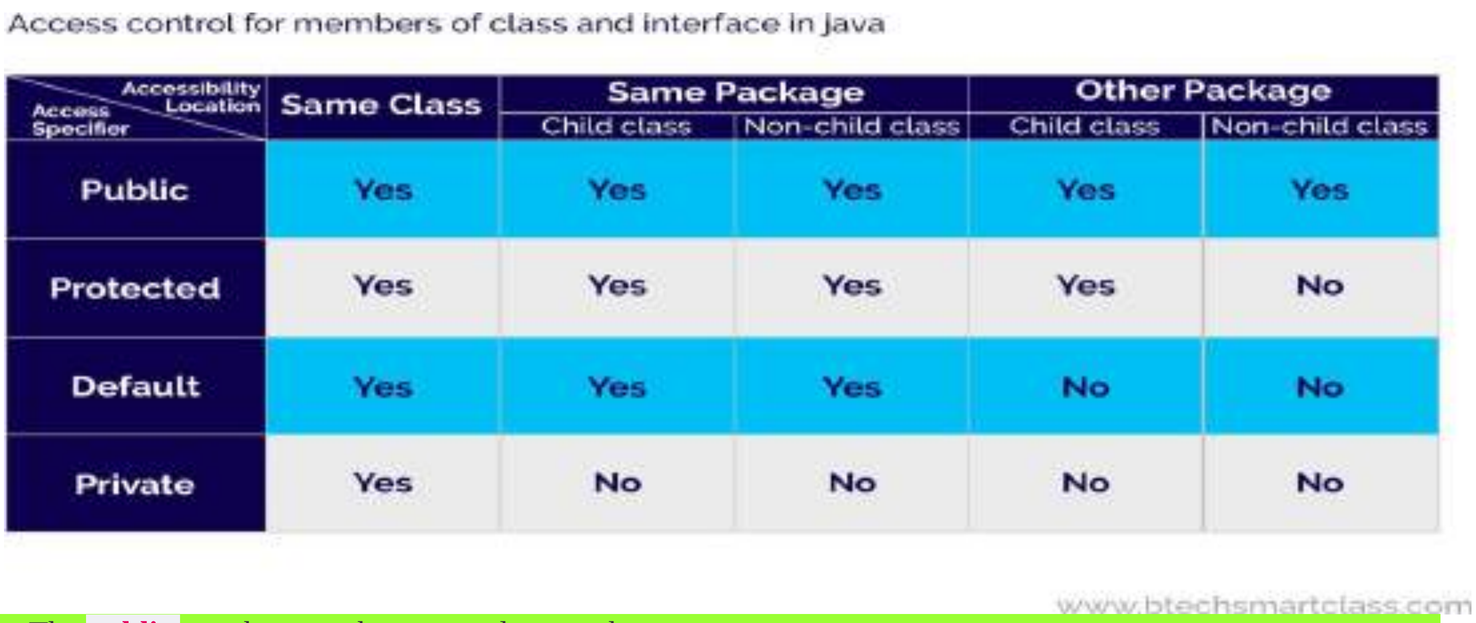
www.btechsmartclass.com

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 \ 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.

**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 Parent Class");
                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 Child Class");
                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 = newChildClass();
                obj.showData();   obj.accessData();

        }
}
```

## Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before introducing generics, we used to store any type of objects in the collection, i.e., non-generic. Now generics force the Java programmer to store a specific type of objects.

### Advantage of Java Generics

There are mainly three advantages of generics. They are as follows:

**1) Type-safety:**

We can hold only a single type of objects in generics. It does not allow to store other objects. Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10"); // compile-time error
```

**2) Type casting is not required:**

There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0); //typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

**3) Compile -Time Checking:**

It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32); //Compile Time Error
```

**Syntax** to use generic collection

```
Class Or Interface<Type>
```

**Example** to use Generics in java

```
ArrayList<String>
```

**Example of Generics in Java**

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();
        list.add("rahul");
        list.add("jai");
        //list.add(32);//compile time error
```

```
                String s=list.get(1);//type casting is not required
                System.out.println("element is: "+s);
                Iterator<String> itr=list.iterator();
                while(itr.hasNext()){
                System.out.println(itr.next());
               }
           }
      }
```

## Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type. Let's see a simple example to create and use the generic class.

**Creating a generic class:**

```
class MyGen<T>
{
    T obj;
    void add(T obj)
    {     this.obj=obj;   }

   T get()
    {   return obj;       }
 }
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

**Using generic class**

Let's see the code to use the generic class.

```
class TestGenerics3
{
        public static void main(String args[])
        {
                MyGen<Integer> m=new MyGen<Integer>();
                m.add(2);
                //m.add("vivek");  //Compile time error
                System.out.println(m.get());
        }
}
```

**Type Parameters**

The type parameters naming conventions are important to learn generics thoroughly.

The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

**Generic Method**

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4
{
    public static < E > void printArray(E[] elements)
    {
        for ( E element : elements)
         {
              System.out.println(element );
        }
        System.out.println();
     }
   public static void main( String args[] )
   {
         Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[]  charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

        System.out.println( "Printing Character Array" );
        printArray( charArray );
   }
 }
```

## Inner Class in Java

In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced, to sum up, same logically relatable classes as Java is object-oriented so bringing it closer to the real world.

**There are certain advantages associated with inner classes are as follows:**

- Making code clean and readable.
- Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to the real world.
- Optimizing the code module.

*We do use them often as we go advance in java object-oriented programming where we want certain operations to be performed, granting access to limited classes and many more which will be clear as we do discuss and implement all types of inner classes in Java.*

**Types of Inner Classes**

There are basically four types of inner classes in java.

1. Nested Inner Class
2. Method Local Inner Classes
3. Static Nested Classes
4. Anonymous Inner Classes

**Syntax**

```
// Java Program to Demonstrate Nested class
// Class 1
// Helper classes
class Outer {
    // Class 2
    // Simple nested inner class
    class Inner {

        // show() method of inner class
        public void show()
        {
            // Print statement
            System.out.println("In a nested class method");
        }
    }
}

// Class 2
// Main class
class Main {
    // Main driver method
    public static void main(String[] args)
    {
        // Note how inner class object is created inside
        // main()
        Outer.Inner in = new Outer().new Inner();
        // Calling show() method over above object created
        in.show();
    }
}
```

**Example :**

```
        // Java Program to Demonstrate Nested class
        // Where Error is thrown

        // Class 1
        // Outer class
        class Outer {

         // Method defined inside outer class
          void outerMethod()
           {
             // Print statement
            System.out.println("inside outerMethod");
           }
         // Class 2
         // Inner class
        class Inner {
        // Main driver method
        public static void main(String[] args)
        {
            // Display message for better readability
            System.out.println("inside inner class Method");
        }
       }
      }
```

## Java String Class

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 cannot 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 = "btech smart class.com";

siteName = "www.btech smart class.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**

```
Stringtitle="JavaTutorials";              // Usingliterals


StringsiteName =newString("www.btechsmartclass.com");        // Usingconstructor
```

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 containing 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 | ReturnValue |
|---|---|---|
| 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 |

| | | |
|---|---|---|
| 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 white space 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.

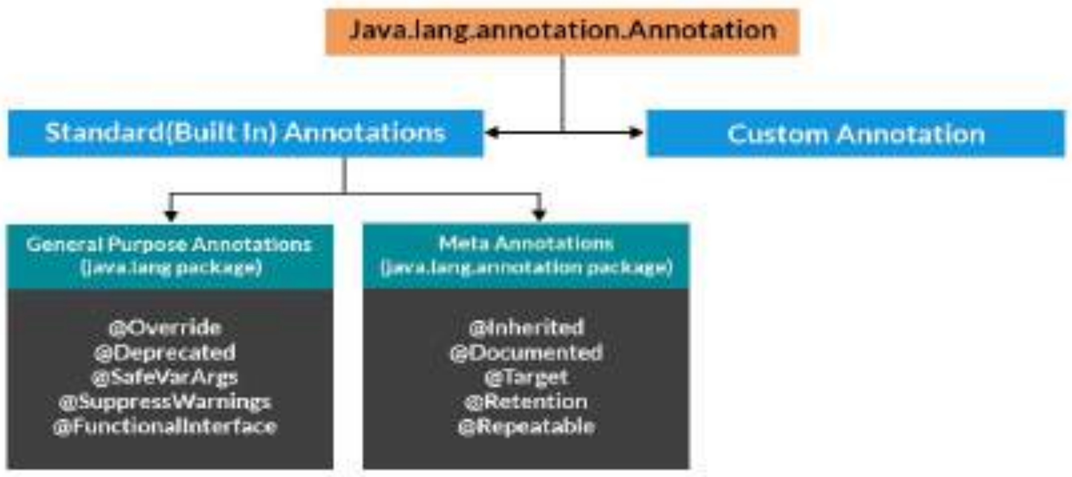
## Java Program

```
Public class JavaStringExample {

        public static void main(String[] args)
        {           String title="Java  Programming";
                    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("Endswith'.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());

        }
    }
```

## Annotations in Java

- Annotations are used to provide supplemental information about a program.
- Annotations start with '**@**'.
- Annotations do not change the action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
- Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.

### Hierarchy of Annotations in Java


Java.lang.annotation.Annotation
Standard(Built In) Annotations
Custom Annotation
General Purpose Annotations (java.lang package)
@Override
@Deprecated
@SafeVarArgs
@SuppressWarnings
@FunctionalInterface
Meta Annotations (java.lang.annotation package)
@Inherited
@Documented
@Target
@Retention
@Repeatable

**Implementation:**

***Note:*** *This program throws compiler error because we have mentioned override, but not overridden, we have overloaded display.*

**Example:**

```
// Java Program to Demonstrate that Annotations
// are Not Barely Comments

// Class 1
class Base {

    // Method
    public void display()
    {
        System.out.println("Base display()");
    }
}
```

```
// Class 2
// Main class
class Derived extends Base {

    // Overriding method as already up in above class

    @Override public void display(int x)
    {
        // Print statement when this method is called

       System.out.println("Derived display(int )");
    }

  // Method 2
    // Main driver method

 public static void main(String args[])
    {
        // Creating object of this class inside main()
        Derived obj = new Derived();

        // Calling display() method inside main()
        obj.display();
    }
}
```

**Output:**

```
10: error: method does not override or implement
a method from a super type
```

If we remove parameter (int x) or we remove @override, the program compiles fine.

## Categories of Annotations

There are broadly 5 categories of annotations as listed:

1. Marker Annotations
2. Single value Annotations
3. Full Annotations
4. Type Annotations
5. Repeating Annotations

Let us discuss and we will be appending code wherever required if so.

### Category 1: Marker Annotations

- The only purpose is to mark a declaration. These annotations contain no members and do not consist of any data. Thus, its presence as an annotation is sufficient.

- Since the marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

**Example**

```
@TestAnnotation()
```

## Category 2: Single value Annotations

- These annotations contain only one member and allow a shorthand form of specifying the value of the member.
- We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be a value.

**Example**

```
@TestAnnotation("testing");
```

## Category 3: Full Annotations

These annotations consist of multiple data members, names, values, pairs.

**Example**

```
@TestAnnotation(owner="Rahul", value="Class Geeks")
```

## Category 4: Type Annotations

These annotations can be applied to any place where a type is being used. For example, we can annotate the return type of a method. These are declared annotated with ***@Target** annotation*.

**Example**

```
// Java Program to Demonstrate Type Annotation

// Importing required classes
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

// Using target annotation to annotate a type
@Target(ElementType.TYPE_USE)

// Declaring a simple type annotation
@interface TypeAnnoDemo{}
// Main class
public class GFG {

// Main driver method
public static void main(String[] args) {
// Annotating the type of a string
@TypeAnnoDemo String string = "I am annotated with a type annotation";
System.out.println(string);
abc();
}

// Annotating return type of a function
static @TypeAnnoDemo int abc() {
System.out.println("This function's  return type is annotated");

return 0;
}
}
```

**Output:**

```
I am annotated with a type annotation

This function's  return type is annotated
```

## Category 5: Repeating Annotations

- These are the annotations that can be applied to a single item more than once.
- For an annotation to be repeatable it must be annotated with the **@Repeatable** annotation, which is defined in the **java.lang.annotation** package.
- Its value field specifies the **container type** for the repeatable annotation. **The container is specified as an annotation whose value field is an array of the repeatable annotation type.**
- Hence, to create a repeatable annotation, firstly the container annotation is created, and then the annotation type is specified as an argument to the @Repeatable annotation.

**Example:**

**// Java Program to Demonstrate a Repeatable Annotation**

```
// Importing required classes
import java.lang.annotation.Annotation;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;

// Make Words annotation repeatable
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface Words
{
        String word() default "Hello";
        int value() default 0;
}

// Create container annotation
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos
{
            Words[] value();
}

public class Main {

   // Repeat Words on newMethod
   @Words(word = "First", value = 1)
   @Words(word = "Second", value = 2)

  public static void newMethod()
   {
               Main obj = new Main();

        try {
          Class<?> c = obj.getClass();

          // Obtain the annotation for newMethod
          Method m = c.getMethod("newMethod");

          // Display the repeated annotation
          Annotation anno
             = m.getAnnotation(MyRepeatedAnnos.class);
          System.out.println(anno);
       }
       catch (NoSuchMethodException e) {
          System.out.println(e);
       }
    }
    public static void main(String[] args) { newMethod(); }
 }
```

**Output:**

```
@MyRepeatedAnnos(value={@Words(value=1, word="First"), @Words(value=2, word="Second")})
```

## Predefined/ Standard Annotations

- Java popularly defines seven built-in annotations as we have seen up in the hierarchy diagram.
- Four are imported from java.lang.annotation: **@Retention**, **@Documented**, **@Target**, and **@Inherited**.
- Three are included in java.lang: **@Deprecated, @Override** and **@SuppressWarnings**

### Annotation 1: @Deprecated

- It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
- The Javadoc @deprecated tag should be used when an element has been deprecated.
- @deprecated tag is for documentation and @Deprecated annotation is for runtime reflection.

- @deprecated tag has higher priority than @Deprecated annotation when both are together used.

**Example:**

```
public class DeprecatedTest
{
   @Deprecated
   public void Display()
   {
      System.out.println("Deprecated test display()");
   }


   public static void main(String args[])
   {
      DeprecatedTest d1 = new DeprecatedTest();
      d1.Display();
   }
}
```

**Output**

```
Deprecatedtest display()
```

## Annotation 2: @Override

- It is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a super class.
- If it doesn't, a compile-time error will result.
- It is used to ensure that a super class method is actually overridden, and not simply overloaded.

**Example**

```
// Java Program to Illustrate Override Annotation
// Class 1
class Base
{
    public void Display()
    {
       System.out.println("Base display()");
    }

    public static void main(String args[])
    {
       Base t1 = new Derived();
       t1.Display();
    }
}

// Class 2
// Extending above class
```

```
class Derived extends Base
{
    @Override
    public void Display()
    {
        System.out.println("Derived display()");
    }
}
```

**Output**

```
Derived display()
```

## Annotation 3: @SuppressWarnings

- It is used to inform the compiler to suppress specified compiler warnings.
- The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.
- Java groups warnings under two categories.
- They are **deprecated** and **unchecked**. Any unchecked warning is generated when a legacy code interfaces with a code that uses generics.

**Example:**

```
// Java Program to illustrate SuppressWarnings Annotation

// Class 1
class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }
}

// Class 2
public class SuppressWarningTest
{
    // If we comment below annotation, program generates
    // warning
    @SuppressWarnings({"checked", "deprecation"})
    public static void main(String args[])
    {
        DeprecatedTest d1 = new DeprecatedTest();
        d1.Display();
    }
}
```

**Output**

```
Deprecatedtest display()
```

## Annotation 4: @Documented

- It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included in 'Javadoc' comments.
- The use of @Documented annotation in the code enables tools like Java doc to process it and include the annotation type information in the generated document.

## Annotation 5: @Target

- It is designed to be used only as an annotation to another annotation.
- **@Target** takes one argument, which must be constant from the **ElementType** enumeration.
- This argument specifies the type of declarations to which the annotation can be applied.
- The constants are shown below along with the type of the declaration to which they correspond.

| **Target Constant** | **Annotations Can Be Applied To** |
|---|---|
| ANNOTATION_TYPE | Another annotation |
| CONSTRUCTOR | Constructor |
| FIELD | Field |
| LOCAL_VARIABLE | Local variable |
| METHOD | Method |
| PACKAGE | Package |
| PARAMETER | Parameter |
| TYPE | Class, Interface, or enumeration |

- We can specify one or more of these values in a **@Target**annotation.

- To specify multiple values, we must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this @Target annotation:

@Target({ElementType.FIELD, ElementType.LOCAL_VARIABLE}) **@Retention Annotation** It determines where and how long the annotation is retent. The 3 values that the @Retention annotation can have:

- **SOURCE:** Annotations will be retained at the source level and ignored by the compiler.
- **CLASS:** Annotations will be retained at compile-time and ignored by the JVM.
- **RUNTIME:** These will be retained at runtime.

## Annotation 6: @Inherited

- @Inherited is a marker annotation that can be used only on annotation declaration.
- It affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a super class to be inherited by a subclass.
- Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its super class is checked.
- If that annotation is present in the super class, and if it is annotated with **@Inherited,** then that annotation will be returned.

## Annotation 7: User-defined (Custom)

- User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc.
- These annotations can be applied just before the declaration of an element (constructor, method, classes, etc).

**Syntax:** Declaration

```
[Access Specifier] @interface<AnnotationName>
{
  DataType <Method Name>() [default value];
}
```

Do keep these certain points as rules for custom annotations before implementing user-defined annotations.

1. **Annotation Name** is an interface.
2. The parameter should not be associated with method declarations and **throws** clause should not be used with method declaration.
3. Parameters will not have a null value but can have a default value.
4. ***default value*** is optional.
5. The return type of method should be either primitive, enum, string, class name, or array of primitive, enum, string, or class name type.

**Example:**

```
// Java Program to Demonstrate User-defined Annotations

package source;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

// User-defined annotation
@Documented
```

```
@Retention(RetentionPolicy.RUNTIME)
@ interface TestAnnotation
{
   String Developer() default "Rahul";
   String Expirydate();
} // will be retained at runtime


// Driver class that uses @TestAnnotation
public class Test
{
   @TestAnnotation(Developer="Rahul", Expirydate="01-10-2020")

   void fun1()
   {
      System.out.println("Test method 1");
   }


   @TestAnnotation(Developer="Anil", Expirydate="01-10-2021")
   void fun2()
   {
      System.out.println("Test method 2");
   }


   public static void main(String args[])
   {
      System.out.println("Hello");
   }
}
```

**Output:**

```
Hello
```

## OOP Principles

## Encapsulation Concept

- Encapsulation is one of the fundamental concept in object-oriented programming (OOP).
- Encapsulation describes bundling data and methods that work on that data within one unit, like a class.
- We often often use this concept to hide an object’s internal representation or state from the outside. This is called information hiding.
- The general idea of this mechanism is simple.
- For example, you have an attribute that is not visible from the outside of an object.
- You bundle it with methods that provide read or write access.
- Encapsulation allows you to hide specific information and control access to the object’s internal state.
- If you’re familiar with any object-oriented programming language, you probably know these methods as getter and setter methods.
- As the names indicate, a getter method retrieves an attribute and a setter method changes it.

- Depending on the methods that you implement, you can decide if an attribute can be read and changed.
- You may also control if the attribute is read-only or not visible at all. Later, we'll show you how you can also use the setter method to implement additional validation rules to ensure that your object always has a valid state.

***Data Encapsulation** : can be defined as wrapping the code or methods(properties) and the related fields or variables together as a single unit.* In object-oriented programming, we call this single unit – a class, interface, etc. We can visualize it like a medical capsule (as the name suggests, too), wherein the enclosed medicine can be compared to fields and methods of a class.

## Setter and Getter Method Usage

- Getter and setter methods are used to access and modify the private variables (fields) of a class, respectively.
- They help achieve encapsulation by providing controlled access to the class's attributes.
- In Java, Getter and Setter are methods used to protect your data and make your code more secure. Getter and Setter make the programmer convenient in setting and getting the value for a particular data type.

- **Getter in Java:** Getter returns the value (accessors), it returns the value of data type int, String, double, float, etc. For the program's convenience, the getter starts with the word "get" followed by the variable name.
- **Setter in Java:** While Setter sets or updates the value (mutators). It sets the value for any variable used in a class's programs. and starts with the word "set" followed by the variable name.

**Syntax**

```
class ABC{
   private variable;

   public void setVariable(int x){
      this.variable=x;
   }

   public int getVariable{
      return variable;
   }
}
```

- ***Note:** In both getter and setter, the first letter of the variable should be capital.*

Example

```
public class Person {
    private String name;
    private int age;

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
public class Main {
    public static void main(String[] args) {
        Person person = new Person();

        // Set values using setter methods
        person.setName("John");
        person.setAge(30);

        // Retrieve values using getter methods
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

**Output**

```
Name: John
Age: 30
```

- The **main()** method creates an instance of the **Person** class named **person**.
- The setter methods **setName("John")** and **setAge(30)** are used to set the values of the **name** and **age** attributes of the **person** object.
- The getter methods **getName()** and **getAge()** are used to retrieve the values of the **name** and **age** attributes, respectively.
- The retrieved values are then printed to the console.

## Implementation of Encapsulation

Example

```
public class Person {
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        } else {
            System.out.println("Age cannot be negative.");
        }
    }

    // Method to display person details
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
public class Main {
    public static void main(String[] args) {
        Person person = new Person("John", 30);

        // Display initial details
        person.display();

        // Update name
        person.setName("Alice");
        // Update age
        person.setAge(25);

        // Display updated details
        person.display();
    }
}
```

**Output**

```
Name: John
Age: 30
Name: Alice
Age: 25
```

- **name** and **age** are private instance variables, accessible only within the **Person** class.
- **getName()** and **setName()** methods provide access to the **name** variable, ensuring encapsulation by controlling how the **name** can be accessed and modified.
- **getAge()** and **setAge()** methods provide access to the **age** variable, with additional validation to ensure the age is non-negative.
- **display()** method allows displaying the person's details without exposing the internal implementation.

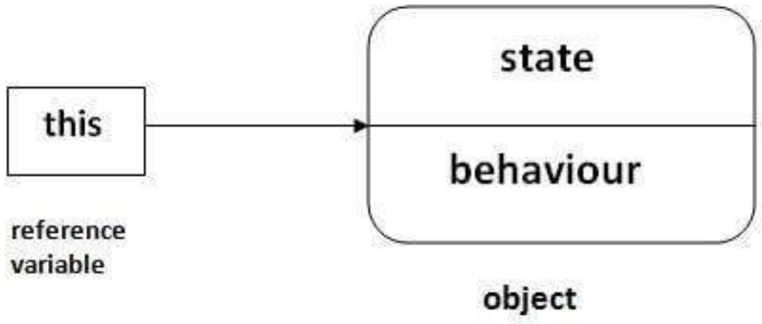
## Advantages of Encapsulation

Encapsulation offers several advantages, which can be summarized in bullet points:

- **Data Hiding**: Encapsulation hides the internal state of objects by making fields private, preventing direct access from outside the class.
- **Enhanced Security**: With encapsulation, sensitive data can be protected from unauthorized access and modification, enhancing security.
- **Modularity and Maintainability**: Encapsulation promotes modularity by isolating the implementation details within a class, making it easier to understand, maintain, and modify the codebase.
- **Code Reusability**: Encapsulation encourages code reuse by allowing classes to be used as components in larger systems without exposing their internal workings.
- **Easier Testing**: Encapsulation facilitates unit testing by providing clear boundaries between different components of the system, allowing for easier isolation and testing of individual units.
- **Flexibility and Evolution**: Encapsulation provides flexibility in changing the internal implementation of a class without affecting its external interface, enabling the evolution of the codebase over time.
- **Reduced Coupling**: Encapsulation reduces coupling between different parts of the system by limiting dependencies on the internal implementation of classes, promoting a more loosely coupled architecture.
- **Encapsulation of Invariants**: Encapsulation allows the encapsulation of invariants and business rules within a class, ensuring that data remains consistent and valid throughout its lifecycle.
- **Facilitates Abstraction**: Encapsulation facilitates abstraction by exposing only essential information through well-defined interfaces, hiding unnecessary details and complexity from the users of the class.
- **Supports Information Hiding**: Encapsulation supports the principle of information hiding, enabling classes to hide their internal implementation details while providing a clear and consistent interface to interact with.

## this keyword in Java

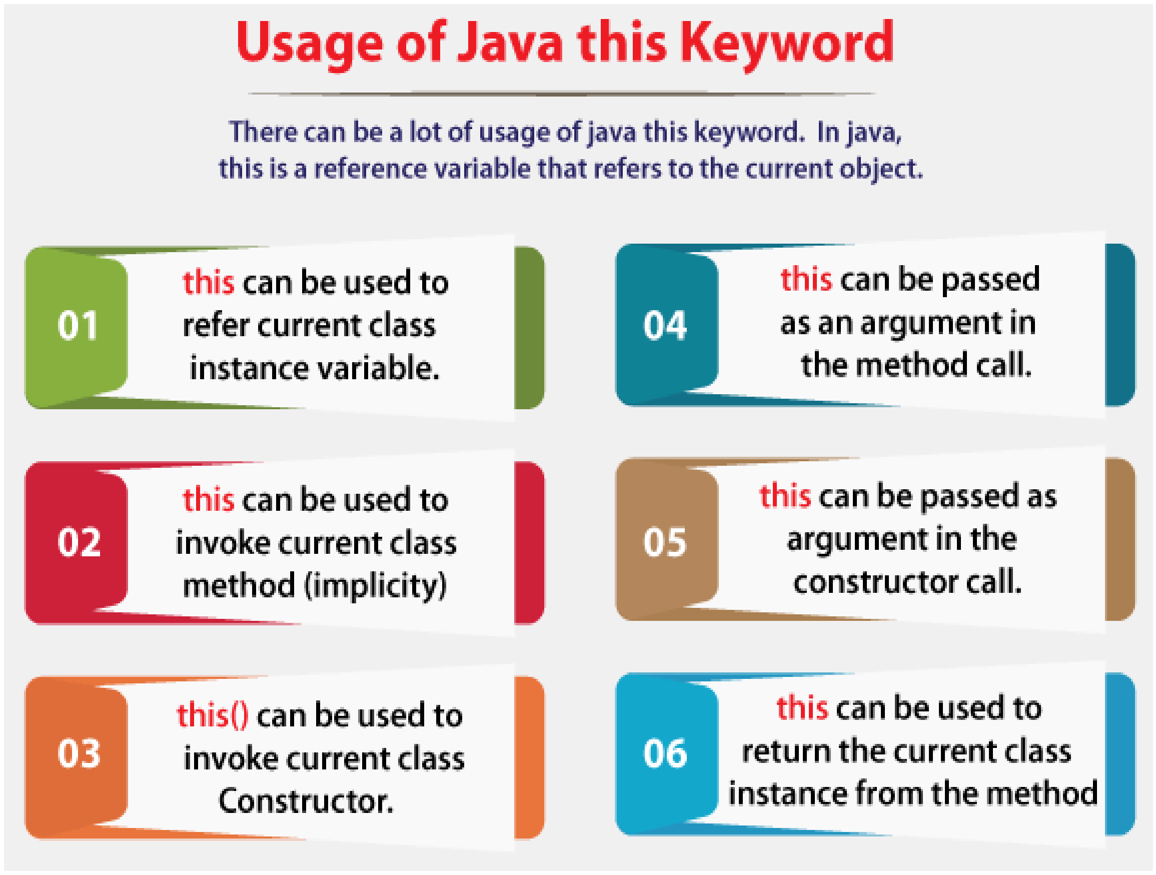
There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



### Usage of Java this keyword

Here is given the 6 usage of java this keyword.

**Suggestion:** If you are beginner to java, lookup only three usages of this keyword.


Usage of Java this Keyword
There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.
01 this can be used to refer current class instance variable.
02 this can be used to invoke current class method (implicity)
03 this() can be used to invoke current class Constructor.
04 this can be passed as an argument in the method call.
05 this can be passed as argument in the constructor call.
06 this can be used to return the current class instance from the method

**1) this: to refer current class instance variable**

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

**Understanding the problem without this keyword**

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student
{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
            rollno=rollno;
            name=name;
            fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1
{
    public static void main(String args[]){
    Student s1=new Student(111,"ankit",5000f);
    Student s2=new Student(112,"sumit",6000f);
    s1.display();
    s2.display();
    }
}
```

**Output:**

```
      0 null 0.0
      0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

**Solution of the above problem by this keyword**

```
class Student
{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee)
    {
            this.rollno=rollno;
            this.name=name;
            this.fee=fee;
    }
    void display()
    {
            System.out.println(rollno+" "+name+" "+fee);
    }
}
```

```
class TestThis2
{
    public static void main(String args[])
    {
            Student s1=new Student(111,"ankit",5000f);
            Student s2=new Student(112,"sumit",6000f);
            s1.display();
            s2.display();
    }
}
```

**Output:**

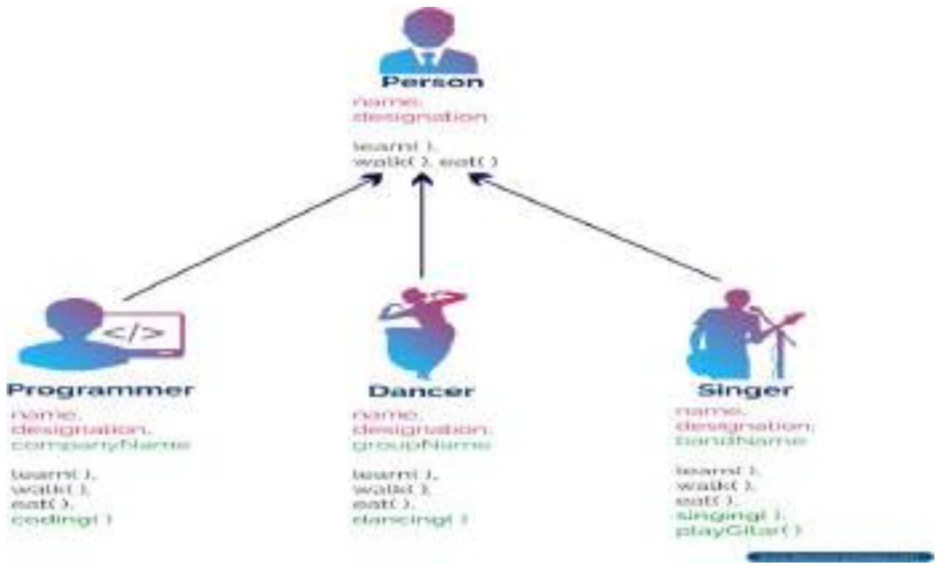
```
    111 ankit 5000.0
    112 sumit 6000.0
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

## Inheritance Concept

### Inheritance

- 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

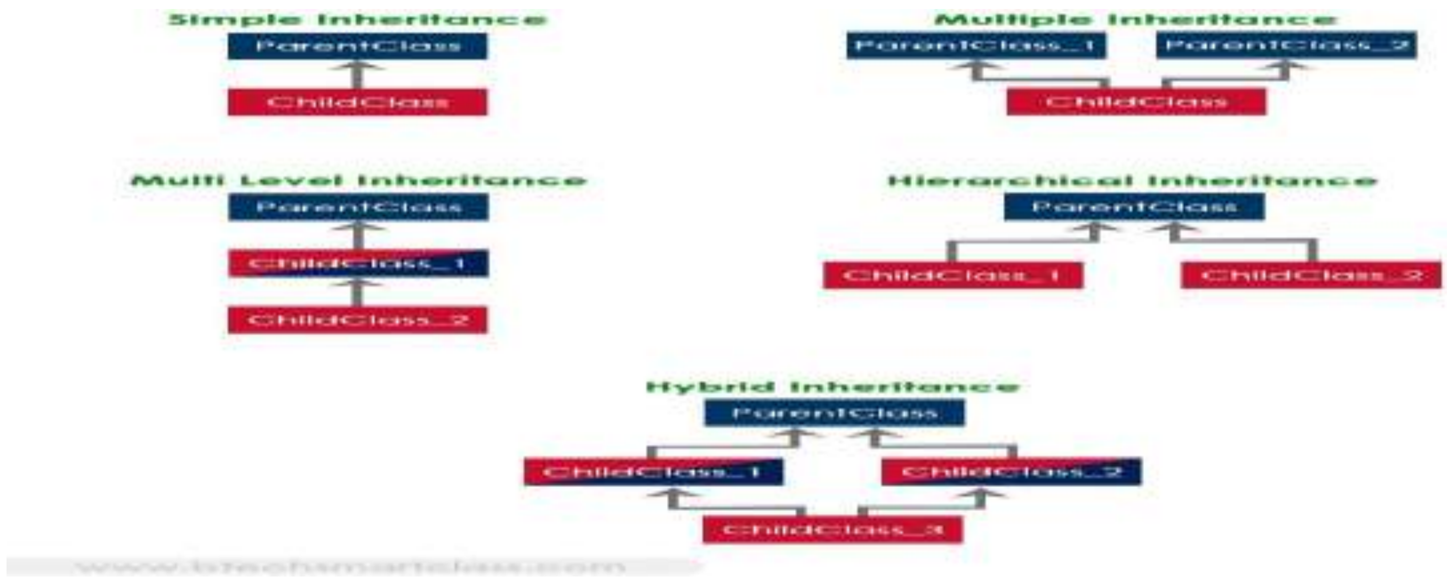
- In inheritance, we use the terms like parent class, child class, base class, derived class, super class, and subclass.
- The **Parent class** is the class which provides features to another class.
- The parent class is also known as **Base class** or **Super class**.
- The **Child class** is the class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

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.



## 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 classextends only oneclass. Extending multiple classes isnot 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{
            inta;
            voidsetData(inta){
                        this.a=a;
            }
}
Class ChildClass extends ParentClass{
             void showData() {
             System.out.println("Valueof ais"+a);
            }
}
Public class SingleInheritance{
            public static void main(String[] args) {
                        ChildClassobj=newChildClass();
                        obj.setData(100);
                        obj.showData();
            }
}
```

## Multi-level Inheritance in java

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

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(inta)  {  this.a=a;

            }

}

Class ChildClass extends ParentClass

   { void showData() {

                        System.out.println("Valueof ais"+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(inta){
                this.a=a;
        }
}
class ChildClass extends ParentClass{ void showData() {
                System.out.println("Inside Child Class!");
                System.out.println("Value of a is "+ a);
        }
}
class ChildClassToo extends ParentClass{ void display() {
                System.out.println("Inside Child Class Too!");
```

```
                        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.,

### 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

## 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) {
                    ChildClassobj=newChildClass();
          }
}
```

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 key word 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 key word 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("ParentClassnum="+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("ChildClassnum="+obj.num);
                //System.out.println("ParentClassnum="+super.num);//supercan'tbeused 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();        //supercan'tbeusedhere
        }
}
```

**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** key word 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(intvalue){
                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 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 methodoverriding.

### 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);          //Callingwith Stringarray
        }
}
```

## 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) {
                ParentClassobj = new ParentClass();
                obj.showData();
                obj=newChildClass();
                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=newChildClass();
                obj.showData();
        }    }
```

## 10 Rules for method overriding

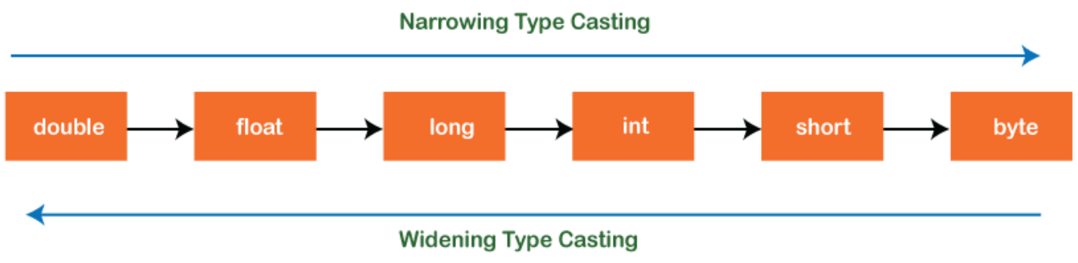
While overriding a method, we must follow the below list of rules.

- 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.

## Type Casting in Java

- **Type casting in Java** is a fundamental concept that allows developers to convert data from one data type to another.
- It is essential for handling data in various situations, especially when dealing with different types of variables, expressions, and methods.
- In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically.
- The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss type casting and its types with proper examples.



Type Casting in Java

### Type Casting

Convert a value from one data type to another data type is known as **type casting**.

### Types & Rules of Typecasting

#### Widening Conversion (Implicit)

- No explicit notation is required.
- Conversion from a smaller data type to a larger data type is allowed.
- No risk of data loss.

#### Narrowing Conversion (Explicit)

- Requires explicit notation using parentheses and casting.
- Conversion from a larger data type to a smaller data type is allowed.
- Risk of data loss due to truncation.

#### WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
```

```
public static void main(String[] args)
{
        int x = 7;
        //automatically converts the integer type into long type
        long y = x;
        //automatically converts the long type into float type
        float z = y;
        System.out.println("Before conversion, int value "+x);
        System.out.println("After conversion, long value "+y);
        System.out.println("After conversion, float value "+z);
}
}
```

**Output**

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

**NarrowingTypeCastingExample.java**

```
public class NarrowingTypeCastingExample
{
        public static void main(String args[])
        {
                double d = 166.66;
                //converting double data type into long data type
                long l = (long)d;
                //converting long data type into int data type
                int i = (int)l;
                System.out.println("Before conversion: "+d);
                //fractional part lost
                System.out.println("After conversion into long type: "+l);
                //fractional part lost
                System.out.println("After conversion into int type: "+i);
        }
}
```

**Output**

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

## 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();//BaseclassreferencetoChildclassinstance
                        obj.printArea();
                        obj = newTriangle();
                        obj.printArea();
                        obj = new Cricle();
                        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 cannot be overridden.
- An abstract class may contain static methods, but the abstract method cannot 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 cannot be instantiated but can be referenced.

## Interface in Java

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables.
- It cannot have a method body.
- Java Interface also represents the IS-A relationship.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have default and static methods in an interface.
- Since Java 9, we can have private methods in an interface.

## Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

It is used to achieve abstraction.
1
2
By interface, we can support the functionality of multiple inheritance.
It can be used to achieve loose coupling.
3

## How to declare an interface?

- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.

**Syntax:**

```
interface <interface_name>
{

        // declare constant fields
        // declare methods that abstract
        // by default.
}
```

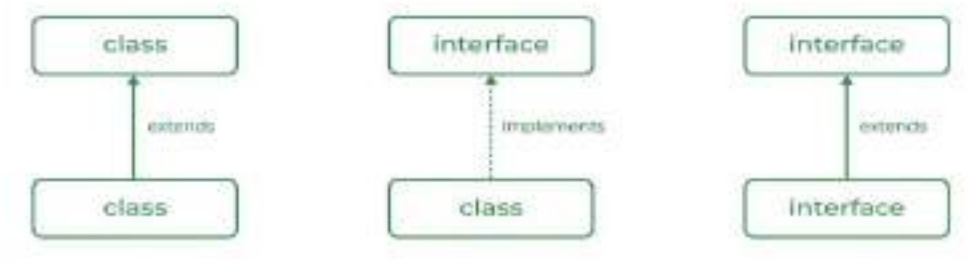
**Example:**

```
interface Animal
{
       void eat();
       void sleep();
}
```

In this example, the Animal interface declares two method signatures: eat() and sleep(). Any class implementing the Animal interface must provide concrete implementations for these methods.

**Relationship Between Class and Interface**

A class can extend another class, and similarly, an interface can extend another interface.
However, only a class can implement an interface, and the reverse (an interface implementing a class) is not allowed.



**Difference Between Class and Interface**

Although Class and Interface seem the same there have certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

| Class | Interface |
|---|---|
| In class, you can instantiate variables and create an object. | In an interface, you must initialize variables as they are final but you can't create an object. |
| A class can contain concrete (with implementation) methods | The interface cannot contain concrete (with implementation) methods. |
| The access specifiers used with classes are private, protected, and public. | In Interface only one specifier is used- Public. |

***Implementation:*** *To implement an interface, we use the keyword* ***implements.***

Let's consider the example of Vehicles like bicycles, cars, and bikes share common functionalities, which can be defined in an interface, allowing each class (e.g., Bicycle, Car, Bike) to implement them in its own way. This approach ensures code reusability, scalability, and consistency across different vehicle types.

**Example :**

```
import java.io.*;
interface Vehicle {

// Abstract methods defined

        void changeGear(int a);
        void speedUp(int a);
        void applyBrakes(int a);
}
```

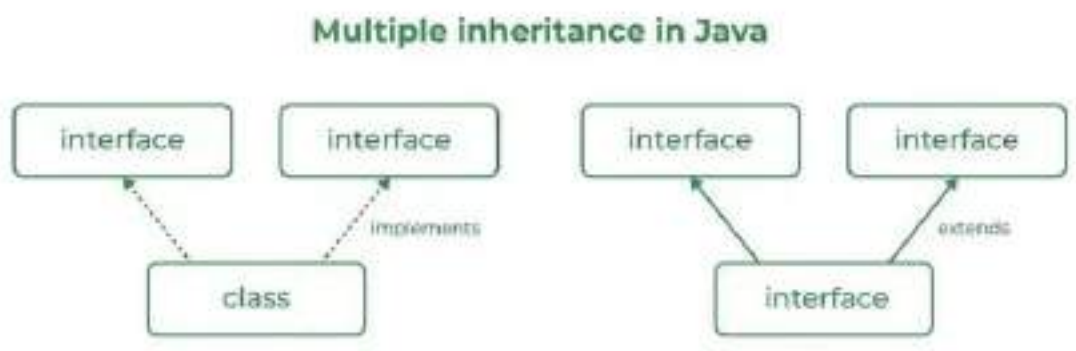
```java
// Class implementing vehicle interface
class Bicycle implements Vehicle
{
         int speed;
         int gear;
        // Change gear
        @Override
        public void changeGear(int newGear)
        {
                    gear = newGear;
        }

        // Increase speed
        @Override
        public void speedUp(int increment)
        {
                    speed = speed + increment;
        }

        // Decrease speed
        @Override
        public void applyBrakes(int decrement)
        {
                    speed = speed - decrement;
        }
        public void printStates()
        {
                    System.out.println("speed: " + speed
                                + " gear: " + gear);
        }
}

// Class implementing vehicle interface
class Bike implements Vehicle
{
        int speed;
        int gear;
        // Change gear
        @Override
        public void changeGear(int newGear){
                    gear = newGear;
        }

        // Increase speed
        @Override
        public void speedUp(int increment){
                    speed = speed + increment;
        }

        // Decrease speed
        @Override
        public void applyBrakes(int decrement){
                    speed = speed - decrement;
        }
        public void printStates() {
                System.out.println("speed: " + speed
          + " gear: " + gear);
        }
}

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

                // Instance of Bicycle(Object)
                Bicycle bicycle = new Bicycle();
                bicycle.changeGear(2);
                bicycle.speedUp(3);
                bicycle.applyBrakes(1);
                System.out.print("Bicycle present state : ");
                bicycle.printStates();

                // Instance of Bike (Object)
                Bike bike = new Bike();
                bike.changeGear(1);
                bike.speedUp(4);
                bike.applyBrakes(3);

                System.out.print("Bike present state : ");
                bike.printStates();
        }
}
```

**Output**

```
Bicycle present state : speed: 2 gear: 2

Bike present state : speed: 1 gear: 1
```

**Multiple Inheritance in Java Using Interface**

Multiple Inheritance is an OOPs concept that can’t be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. let us check this with an example.



**Example:**

```
import java.io.*;

// Add interface
interface Add
{
                int add(int a,int b);
}

// Sub interface
interface Sub
{
               int sub(int a,int b);
}


// Calculator class implementing

// Add and Sub
class Cal implements Add , Sub
{
   // Method to add two numbers
  public int add(int a,int b){
               return a+b;
 }

// Method to sub two numbers
public int sub(int a,int b){
                return a-b;
}
}

class GFG{
 // Main Method
  public static void main (String[] args)
 {
                // instance of Cal class
               Cal x = new Cal();
               System.out.println("Addition : " + x.add(2,1));
               System.out.println("Substraction : " + x.sub(2,1));

                }
}
```

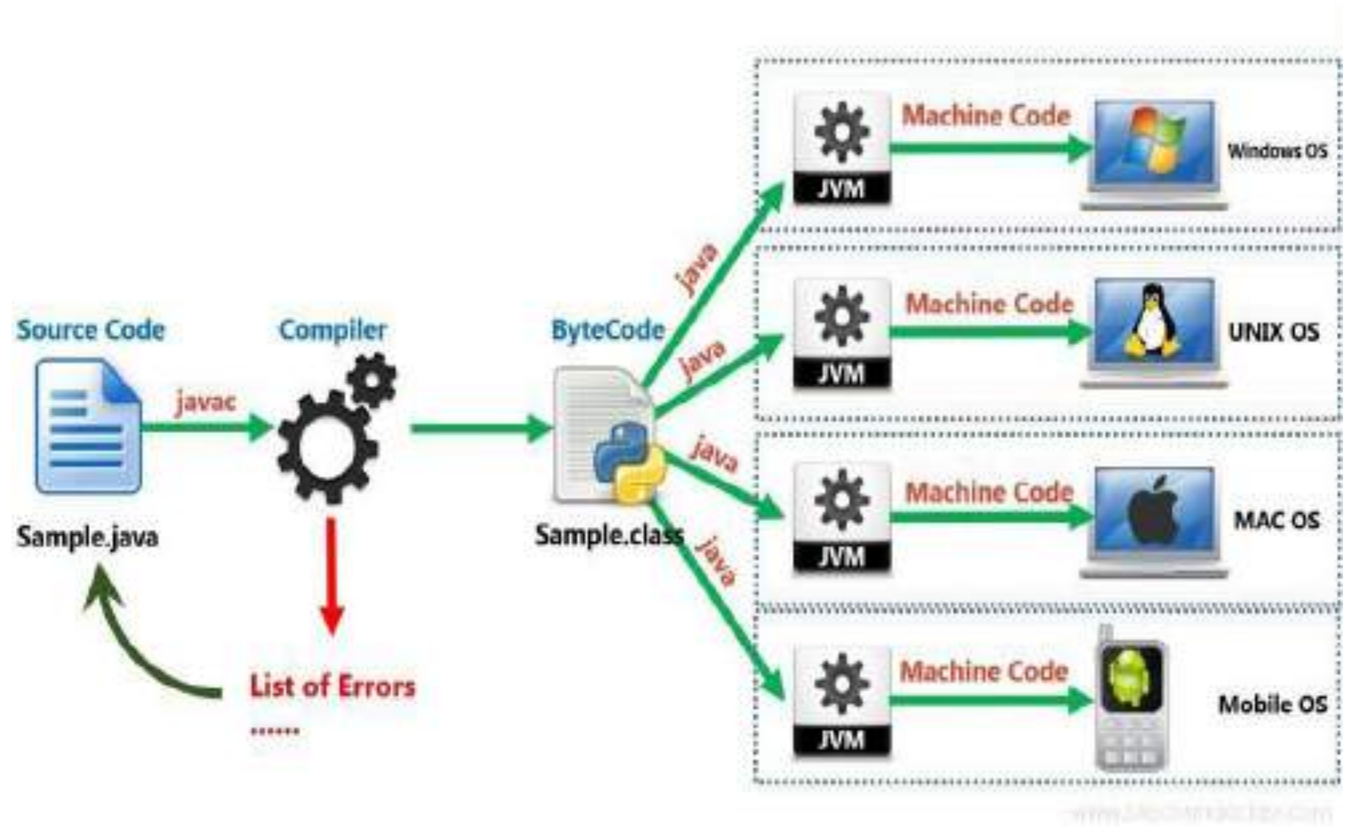
**Output**

```
Addition : 3

Substraction : 1
```

## Execution Process of JavaProgram

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.



# DEPARTMENT OF INFORMATION TECHNOLOGY
# Java Programming (R22CSI2215)

**UNIT-II**

**Exception Handling :** Exception and Error, Exception Types, Exception Handler, Exception Handling Clauses – try, catch, finally, throws and the throw statement, Built-in-Exceptions and Custom Exceptions.

**Files and I/O Streams:** The file class, Streams, The Byte Streams, Filtered Byte Streams, The Random Access File class.

## 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.

## Exception and Error

Both exceptions and errors are the subclasses of a throwable class. The error implies a problem that mostly arises due to the shortage of system resources. On the other hand, the exceptions occur during runtime and compile time. Let's find out some major differences between exceptions and errors.

**What are Errors?**

- The error signifies a situation that mostly happens due to the absence of system resources.
- The system crash and memory errors are an example of errors. It majorly occurs at runtime.

**What are Exceptions?**

- The exceptions are the issues that can appear at runtime and compile time.
- It majorly arises in the code or program authored by the developers.
- There are two types of exceptions: Checked exceptions and Unchecked exceptions.

**Difference between Errors and Exceptions in Java**

| S.No | Errors | Exceptions |
|---|---|---|
| **1.** | The error indicates trouble that primarily occurs due to the scarcity of system resources. | The exceptions are the issues that can appear at runtime and compile time. |
| **2.** | It is not possible to recover from an error. | It is possible to recover from an exception. |
| **3.** | In java, all the errors are unchecked. | In java, the exceptions can be both checked and unchecked. |
| **4.** | The system in which the program is running is responsible for errors. | The code of the program is accountable for exceptions. |
| **5.** | They are described in the java.lang.Error package. | They are described in |

## Exception Types in Java

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***.

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.

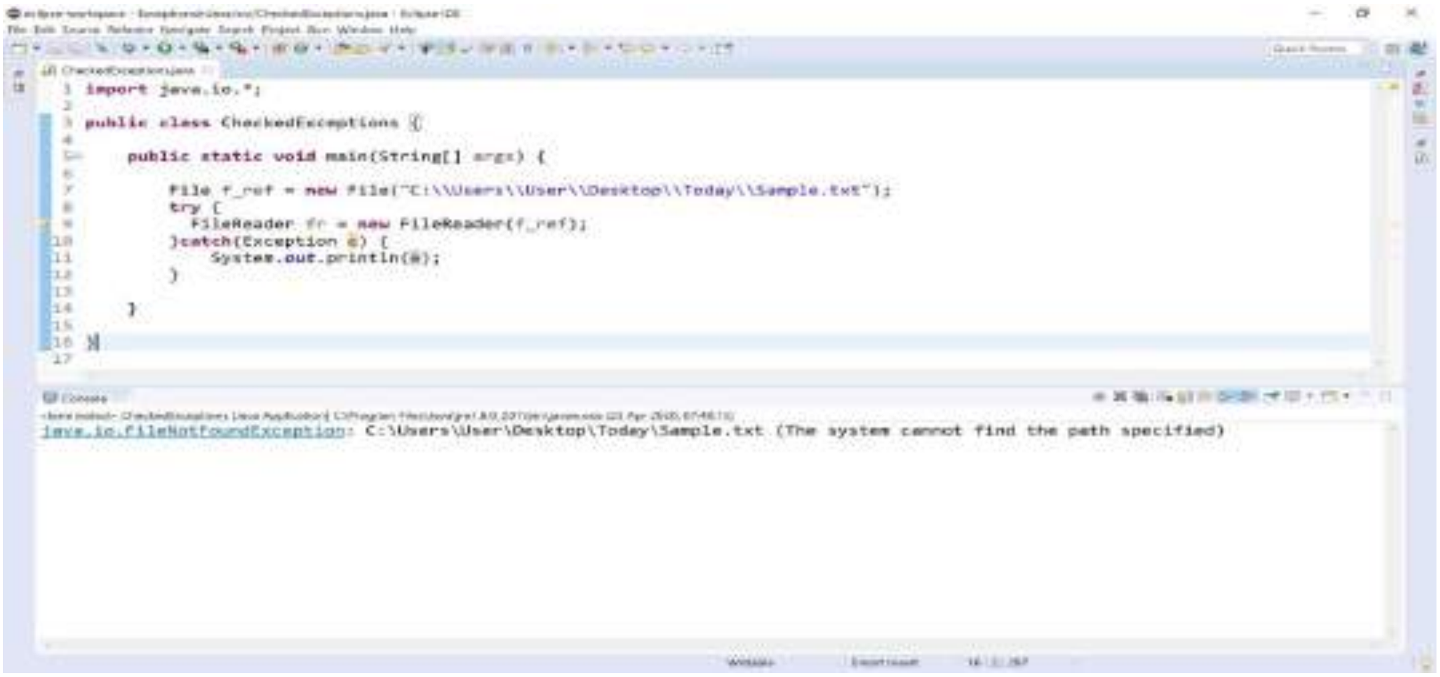
- IOException
- FileNotFoundException
- ClassNotFoundException
- SQLException
- DataAccessException
- InstantiationException
- UnknownHostException

☐ 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**



```
import java.io.*;

public class CheckedExceptions {

    public static void main(String[] args) {

        File f_ref = new File("C:\\Users\\User\\Desktop\\Today\\Sample.txt");
        try {
          FileReader fr = new FileReader(f_ref);
        }catch(Exception e) {
            System.out.println(e);
        }

    }

}
```

```
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.

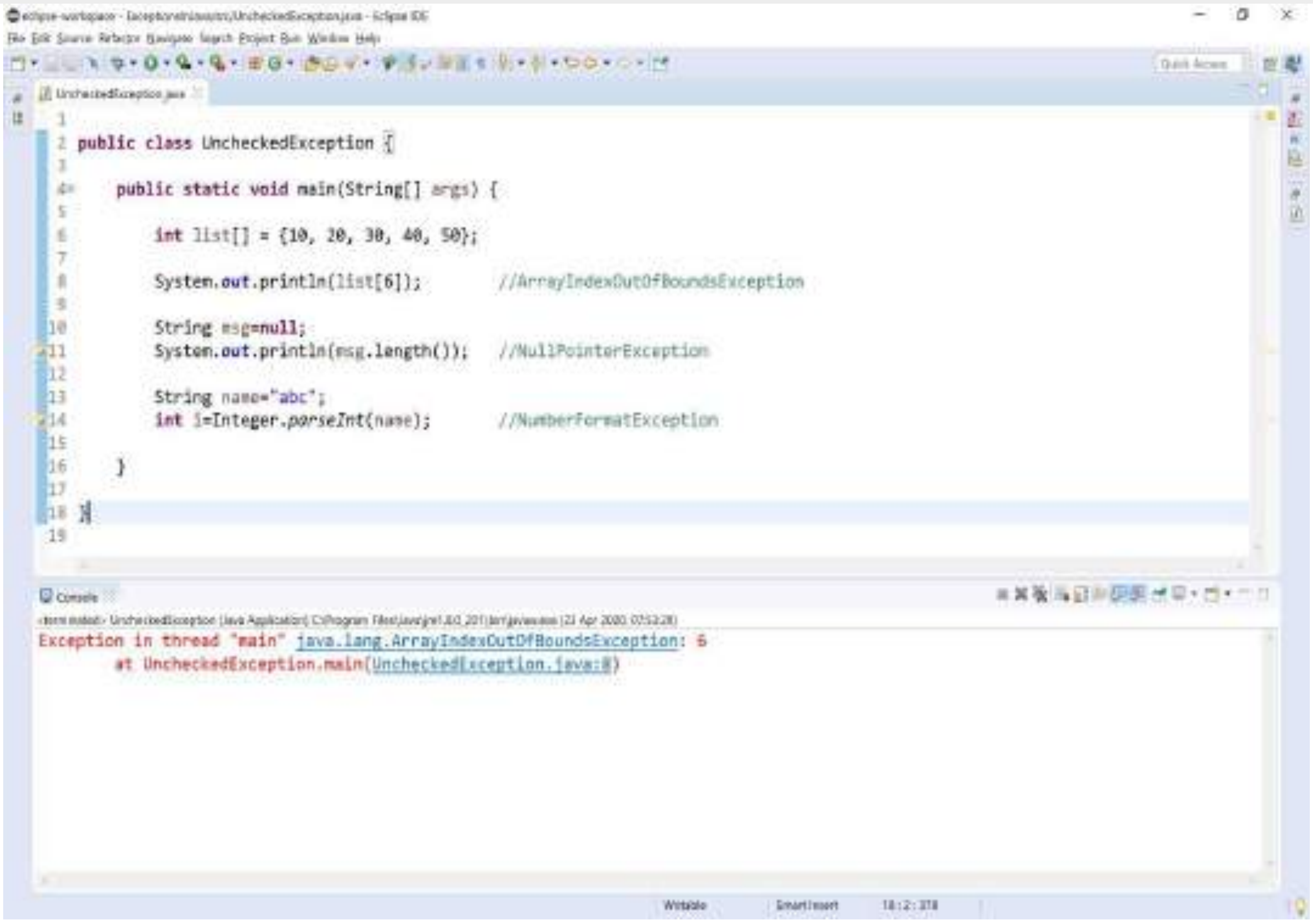
The following are a few built-in classes used to handle unchecked exceptions in java.

- ArithmeticException
- NullPointerException
- NumberFormatException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException

☐In the exception class hierarchy, the unchecked exception classes are the children of Run time Exception class, which is a child class of Exception class.
The unchecked exception is also known as a run time exception.

Let's look at the following example program for the unchecked exceptions.

### Example - Unchecked Exceptions

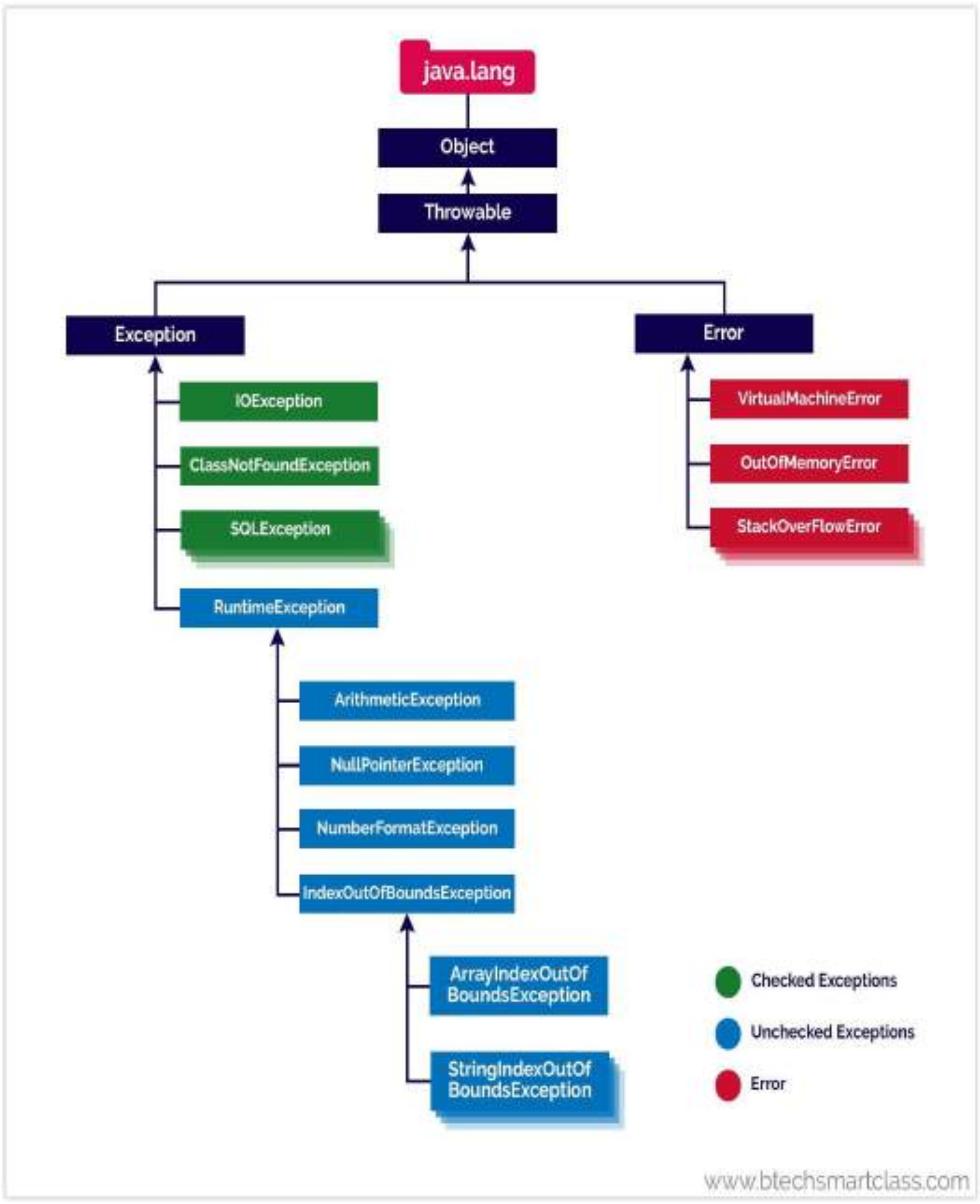

File Edit Source Refactor Navigate Search Project Run Window Help
UncheckedException.java
Console
Writable
Smart Insert
18 : 2 : 378

```java

public class UncheckedException {

    public static void main(String[] args) {

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

        System.out.println(list[6]);          //ArrayIndexOutOfBoundsException

        String msg=null;
        System.out.println(msg.length());     //NullPointerException

        String name="abc";
        int i=Integer.parseInt(name);         //NumberFormatException

    }

}
```

```
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 suports 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

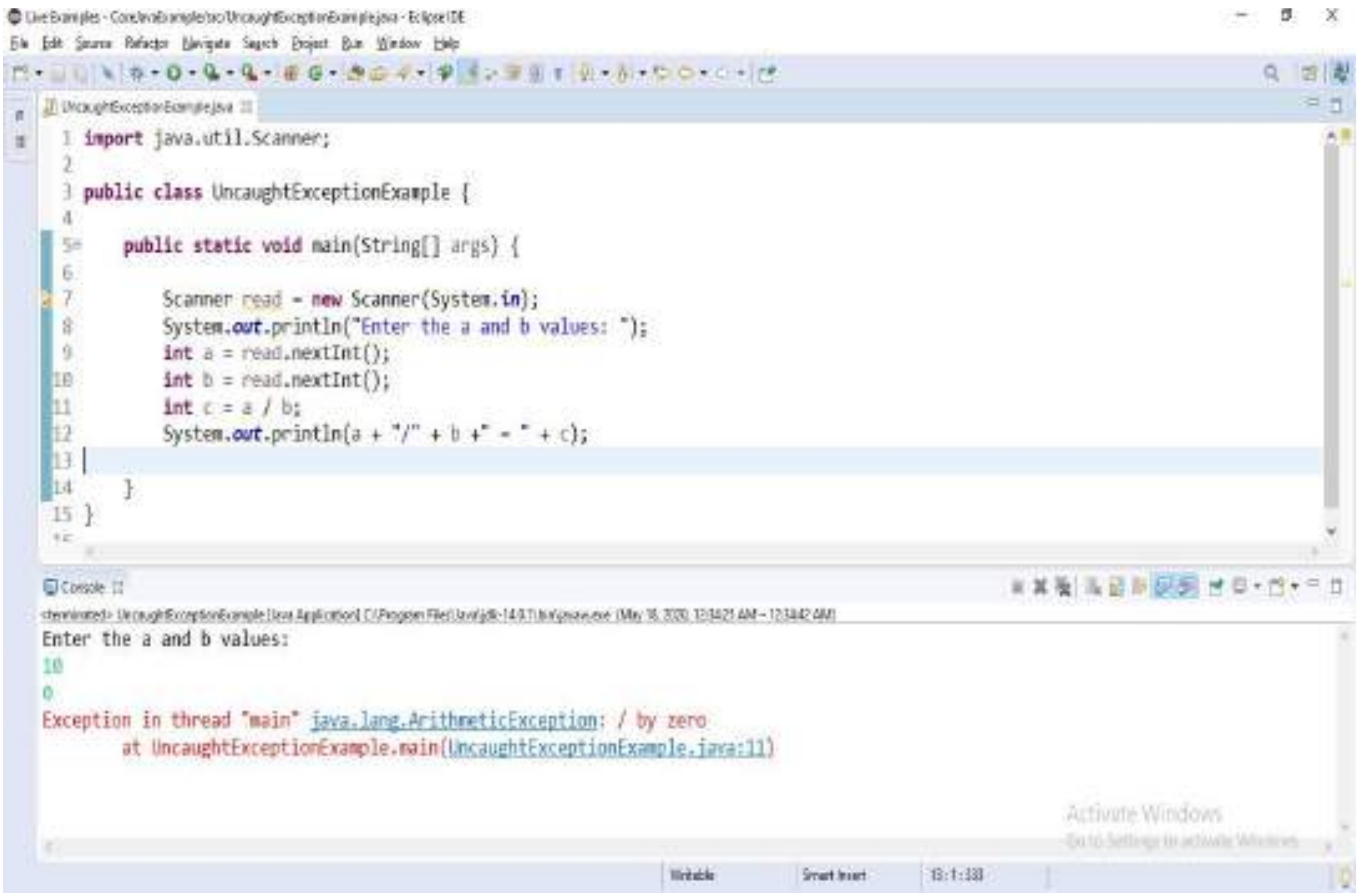
- The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stable the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.
- In resumptive model we may use a method call that want resumption like behavior. We may also place the try block in a while loop that keeps re-entering the try block util 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 a exception message with the help of 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



```java
import java.util.Scanner;

public class UncaughtExceptionExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        int a = read.nextInt();
        int b = read.nextInt();
        int c = a / b;
        System.out.println(a + "/" + b +" = " + c);

    }
}
```

```
Enter the a and b values:
10
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at UncaughtExceptionExample.main(UncaughtExceptionExample.java:11)
```

When we execute the above code, it produce the following output for the value a = 10 and b = 0.

In the above example code, we are not used try and catch blocks, but when the value of b is zero the division by zero exception occurs and it 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 tests the occurence of an exception.
- The keyword catch is used to define a block of code that handles the exception occured in the respective try block.
- 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.
- Both try and catch are used as a pair. Every try block must have one or more catch blocks. We can not use try without atleast one catch, and catch alone can be used (catch without try is not allowed).

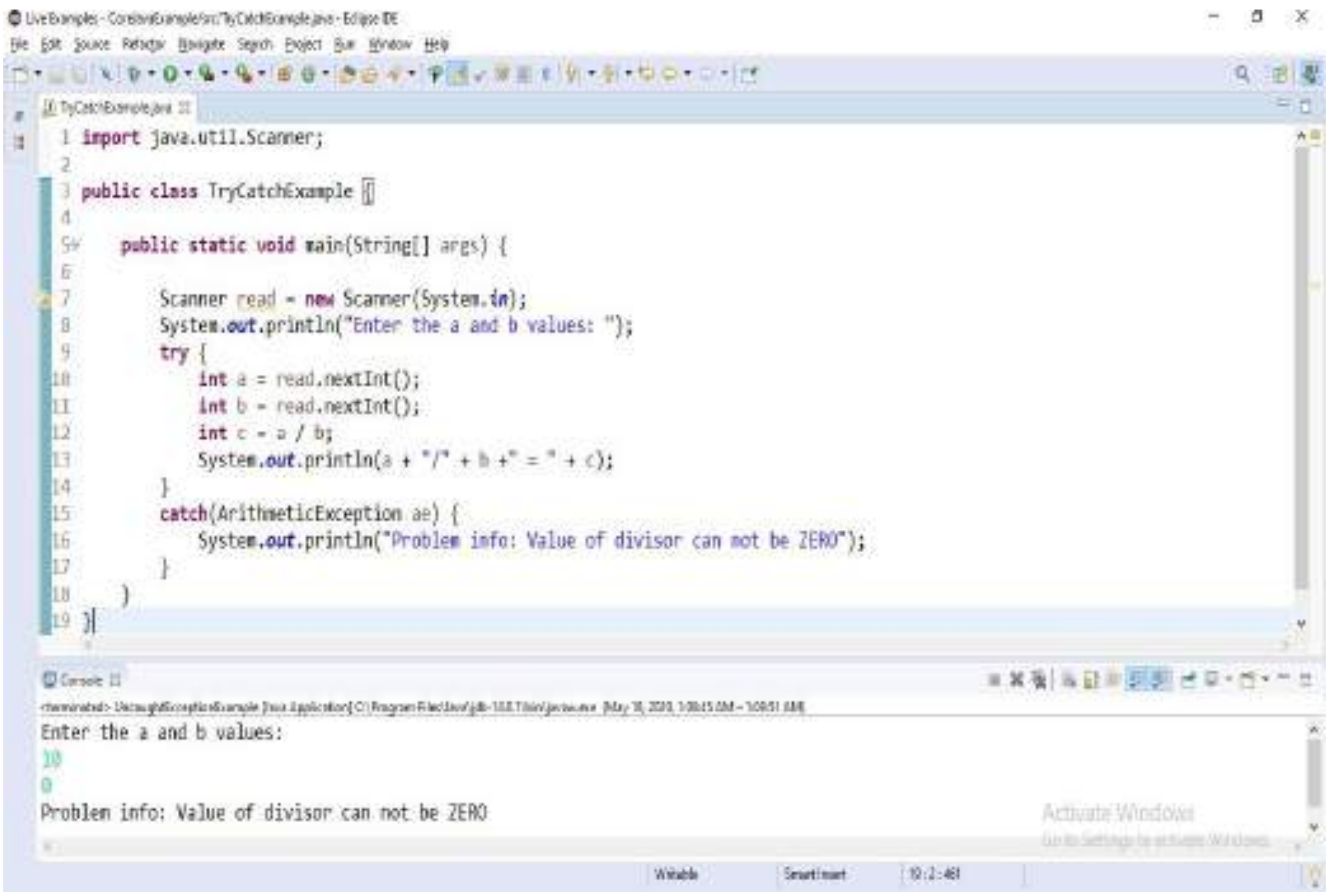
The following is the syntax of try and catch blocks.

**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**



```java
import java.util.Scanner;

public class TryCatchExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        try {
            int a = read.nextInt();
            int b = read.nextInt();
            int c = a / b;
            System.out.println(a + "/" + b +" = " + c);
        }
        catch(ArithmeticException ae) {
            System.out.println("Problem info: Value of divisor can not be ZERO");
        }
    }
}
```

```
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 transfered 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.

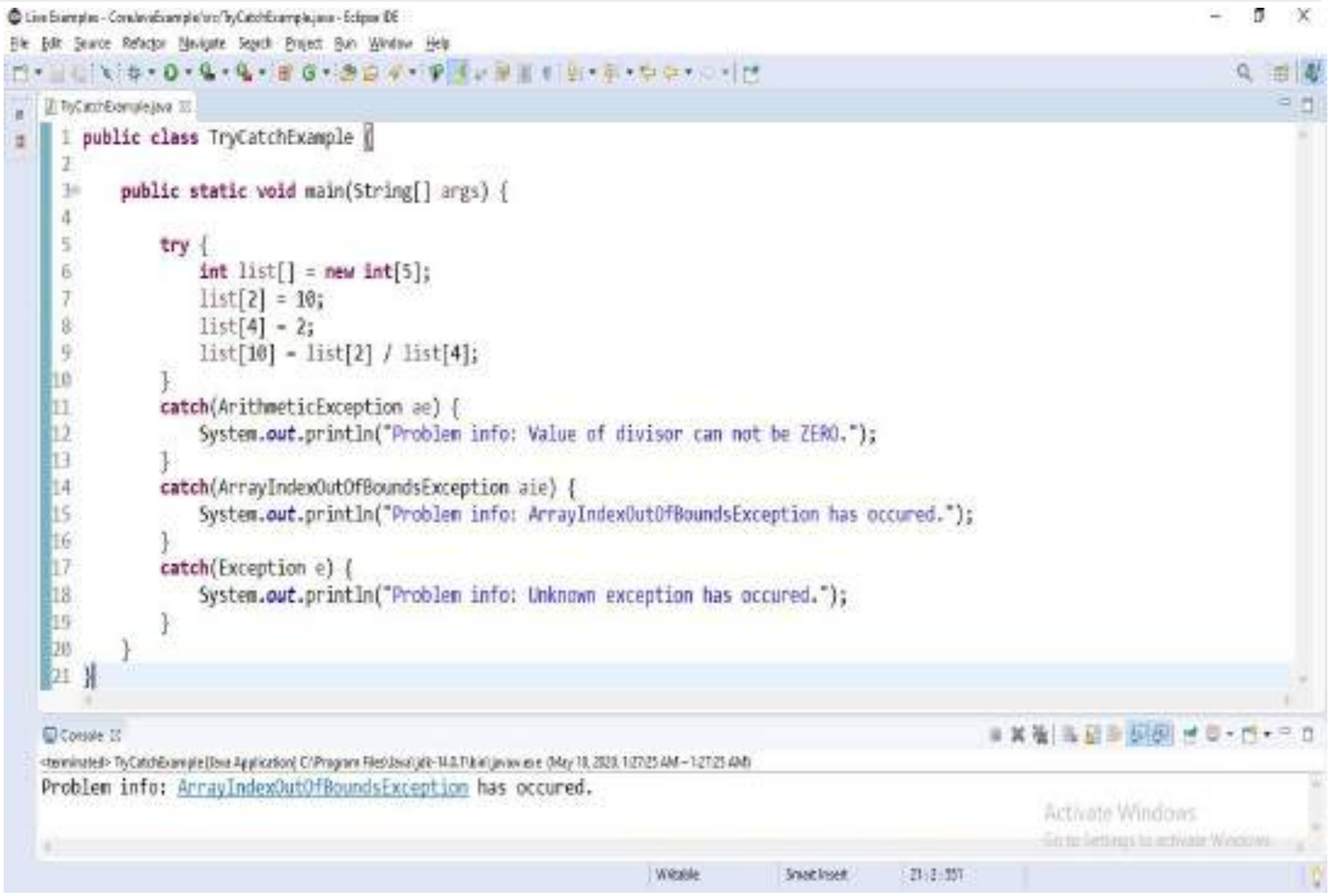
 The try block generates only one exception at a time, and at a time only one catch block is executed.

 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'slookatthefollowingexampleJavacodetoillustratemultiplecatchclauses.

**Example**



```
public class TryCatchExample {

    public static void main(String[] args) {

        try {
            int list[] = new int[5];
            list[2] = 10;
            list[4] = 2;
            list[10] = list[2] / list[4];
        }
        catch(ArithmeticException ae) {
            System.out.println("Problem info: Value of divisor can not be ZERO.");
        }
        catch(ArrayIndexOutOfBoundsException aie) {
            System.out.println("Problem info: ArrayIndexOutOfBoundsException has occured.");
        }
        catch(Exception e) {
            System.out.println("Problem info: Unknown exception has occured.");
        }
    }
}
```

```
<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 occured.
```

## 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.

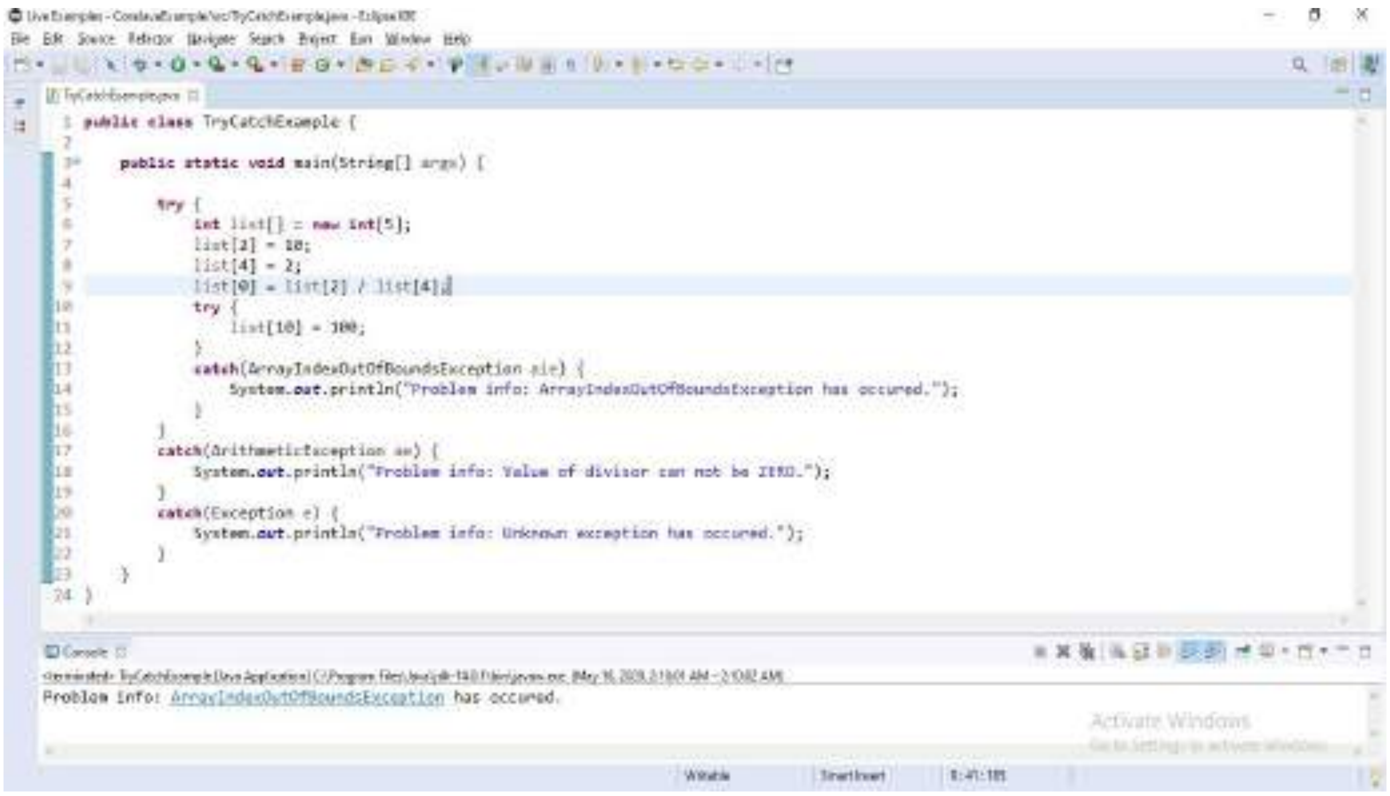
When there are nested try blocks, each try block must have one or more separate catch blocks.

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: ArrayIndexOutOfBounds Exception has occured.");
            }
        }
        catch(ArithmeticException ae){
            System.out.println("Problem info: Value of divisor cannot be ZERO.");
        }
        catch(Exception e){
            System.out.println("Problem info: Unknown exception has occured.");
        }
    }
}
```

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



```
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 occured.");
            }
        }
        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 occured.");
        }
    }
}
```

```
Problem info: ArrayIndexOutOfBoundsException has occured.
```

☐In case of nested try blocks, if an exception occured in the inner try block and it's 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 subclasses 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.

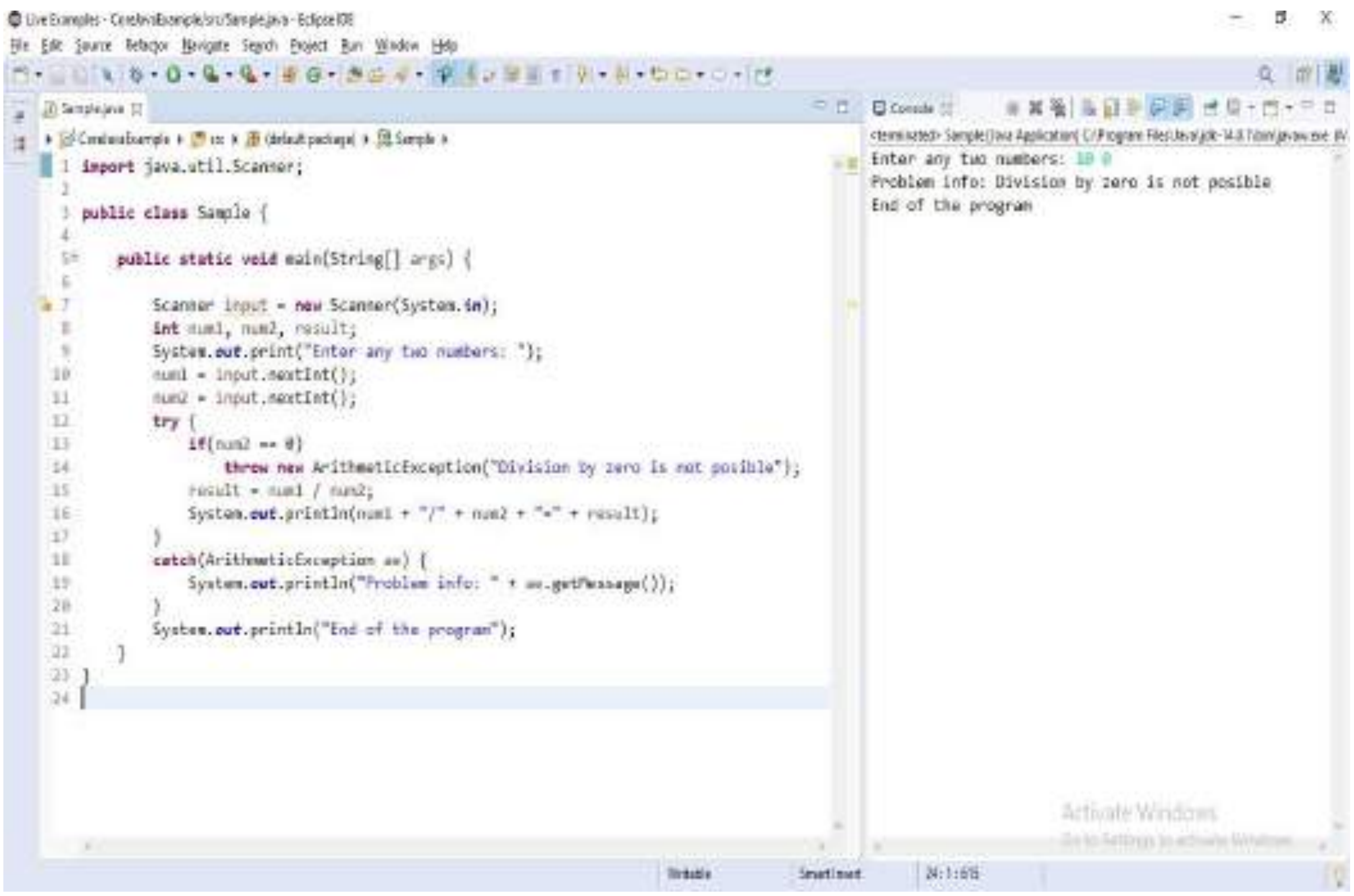
**Syntax**

```
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**



```
import java.util.Scanner;

public class Sample {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        int num1, num2, result;
        System.out.print("Enter any two numbers: ");
        num1 = input.nextInt();
        num2 = input.nextInt();
        try {
            if(num2 == 0)
                throw new ArithmeticException("Division by zero is not posible");
            result = num1 / num2;
            System.out.println(num1 + "/" + num2 + "=" + result);
        }
        catch(ArithmeticException ae) {
            System.out.println("Problem info: " + ae.getMessage());
        }
        System.out.println("End of the program");
    }
}
```

```
<terminated> Sample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe
Enter any two numbers: 10 0
Problem info: Division by zero is not posible
End of the program
```

## 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.

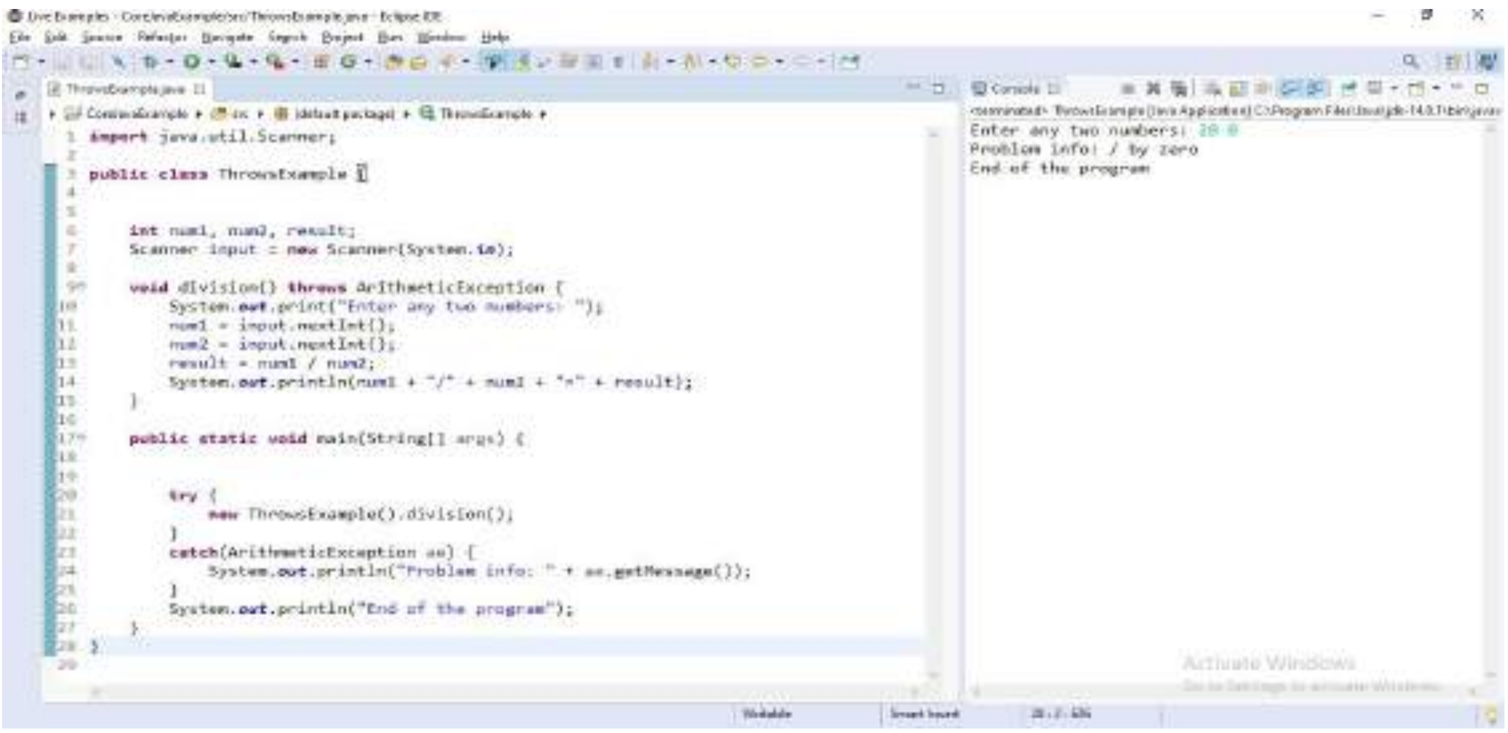
Let's look at the following example Java code to illustrate throws keyword.

### Example

```
import java.util.Scanner;
publicclassThrowsExample{
```

```java
    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.



```java
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");
    }
}

```

```
<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

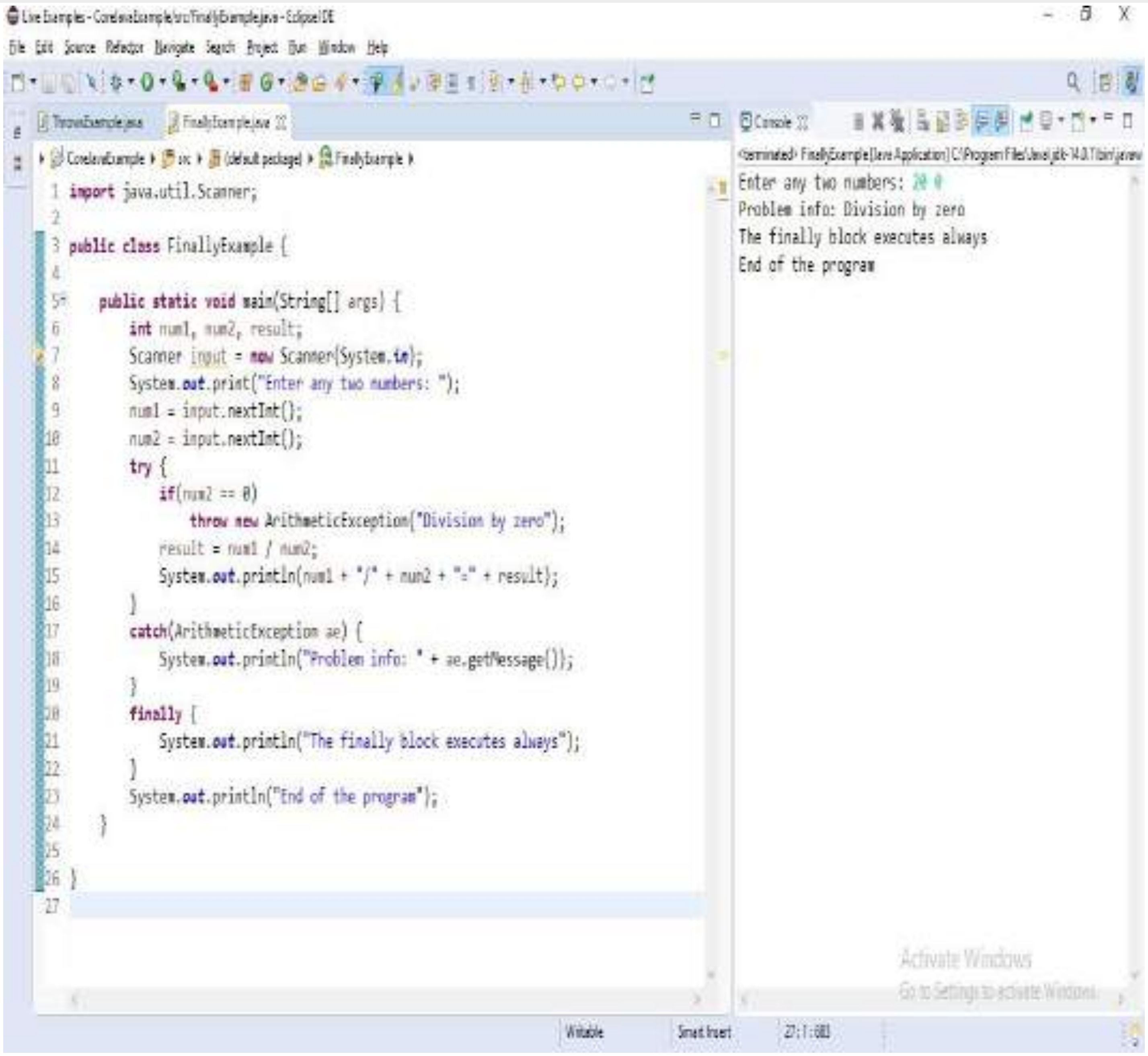
- The finally keyword used to define a block that must be executed irrespective of exception occurence.
- The basic purpose of finally keyword is to cleanup resources allocated by try block, such as closing file, closing database connection, etc.

☐ Only one finally block is allowed for each try block.

☐ Use of finally block is optional.

Let's look at the following example Java code to illustrate throws keyword.

### Example


Live Examples - CoreJavaExample/src/FinallyExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
ThrowsExample.java
FinallyExample.java
CoreJavaExample ▸ src ▸ (default package) ▸ FinallyExample ▸
Console
<terminated> FinallyExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw
Writable
Smart Insert
27 : 1 : 683

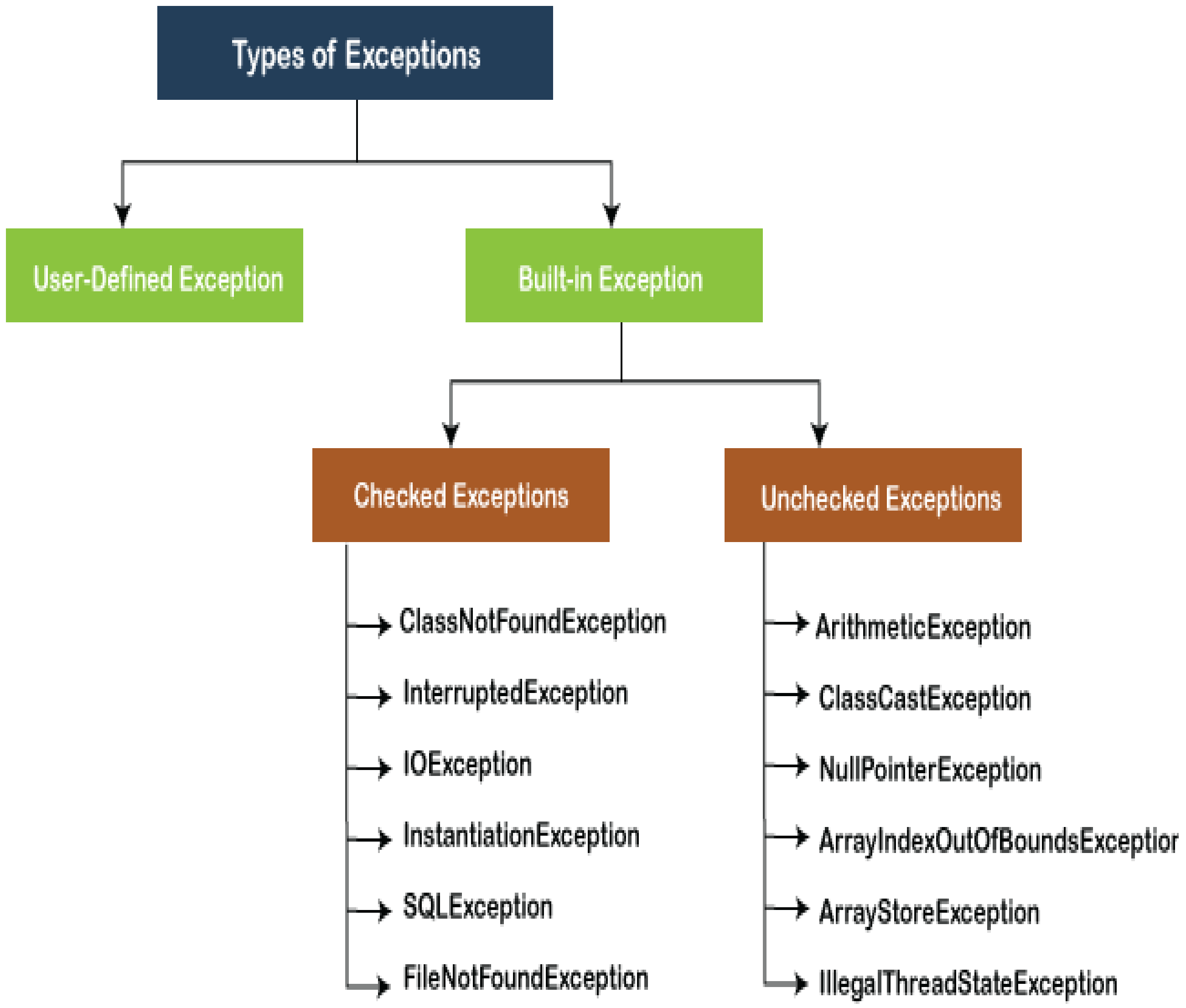
```java
import java.util.Scanner;

public class FinallyExample {

    public static void main(String[] args) {
        int num1, num2, result;
        Scanner input = new Scanner(System.in);
        System.out.print("Enter any two numbers: ");
        num1 = input.nextInt();
        num2 = input.nextInt();
        try {
            if(num2 == 0)
                throw new ArithmeticException("Division by zero");
            result = num1 / num2;
            System.out.println(num1 + "/" + num2 + "=" + result);
        }
        catch(ArithmeticException ae) {
            System.out.println("Problem info: " + ae.getMessage());
        }
        finally {
            System.out.println("The finally block executes always");
        }
        System.out.println("End of the program");
    }

}
```

```
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 we redefined 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**<br><br>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**<br><br>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**<br><br>It is thrown when one attempts to access a method or member that visibility qualifiers do not allow. |
| 4 | **InstantiationException**<br><br>It is thrown when an application tries to create an instance of a class using the new Instance method in class Class, but the specified class object cannot be instantiated because it is an interface or is an abstract class. |
| 5 | **InterruptedException**<br><br>It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted. |
| 6 | **NoSuchFieldException**<br><br>It indicates that the class doesn't have a field of a specified name. |
| 7 | **NoSuchMethodException**<br><br>It is thrown when some JAR file has a different version at runtime that 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**<br><br>It handles the arithmeticexceptions like dividion by zero |
| 2 | **ArrayIndexOutOfBoundsException**<br><br>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**<br><br>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**<br><br>It is used to indicate that an assertion has failed |
| 5 | **ClassCastException**<br><br>It handles the situation when we try to improperly cast a class from one type to another. |
| 6 | **IllegalArgumentException**<br><br>This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument. |
| 7 | **IllegalMonitorStateException**<br><br>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**<br><br>It signals that a method has been invoked at an illegal or inappropriate time. |

| S. No. | Exception Class with Description |
|---|---|
| 9 | **IllegalThreadStateException**<br><br>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**<br><br>It is thrown when attempting to access an invalid index with in a collection, such as an array , vector , string , and so forth |
| 11 | **NegativeArraySizeException**<br><br>It is thrown if an applet tries to create an array with negative size. |
| 12 | **NullPointerException**<br><br>It is thrown when program attempts to use an object reference that has the null value. |
| 13 | **NumberFormatException**<br><br>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**<br><br>It is thrown by the Java Card Virtual Machine to indicate a security violation. |
| 15 | **StringIndexOutOfBounds**<br><br>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**<br><br>It is thrown to indicate that the requested operation is not supported. |

## Examples of Built-in Exception:

**Arithmetic exception :** It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
// Java program to demonstrate ArithmeticException

class ArithmeticException_Demo {

  public static void main(String[] args) {

  try {
              int a = 30, b = 0;
              int c = a / b; // cannot divide by zero
              System.out.println("Result = " + c);
     } catch (ArithmeticException e) {
              System.out.println("Can't divide a number by 0");
   }
  }
}
```

**Output**

```
Can't divide a number by 0
```

**ArrayIndexOutOfBounds Exception:** It is thrown to indicate that 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.

```
// Java program to demonstrate ArrayIndexOutOfBoundException

class ArrayIndexOutOfBound_Demo {

public static void main(String[] args) {

try {
              int[] a = new int[5];
              a[5] = 9; // accessing 6th element in an array of size 5
 } catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Array Index is Out Of Bounds");
}
}
}
```

**Output**

```
Array Index is Out Of Bounds
```

## Java Custom Exception

- In Java, we can create our own exceptions that are derived classes of the Exception class.
- Creating our own Exception is known as custom exception or user-defined exception.
- Basically, Java custom exceptions are used to customize the exception according to user need.
- A **custom exception in Java** is an exception defined by the user to handle specific application requirements.
- These exceptions extend either the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions).

### Why Use Java Custom Exceptions?

- To represent application-specific errors.
- To add clear, descriptive error messages for better debugging.
- To encapsulate business logic errors in a meaningful way.

## 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();
    }
}
```

## Java File Class

- The **File** is a built-in class in Java. In java, the File class has been defined in the **java.io** package.
- Java File class is a representation of a file or directory pathname.
- Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.
- Java File class contains several methods for working with the pathname, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

***Features:***

- It is an abstract representation of files and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative.
- The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it.
- A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

**How to Create a File Object?**

A File object is created by passing in a string that represents the name of a file, a String, or another File object. For example,

```
File a = new File("/usr/local/bin/geeks");
```

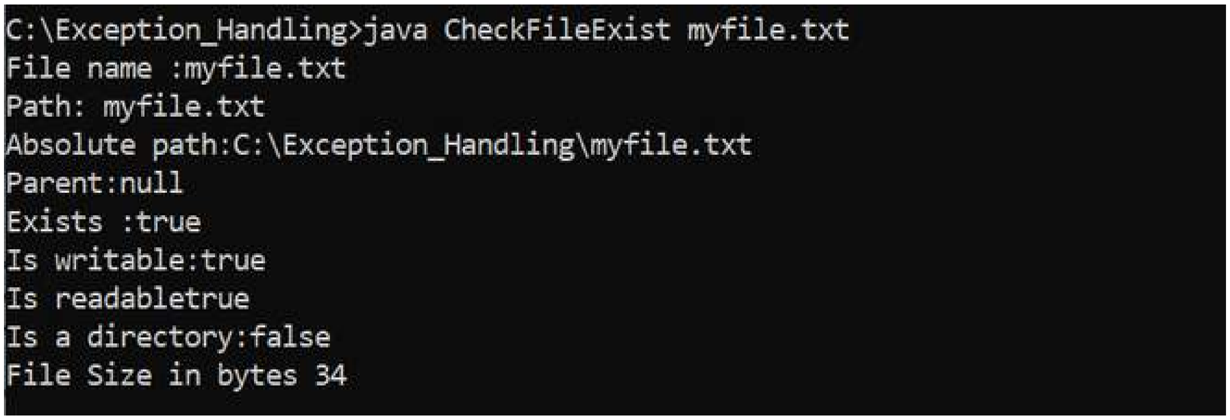
This defines an abstract file name for the geeks file in the directory /usr/local/bin. This is an absolute abstract file name.

**Example :** Program to check if a file or directory physically exists or not.

```
 //  In this Java program, we accepts a file or directory name from command line arguments.
// Then the program will check if that file or directory physically exist or not and it displays
// the property of that file or directory.
```

```
import java.io.File;

// Displaying file property

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

                // Accept file name or directory name through command line args
                String fname = args[0];

                // pass the filename or directory name to File  object
                File f = new File(fname);

                // apply File class methods on File object
                System.out.println("File name :" + f.getName());
                System.out.println("Path: " + f.getPath());
                System.out.println("Absolute path:" + f.getAbsolutePath());
                System.out.println("Parent:" + f.getParent());
                System.out.println("Exists :" + f.exists());

                if (f.exists())
                {
                        System.out.println("Is writable:" + f.canWrite());
                        System.out.println("Is readable" + f.canRead());
                        System.out.println("Is a directory:" + f.isDirectory());
                        System.out.println("File Size in bytes " + f.length());
                }
        }
}
```

***Output:***



```
C:\Exception_Handling>java CheckFileExist myfile.txt
File name :myfile.txt
Path: myfile.txt
Absolute path:C:\Exception_Handling\myfile.txt
Parent:null
Exists :true
Is writable:true
Is readabletrue
Is a directory:false
File Size in bytes 34
```

## Constructors of Java File Class

- **File(File parent, String child):** Creates a new File instance from a parent abstract pathname and a child pathname string.
- **File(String pathname):** Creates a new File instance by converting the given pathname string into an abstract pathname.
- **File(String parent, String child):** Creates a new File instance from a parent pathname string and a child pathname string.
- **File(URI uri):** Creates a new File instance by converting the given file: URI into an abstract pathname.

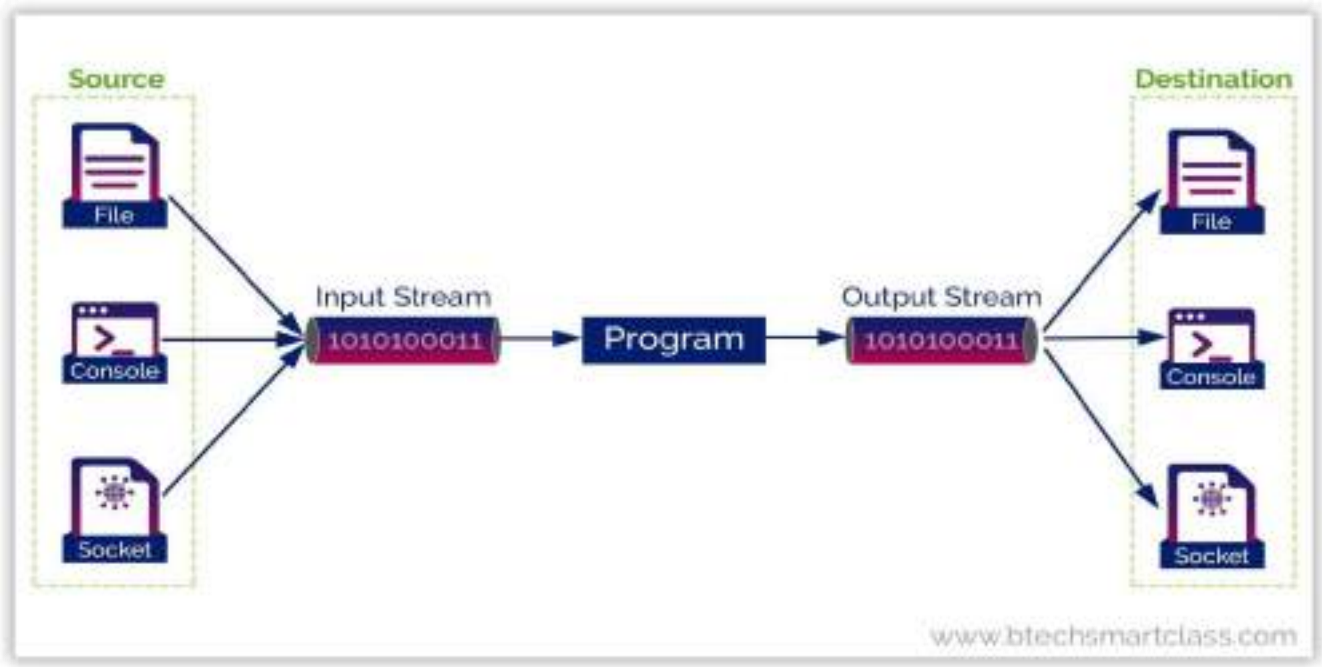
## Methods of File Class in Java

| Method | Description | Return Type |
|---|---|---|
| **canExecute()** | Tests whether the application can execute the file denoted by this abstract pathname. | boolean |
| **canRead()** | Tests whether the application can read the file denoted by this abstract pathname. | boolean |
| **canWrite()** | Tests whether the application can modify the file denoted by this abstract pathname. | boolean |
| **compareTo(File pathname)** | Compares two abstract pathnames lexicographically. | int |
| **createNewFile()** | Atomically creates a new, empty file named by this abstract pathname. | boolean |
| **delete()** | Deletes the file or directory denoted by this abstract pathname. | boolean |
| **equals(Object obj)** | Tests this abstract pathname for equality with the given object. | boolean |

| Method | Description | Return Type |
|---|---|---|
| **exists()** | Tests whether the file or directory denoted by this abstract pathname exists. | boolean |
| **getAbsolutePath()** | Returns the absolute pathname string of this abstract pathname. | String |
| **list()** | Returns an array of strings naming the files and directories in the directory. | String[] |
| **getName()** | Returns the name of the file or directory denoted by this abstract pathname. | String |
| **getParent()** | Returns the pathname string of this abstract pathname's parent. | String |
| **getParentFile()** | Returns the abstract pathname of this abstract pathname's parent. | File |
| **getPath()** | Converts this abstract pathname into a pathname string. | String |
| **isDirectory()** | Tests whether the file denoted by this pathname is a directory. | boolean |
| **isFile()** | Tests whether the file denoted by this abstract pathname is a normal file. | boolean |
| **length()** | Returns the length of the file denoted by this abstract pathname. | long |
| **listFiles()** | Returns an array of abstract pathnames denoting the files in the directory. | File[] |
| **mkdir()** | Creates the directory named by this abstract pathname. | boolean |

## Streams 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.

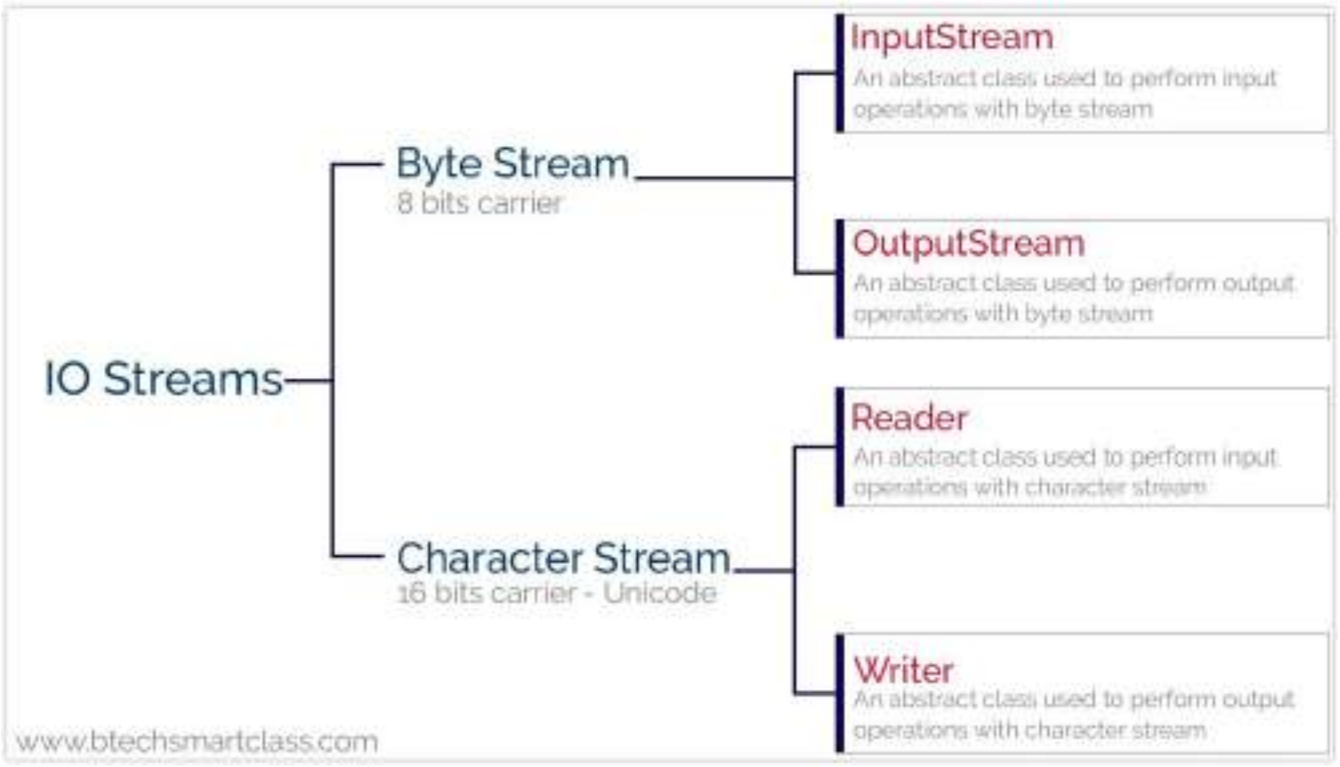


- In java, the stream-based IO operations are performed using two separate streams input stream and output stream.
- The input stream is used for input operations, and the output stream is used for output operations.
- The java stream is composed of bytes.
- In Java, every program creates 3 streams automatically, 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, they are as follows.

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

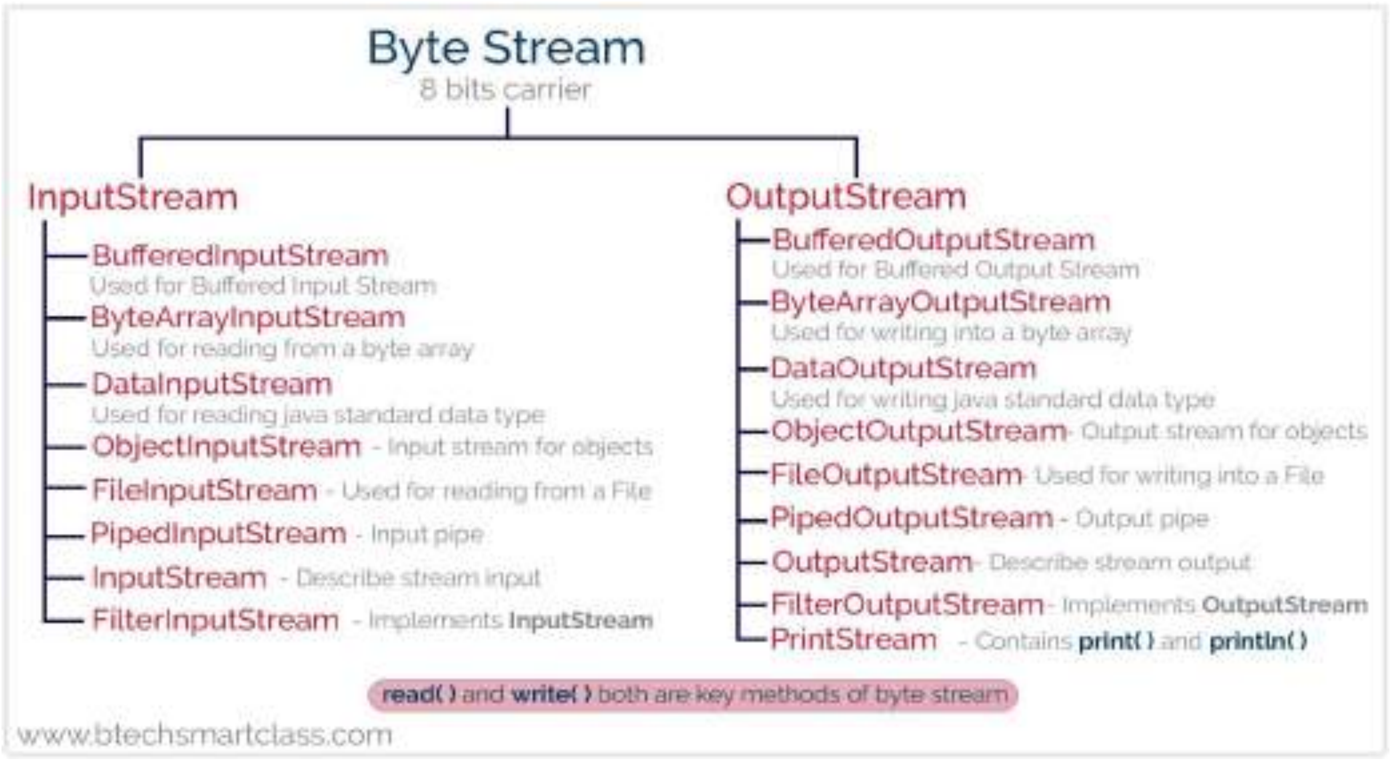
The following picture shows how streams are categorized, and various built-in classes used by the java IO system.


IO Streams
Byte Stream
8 bits carrier
Character Stream
16 bits carrier - Unicode
InputStream
An abstract class used to perform input operations with byte stream
OutputStream
An abstract class used to perform output operations with byte stream
Reader
An abstract class used to perform input operations with character stream
Writer
An abstract class used to perform output operations with character stream
www.btechsmartclass.com

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 8bits carrier.
- The byte stream in java allows us to transmit 8bits of data.
- In Java1.0 version all IO operations were byte oriented, there was no other stream (character stream).
- The java byte stream is defined by two abstract classes, **Input Stream** and **Output Stream**.
- The Input Stream class used for byte stream based input operations, and the Output Stream class used for byte stream based output operations.
- The Input Stream and Output Stream 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()**<br>It returns the number of bytes that can be read from the input stream. |
| 2 | **int read()**<br>It reads the next byte from the input stream. |
| 3 | **int read(byte[]b)**<br>It reads a chunk of bytes from the input stream and store them in its byte array, b. |
| 4 | **void close()**<br>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.

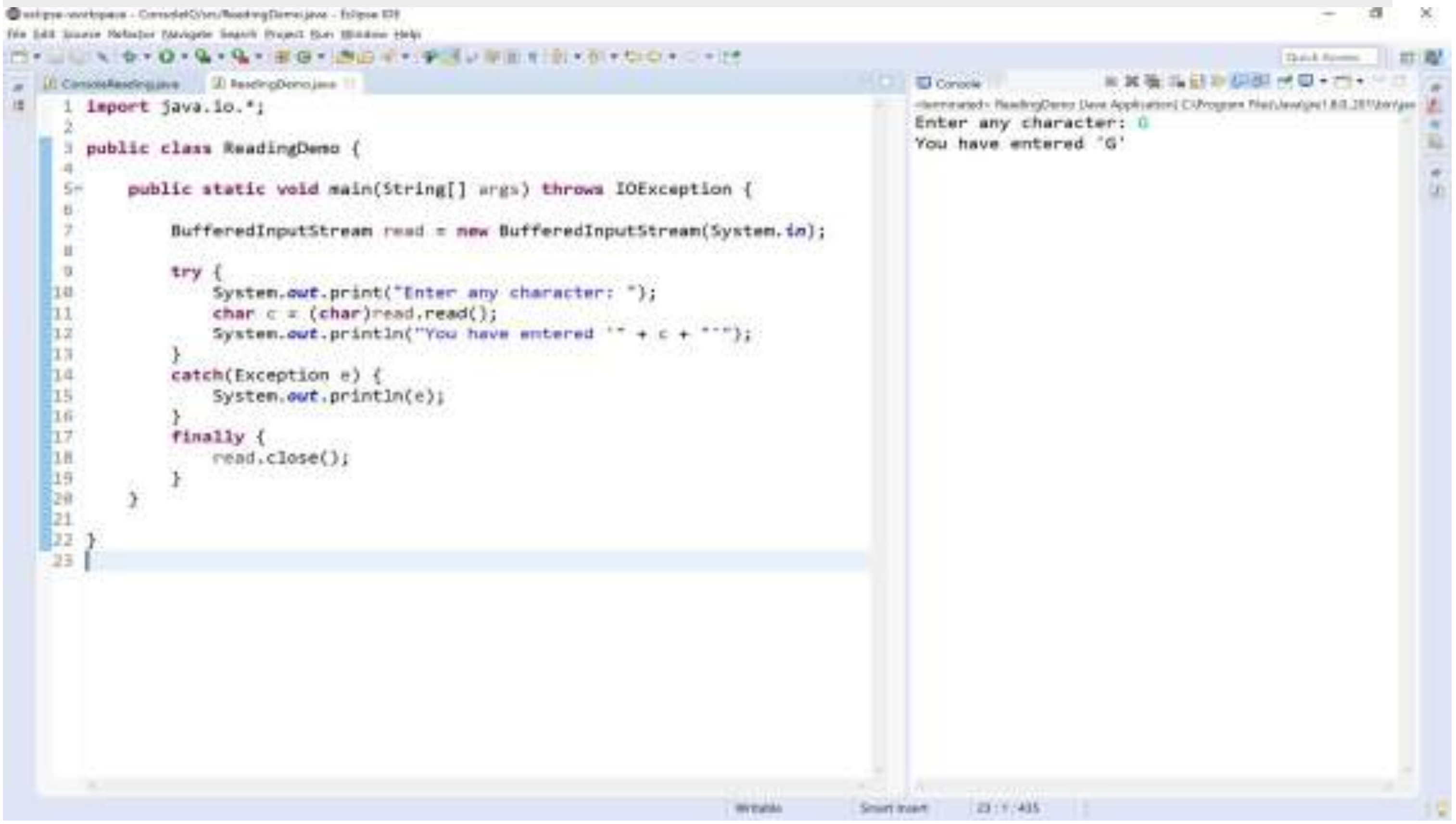
| S.No. | Method with Description |
|---|---|
| 1 | **void write(intn)**<br>It writes byte (contained in an int) to the outputstream. |
| 2 | **void write(byte[]b)**<br>It writes a whole byte array(b) to the output stream. |
| 3 | **void flush()**<br>It flushes the output steam by forcing out buffered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

## 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.

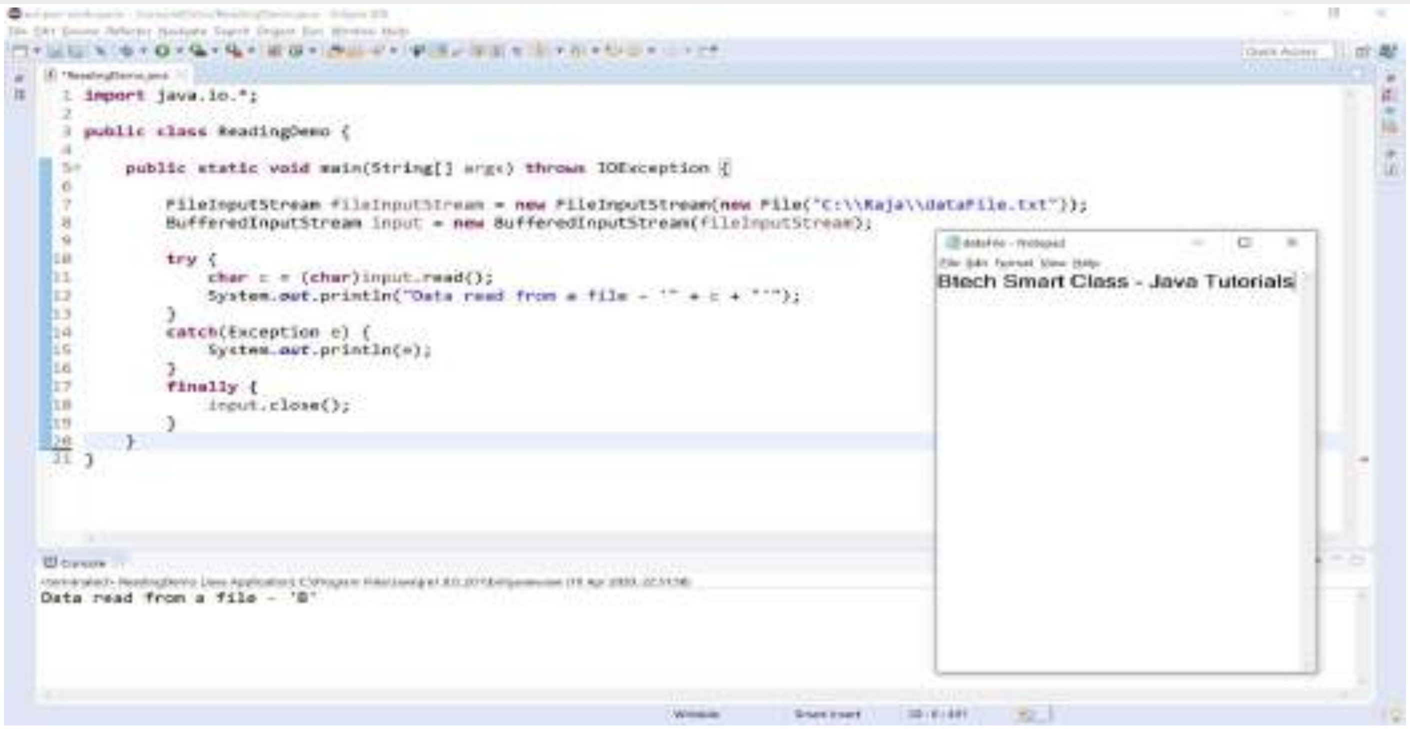
### Example1 – Reading from console



```java
import java.io.*;

public class ReadingDemo {

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

        BufferedInputStream read = new BufferedInputStream(System.in);

        try {
            System.out.print("Enter any character: ");
            char c = (char)read.read();
            System.out.println("You have entered '" + c + "'");
        }
        catch(Exception e) {
            System.out.println(e);
        }
        finally {
            read.close();
        }
    }

}
```

```
Enter any character: G
You have entered 'G'
```

## Example 2 - Reading from a file



```
import java.io.*;

public class ReadingDemo {

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

        FileInputStream fileInputStream = new FileInputStream(new File("C:\\Raja\\dataFile.txt"));
        BufferedInputStream input = new BufferedInputStream(fileInputStream);

        try {
            char c = (char)input.read();
            System.out.println("Data read from a file - '" + c + "'");
        }
        catch(Exception e) {
            System.out.println(e);
        }
        finally {
            input.close();
        }
    }
}
```

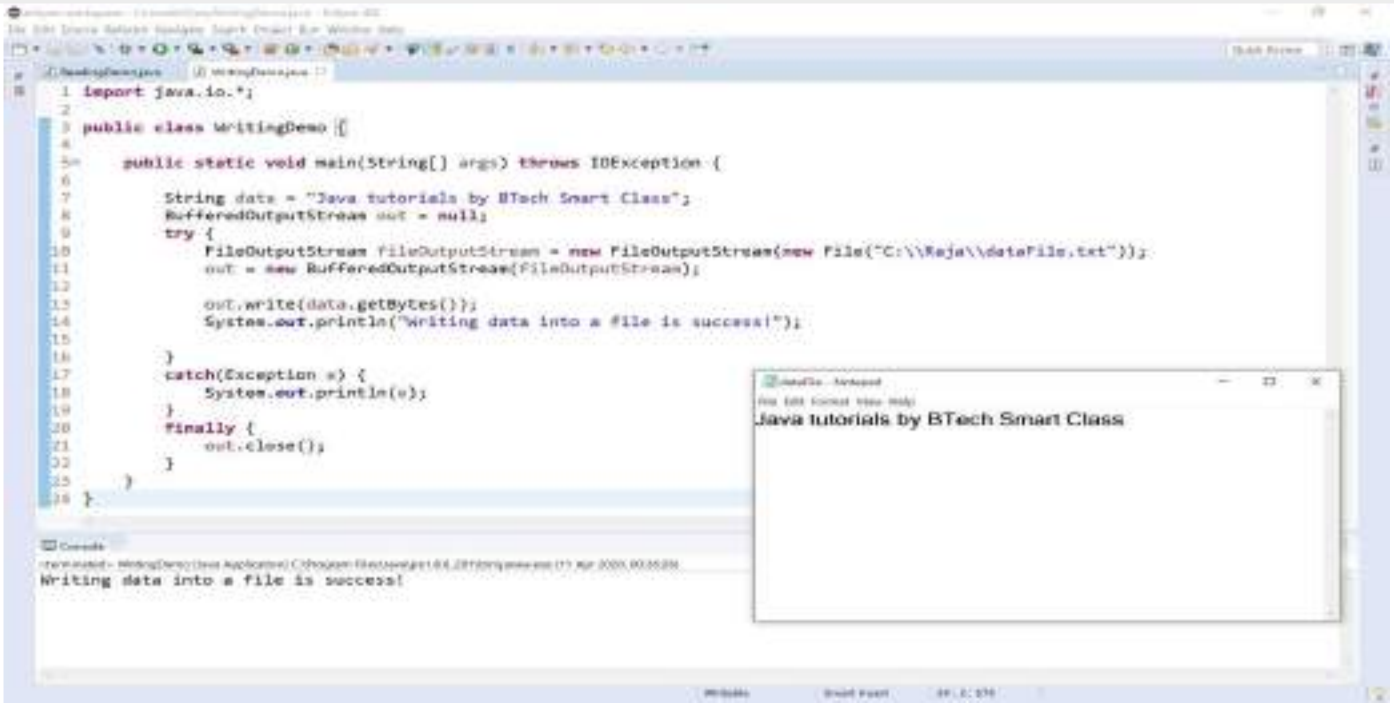
```
Data read from a file - 'B'
```

```
Btech Smart Class - Java Tutorials
```

## 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



```
import java.io.*;

public class WritingDemo {

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

        String data = "Java tutorials by BTech Smart Class";
        BufferedOutputStream out = null;
        try {
            FileOutputStream fileOutputStream = new FileOutputStream(new File("C:\\Raja\\dataFile.txt"));
            out = new BufferedOutputStream(fileOutputStream);

            out.write(data.getBytes());
            System.out.println("Writing data into a file is success!");

        }
        catch(Exception e) {
            System.out.println(e);
        }
        finally {
            out.close();
        }
    }
}
```

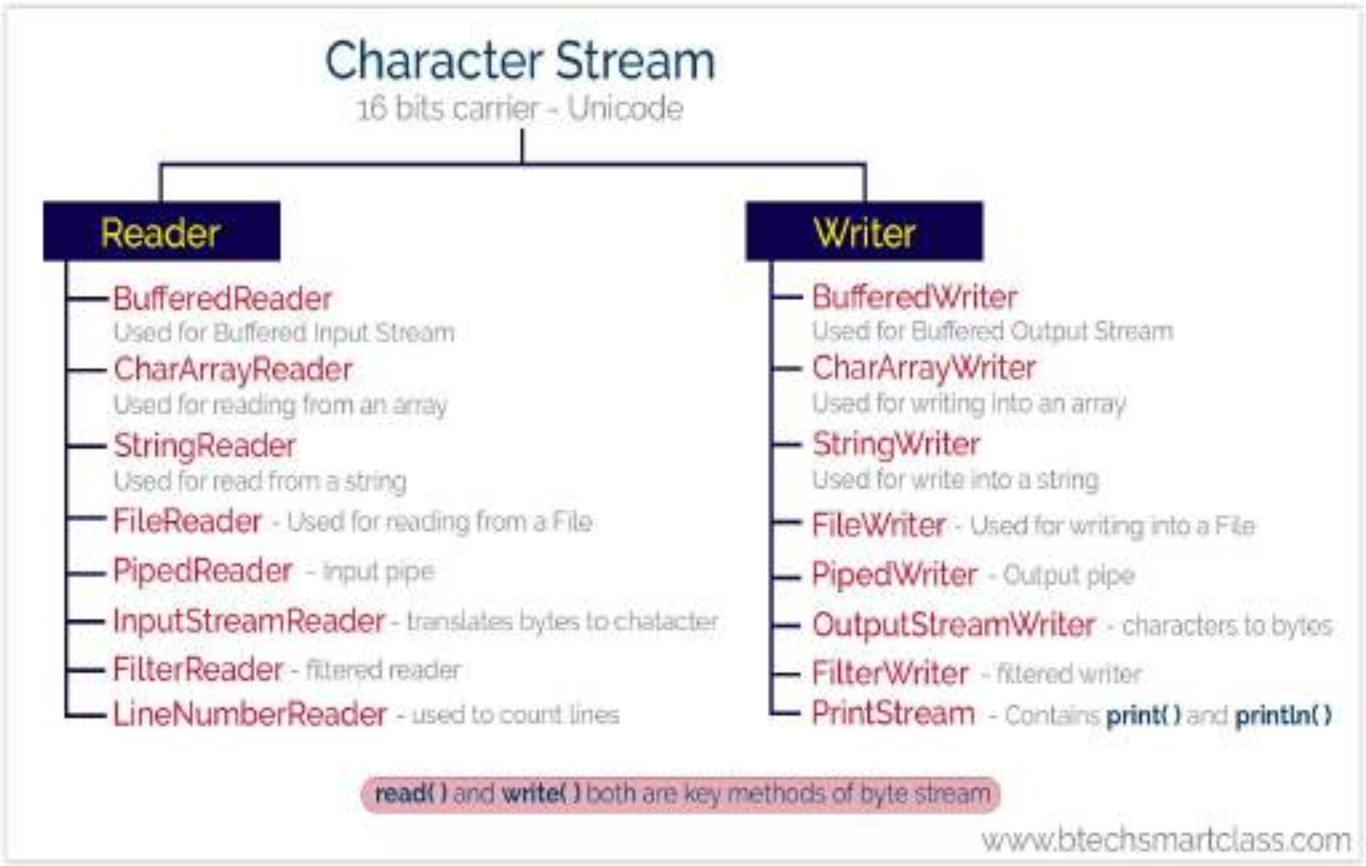
```
Writing data into a file is success!
```

```
Java tutorials by BTech Smart Class
```

## 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 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 charater stream based output operations.
- The Reader and Writer classes have several concreate classes to perform various IO operations based on the character stream.

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


Character Stream
16 bits carrier - Unicode
Reader
BufferedReader
Used for Buffered Input Stream
CharArrayReader
Used for reading from an array
StringReader
Used for read from a string
FileReader - Used for reading from a File
PipedReader - input pipe
InputStreamReader - translates bytes to chatacter
FilterReader - filtered reader
LineNumberReader - used to count lines
Writer
BufferedWriter
Used for Buffered Output Stream
CharArrayWriter
Used for writing into an array
StringWriter
Used for write into a string
FileWriter - Used for writing into a File
PipedWriter - Output pipe
OutputStreamWriter - characters to bytes
FilterWriter - filtered writer
PrintStream - Contains print( ) and println( )
read( ) and write( ) both are key methods of byte stream
www.btechsmartclass.com

## 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()**<br>It reads the next character from the input stream. |
| 2 | **int read(char[]cbuffer)**<br>It reads a chunk of characters from the input stream and store them in its byte array, cbuffer. |
| 3 | **int read(char[]cbuf, int off, int len)**<br>It reads charaters into a portion of an array**.** |
| 4 | **int read(Char Buffer target)**<br>It reads charaters into the specified character buffer. |
| 5 | **String readLine()**<br>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 linefeed. |
| 6 | **boolean ready()**<br>It tells whether the stream is ready to be read. |
| 7 | **void close()**<br>It closes the input stream and also frees any resources connected with this inputstream. |

## 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 |
|---|---|
| 1 | **void flush()**<br>It flushes the output steam by forcing out buffered bytes to be written out. |
| 2 | **void write(char[]cbuf)**<br>It writes a whole array (cbuf)to the output |
| 3 | **void write(char[]cbuf, int off, int len)**<br>It writes a portion of an array of |

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

## Reading data using BufferedReader

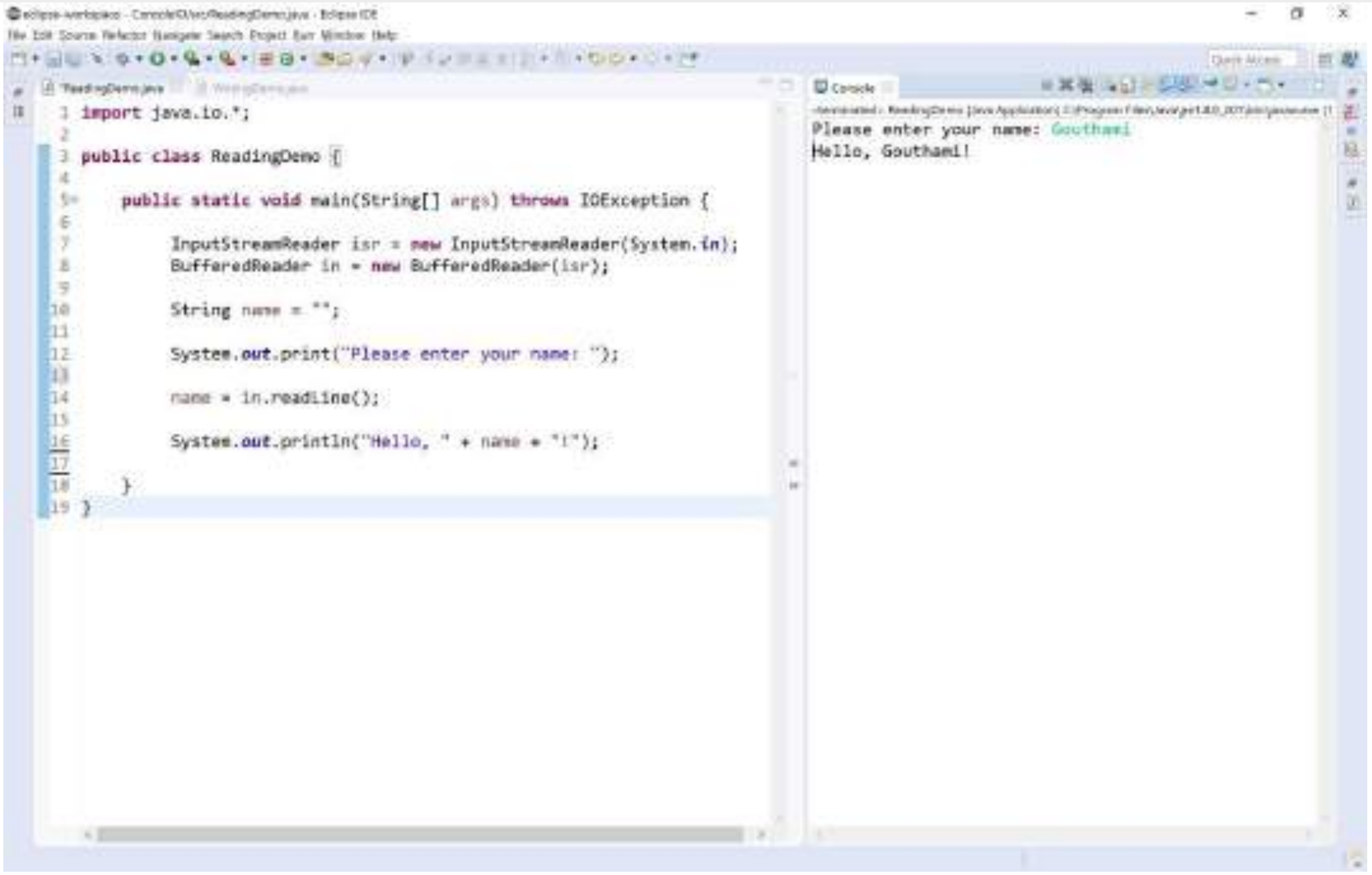
We can use the BufferedReader class to read data from the console.

The BufferedInputStream class needs InputStreamReaderclass.

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.

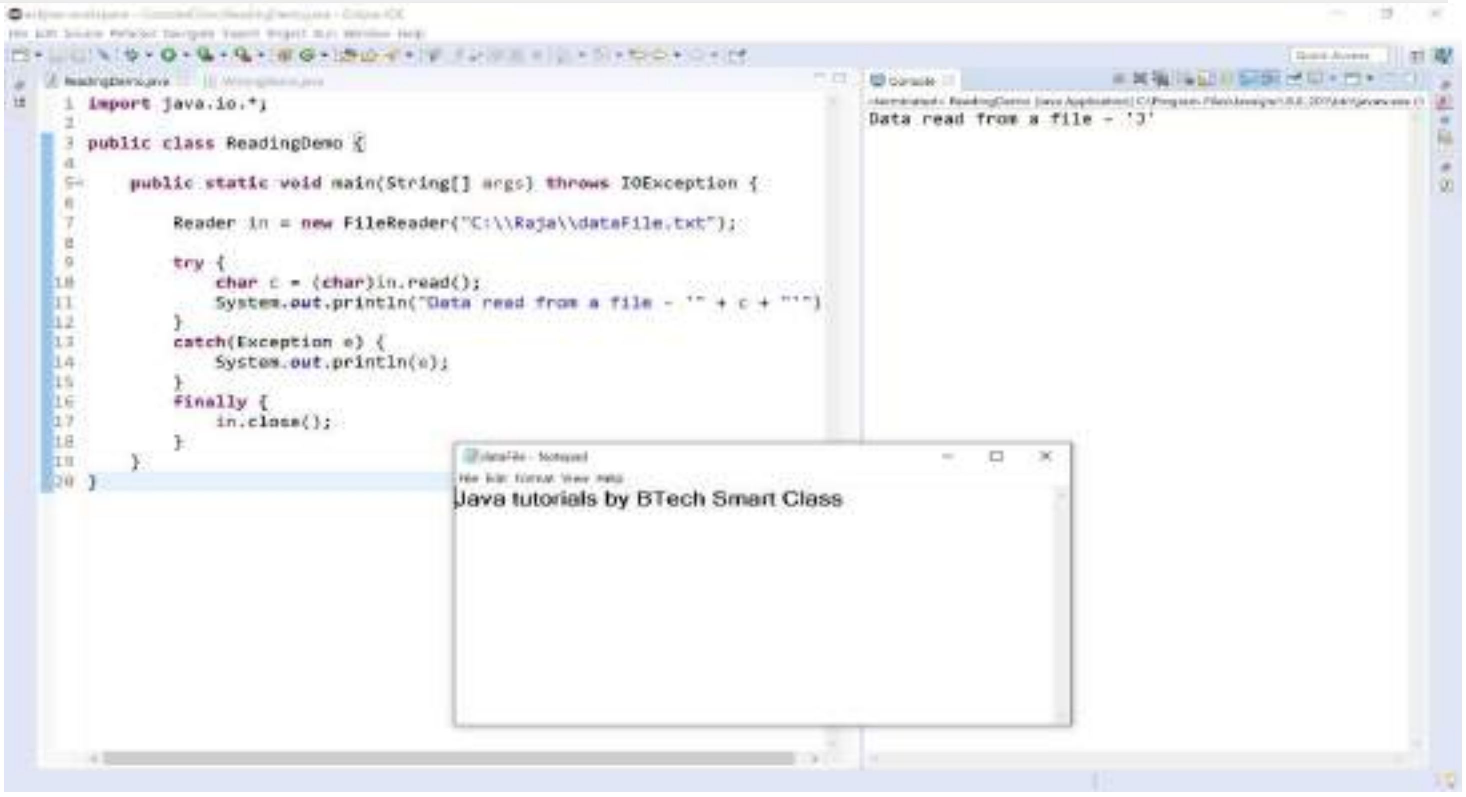
## Example1-Readingfromconsole



```java
import java.io.*;

public class ReadingDemo {

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

        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(isr);

        String name = "";

        System.out.print("Please enter your name: ");

        name = in.readLine();

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

    }
}
```

```
Please enter your name: Gouthami
Hello, Gouthami!
```

## Example 2-Readingfrom a file



```java
import java.io.*;

public class ReadingDemo {

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

        Reader in = new FileReader("C:\\Raja\\dataFile.txt");

        try {
            char c = (char)in.read();
            System.out.println("Data read from a file - '" + c + "'")
        }
        catch(Exception e) {
            System.out.println(e);
        }
        finally {
            in.close();
        }
    }
}
```

```
Data read from a file - 'J'
```

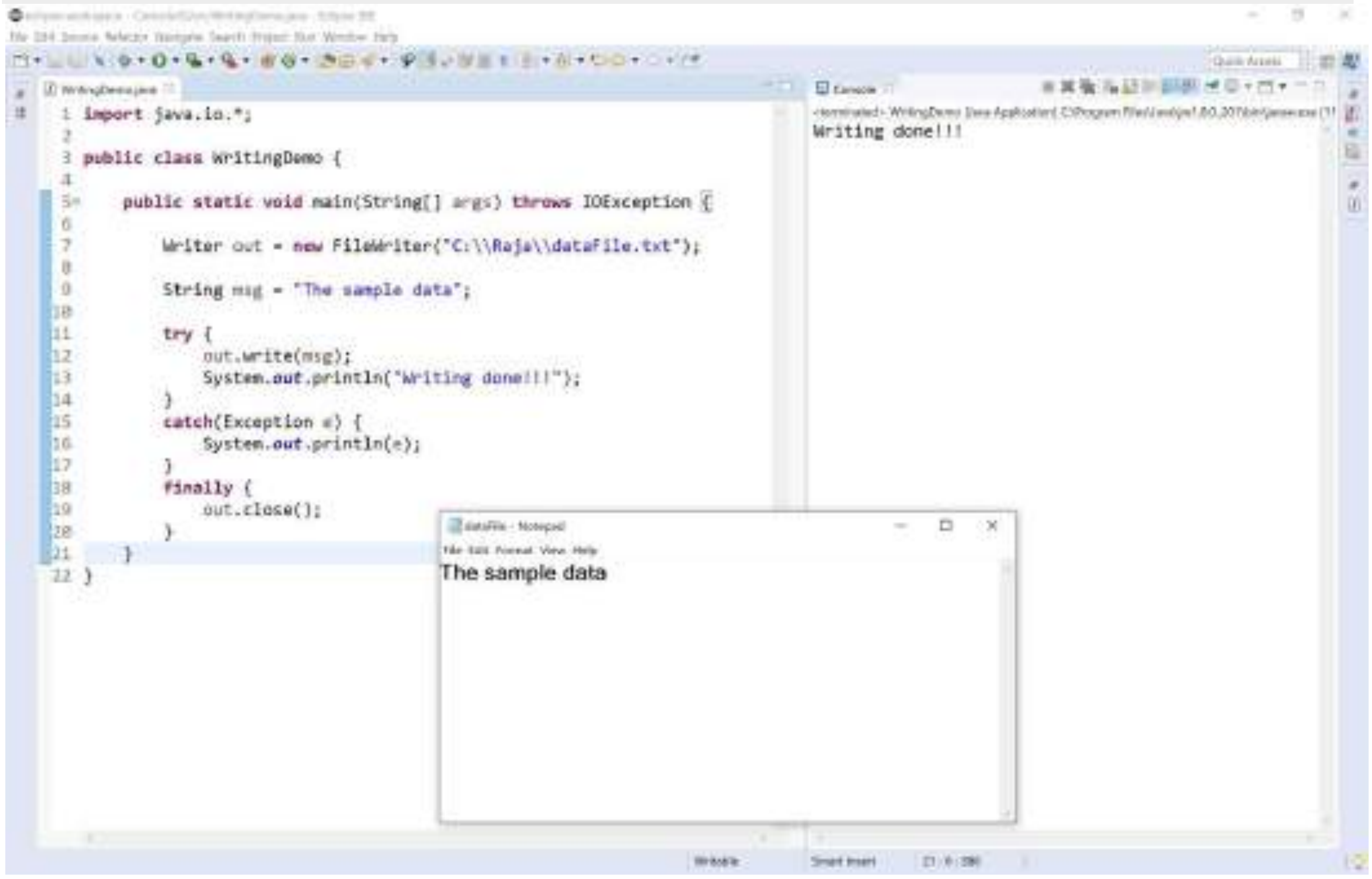
Java tutorials by BTech Smart Class

## Writing data using FileWriter

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

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

## Example- Writingdata intoafile



```java
import java.io.*;

public class WritingDemo {

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

		Writer out = new FileWriter("C:\\Raja\\dataFile.txt");

		String msg = "The sample data";

		try {
			out.write(msg);
			System.out.println("Writing done!!!");
		}
		catch(Exception e) {
			System.out.println(e);
		}
		finally {
			out.close();
		}
	}
}
```

```
Writing done!!!
```

```
The sample data
```

## 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 toanother file using FileInoutStream and FileOutputStream classes.

Example

```
import ava.io.*;


public class FileReadingTest
{
      public static void main(Stringargs[]) throws IOException
      {
               FileInputStream in = null;
               FileOutputStream out=null;
              try
              {
                         in=new FileInputStream("C:\\II-IT\\Input-File.txt");
                         out=new FileOutputStream("C:\\II-IT\\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 implementedbasedonthecharacterstream.TheFileWriterclassprovidesa 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(Stringargs[]) throws IOException{
      FileReader in = null;
      FileWriterout=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();
         }
      }
   }
}
```

## Filtered Byte Stream

- **Filter streams** are streams that wrap around underlying input or output streams and adds new features. In other words, filter streams are streams that filter byte input or output streams for some purpose.
- For example, the basic byte input stream provides a read method that can be used only for reading bytes. If we want to read integers, doubles, or strings, we require a filter class that could wrap the byte input stream.
- Using a filter class, we can read integers, doubles, and strings instead of bytes and characters.

### Types of Filter Streams in Java

There are basically two types of filter streams in Java for filtering data. They are:

- FilterInputStream
- FilterOutputStream

- FilterInputStream and FilterOutputStream are the base classes for filtering data.
- If you want to process primitive numeric types, use *DataInputStream* and *DataOutputStream* to filter bytes.

### FilterInputStream in Java

- FilterInputStream in Java is a concrete subclass class of InputStream class that filters data of an underlying stream.
- It implements Closeable and AutoCloseable interfaces.
- FilterInputStream class has four important subclasses that are as follows:
    - BufferedInputStream
    - DataInputStream
    - LineNumberInputStream
    - PushbackInputStream

The general syntax for declaration of FilterInputStream class is given below:

```
public class FilterInputStream extends InputStream
                implements Closeable, AutoCloseable
```

- FilterInputStream was added in Java 1.0 version.
- It is present in java.io.FilterInputStream package.

## Constructor of FilterInputStream class in Java

FilterInputStream class provides a single protected constructor that specifies the underlying stream from which filter stream reads data.

**FilterInputStream(InputStream in):**

- This constructor creates a FilterInputStream built on top of the underlying InputStream in.
- Since the constructor provided by filter input stream is protected, we cannot use FilterInputStream class directly.
- We need to create an instance of subclasses of FilterInputStream class to filter data.

### FilterInputStream Methods in Java

FilterInputStream class does not define any new methods. All the methods available in filter input stream are inherited from its superclass InputStream.

Some important inherited methods are as follows:

1. **int available():** This method returns an estimated number of bytes that can be read from the input stream.
2. **int read():** This method is used to read the next byte of data from the input stream.
3. **int read(byte[ ] b):** This method reads up to byte.length bytes of data from the input stream.
4. **long skip(long n):** It skips over and discards n bytes of data from the input stream.
5. **boolean markSupported():** This method evaluates if the input stream support mark and reset method.
6. **void mark(int readlimit):** This method marks the current position in the input stream.
7. **void reset():** This method is used to reset the input stream.
8. **void close():** This method is used to close the underlying input stream.

All the above methods will throw an exception named IOException when an error occurs.

### Example Program based on FilterInputStream

Let's take an example program based on the filter input stream.

**Example 1:**

```
package javaProgram;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FilterInputStream;
import java.io.IOException;
public class BufferedOutputStreamEx {
```

```
public static void main(String[ ] args) throws IOException
{
  String filepath = "D:\\myfile.txt";
  FileInputStream  fis = new FileInputStream(filepath);
  FilterInputStream filter = new BufferedInputStream(fis);
  int i = 0;
  while((i = filter.read()) != -1){
    System.out.print((char)i);
  }
  filter.close();
 }
}
```

Output:

```
Welcome to Scientech Easy
```

Here, we are supposing that you have the following data in “myfile.txt” file: Welcome to Scientech Easy.

Let’s create a Java program to get the actual number of available bytes in the filter input stream.

**Example 2:**

```
package javaProgram;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FilterInputStream;
import java.io.IOException;
public class BufferedOutputStreamEx {
public static void main(String[ ] args) throws IOException
{
 String filepath = "D:\\myfile.txt";
 FileInputStream  fis = new FileInputStream(filepath);
 FilterInputStream filter = new BufferedInputStream(fis);

 int availableBytes = filter.available();
 System.out.println("Initially, Available bytes: " +availableBytes);

 filter.read();
 filter.read();
 filter.read();

 int available = filter.available();
```

```
  System.out.println("Available bytes after reading three bytes: " +available);
  filter.close();
 }
}

Output:
       Initially, Available bytes: 25
       Available bytes after reading three bytes: 22
```

## FilterOutputStream in Java

**Java.io.FilterOutputStream** class is the superclass of all those classes which filters output streams. The write() method of FilterOutputStream Class filters the data and write it to the underlying stream, filtering which is done depending on the Streams.

**Declaration :**

```
public class FilterOutputStream
     extends OutputStream
```

**Constructors :**

- **FilterOutputStream(OutputStream geekout) :** Creates an output stream filter.

**Methods:**

- **write(int arg) : java.io.FilterOutputStream.write(int arg)** writes specified byte to the Outpu stream.

**Syntax :**

```
public void write(int arg)
```

**Parameters :**
arg : Source Bytes
**Return :**
void
**Exception :**
In case any I/O error occurs.

**Implementation :**

```
// Java program illustrating the working of work(int arg) method

import java.io.*;
import java.lang.*;

public class NewClass
{
    public static void main(String[] args) throws IOException
    {
        // OutputStream, FileInputStream & FilterOutputStream initially null
        OutputStream geek_out = null;
        FilterOutputStream geek_filter = null;

        // FileInputStream used here
        FileInputStream geekinput = null;

        char c;
        int a;
        try
        {
            // create output streams
            geek_out = new FileOutputStream("GEEKS.txt");
            geek_filter = new FilterOutputStream(geek_out);

            // write(int arg) : Used to write 'M' in the file
            // - "ABC.txt"
            geek_filter.write(77);

            // Flushes the Output Stream
            geek_filter.flush();

            // Creating Input Stream
            geekinput = new FileInputStream("GEEKS.txt");

            // read() method of FileInputStream :
            // reading the bytes and converting next bytes to int
            a = geekinput.read();

            /* Since, read() converts bytes to int, so we
               convert int to char for our program output*/
            c = (char)a;
            // print character
            System.out.println("Character written by" +
                               " FilterOutputStream : " + c);
         }
        catch(IOException except)
        {
            // if any I/O error occurs
            System.out.print("Write Not working properly");
        }
        finally{

            // releases any system resources associated with
            // the stream
            if (geek_out != null)
                geek_out.close();
            if (geek_filter != null)
                geek_filter.close();
        }
    }
}
```

## Random Access File Class 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)**<br>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)**<br>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 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 is written to the disk synchronously.
- **rws** – Creates the file with read and write mode - synchronously. All updates to file content or meta data is written to the disk synchronously.

## RandomAccessFile methods

The RandomAccessFile class in java has the following methods.

| S.No. | Methods with Description |
|---|---|
| 1 | **int read()**<br>It reads byte of data from a file. The byte is returned as an integer in the range 0-255. |
| 2 | **int read(byte[]b)**<br>It reads byte of data from file up to b.length, -1 if end of file is reached. |
| 3 | **int read(byte[]b, int offset, int len)**<br>It reads bytes initializing from off set position up to b.length from the buffer. |
| 4 | **boolean readBoolean()**<br>It reads a Boolean value from the file. |
| 5 | **byte readByte()**<br>It reads signed eight-bit value from file. |
| 6 | **char readChar()**<br>It reads a character value from file. |
| 7 | **double readDouble()**<br>It reads a double value from file. |
| 8 | **float readFloat()**<br>It reads a float value from file. |
| 9 | **long readLong()**<br>It reads a long value from file. |
| 10 | **int readInt()**<br>It reads a integer value from file. |

| | |
|---|---|
| 11 | **void readFully(byte[]b)**<br>It reads bytes initializing from offset position upto b.length from the buffer. |
| 12 | **void readFully(byte[]b, int offset, int len)**<br>It reads bytes initializing from offset position upto b.length from the buffer. |
| 13 | **String readUTF()**<br>It reads in a string from the file. |
| 14 | **void seek(long pos)**<br>It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs. |
| 15 | **long length()**<br>It returns the length of the file. |
| 16 | **void write(int b)**<br>It writes the specified byte to the file from the current cursor position. |
| 17 | **void writeFloat(float v)**<br>It converts the float argument to an int using the floatToInt Bits 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)**<br>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:\\IIIT\\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("Useofread()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("UseofreadBoolean():"+f_ref.readBoolean());
//readByte()method:
System.out.println("UseofreadByte():"+f_ref.readByte());


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


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


f_ref.seek(0);
f_ref.writeDouble(d);
f_ref.seek(0);
//readdouble
System.out.println("UseofreadDouble():"+f_ref.readDouble());
f_ref.seek(0);
```

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

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

                f_ref.seek(0);
                // Createarrayupto geek.length
                byte[]arr=newbyte[(int)f_ref.length()];
                //        readFully()         :
                f_ref.readFully(arr);
               String str1 = new String(arr);
               System.out.println("UseofreadFully():"+str1);

               f_ref.seek(0);

               //readFully(byte[]b,intoff,intlen):
               f_ref.readFully(arr, 0, 8);
               Stringstr2=newString(arr);
               System.out.println("UseofreadFully(byte[]b,intoff,intlen):"+str2);
           }
       catch(IOExceptionex)
       {
          System.out.println("SomethingwentWrong");
          ex.printStackTrace();
       }
    }
}
```

# DEPARTMENT OF INFORMATION TECHNOLOGY

# Java Programming (R22CSI2215)

## UNIT – III

**Packages:** Defining a Package, CLASSPATH, Access Specifiers, importing packages. Few Utility Classes - String Tokenizer, BitSet, Date, Calendar, Random, Formatter, Scanner.

**Collections:** Collections overview, Collection Interfaces, Collections Implementation Classes, Sorting in Collections, Comparable and Comparator Interfaces.

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

## 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.

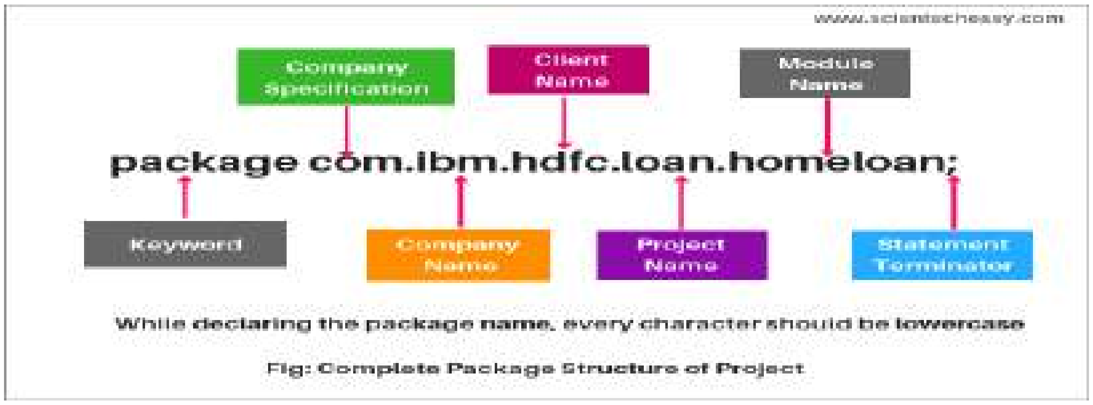
For example:

```
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.


www.scientecheasy.com
Company Specification
Client Name
Module Name
package com.ibm.hdfc.loan.homeloan;
Keyword
Company Name
Project Name
Statement Terminator
While declaring the package name, every character should be lowercase
Fig: Complete Package Structure of 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:

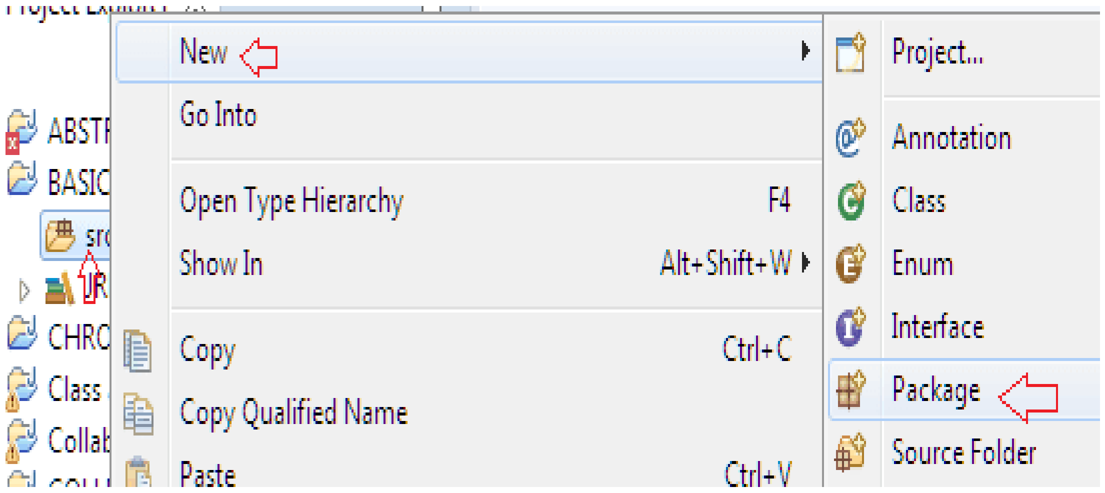
```
package   com.tcs.icici.loan.carloan.penalty;
```

**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 Microsystem. 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:

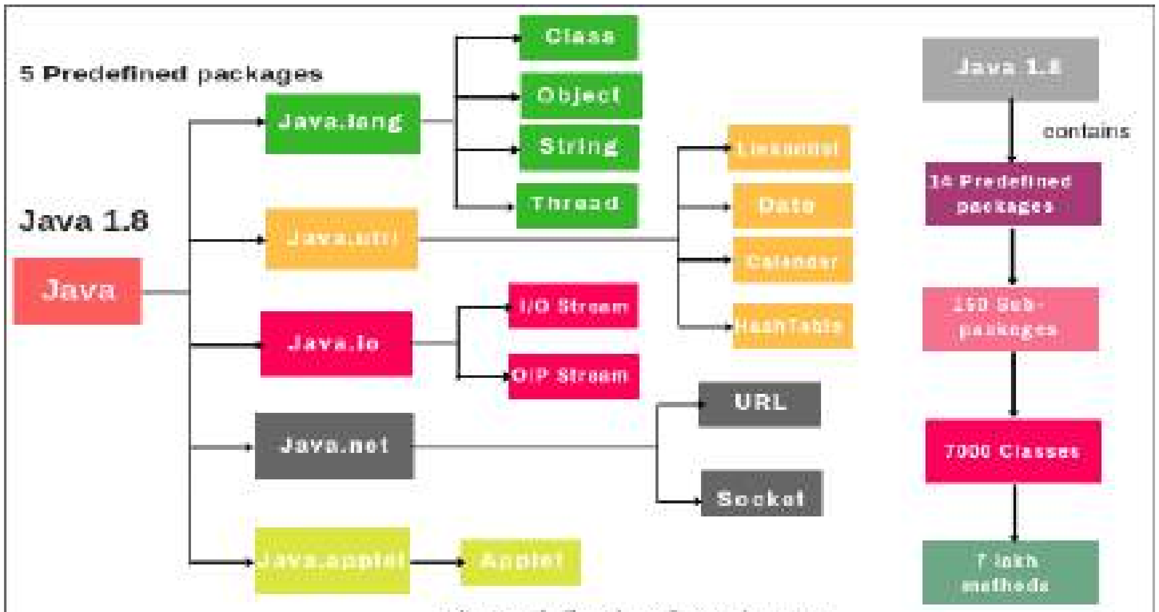


Fig: Predefined packages in Java

## 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, Calender, 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.

**Example:**

1. **Create the Package:**

First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names.**

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

2. **Use the Class in Program:**

Now we will use the **MyClass** class in our program.

```
// import 'MyClass' class from 'names' myPackage
import myPackage.MyClass;

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

      // Initializing the String variable
      // with a value
      String s = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass o = new MyClass();

      o.getNames(s);
   }
}
```

**Note:** MyClass.java must be saved inside the myPackage directory since it is a part of the package.

**Static Import In Java**

Static Import in Java is about simplifying access to static members and separates it from the broader discussion of user-defined packages.
Static import is a feature introduced in Java programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.

**Example:**

```
// Note static keyword after import.
import static java.lang.System.*;

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

      // We don't need to use 'System.out'
      // as imported using static.
      out.println("GeeksforGeeks");
   }
}
```

**Output**

```
GeeksforGeeks
```

## Directory Structure and CLASSPATH

Package names correspond to a directory structure. For example, a class Circle in package com.zzz.project1.subproject2 is stored as:

*$BASE_DIR/com/zzz/project1/subproject2/Circle.class*

- Here **$BASE_DIR** represents the base directory of the package.
- The "dot" in the package name corresponds to a sub-directory of the file system.
- The base directory (**$BASE_DIR**) could be located anywhere in the file system.
- Hence, the Java compiler and runtime must be informed about the location of the $BASE_DIR so as to locate the classes.
- It is is accomplished by an environment variable called **CLASSPATH**.
- CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

**Setting CLASSPATH**

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment the steps In Windows is

*Go to **Control Panel** -> **System** -> **Advanced** -> **Environment Variables**.*

- Select **"System Variables"** to apply the CLASSPATH for all users on the system.

- Select **"User Variables"** to apply it only for the currently logged-in user.
- **Edit or Create CLASSPATH :** If CLASSPATH already exists, select it and click **"Edit" or** If it doesn't exist, click **"New"**
- **Enter CLASSPATH Details**: In the **"Variable name"** field, enter: **"CLASSPATH"**, In the **"Variable value"** field, enter the directories and JAR files separated by semicolons.
- In the **"Variable value"** field, enter the directories and JAR files separated by semicolons.
- **Example:**
  *.c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar*

- The dot (.) represents the current working directory.
- To check the current setting of the CLASSPATH, issue the following command:

*> SET CLASSPATH*

CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:

*> SET  CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar*

Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,

*> java –classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3*

**Illustration of user-defined packages:** Creating our first package: File name – ClassOne.java

```
package package_name;

public class ClassOne {
   public void methodClassOne()
   {
      System.out.println("Hello there its ClassOne");
   }
}
```

Creating our second package: File name – ClassTwo.java

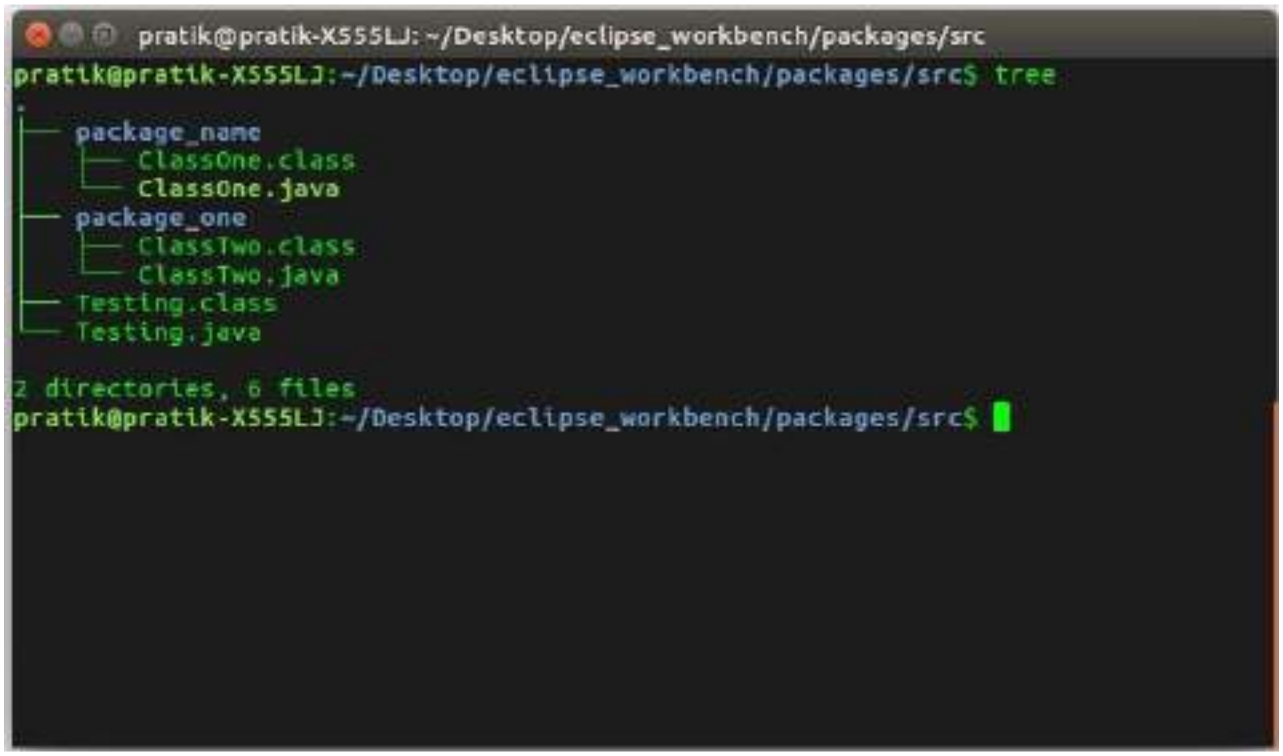
```
package package_one;

public class ClassTwo {
   public void methodClassTwo()
   {
      System.out.println("Hello there i am ClassTwo");
   }
}
```

Making use of both the created packages: File name – Testing.java

```
import package_name.ClassOne;
import package_one.ClassTwo;
```

```
public class Testing {
    public static void main(String[] args)
    {
        ClassTwo a = new ClassTwo();
        ClassOne b = new ClassOne();
        a.methodClassTwo();
        b.methodClassOne();
    }
}
```

Now having a look at the directory structure of both the packages and the testing class file:



```
pratik@pratik-X555LJ: ~/Desktop/eclipse_workbench/packages/src
pratik@pratik-X555LJ:~/Desktop/eclipse_workbench/packages/src$ tree
.
├── package_name
│   ├── ClassOne.class
│   └── ClassOne.java
├── package_one
│   ├── ClassTwo.class
│   └── ClassTwo.java
├── Testing.class
└── Testing.java

2 directories, 6 files
pratik@pratik-X555LJ:~/Desktop/eclipse_workbench/packages/src$
```

**Access Modifiers in the Context of Packages**

1. **Public:** Members with the public modifier are accessible from anywhere, regardless of whether the accessing class is in the same package or not**.**
2. **Protected:** Memebers with the protected modifier are accessible within the same package, In subclasses
3. **Default:** Members with no modifier are accessible only within the same package
4. **Private:** Members with the private modifier are accessible only within the same class. They cannot be accessed by classes in the same package, subclasses, or different packages.

**Important points:**

- Every class is part of some package.
- If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).

- All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
- If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
- We can access public classes in another (named) package using: **package-name.class-name**

## 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.

## Few Utility Classes in Java

A utility class in Java is a class that provides common utility methods that are not tied to a specific object or instance. These classes often contain static methods meaning you can call them directly on the class without creating an instance of the class.

**Here are some frequently used utility classes in Java and examples**

- java.util.Arrays
- java.util.Collections
- java.util.Date
- java.util.StringTokenizer
- java.util.Random
- java.util.regex.Pattern
- java.util.Scanner
- java.util.Calendar

## 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.

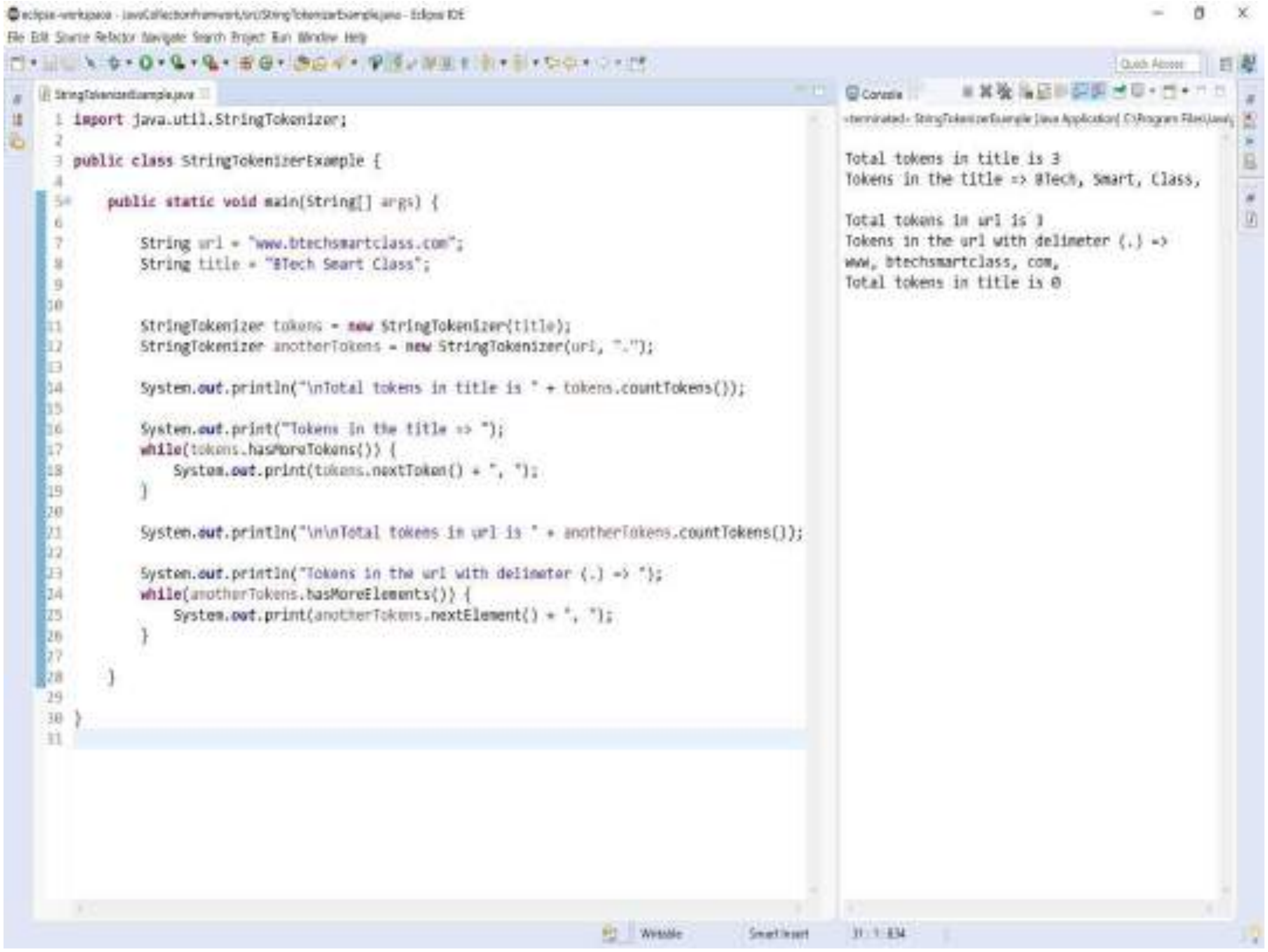
| S. No. | Constructor with Description |
|---|---|
| 1 | **StringTokenizer(String str)**<br><br>It creates StringTokenizer object for the specified string str with default delimeter. |
| 2 | **StringTokenizer(String str, String delimeter)**<br>It creates StringTokenizer object for the specified string str with specified delimeter. |
| 3 | **StringTokenizer(String str, String delimeter, boolean returnValue)**<br>It creates StringTokenizer object with specified string, delimeter and returnValue. |

The StringTokenizer class in java has the following methods.

| S.No. | Methods with Description |
|---|---|
| 1 | **String nextToken()**<br><br>It returns the next token from the StringTokenizer object. |
| 2 | **String nextToken(String delimeter)**<br>It returns the next token from the StringTokenizer object based on the delimeter. |
| 3 | **Object nextElement()**<br>It returns the next token from the StringTokenizer object. |
| 4 | **boolean hasMoreTokens()**<br>It returns true if there are more tokens in the StringTokenizer object. otherwise returns false. |
| 5 | **boolean hasMoreElements()**<br>It returns true if there are more tokens in the StringTokenizer object. otherwise returns false. |
| 6 | **int countTokens()**<br>It returns total number of tokens in the StringTokenizer object. |

Let's consider an example program to illustrate methods of StringTokenizer class.

**Example :**



```java
import java.util.StringTokenizer;

public class StringTokenizerExample {

    public static void main(String[] args) {

        String url = "www.btechsmartclass.com";
        String title = "BTech Smart Class";


        StringTokenizer tokens = new StringTokenizer(title);
        StringTokenizer anotherTokens = new StringTokenizer(url, ".");

        System.out.println("\nTotal tokens in title is " + tokens.countTokens());

        System.out.print("Tokens in the title => ");
        while(tokens.hasMoreTokens()) {
            System.out.print(tokens.nextToken() + ", ");
        }

        System.out.println("\n\nTotal tokens in url is " + anotherTokens.countTokens());

        System.out.println("Tokens in the url with delimeter (.) => ");
        while(anotherTokens.hasMoreElements()) {
            System.out.print(anotherTokens.nextElement() + ", ");
        }

    }

}
```

```
<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 delimeter (.) =>
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 |
|---|---|
| 1 | **BitSet( )**<br><br>It creates a default BitSet |
| **2** | **BitSet(int noOfBits)**<br>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.

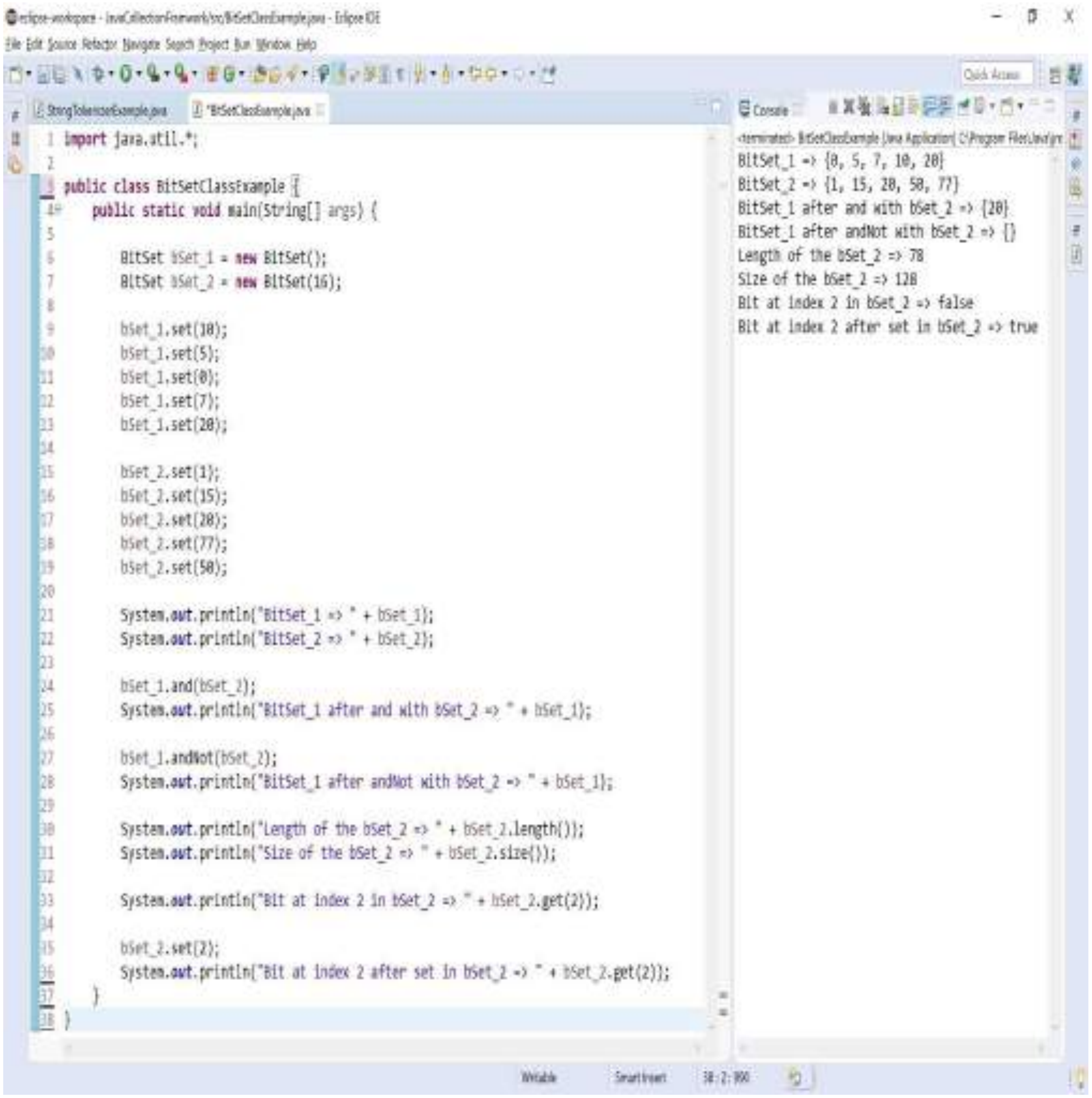
| S.No. | Methods with Description |
|---|---|
| 1 | **void and(BitSet bitSet)**<br>It performs AND operation on the contents of the invoking BitSet object with those specified by bitSet. |
| 2 | **void andNot(BitSet bitSet)**<br>For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared. |
| 3 | **void flip(int index)**<br>It reverses the bit at specified index. |
| 4 | **void flip(int startIndex, int endIndex)**<br>It reverses all the bits from specified startIndex to endIndex. |
| 5 | **void or(BitSet bitSet)**<br>It performs OR operation on the contents of the invoking BitSet object with those specified by bitSet. |
| 6 | **void xor(BitSet bitSet)**<br>It performs XOR operation on the contents of the invoking BitSet object with those specified by bitSet. |
| 7 | **int cardinality( )**<br>It returns the number of bits set to true in the invoking BitSet. |
| 8 | **void clear( )**<br>It sets all the bits to zeros of the invoking BitSet. |
| 9 | **void clear(int index)**<br>It set the bit specified by given index to zero. |
| | |

| | |
|---|---|
| 10 | **void clear(int startIndex, int endIndex)**<br>It sets all the bits from specified startIndex to endIndex to zero. |
| 11 | **Object clone( )**<br>It duplicates the invoking BitSet object. |
| 12 | **boolean equals(Object bitSet)**<br>It retruns true if both invoking and argumented BitSets are equal, otherwise returns false. |
| 13 | **boolean get(int index)**<br>It retruns the present state of the bit at given index in the invoking BitSet. |
| 14 | **BitSet get(int startIndex, int endIndex)**<br>It returns a BitSet object that consists all the bits from startIndex to endIndex. |
| 15 | **int hashCode( )**<br>It returns the hash code of the invoking BitSet. |
| 16 | **boolean intersects(BitSet bitSet)**<br>It returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1. |
| 17 | **boolean isEmpty( )**<br>It returns true if all bits in the invoking object are zero, otherwise returns false. |
| 17 | **int length( )**<br>It returns the total number of bits in the invoking BitSet. |
| 18 | **int nextClearBit(int startIndex)**<br>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)**<br>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)**<br>It sets the bit specified by index. |
| 21 | **void set(int index, boolean value)**<br>It sets the bit specified by index to the value passed in value. |
| 22 | **void set(int startIndex, int endIndex)**<br>It sets all the bits from startIndex to endIndex. |

| | |
|---|---|
| 23 | **void set(int startIndex, int endIndex, boolean value)**<br>It sets all the bits to specified value from startIndex to endIndex. |
| 24 | **int size( )**<br>It returns the total number of bits in the invoking BitSet. |
| 25 | **String toString( )**<br>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
StringTokenizerExample.java
*BitSetClassExample.java
Console
Quick Access
Writable
Smart Insert
38 : 2 : 900

```java
import java.util.*;

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

        BitSet bSet_1 = new BitSet();
        BitSet bSet_2 = new BitSet(16);

        bSet_1.set(10);
        bSet_1.set(5);
        bSet_1.set(0);
        bSet_1.set(7);
        bSet_1.set(20);

        bSet_2.set(1);
        bSet_2.set(15);
        bSet_2.set(20);
        bSet_2.set(77);
        bSet_2.set(50);

        System.out.println("BitSet_1 => " + bSet_1);
        System.out.println("BitSet_2 => " + bSet_2);

        bSet_1.and(bSet_2);
        System.out.println("BitSet_1 after and with bSet_2 => " + bSet_1);

        bSet_1.andNot(bSet_2);
        System.out.println("BitSet_1 after andNot with bSet_2 => " + bSet_1);

        System.out.println("Length of the bSet_2 => " + bSet_2.length());
        System.out.println("Size of the bSet_2 => " + bSet_2.size());

        System.out.println("Bit at index 2 in bSet_2 => " + bSet_2.get(2));

        bSet_2.set(2);
        System.out.println("Bit at index 2 after set in bSet_2 => " + bSet_2.get(2));
    }
}
```

```
<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
```

## 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.

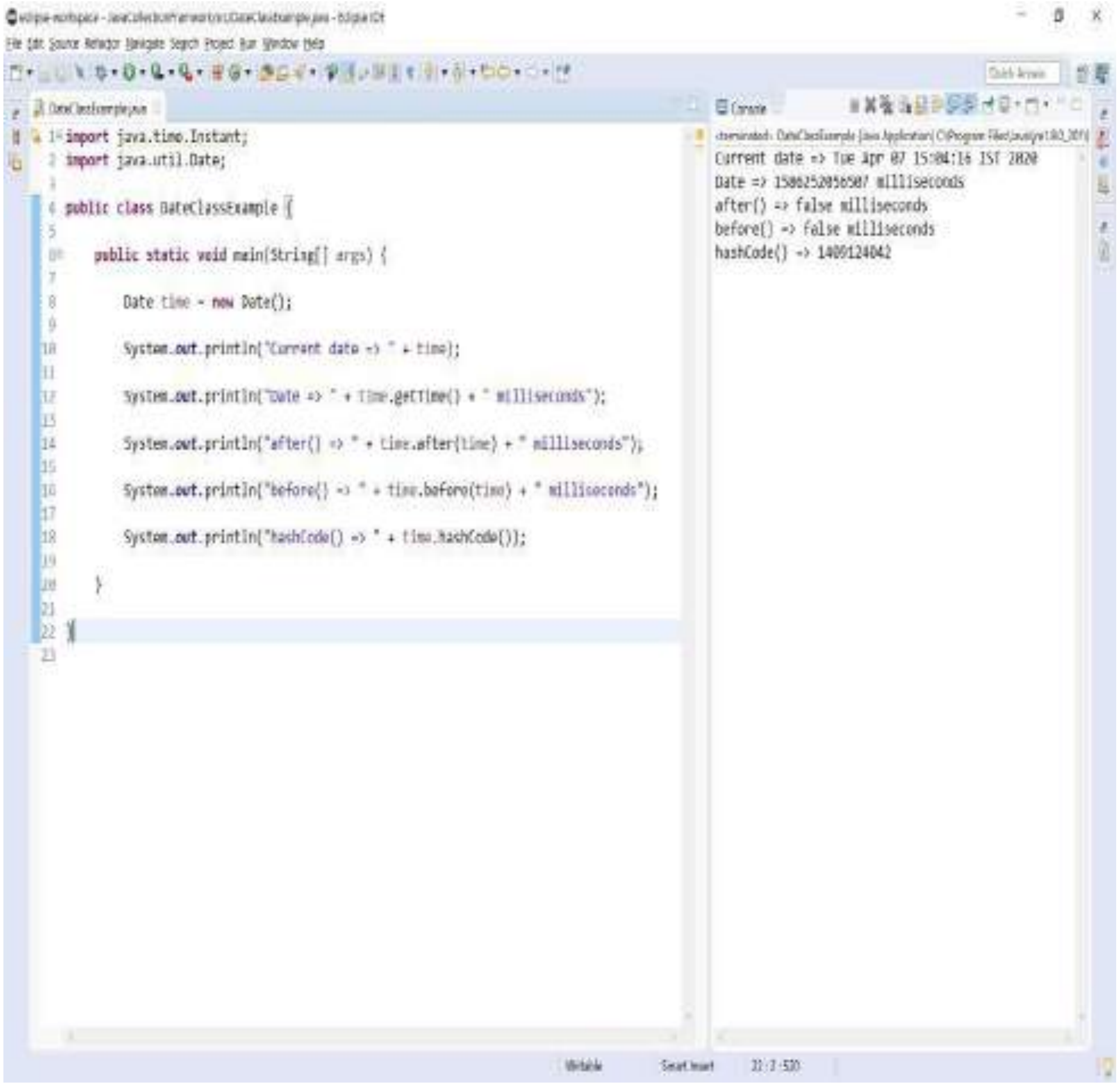
The Date class in java has the following constructor.

| S. No. | Constructor with Description |
|---|---|
| 1 | **Date( )**<br><br>It creates a Date object that represents current date and time. |
| 2 | **Date(long milliseconds)**<br>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) -** Depricated<br>It creates a date object with the specified year, month, and date. |
| 4 | **Date(int year, int month, int date, int hrs, int min) -** Depricated<br>It creates a date object with the specified year, month, date, hours, and minuts. |
| 5 | **Date(int year, int month, int date, int hrs, int min, int sec) -** Depricated<br>It creates a date object with the specified year, month, date, hours, minuts and seconds. |
| 6 | **Date(String s) -** Depricated<br>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()**<br>It returns the time represented by this date object. |
| 2 | **boolean after(Date date)**<br>It returns true, if the invoking date is after the argumented date. |
| 3 | **boolean before(Date date)**<br>It returns true, if the invoking date is before the argumented date. |
| 4 | **Date from(Instant instant)**<br>It returns an instance of Date object from Instant date. |
| 5 | **void setTime(long time)**<br>It changes the current date and time to given time. |
| 6 | **Object clone( )**<br>It duplicates the invoking Date object. |
| 7 | **int compareTo(Date date)**<br>It compares current date with given date. |
| 8 | **boolean equals(Date date)**<br>It compares current date with given date for equality. |
| 9 | **int hashCode()**<br>It returns the hash code value of the invoking date object. |
| 10 | **Instant toInstant()**<br>It converts current date into Instant object. |
| 11 | **String toString()**<br>It converts this date into Instant object. |

Let's consider an example program to illustrate methods of Date class.



```java
import java.time.Instant;
import java.util.Date;

public class DateClassExample {

    public static void main(String[] args) {

        Date time = new Date();

        System.out.println("Current date => " + time);

        System.out.println("Date => " + time.getTime() + " milliseconds");

        System.out.println("after() => " + time.after(time) + " milliseconds");

        System.out.println("before() => " + time.before(time) + " milliseconds");

        System.out.println("hashCode() => " + time.hashCode());

    }

}
```

```
<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()**<br><br>It returns a calendar using the default time zone and locale. |
| 2 | **Date getTime()**<br>It returns a Date object representing the invoking Calendar's time value. |
| 3 | **TimeZone getTimeZone()**<br>It returns the time zone object associated with the invoking calendar. |
| 4 | **String getCalendarType()**<br>It returns an instance of Date object from Instant date. |
| 5 | **int get(int field)**<br>It rerturns the value for the given calendar field. |
| 6 | **int getFirstDayOfWeek()**<br>It returns the day of the week in integer form. |
| 7 | **int getWeeksInWeekYear()**<br>It retruns the total weeks in week year. |
| 8 | **int getWeekYear()**<br>It returns the week year represented by current Calendar. |
| 9 | **void add(int field, int amount)**<br>It adds the specified amount of time to the given calendar field. |
| 10 | **boolean after (Object when)**<br>It returns true if the time represented by the Calendar is after the time represented by when Object. |

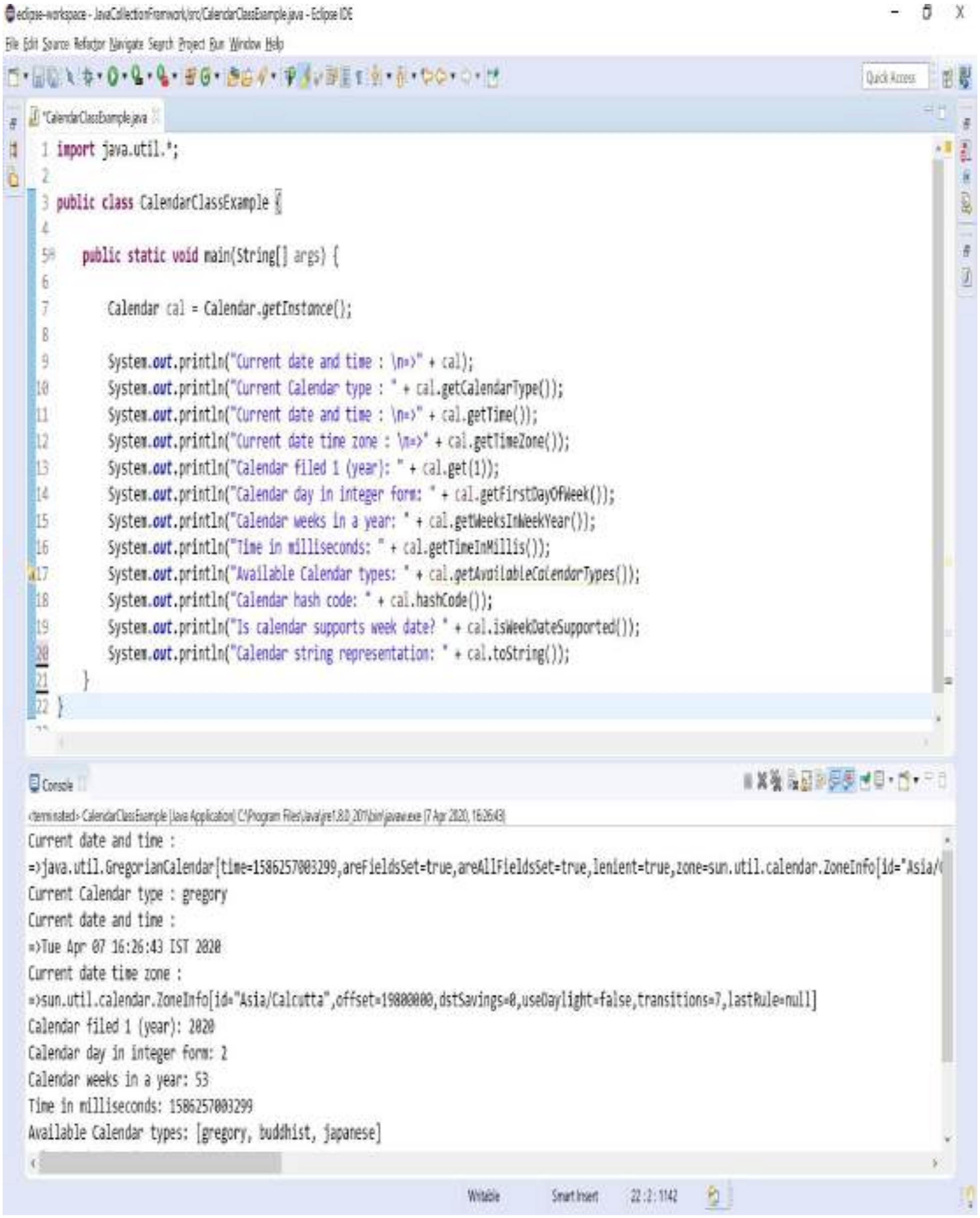
| S.No. | Methods with Description |
|---|---|
| 11 | **boolean before(Object when)**<br>It returns true if the time represented by the Calendar is before the time represented by when Object. |
| 12 | **void clear(int field)**<br>It sets the given calendar field value and the time value of this Calendar undefined. |
| 13 | **Object clone()**<br>It retruns the copy of the current object. |
| 14 | **int compareTo(Calendar anotherCalendar)**<br>It compares and retruns the time values (millisecond offsets) between two calendar object. |
| 15 | **void complete()**<br>It sets any unset fields in the calendar fields. |
| 16 | **void computeFields()**<br>It converts the current millisecond time value time to calendar field values in fields[]. |
| 17 | **void computeTime()**<br>It converts the current calendar field values in fields[] to the millisecond time value time. |
| 18 | **boolean equals(Object object)**<br>It returns true if both invoking object and argumented object are equal. |
| 19 | **int getActualMaximum(int field)**<br>It returns the Maximum possible value of the specified calendar field. |
| 20 | **int getActualMinimum(int field)**<br>It returns the Minimum possible value of the specified calendar field. |
| 21 | **Set getAvailableCalendarTypes()**<br>It returns a string set of all available calendar type supported by Java Runtime Environment. |
| 22 | **Locale[] getAvailableLocales()**<br>It returns an array of all locales available in java runtime environment. |
| 23 | **String getDisplayName(int field, int style, Locale locale)**<br>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)**<br>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)**<br>It returns the highest minimum value of the specified Calendar field. |
| 26 | **int getLeastMaximum(int field)**<br>It returns the highest maximum value of the specified Calendar field. |
| 27 | **int getMaximum(int field)**<br>It returns the maximum value of the specified calendar field. |
| 28 | **int getMinimalDaysInFirstWeek()**<br>It returns required minimum days in integer form. |
| 29 | **int getMinimum(int field)**<br>It returns the minimum value of specified calendar field. |
| 30 | **long getTimeInMillis()**<br>It returns the current time in millisecond. |
| 31 | **int hashCode()**<br>It returns the hash code of the invoking object. |
| 32 | **int internalGet(int field)**<br>It returns the value of the given calendar field. |
| 33 | **boolean isLenient()**<br>It returns true if the interpretation mode of this calendar is lenient; false otherwise. |
| 34 | **boolean isSet(int field)**<br>If not set then it returns false otherwise true. |
| 35 | **boolean isWeekDateSupported()**<br>It returns true if the calendar supports week date. The default value is false. |
| 36 | **void roll(int field, boolean up)**<br>It increase or decrease the specified calendar field by one unit without affecting the other field |
| 37 | **void set(int field, int value)**<br>It sets the specified calendar field by the specified value. |
| 38 | **void setFirstDayOfWeek(int value)**<br>It sets the first day of the week. |

| S.No. | Methods with Description |
|---|---|
| 39 | **void setMinimalDaysInFirstWeek(int value)**<br>It sets the minimal days required in the first week. |
| 40 | **void setTime(Date date)**<br>It sets the Time of current calendar object. |
| 41 | **void setTimeInMillis(long millis)**<br>It sets the current time in millisecond. |
| 42 | **void setTimeZone(TimeZone value)**<br>It sets the TimeZone with passed TimeZone value. |
| 43 | **void setWeekDate(int weekYear, int weekOfYear, int dayOfWeek)**<br>It sets the current date with specified integer value. |
| 44 | **Instant toInstant()**<br>It converts the current object to an instant. |
| 45 | **String toString()**<br>It returns a string representation of the current object. |

Let's consider an example program to illustrate methods of Calendar class.

**Example**


eclipse-workspace - JavaCollectionFramework/src/CalendarClassExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
*CalendarClassExample.java
Console
Writable
Smart Insert
22 : 2 : 1142

```java
import java.util.*;

public class CalendarClassExample {

    public static void main(String[] args) {

        Calendar cal = Calendar.getInstance();

        System.out.println("Current date and time : \n=>" + cal);
        System.out.println("Current Calendar type : " + cal.getCalendarType());
        System.out.println("Current date and time : \n=>" + cal.getTime());
        System.out.println("Current date time zone : \n=>" + cal.getTimeZone());
        System.out.println("Calendar filed 1 (year): " + cal.get(1));
        System.out.println("Calendar day in integer form: " + cal.getFirstDayOfWeek());
        System.out.println("Calendar weeks in a year: " + cal.getWeeksInWeekYear());
        System.out.println("Time in milliseconds: " + cal.getTimeInMillis());
        System.out.println("Available Calendar types: " + cal.getAvailableCalendarTypes());
        System.out.println("Calendar hash code: " + cal.hashCode());
        System.out.println("Is calendar supports week date? " + cal.isWeekDateSupported());
        System.out.println("Calendar string representation: " + cal.toString());
    }
}
```

```
<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/
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()**<br><br>It creates a new random number generator. |
| 2 | **Random(long seedValue)**<br>It creates a new random number generator using a single long seedValue. |

The Random class in java has the following methods.

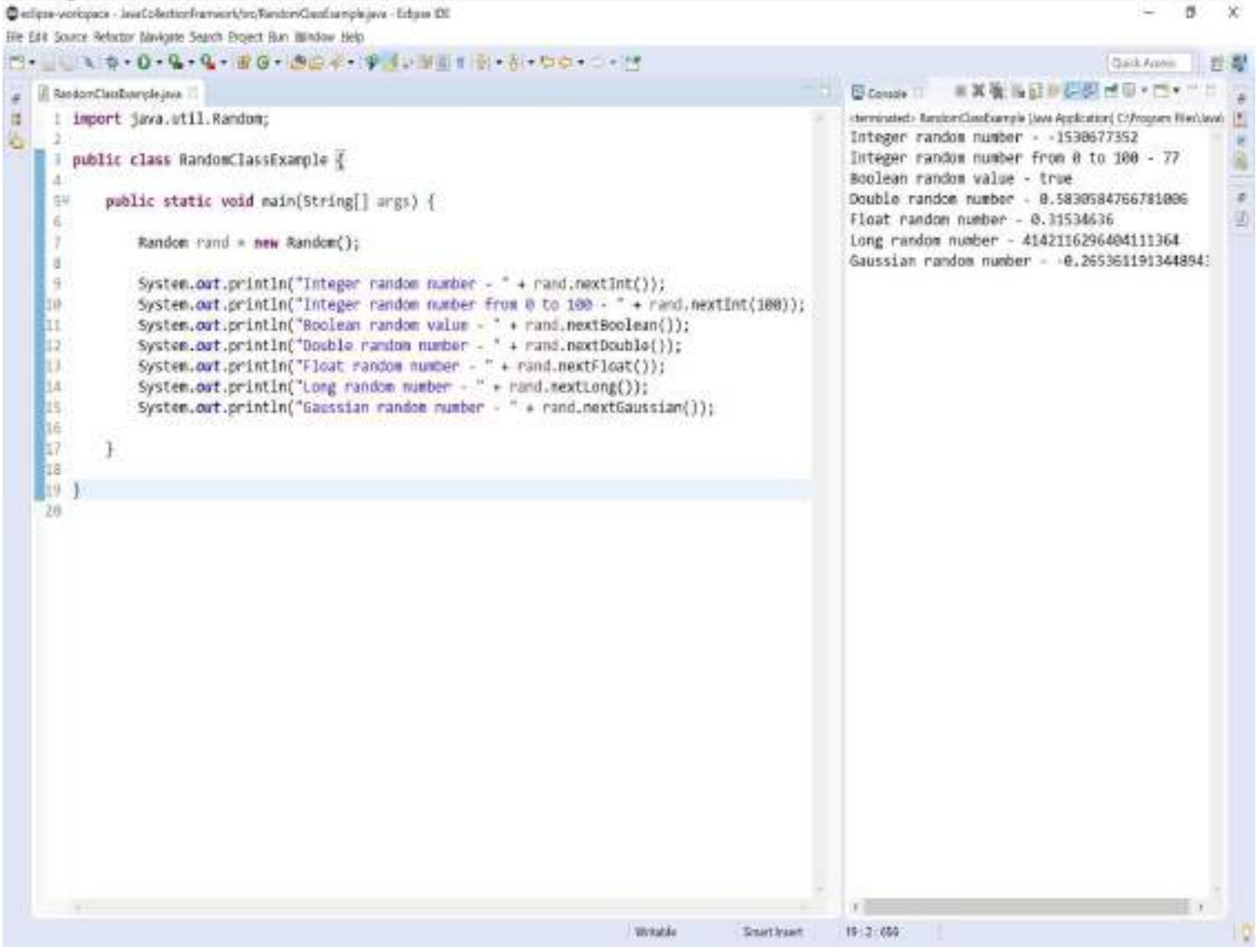
| S.No. | Methods with Description |
|---|---|
| 1 | **int next(int bits)**<br><br>It generates the next pseudo-random number. |
| 2 | **Boolean nextBoolean()**<br>It generates the next uniformly distributed pseudo-random boolean value. |
| 3 | **double nextDouble()**<br>It generates the next pseudo-random double number between 0.0 and 1.0. |
| 4 | **void nextBytes(byte[] bytes)**<br>It places the generated random bytes into an user-supplied byte array. |
| 5 | **float nextFloat()**<br>It generates the next pseudo-random float number between 0.0 and 1.0.. |

| S.No. | Methods with Description |
|---|---|
| 6 | **int nextInt()**<br>It generates the next pseudo-random int number. |
| 7 | **int nextInt(int n)**<br>It generates the next pseudo-random integer value between zero and n. |
| 8 | **long nextLong()**<br>It generates the next pseudo-random, uniformly distributed long value. |
| 9 | **double nextGaussian()**<br>It generates the next pseudo-random Gaussian distributed double number with mean 0.0 and standard deviation 1.0. |
| 10 | **void setSeed(long seedValue)**<br>It sets the seed of the random number generator using a single long seedValue. |
| 11 | **DoubleStream doubles()**<br>It returns a stream of pseudo-random double values, each conforming between 0.0 and 1.0. |
| 12 | **DoubleStream doubles(double start, double end)**<br>It retruns an unlimited stream of pseudo-random double values, each conforming to the given start and end. |
| 13 | **DoubleStream doubles(long streamSize)**<br>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)**<br>It returns a stream producing the given streamSizenumber of pseudo-random double values, each conforming to the given start and end. |
| 15 | **IntStream ints()**<br>It returns a stream of pseudo-random integer values. |
| 16 | **IntStream ints(int start, int end)**<br>It retruns an unlimited stream of pseudo-random integer values, each conforming to the given start and end. |
| 17 | **IntStream ints(long streamSize)**<br>It returns a stream producing the pseudo-random integer values for the given streamSize number. |
| 18 | **IntStream ints(long streamSize, int start, int end)**<br>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()**<br>It returns a stream of pseudo-random long values. |
| 20 | **LongStream longs(long start, long end)**<br>It retruns an unlimited stream of pseudo-random long values, each conforming to the given start and end. |
| 21 | **LongStream longs(long streamSize)**<br>It returns a stream producing the pseudo-random long values for the given streamSize number. |
| 22 | **LongStream longs(long streamSize, long start, long end)**<br>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**



```java
import java.util.Random;

public class RandomClassExample {

    public static void main(String[] args) {

        Random rand = new Random();

        System.out.println("Integer random number - " + rand.nextInt());
        System.out.println("Integer random number from 0 to 100 - " + rand.nextInt(100));
        System.out.println("Boolean random value - " + rand.nextBoolean());
        System.out.println("Double random number - " + rand.nextDouble());
        System.out.println("Float random number - " + rand.nextFloat());
        System.out.println("Long random number - " + rand.nextLong());
        System.out.println("Gaussian random number - " + rand.nextGaussian());

    }

}
```

```
<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 - 4142116296404111364
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()**<br><br>It creates a new formatter. |
| 2 | **Formatter(Appendable a)**<br>It creates a new formatter with the specified destination. |
| 3 | **Formatter(Appendable a, Locale l)**<br>It creates a new formatter with the specified destination and locale. |
| 4 | **Formatter(File file)**<br>It creates a new formatter with the specified file. |
| 5 | **Formatter(File file, String charset)**<br>It creates a new formatter with the specified file and charset. |
| 6 | **Formatter(File file, String charset, Locale l)**<br>It creates a new formatter with the specified file, charset, and locale. |
| 7 | **Formatter(Locale l)**<br>It creates a new formatter with the specified locale. |
| 8 | **Formatter(OutputStream os)**<br>It creates a new formatter with the specified output stream. |
| 9 | **Formatter(OutputStream os, String charset)**<br>It creates a new formatter with the specified output stream and charset. |
| 10 | **Formatter(OutputStream os, String charset, Locale l)**<br>It creates a new formatter with the specified output stream, charset, and locale. |
| 11 | **Formatter(PrintStream ps)**<br>It creates a new formatter with the specified print stream. |

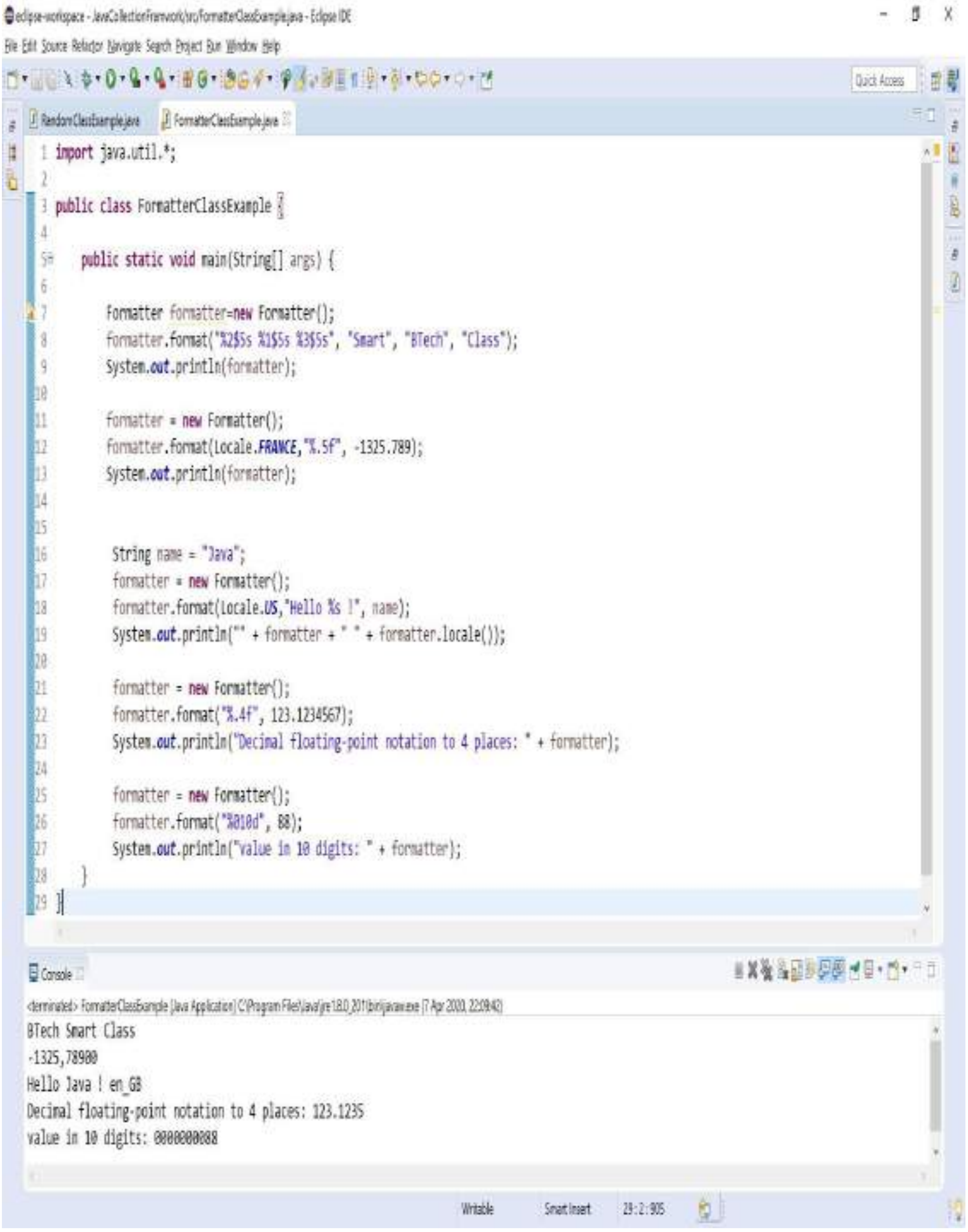
| S.No. | Constructor with Description |
|---|---|
| 12 | **Formatter(String fileName)**<br>It creates a new formatter with the specified file name. |
| 13 | **Formatter(String fileName, String charset)**<br>It creates a new formatter with the specified file name and charset. |
| 14 | **Formatter(String fileName, String charset, Locale l)**<br>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)**<br><br>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)**<br>It writes a formatted string to the invoking object's destination using the specified format string and arguments. |
| 3 | **void flush()**<br>It flushes the invoking formatter. |
| 4 | **Appendable out()**<br>It returns the destination for the output. |
| 5 | **Locale locale()**<br>It returns the locale set by the construction of the invoking formatter. |
| 6 | **String toString()**<br>It converts the invoking object to string. |
| 7 | **IOException ioException()**<br>It returns the IOException last thrown by the invoking formatter's Appendable. |
| 8 | **void close()**<br>It closes the invoking formatter. |

Let's consider an example program to illustrate methods of Formatter class.

**Example**



```
import java.util.*;

public class FormatterClassExample {

    public static void main(String[] args) {

        Formatter formatter=new Formatter();
        formatter.format("%2$5s %1$5s %3$5s", "Smart", "BTech", "Class");
        System.out.println(formatter);

        formatter = new Formatter();
        formatter.format(Locale.FRANCE,"%.5f", -1325.789);
        System.out.println(formatter);


         String name = "Java";
         formatter = new Formatter();
         formatter.format(Locale.US,"Hello %s !", name);
         System.out.println("" + formatter + " " + formatter.locale());

         formatter = new Formatter();
         formatter.format("%.4f", 123.1234567);
         System.out.println("Decimal floating-point notation to 4 places: " + formatter);

         formatter = new Formatter();
         formatter.format("%010d", 88);
         System.out.println("value in 10 digits: " + formatter);
    }
}
```

<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)**<br><br>It creates a new Scanner that produces values read from the specified input stream. |
| 2 | **Scanner(InputStream source, String charsetName)**<br>It creates a new Scanner that produces values read from the specified input stream. |
| 3 | **Scanner(File source)**<br>It creates a new Scanner that produces values scanned from the specified file. |
| 4 | **Scanner(File source, String charsetName)**<br>It creates a new Scanner that produces values scanned from the specified file. |
| 5 | **Scanner(String source)**<br>It creates a new Scanner that produces values scanned from the specified string. |
| 6 | **Scanner(Readable source)**<br>It creates a new Scanner that produces values scanned from the specified source. |
| 7 | **Scanner(ReadableByteChannel source)**<br>It creates a new Scanner that produces values scanned from the specified channel. |
| 8 | **Scanner(ReadableByteChannel source, String charsetName)**<br>It creates a new Scanner that produces values scanned from the specified channel. |

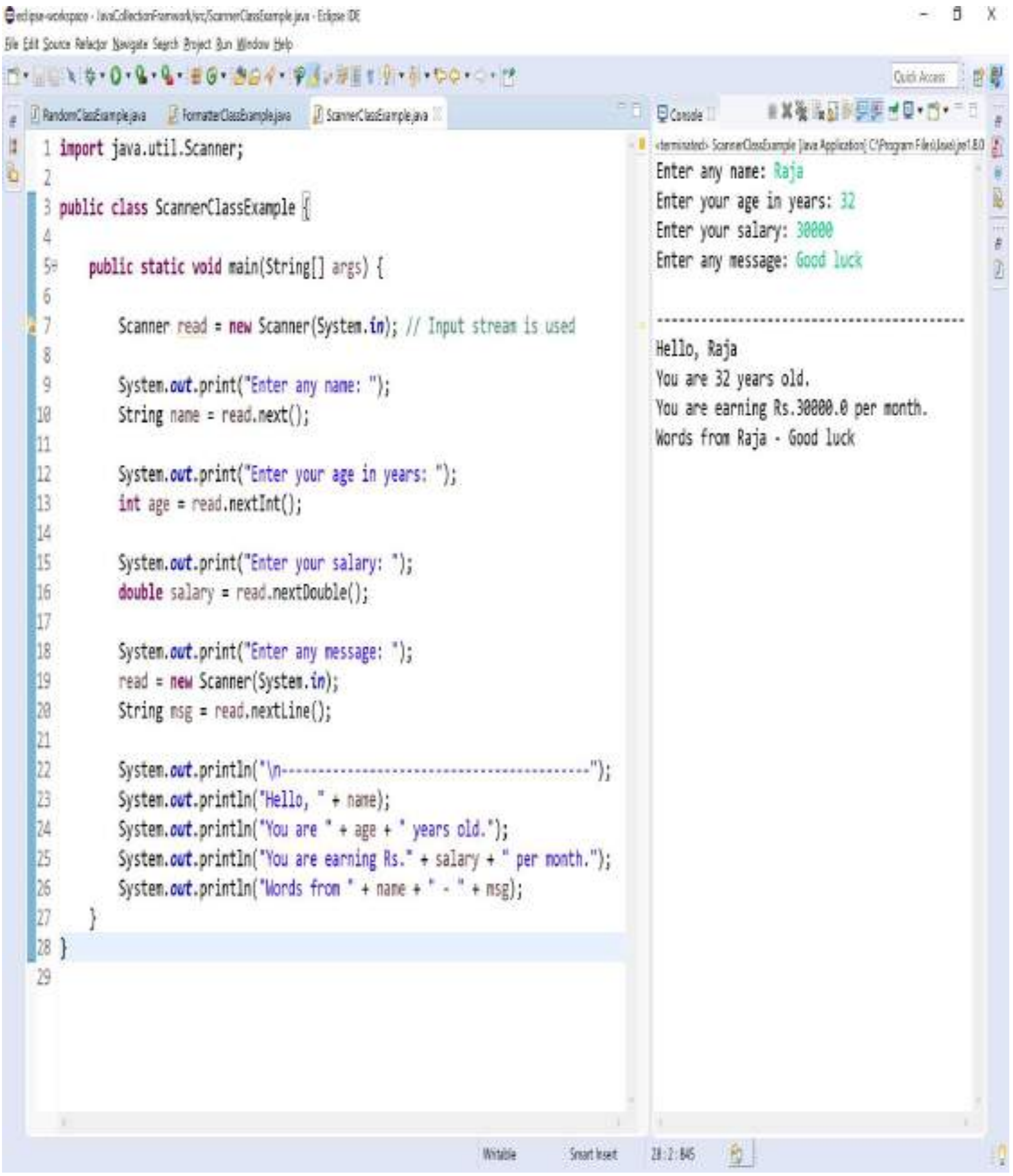
The Scanner class in java has the following methods.

| S.No | Methods with |
|---|---|
| 1 | **String**<br><br>It reads the next complete token from the invoking |

| S.No. | Methods with Description |
|---|---|
| 2 | **String next(Pattern pattern)**<br>It reads the next token if it matches the specified pattern. |
| 3 | **String next(String pattern)**<br>It reads the next token if it matches the pattern constructed from the specified string. |
| 4 | **boolean nextBoolean()**<br>It reads a boolean value from the user. |
| 5 | **byte nextByte()**<br>It reads a byte value from the user. |
| 6 | **double nextDouble()**<br>It reads a double value from the user. |
| 7 | **float nextFloat()**<br>It reads a floating-point value from the user. |
| 8 | **int nextInt()**<br>It reads an integer value from the user. |
| 9 | **long nextLong()**<br>It reads a long value from the user. |
| 10 | **short nextShort()**<br>It reads a short value from the user. |
| 11 | **String nextLine()**<br>It reads a string value from the user. |
| 12 | **boolean hasNext()**<br>It returns true if the invoking scanner has another token in its input. |
| 13 | **void remove()**<br>It is used when remove operation is not supported by this implementation of Iterator. |
| 14 | **void close()**<br>It closes the invoking scanner. |

Let's consider an example program to illustrate methods of Scanner class.

**Example**

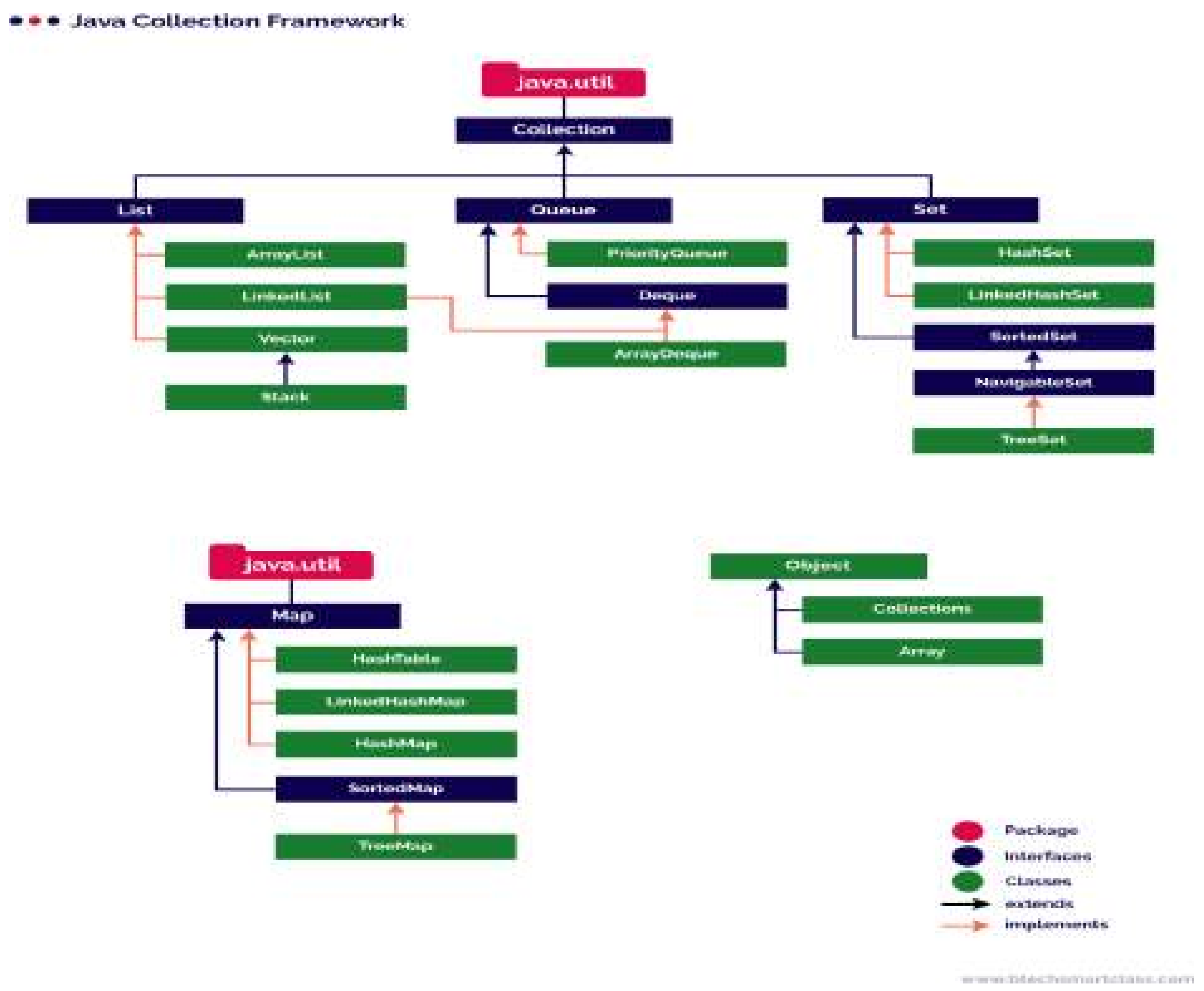


```java
import java.util.Scanner;

public class ScannerClassExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in); // Input stream is used

        System.out.print("Enter any name: ");
        String name = read.next();

        System.out.print("Enter your age in years: ");
        int age = read.nextInt();

        System.out.print("Enter your salary: ");
        double salary = read.nextDouble();

        System.out.print("Enter any message: ");
        read = new Scanner(System.in);
        String msg = read.nextLine();

        System.out.println("\n----------------------------------------");
        System.out.println("Hello, " + name);
        System.out.println("You are " + age + " years old.");
        System.out.println("You are earning Rs." + salary + " per month.");
        System.out.println("Words from " + name + " - " + msg);
    }
}
```

```
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
```

## Collections: Collections overview

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).



- In Java, collections are objects that group multiple elements into a single unit, providing a way to store, retrieve, manipulate, and communicate data. The Java Collections Framework is a unified architecture for representing and manipulating these collections, offering a wide array of interfaces and classes for various data structures like lists, sets, queues, and maps.

### 1. Collections: The Foundation

- **Definition:** A collection is an object that acts as a container for a group of objects. It's like a basket, bag, or any other container that holds multiple items.
- **Purpose:** Collections are used to store, retrieve, manipulate, and communicate data.
- **Example:** A list of names, a set of unique numbers, or a map of employee IDs to employee objects.

**2. The Java Collections Framework**

- **Definition:**

  A framework that provides a standard way to work with collections in Java. It's a library of classes and interfaces that implement various data structures.

- **Components:**

  **Interfaces:** Define the behavior of collections (e.g., Collection, List, Set, Queue, Map).

  **Classes:** Implement the interfaces, providing concrete implementations of the data structures (e.g., ArrayList, LinkedList, HashSet, HashMap).

3. **Key Interfaces:**
   - ✓ Collection: The root interface for all collections.
   - ✓ List: An ordered collection (allows duplicates).
   - ✓ Set: An unordered collection (does not allow duplicates).
   - ✓ Queue: A collection that processes elements in a FIFO (First-In, First-Out) order.
   - ✓ Map: A collection that stores key-value pairs.

4. **Common Collection Implementations**
   - ✓ ArrayList: A dynamic array implementation of the List interface.
   - ✓ LinkedList: A linked list implementation of the List interface.
   - ✓ HashSet: A hash table implementation of the Set interface.
   - ✓ HashMap: A hash table implementation of the Map interface.
   - ✓ Queue: A queue implementation for FIFO operations (e.g., LinkedList can be used as a queue).

## Advantages of the Java Collection Framework

The Java Collections Framework offers significant advantages that enhance development practices, code quality, and application performance:

1. **Reusability:** The framework provides a comprehensive set of common classes and utility methods applicable across various types of collections. This feature promotes code reusability, sparing developers the need to write duplicate code for common operations.
2. **Quality:** Leveraging the Java Collections Framework elevates the quality of programs. The components within the framework have been extensively tested and are widely used by a vast community of developers, ensuring reliability and stability in your applications.
3. **Speed:** Developers often report an increase in development speed when using the Collections Framework. It allows them to concentrate on the core business logic of their applications rather than on implementing generic collection functionalities, thus speeding up the development process.
4. **Maintenance:** The open-source nature of the Java Collections Framework, coupled with readily available API documentation, facilitates easier code maintenance. Code written using the framework can be easily understood and taken over by other developers, ensuring continuity and ease of maintenance.

5. **Reduces Effort to Design New APIs:** An additional benefit is the reduced necessity for API designers and implementers to create new collection mechanisms for each new API.

**Class:** A class is a blueprint from which individual objects are created. It encapsulates data for objects through fields (attributes) and defines behavior via methods. Classes support inheritance, allowing one class to inherit the properties and methods of another, facilitating code reuse and polymorphism.

**Interface:** An interface is a reference type in Java that can contain constants and abstract methods (methods without a body). Interfaces specify what a class must do but not how it does it, enforcing a set of methods that the class must implement. Interfaces are used to achieve abstraction and multiple inheritance in Java.

## What is Collection Framework?

Java 1.2 provided **Collections Framework** that is the architecture to represent and manipulate Collections in java in a standard way. Java Collections Framework consists of the following parts:

### 1. Interfaces

- Java Collections Framework interfaces provides the abstract data type to represent collection.
- *java.util.Collection* is the root interface of Collections Framework. It is on the top of the Collections framework hierarchy. It contains some important methods such as size(), iterator(), add(), remove(), clear() that every Collection class must implement.
- Some other important interfaces are java.util.List, java.util.Set, java.util.Queue and java.util.Map. The Map is the only interface that doesn't inherit from the Collection interface but it's part of the Collections framework. All the collections framework interfaces are present in java.util package.

### 2. Implementation Classes

- Java Collections framework provides implementation classes for core collection interfaces. We can use them to create different types of collections in the Java program.
- Some important collection classes are ArrayList, LinkedList, HashMap, TreeMap, HashSet, and TreeSet. These classes solve most of our programming needs but if we need some special collection class, we can extend them to create our custom collection class.
- Java 1.5 came up with thread-safe collection classes that allowed us to modify Collections while iterating over them. Some of them are CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet. These classes are in java.util.concurrent package.
- All the collection classes are present in java.util and java.util.concurrent package.

### 3. Algorithms

- Algorithms are useful methods to provide some common functionalities such as searching, sorting and shuffling.

## Collection Interfaces

- The Collection interface is the interface which is implemented by all the classes in the collection framework.
- It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
- Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. that are implemented by all the subclasses of Collection interface.

## List Interface

- List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects.
- It can have duplicate values.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- To instantiate the List interface, we must use:

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

- There are various methods in List interface that can be used to insert, delete, and access the elements from the list.
- The classes that implement the List interface are given below.

### ArrayList

- The ArrayList class implements the List interface.
- It uses a dynamic array to store the duplicate element of different data types.
- The ArrayList class maintains the insertion order and is non-synchronized.
- The elements stored in the ArrayList class can be randomly accessed.
- Consider the following example.

**Example**

```
import java.util.*;
class Main{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}  }
```

**Output:**

*Ravi*
*Vijay*
*Ravi*
*Ajay*

## LinkedList

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.
- In LinkedList, the manipulation is fast because no shifting is required.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

**Output:**

*Ravi*
*Vijay*
*Ravi*
*Ajay*

## Vector

- Vector uses a dynamic array to store the data elements.
- It is similar to ArrayList.
- It is synchronized and contains many methods that are not the part of Collection framework.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

**Output:**

```
Ayush
Amit
Ashish
Garima
```

## Stack

- The stack is the subclass of Vector.
- It implements the last-in-first-out data structure, i.e., Stack.
- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

**Output:**

```
Ayush
Garvit
Amit
Ashish
```

## Queue Interface

- Queue interface maintains the first-in-first-out order.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
- Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

### PriorityQueue

- The PriorityQueue class implements the Queue interface.
- It holds the elements or objects which are to be processed by their priorities.
- PriorityQueue doesn't allow null values to be stored in the queue.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

**Output:**

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
```

```
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

## Deque Interface

- Deque interface extends the Queue interface.
- In Deque, we can remove and add the elements from both the side.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.
- Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

## ArrayDeque

- ArrayDeque class implements the Deque interface.
- It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

**Output:**

```
Gautam
Karan
Ajay
```

## Set Interface

- Set Interface in Java is present in java.util package.
- It extends the Collection interface.
- It represents the unordered set of elements which doesn't allow us to store the duplicate items.

- We can store at most one null value in Set.
- Set is implemented by HashSet, LinkedHashSet, and TreeSet.
- Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

## HashSet

- HashSet class implements Set Interface.
- It represents the collection that uses a hash table for storage.
- Hashing is used to store the elements in the HashSet.
- It contains unique items.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

**Output:**

```
Vijay
Ravi
Ajay
```

## LinkedHashSet

- LinkedHashSet class represents the LinkedList implementation of Set Interface.
- It extends the HashSet class and implements Set interface.
- Like HashSet, It also contains unique elements.
- It maintains the insertion order and permits null elements.
- Consider the following example.

**Example**

```
import java.util.*;
public class Main{
```

```
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

**Output:**

```
Ravi
Vijay
Ajay
```

## SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.
- The elements of the SortedSet are arranged in the increasing (ascending) order.
- The SortedSet provides the additional methods that inhibit the natural ordering of the elements.
- The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

## TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.
- Consider the following example:

**Example**

```
import java.util.*;
public class Main{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
```

```java
} } }
```

**Output:**

```
Ajay
Ravi
Vijay
```

## Map Interface

- The Map interface in Java is part of the Java Collections Framework.
- It represents a mapping between a set of keys and their corresponding values.
- A Map cannot contain duplicate keys; each key can map to at most one value.
- The Map interface is used to store key-value pairs, where each key is unique, and it provides an efficient way to retrieve, update, and manipulate data based on keys.

**Syntax:**

```java
Map<T, T> hm = new HashMap<>();
Map<T, T> tm = new TreeMap<>();
```

### HashMap

- HashMap in Java is a key-value data structure offering efficient data access via keys using hashing.
- Hashing converts large strings or other objects into smaller, consistent values for quick indexing and searching.
- HashMap implements the Map interface and is used for managing large datasets efficiently.
- Additionally, HashSet uses HashMap internally to store elements uniquely, demonstrating the utility of hashing in Java's collections framework for fast data retrieval and management.

**Example**

```java
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
        // Creating a HashMap
        Map<String, Integer> map = new HashMap<>();
        // Adding key-value pairs to the HashMap
        map.put("Alice", 10);
        map.put("Bob", 20);
        map.put("Charlie", 30);
        // Retrieving a value
        System.out.println("Value for 'Alice': " + map.get("Alice"));
        // Iterating over key-value pairs
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + ": " + value);
        }
        // Removing a key-value pair
        map.remove("Charlie");
        // Checking the presence of a key
        if (map.containsKey("Bob")) {
            System.out.println("Map contains key 'Bob'.");
        }
    }
}
```

**Output:**

```
Value for 'Alice': 10Bob: 20
Alice: 10
Charlie: 30
Map contains key 'Bob'.
```

## Sorting in Collections

- Collections sort in Java provides in-built methods to sort data faster and in an easier manner.
- *Collections* sort is a method of *Java Collections* class used to sort a list, which implements the *List* interface.
- All the elements in the list must be mutually comparable.
- If a list consists of string elements, then it will be sorted in alphabetical order.
- If it consists of a date element, it will be sorted into chronological order.
- The sort is guaranteed to be stable, i.e., the order of equal elements after performing sort will remain unchanged.
- By default, this method sorts the list in ascending order of elements.
- *Collections.sort* is used to sort all data structures such as linkedList, ArrayList, and queue.

### How Does Collections Sort Work in Java?

- The *sort* method transfers control to the *compare* method, and *compare* method returns values based on the arguments passed:
    - If both the objects are equal, returns 0
    - If the first object is greater than the second, returns a value > 0
    - If the second object is greater than the first, returns a value < 0
- Based on the values returned, the function decides whether to swap the values.
- To sort objects you need to specify a rule that decides how objects should be sorted.
- For example, if you have a list of cars you might want to sort them by year, the rule could be that cars with an earlier year go first.
- The Comparator and Comparable interfaces allow you to specify what rule is used to sort objects.
- Being able to specify a sorting rule also allows you to change how strings and numbers are sorted.

## Comparators Interface

- An object that implements the Comparator interface is called a comparator.
- The Comparator interface allows you to create a class with a compare() method that compares two objects to decide which one should go first in a list.
- The compare() method should return a number which is:
  - Negative if the first object should go first in a list.
  - Positive if the second object should go first in a list.
  - Zero if the order does not matter.
- A class that implements the Comparator interface might look something like this:

**Example**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

// Define a Car class
class Car {
  public String brand;
  public String model;
  public int year;

  public Car(String b, String m, int y) {
    brand = b;
    model = m;
    year = y;
  }
}

// Create a comparator

class SortByYear implements Comparator
{
  public int compare(Object obj1, Object obj2)
{
    // Make sure that the objects are Car objects
    Car a = (Car) obj1;
    Car b = (Car) obj2;

    // Compare the year of both objects
    if (a.year < b.year) return -1; // The first car has a smaller year
    if (a.year > b.year) return 1;  // The first car has a larger year
    return 0; // Both cars have the same year
  }
}
```

```
public class Main
{
  public static void main(String[] args)
{
    // Create a list of cars

    ArrayList<Car> myCars = new ArrayList<Car>();
    myCars.add(new Car("BMW", "X5", 1999));
    myCars.add(new Car("Honda", "Accord", 2006));
    myCars.add(new Car("Ford", "Mustang", 1970));

    // Use a comparator to sort the cars

    Comparator myComparator = new SortByYear();
    Collections.sort(myCars, myComparator);

    // Display the cars

    for (Car c : myCars) {
      System.out.println(c.brand + " " + c.model + " " + c.year);
    }
  }
}
```

**Output :**

```
Ford Mustang 1970
BMW X5 1999
Honda Accord 2006
```

**Using a Lambda Expression**

To make the code shorter, the comparator can be replaced with a lambda expression which has the same arguments and return value as the compare() method:

**Example**

Use a lambda expression as a comparator:

```
Collections.sort(myCars, (obj1, obj2) -> {
  Car a = (Car) obj1;
  Car b = (Car) obj2;
  if (a.year < b.year) return -1;
  if (a.year > b.year) return 1;
  return 0;
});
```

## Comparable Interface

- The Comparable interface allows an object to specify its own sorting rule with a compareTo() method.

- The compareTo() method takes an object as an argument and compares the comparable with the argument to decide which one should go first in a list.

- Like the comparator, the compareTo() method returns a number which is:

  - Negative if the comparable should go first in a list.
  - Positive if the other object should go first in a list.
  - Zero if the order does not matter.

Many native Java classes implement the Comparable interface, such as String and Integer.

This is why strings and numbers do not need a comparator to be sorted.

- An object that implements the Comparable interface might look something like this:

- Here is the same example as before but using the Comparable interface instead of a comparator:

**Example**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

// Define a Car class which is comparable

class Car implements Comparable
{
  public String brand;
  public String model;
  public int year;

  public Car(String b, String m, int y)
 {
    brand = b;
    model = m;
    year = y;
  }

  // Decide how this object compares to other objects

  public int compareTo(Object obj)
{
    Car other = (Car)obj;
    if(year < other.year) return -1; // This object is smaller than the other one
    if(year > other.year) return 1;  // This object is larger than the other one
    return 0; // Both objects are the same
  }
}

public class Main
{
  public static void main(String[] args)
  {
    // Create a list of cars
    ArrayList<Car> myCars = new ArrayList<Car>();
    myCars.add(new Car("BMW", "X5", 1999));
    myCars.add(new Car("Honda", "Accord", 2006));
    myCars.add(new Car("Ford", "Mustang", 1970));
    // Sort the cars
    Collections.sort(myCars);
    // Display the cars
    for (Car c : myCars) {
      System.out.println(c.brand + " " + c.model + " " + c.year);
    }
  }
}
```

**OUTPUT:**

```
Ford Mustang 1970
BMW X5 1999
Honda Accord 2006
```

## Comparator vs. Comparable

- A comparator is an object with one method that is used to compare two different objects.
- A comparable is an object which can compare itself with other objects.
- It is easier to use the Comparable interface when possible, but the Comparator interface is more powerful because it allows you to sort any kind of object even if you cannot change its code.
- Comparable interface can be used to provide single way of sorting whereas Comparator interface is used to provide different ways of sorting.
- For using Comparable, Class needs to implement it whereas for using Comparator we don’t need to make any change in the class.
- Comparable interface is in java.lang package whereas Comparator interface is present in java.util package.
- We don’t need to make any code changes at client side for using Comparable, Arrays.sort() or Collection.sort() methods automatically uses the compareTo() method of the class. For Comparator, client needs to provide the Comparator class to use in compare() method.

**DEPARTMENT OF INFORMATION TECHNOLOGY**

# Java Programming (R22CSI2215)

**UNIT – IV**

**Multithreading:** Process and Thread, Differences between thread-based multitasking and process – based multitasking, Java thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication.

**Java Database Connectivity:** Types of Drivers, JDBC architecture, JDBC Classes and Interfaces, Basic steps in Developing JDBC Application, Creating a New Database and Table with JDBC.

## 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.

### Process

- A process is an independent execution environment with its own memory space, resources, and system context. It represents a running program managed by the operating system.

### Thread

- A thread, on the other hand, is a lightweight sub-process that exists within a process. Multiple threads can coexist within the same process, sharing the process's memory space and resources. This shared memory enables efficient communication and data sharing between threads.

### Key Differences

| **Feature** | **Process** | **Thread** |
|---|---|---|
| Memory Space | Separate memory space for each process | Shared memory space within a process |
| Resource Sharing | Limited resource sharing between processes | Extensive resource sharing among threads within a process |
| Context Switching | Higher overhead for context switching between processes | Lower overhead for context switching between threads |

| | | |
|---|---|---|
| Communication | Requires inter-process communication (IPC) mechanisms | Easier communication through shared memory |
| Creation/Termination | Higher cost for process creation and termination | Lower cost for thread creation and termination |

## Multithreading

- Java provides built-in support for multithreading through the Thread class and the Runnable interface. Multithreading enables a program to perform multiple tasks concurrently, improving responsiveness and efficiency.

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.

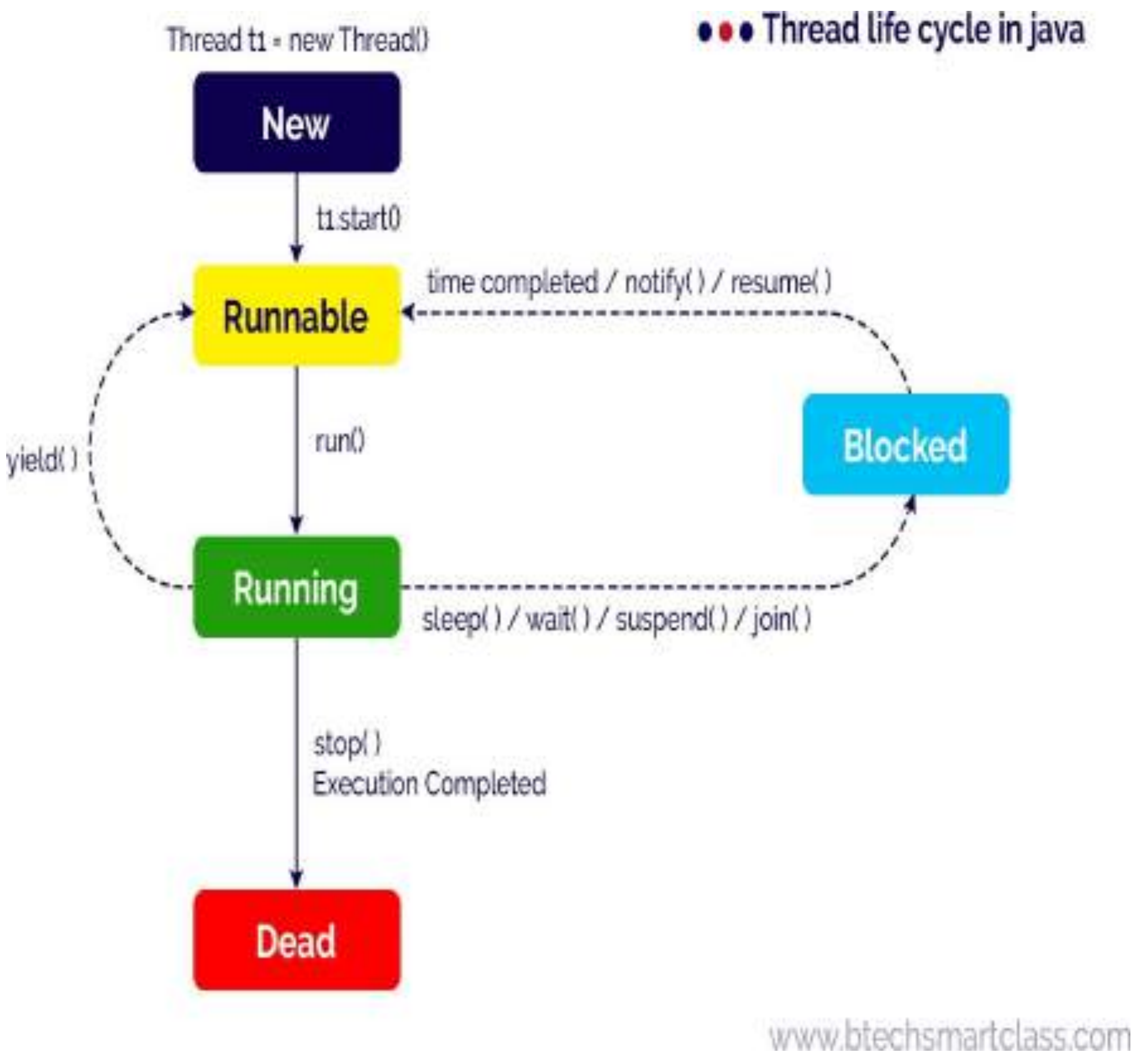
- Process-based multitasking
- Thread-based multitasking

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 / Thread Life Cycle

- 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();    suspened();    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.

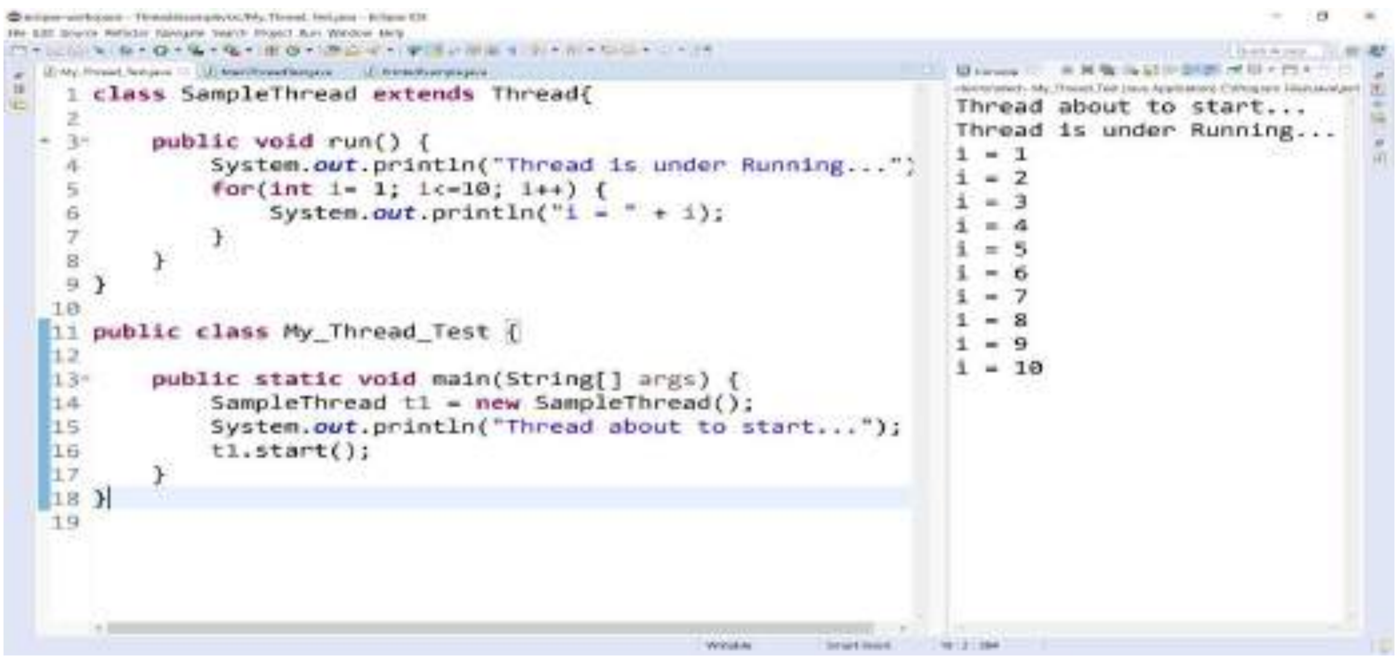
## Extending Thread class

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.



```
class SampleThread extends Thread{

    public void run() {
        System.out.println("Thread is under Running...");
        for(int i= 1; i<=10; i++) {
            System.out.println("i = " + i);
        }
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread t1 = new SampleThread();
        System.out.println("Thread about to start...");
        t1.start();
    }
}
```

```
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
```

## Implementing Runnable interface

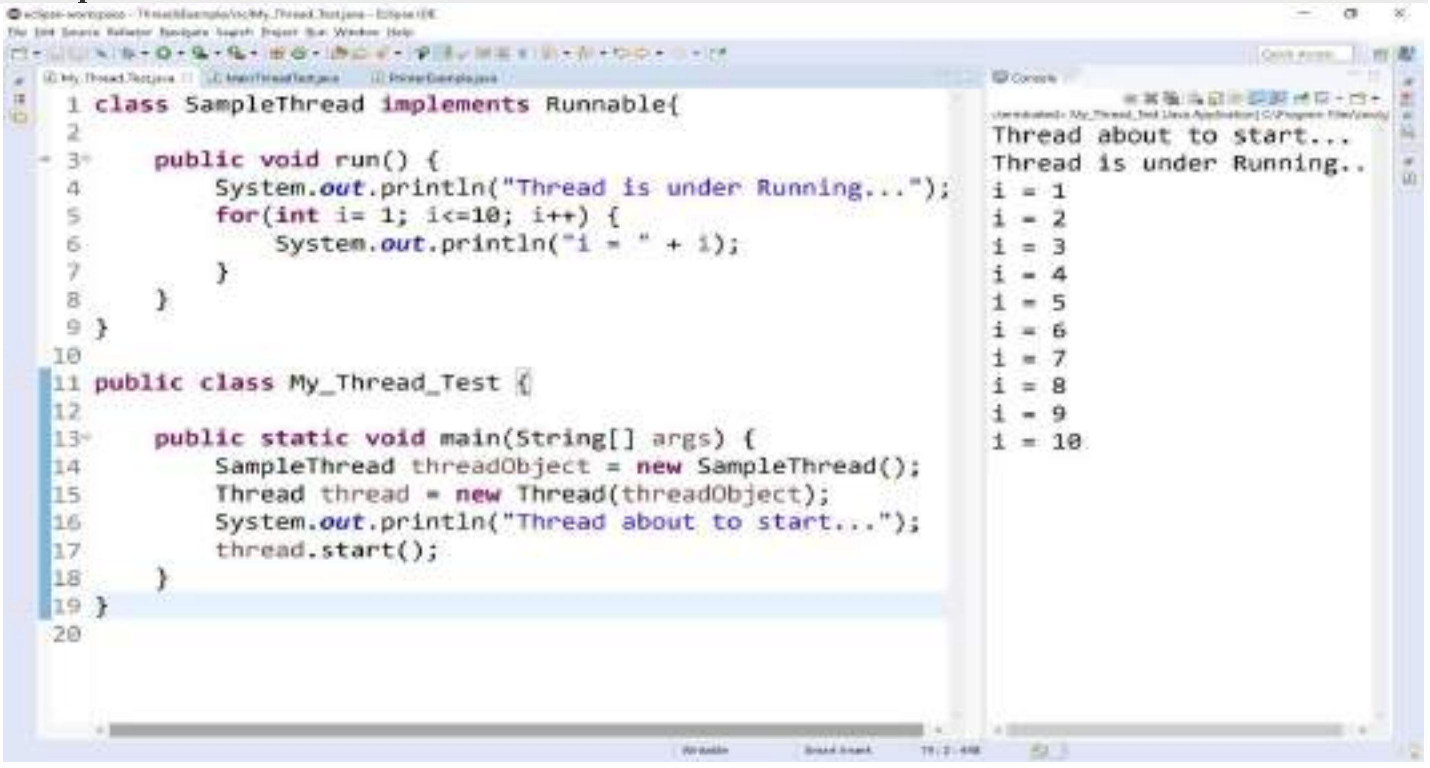
The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

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**



```
class SampleThread implements Runnable{

    public void run() {
        System.out.println("Thread is under Running...");
        for(int i= 1; i<=10; i++) {
            System.out.println("i = " + i);
        }
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread threadObject = new SampleThread();
        Thread thread = new Thread(threadObject);
        System.out.println("Thread about to start...");
        thread.start();
    }
}
```

```
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 consructors.

- **Thread( )**
- **Thread( String threadName )**
- **Thread( Runnable objectName )**
- **Thread( Runnable objectName, String threadName )**

The Thread classes 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 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 |
| 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 |

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.

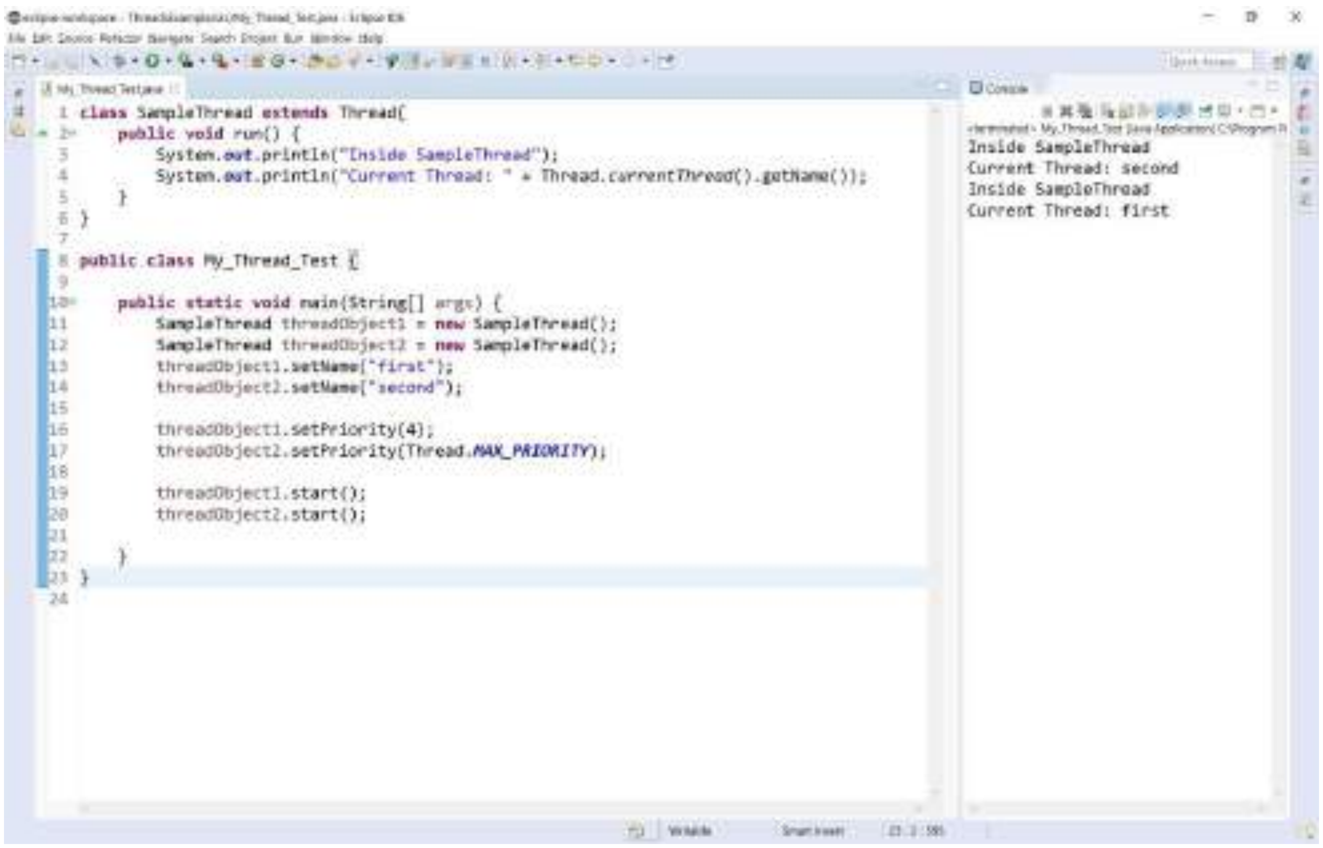
**Example**

```
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();
            }
  }
```



```java
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();

    }
}
```

```
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

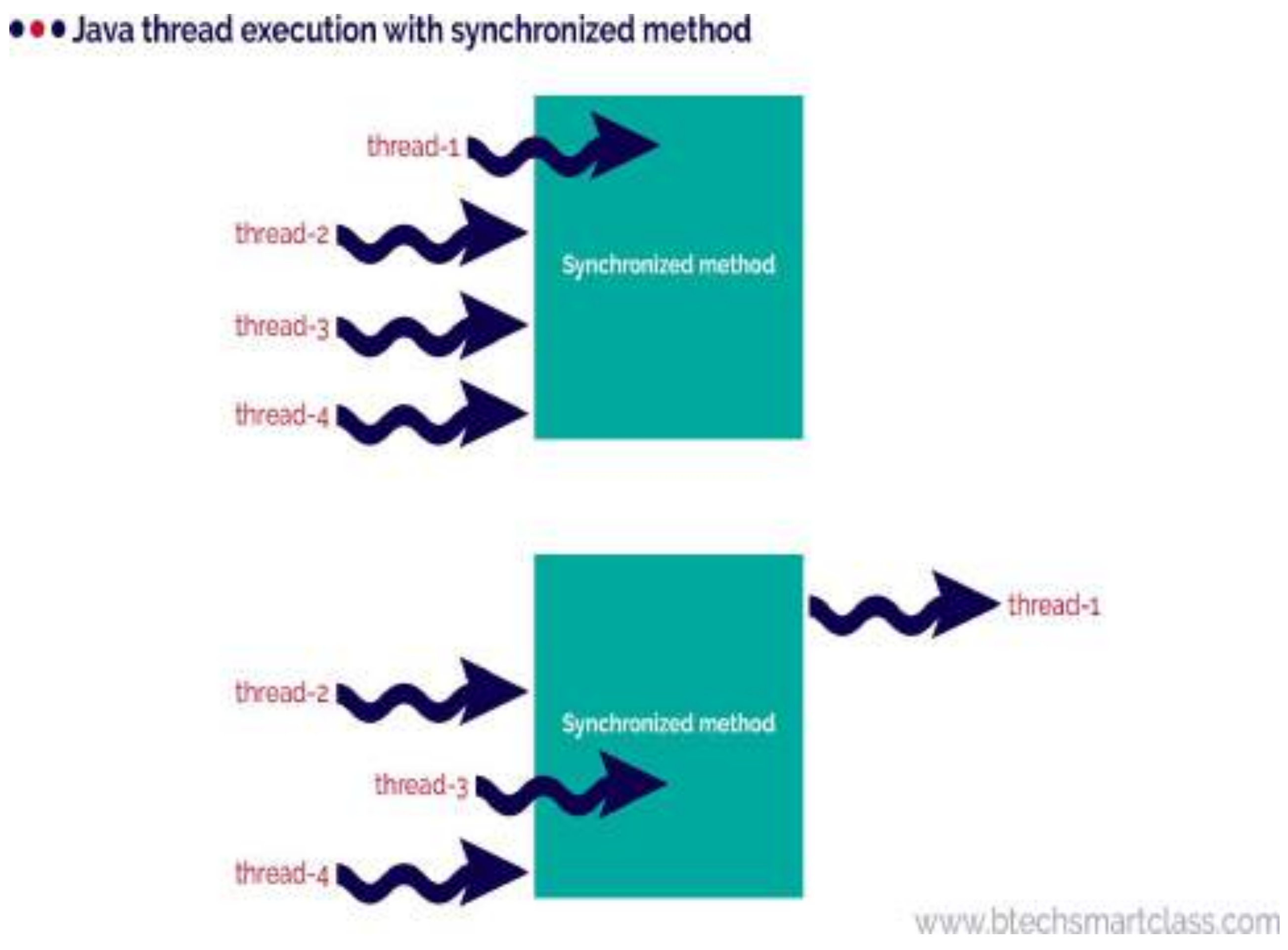
## Mutual Exclusion

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.

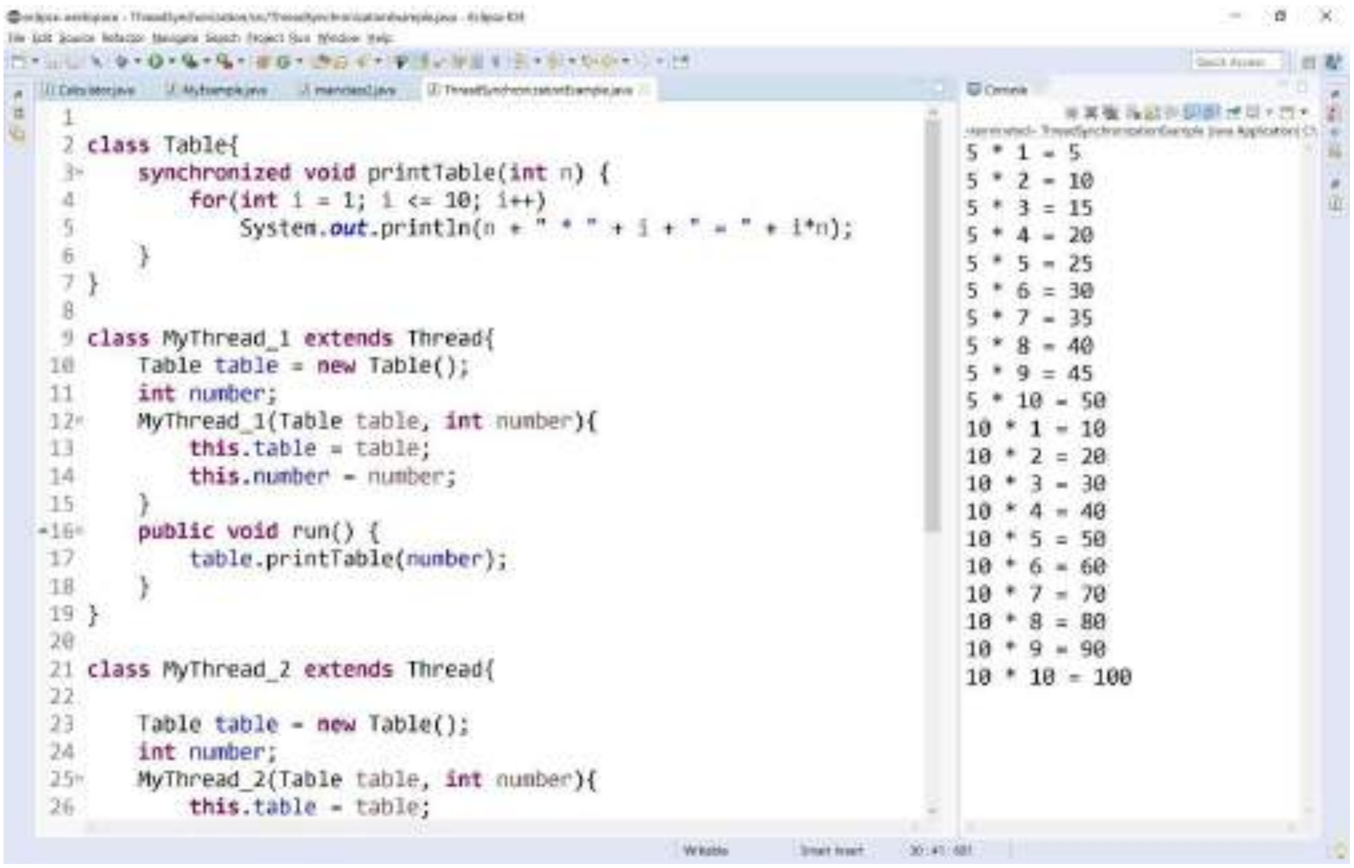


In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method). When thread-1 completes it task, then one of the threads that are waiting is allocated with the synchronized method, 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.



```
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;
```

```
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 folllowing syntax is used to define a synchronized block.

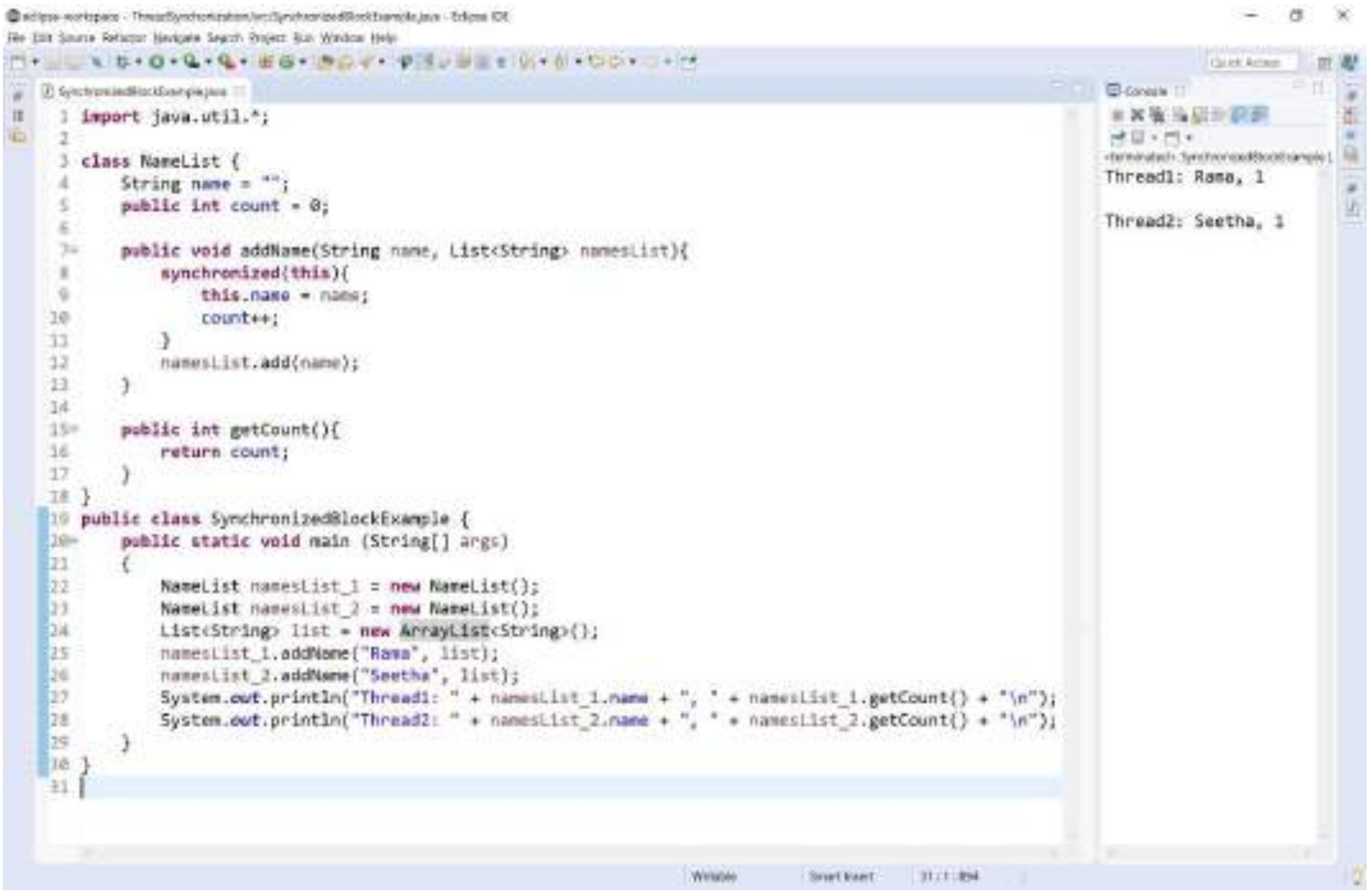
**Syntax**

```
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");
    }
}
```



```
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");
    }
}
```

```
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.
- The 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.

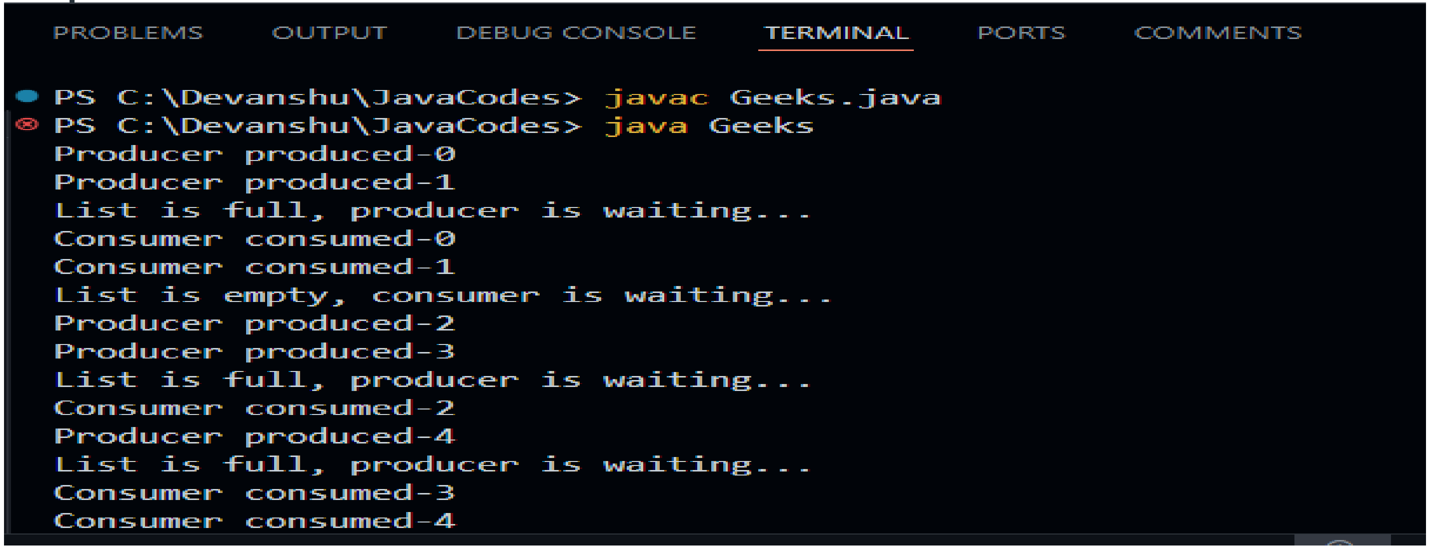
**Example**

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

```
// Java program to implement solution of producer-consumer problem.

import java.util.LinkedList;

public class Geeks
{
    public static void main(String[] args)
        throws InterruptedException
    {
        // Object of a class that has both produce()and consume() methods
        final PC pc = new PC();

        // Create producer thread
        Thread t1 = new Thread(new Runnable()
           {
            @Override
            public void run()
            {
                try {
                    pc.produce();
                }
                catch (InterruptedException e)
                   {
                    e.printStackTrace();
                }
            }
        });
        // Create consumer thread
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                try {
                    pc.consume();
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // t1 finishes before t2
        t1.join();
        t2.join();
    }

    // This class has a list, producer (adds items to list)
    // and consumer (removes items).

           public static class PC {
        // Create a list shared by producer and consumer
        // Size of list is 2.

           LinkedList<Integer> list = new LinkedList<>();
        int capacity = 2;

        // Function called by producer thread
        public void produce() throws InterruptedException
        {
            int value = 0;
            while (true) {
                synchronized (this)
                {
                    // producer thread waits while list is full
                    if (list.size() == capacity) {
                        System.out.println("List is full, producer is waiting...");
                        // Signal any waiting consumer before waiting
                        notify();
                        wait();
                    }

                    // to insert the jobs in the list
                    list.add(value);
                    System.out.println("Producer produced-" + value);
                    value++;
```

```
                        // notifies the consumer thread that now it can start consuming
                        notify();

                        // makes the working of program easier to understand
                        Thread.sleep(1000);
                    }
                } }

            // Function called by consumer thread
            public void consume() throws InterruptedException
            {
                while (true) {
                    synchronized (this)
                    {
                        // consumer thread waits while list is empty
                        if (list.size() == 0) {
                            System.out.println("List is empty, consumer is waiting...");
                            // Signal any waiting producer before waiting
                            notify();
                            wait();
                        }
                        // to retrieve the first job in the list
                        int val = list.removeFirst();
                        System.out.println("Consumer consumed-" + val);
                        // Wake up producer thread
                        notify();
                        // and sleep
                        Thread.sleep(1000);
                    }
            }    }
        }
    }
```

**Output:**



PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    COMMENTS

```
PS C:\Devanshu\JavaCodes> javac Geeks.java
PS C:\Devanshu\JavaCodes> java Geeks
Producer produced-0
Producer produced-1
List is full, producer is waiting...
Consumer consumed-0
Consumer consumed-1
List is empty, consumer is waiting...
Producer produced-2
Producer produced-3
List is full, producer is waiting...
Consumer consumed-2
Producer produced-4
List is full, producer is waiting...
Consumer consumed-3
Consumer consumed-4
```

All the methods wait( ), notify( ), and notifyAll( ) can be used only inside the synchronized methods only.

## Java Database Connectivity

- JDBC stands for Java Database Connectivity.
- JDBC is a Java API to connect and execute the query with the database, and processing the results.
- It is a part of **JavaSE** (Java Standard Edition) from Oracle Corporation.
- JDBC API uses JDBC drivers to connect with the database.
- Java Database Connectivity (JDBC) is an Application Programming Interface (API) used to connect Java application with Database.
- JDBC is used to interact with the various type of Database such as Oracle, MS Access, My SQL and SQL Server and it can be stated as the platform-independent interface between a relational database and Java programming.
- It provided methods to query and update data in a database and related towards relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.
- The JDBC library combines APIs for each of the tasks discussed below that are associated with database usage.

    1. Create a connection to a database.
    2. Creating SQL or MySQL statements.
    3. Executing SQL or MySQL queries in the database.
    4. Viewing & modifying the resulting records.

- JDBC is a specification that provides a comprehensive set of interfaces that allows for portable access to an underlying database.
- Java can be used to write different types of executable, such as:
    1. Java Applications
    2. Java Applets
    3. Java Servlets
    4. Java Server Pages (JSPs)
    5. Enterprise JavaBeans (EJBs).

- All of these different executable can use a JDBC driver to access a database and take advantage of the stored data. JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.
- JDBC connection supports to create and execute statements. These statements are an updated statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT.

## Types of JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database.
- JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.
- There are 4 types of JDBC drivers:

    1. Type 1 driver or JDBC-ODBC bridge driver
    2. Type 2 driver or Native-API driver (partially java driver)
    3. Type 3 driver or Network Protocol driver (fully java driver)
    4. Type 4 driver or Thin driver (fully java driver)

### 1) Type 1 driver or JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
- The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
- This is now discouraged because of thin driver.

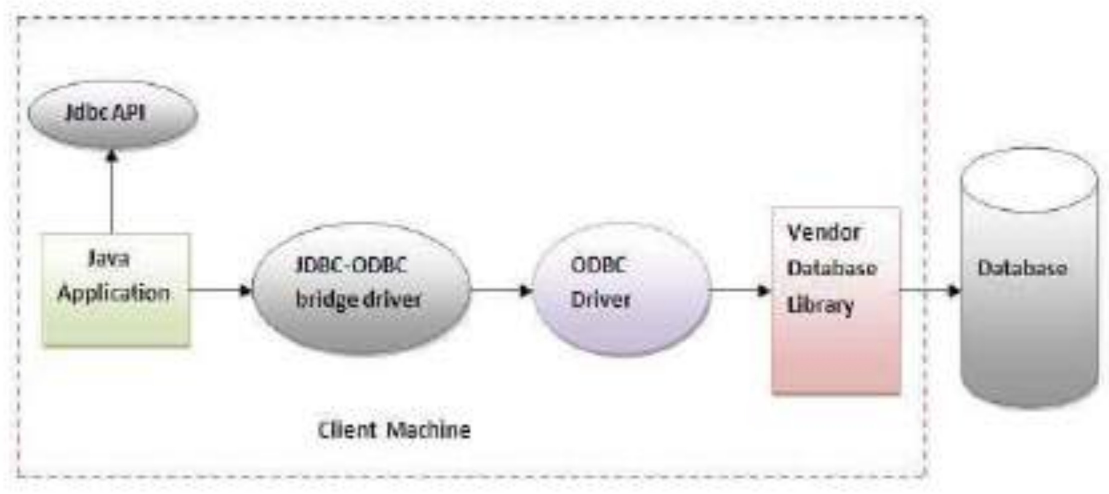


Figure- JDBC-ODBC Bridge Driver

- In Java 8, the JDBC-ODBC Bridge has been removed.
- Oracle does not support the JDBC-ODBC Bridge from Java 8.
- Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

### Advantages:

- easy to use.
- can be easily connected to any database.

### Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

## 2) Type 2 driver or Native-API driver

- The Native API driver uses the client-side libraries of the database.
- The driver converts JDBC method calls into native calls of the database API.
- It is not written entirely in java.

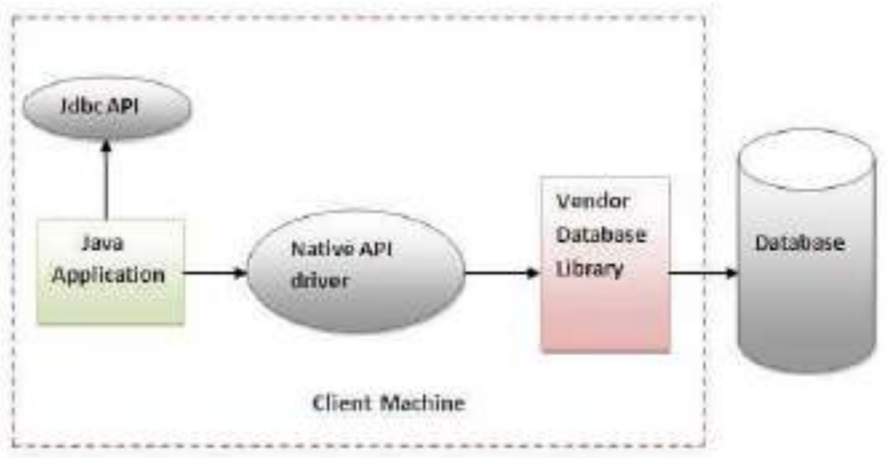


Figure- Native API Driver

### Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

### Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

## 3) Type 3 driver or Network Protocol driver

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- It is fully written in java.

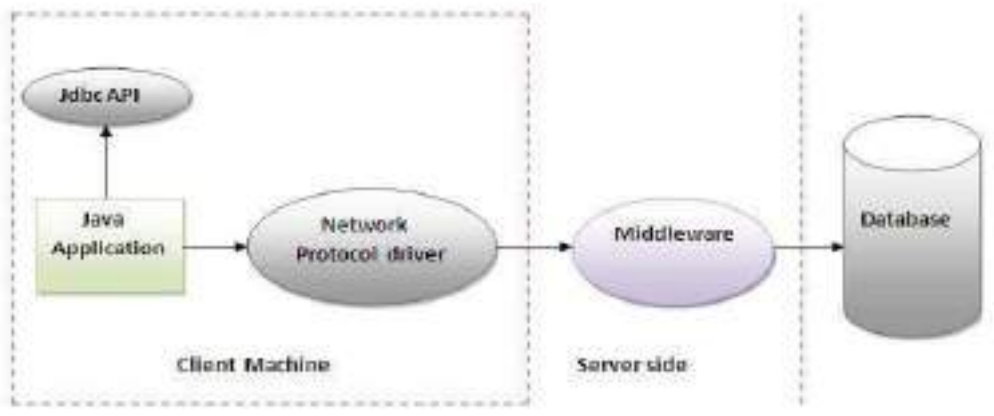


Figure- Network Protocol Driver

**Advantage:**

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

**Disadvantages:**

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

## 4) Type 4 driver or Thin driver

- The thin driver converts JDBC calls directly into the vendor-specific database protocol.
- That is why it is known as thin driver.
- It is fully written in Java language.

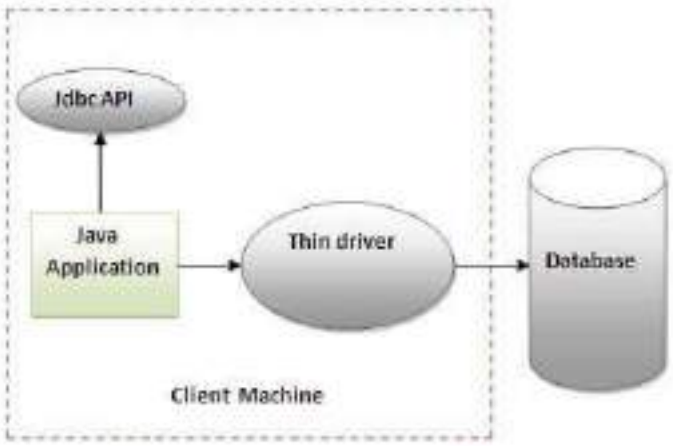


Figure- Thin Driver

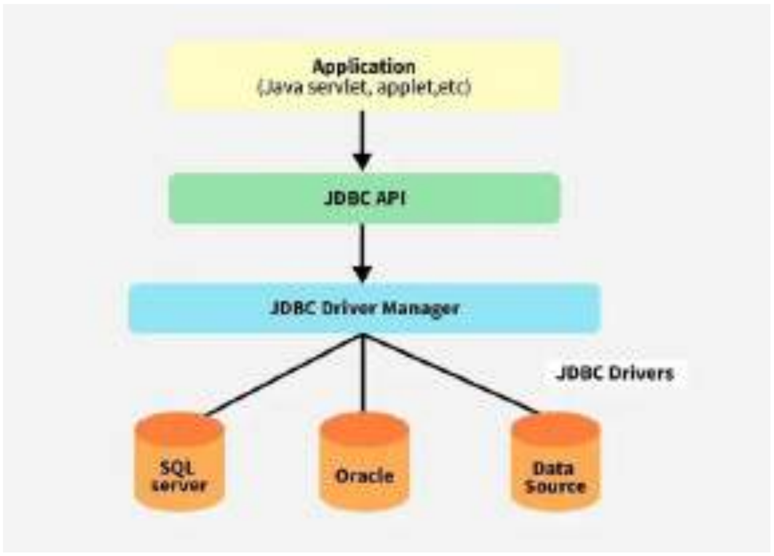
**Advantage:**

- Better performance than all other drivers.
- No software is required at client side or server side.

**Disadvantage:**

- Drivers depend on the Database.

## JDBC Architecture



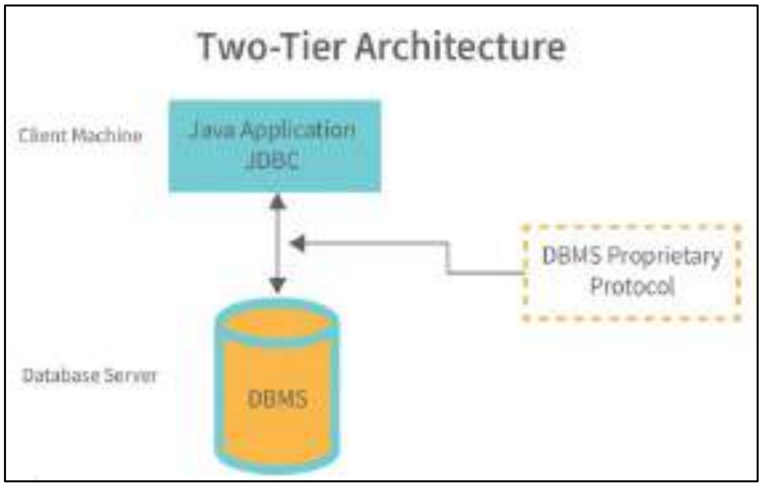
**Explanation:**

- **Application:** It can be a Java application or servlet that communicates with a data source.
- **The JDBC API:** It allows Java programs to execute SQL queries and get results from the database. Some key components of JDBC API include
    - Interfaces like Driver, ResultSet, RowSet, PreparedStatement, and Connection that helps managing different database tasks.
    - Classes like DriverManager, Types, Blob, and Clob that helps managing database connections.
- **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.
- **JDBC drivers:** These drivers handle interactions between the application and the database.

The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

**1. Two-Tier Architecture**

- A Java Application communicates directly with the database using a JDBC driver.
- It sends queries to the database and then the result is sent back to the application.
- For example, in a client/server setup, the user's system acts as a client that communicates with a remote database server.
- A Java applet and application communicates directly with the data source in the two-tier paradigm.
- This necessitates the use of a JDBC driver that can interface with the data source in question.
- The user's commands are transmitted to the database or other data source, and the statements' results are returned to the user.
- The data source could be on another machine to which the user has a network connection.
- A client/server configuration is one in which the user's machine acts as the client and the system that houses the data source acts as the server.
- An intranet, for example, can connect people within a company, or the Internet can be used as a network.

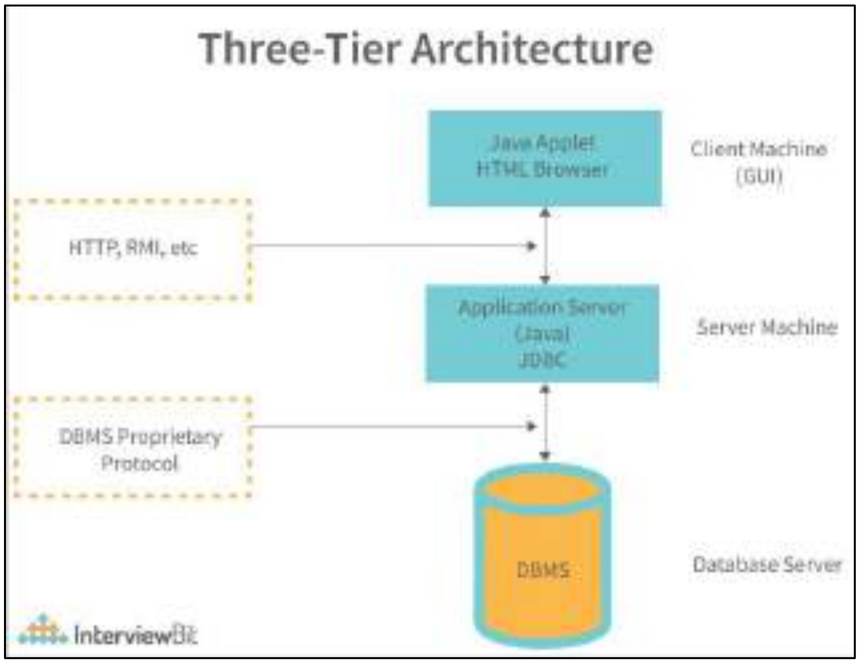


**Structure:**

*Client Application (Java) -> JDBC Driver -> Database*

**2. Three-Tier Architecture**

- In this, user queries are sent to a middle-tier services, which interacts with the database.
- The database results are processed by the middle tier and then sent back to the user.
- Commands are sent to a "middle tier" of services in the three-tier paradigm, which subsequently transmits the commands to the data source.
- The data source interprets the commands and provides the results to the middle tier, which ultimately passes them on to the user.
- The three-tier architecture appeals to MIS directors because the intermediate tier allows them to maintain control over access and the types of changes that can be made to company data.
- Another benefit is that it makes application deployment easier.
- Finally, the three-tier architecture can bring performance benefits in many circumstances.



**Structure:**

*Client Application -> Application Server -> JDBC Driver -> Database*

## The Components of JDBC

- The components of JDBC are listed below.
- These elements assist us in interacting with a database.
- The following are the JDBC components:

1. **JDBC Driver Manager:** In a JDBC application, the Driver Manager loads database-specific drivers. This driver manager makes a database connection. To handle the user request, it additionally makes a database-specific call to the database.

2. **Driver:** A driver is an interface that manages database server connectivity. Communication is handled using DriverManager objects.

3. **JDBC-ODBC Bridge Drivers:** They are used to link database drivers to the database. The JDBC method calls are translated into ODBC method calls by the bridge. To access the ODBC (Open Database Connectivity) characteristics, it uses the sun.jdbc.odbc package, which includes the native library.

4. **JDBC API:** Sun Microsystem has provided JDBC API, which allows you to write a Java program that talks with any database without modifying the code. The JDBC API is implemented by the JDBC Driver.

5. **JDBC Test Suite:** The JDBC Test Suite aids in the testing of JDBC Driver operations such as insertion, deletion, and updating. It aids in determining whether or not the JDBC Drivers will run the program. It ensures that the program will be run by JDBC Drivers with confidence and conformity.

6. **Database Server:** This is the database server that the JDBC client wants to communicate with, such as Oracle, MySQL, SQL Server, and so on.

7. **Statement:** To send SQL statements to the database, you use objects built using this interface. In addition to performing stored procedures, certainly derived interfaces accept parameters.

8. **ResultSet:** These objects retain data retrieved from a database when you use Statement objects to conduct a SQL query. It functions as an iterator, allowing you to cycle through the data it contains.

9. **SQL Exception:** This class is responsible for any errors that occur in a database application.

## JDBC Classes and Interfaces

- JDBC API is available in two packages java.sql, core API and javax.sql JDBC optional packages.
- Following are the important classes and interfaces of JDBC.
- JDBC is a set of classes and interfaces that define a standard method of connecting with a database.
- JDBC's primary classes and interfaces are Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, and SQLException.

| Class/interface | Description |
|---|---|
| **DriverManager** | This class manages the JDBC drivers. You need to register your drivers to this.It provides methods such as registerDriver() and getConnection(). |
| **Driver** | This interface is the Base interface for every driver class i.e.<br>If you want to create a JDBC Driver of your own you need to implement this interface. If you load a Driver class (implementation of this interface), it will create an instance of itself and register with the driver manager. |
| **Statement** | This interface represents a static SQL statement. Using the Statement object and its methods, you can execute an SQL statement and get the results of it. It provides methods such as execute(), executeBatch(), executeUpdate() etc. To execute the statements. |
| **PreparedStatement** | This represents a precompiled SQL statement. An SQL statement is compiled and stored in a prepared statement and you can later execute this multiple times. You can get an object of this interface using the method of the Connection interface named prepareStatement(). This provides methods such as executeQuery(), executeUpdate(), and execute() to execute the prepared statements and getXXX(), setXXX() (where XXX is the datatypes such as long int float etc..) methods to set and get the values of the bind variables of the prepared statement. |
| **CallableStatement** | Using an object of this interface you can execute the stored procedures. This returns single or multiple results. It will accept input parameters too. You can create a CallableStatement using the prepareCall() method of the Connection interface.<br>Just like Prepared statement, this will also provide setXXX() and getXXX() methods to pass the input parameters and to get the output parameters of the procedures. |
| **Connection** | This interface represents the connection with a specific database. SQL statements are executed in the context of a connection. This interface provides methods such as close(), commit(), rollback(), createStatement(), prepareCall(), prepareStatement(), setAutoCommit() setSavepoint() etc. |
| **ResultSet** | This interface represents the database result set, a table which is generated by executing statements. This interface provides getter and update methods to retrieve and update its contents respectively. |
| **ResultSetMetaData** | This interface is used to get the information about the result set such as, number of columns, name of the column, data type of the column, schema of the result set, table name, etc.It provides methods such as getColumnCount(), getColumnName(), getColumnType(), getTableName(), getSchemaName() etc. |

## Basic steps in Developing JDBC Application

- JDBC or Java Database Connectivity is a Java API to connect and execute the query with the database.
- It is a specification from Sun microsystems that provides a standard abstraction(API or Protocol) for java applications to communicate with various databases.
- It provides the language with java database connectivity standards.
- It is used to write programs required to access databases.
- JDBC, along with the database driver, can access databases and spreadsheets.
- The enterprise data stored in a relational database(RDB) can be accessed with the help of JDBC APIs.

### Standard Steps followed for developing JDBC(JDBC4.X) Application

1. Load and register the Driver
2. Establish the Connection b/w java application and database
3. Create a Statement Object
4. Send and execute the Query
5. Process the result from ResultSet
6. Close the Connection

### Step1: Load and register the Driver

- A third-party DB vendor class that implements java.sql.Driver(I) is called a “Driver”.
- This class Object we need to create and register it with JRE to set up the JDBC environment to run JDBC applications.

```
public class com.mysql.cj.jdbc.Driver extends com.mysql.cj.jdbc.NonRegisteringDriver implements
java.sql.Driver {

   public com.mysql.cj.jdbc.Driver() throws java.sql.SQLException;

   static {};

}
```

In MySQL Jar, the Driver class is implementing java.sql.Driver, so Driver class Object should be created and it should be registered to set up the JDBC environment inside JRE.

### Step 2: Establish the Connection b/w java application and database

- public static Connection getConnection(String url, String username,String password) throws SQLException;
- public static Connection getConnection(String url, Properties) throws SQLException;
- public static Connection getConnection(String url) throws SQLException;
- The below creates the Object of Connection interface.

```
Connection connection = DriverManager.getConnection(url,username,password);
```

getConnection(url,username,password) created an object of a class which implements Connection(I) that class object is collected by Connection(I). This feature in java refers to

### Step 3: Create a Statement Object

- public abstract Statement createStatement() throws SQLException;
- public abstract Statement createStatement(int,int) throws SQLException;
- public abstract Statement createStatement(int,int,int) throws SQLException;

```
// create statement object

Statement statement = connection.createStatement();
```

### Step 4: Send and execute the Query

From the DB administrator's perspective queries are classified into 5 types

1. DDL (Create table, alter table, drop table,..)
2. DML(Insert, update, delete)
3. DQL(select)
4. DCL(alter password,grant access)
5. TCL(commit,rollback,savepoint)

According to the java developer perspective, we categorize queries into 2 types

- Select Query
- NonSelect Query

Methods for executing the Query are

- executeQuery() => for the select query we use this method.
- executeUpdate() => for insert, update and delete queries we use this method.
- execute() => for both select and non-select queries we use this method.

```
public abstract ResultSet executeQuery(String sqlSelectQuery) throws SQLException;

String sqlSelectQuery ="select sid,sname,sage,saddr from Student";

ResultSet resultSet = statement.executeQuery(sqlSelectQuery);
```

### Step 5: Process the result from ResultSet

public abstract boolean next() throws java.sql.SQLException; => To check whether the next Record is available or not returns true if available otherwise returns false.

```
System.out.println("SID\tSNAME\tSAGE\tSADDR");

while(resultSet.next()){

   Integer id = resultSet.getInt(1);

   String name = resultSet.getString(2);

   Integer age = resultSet.getInt(3);

   String team = resultSet.getString(4);

   System.out.println(id+"\t"+name+"\t"+age+"\t"+team);

}
```

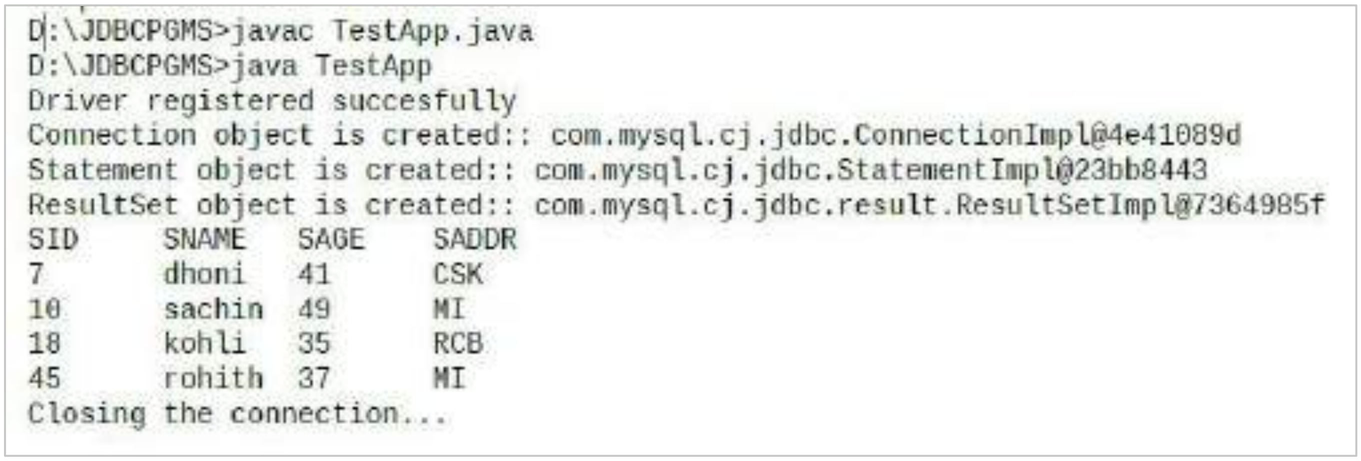
### Step 6: Close the Connection

```
// Close the Connection

connection.close();
```

**Example :**

```
/*Java code to communicate with database and execute select query*/

import com.mysql.cj.jdbc.Driver;
import java.io.*;
import java.sql.*;

class GFG {
    public static void main(String[] args)
        throws SQLException
    {
        // Load and register the Driver
        Driver driver = new Driver(); // Creating driver
                                      // object for MySQLDB
        DriverManager.registerDriver(driver);
        System.out.println("Driver registered successfully");

        // Establish the connection b/w java and Database
        // JDBC URL SYNTAX::
        // <mainprotocol>:<subprotocol>:<subname>
        String url = "jdbc:mysql://localhost:3306/enterprisejavabatch";
        String username = "root";
        String password = "root123";

        Connection connection = DriverManager.getConnection( url, username, password);
        System.out.println("Connection object is created:: " + connection);

        // Create a Statement Object
        Statement statement = connection.createStatement();
        System.out.println("Statement object is created:: "+ statement);

        // Sending and execute the Query
        String sqlSelectQuery = "select sid,sname,sage,saddr from Student";
        ResultSet resultSet = statement.executeQuery(sqlSelectQuery);
        System.out.println("ResultSet object is created:: "+ resultSet);

        // Process the result from ResultSet
        System.out.println("SID\tSNAME\tSAGE\tSADDR");
        while (resultSet.next()) {
            Integer id = resultSet.getInt(1);
            String name = resultSet.getString(2);
            Integer age = resultSet.getInt(3);
            String team = resultSet.getString(4);
            System.out.println(id + "\t" + name + "\t" + age+ "\t" + team);
        }

        // Close the Connection
        connection.close();
        System.out.println("Closing the connection...");
    }
}
```

**Output:**



```
D:\JDBCPGMS>javac TestApp.java
D:\JDBCPGMS>java TestApp
Driver registered succesfully
Connection object is created:: com.mysql.cj.jdbc.ConnectionImpl@4e41089d
Statement object is created:: com.mysql.cj.jdbc.StatementImpl@23bb8443
ResultSet object is created:: com.mysql.cj.jdbc.result.ResultSetImpl@7364985f
SID     SNAME   SAGE    SADDR
7       dhoni   41      CSK
10      sachin  49      MI
18      kohli   35      RCB
45      rohith  37      MI
Closing the connection...
```

## Creating a New Database and Table with JDBC.

### Creating a New Database with JDBC

The following steps are required to create a new Database using JDBC application –

**Import the packages** − Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice.

**Open a connection** − Requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database server.
To create a new database, you need not give any database name while preparing database URL as mentioned in the below example.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to the database.

**Clean up the environment -** try with resources automatically closes the resources.

## Example: Creating a Database

In this example, we've three static strings containing a dababase connection url, username, password. Now using DriverManager.getConnection() method, we've prepared a database connection. Once connection is prepared, we've created a Statement object using connection.createStatement() method. Then using statement.executeUpdate(), we've run the query to create a new database named Students and printed the success message.

In case of any exception while creating the database, a catch block handled SQLException and printed the stack trace.

Copy and paste the following example in JDBCExample.java, compile and run as follows −

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCExample {
   static final String DB_URL = "jdbc:mysql://localhost/";
   static final String USER = "guest";
   static final String PASS = "guest123";

   public static void main(String[] args) {
      // Open a connection
      try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
         Statement stmt = conn.createStatement();
      ) {
         String sql = "CREATE DATABASE STUDENTS";
         stmt.executeUpdate(sql);
         System.out.println("Database created successfully...");
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```

### Output

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
```

```
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java JDBCExample
Database created successfully...
C:\>
```

## Creating a Table with JDBC

**Import the packages** − Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice.

**Open a connection** − Requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to create a table in a seleted database.

**Clean up the environment** − try with resources automatically closes the resources.

**Example: Creating a Table**

In this example, we've three static strings containing a dababase connection url, username, password. Now using DriverManager.getConnection() method, we've prepared a database connection. Once connection is prepared, we've prepared a Statement object using createStatement() method. As next step, We've prepared a SQL string to create a new table REGISTRATION and created the table in database by calling statement.executeUpdate() method.

Copy and paste the following example in TestApplication.java, compile and run as follows −

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TestApplication {
   static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
   static final String USER = "guest";
   static final String PASS = "guest123";

   public static void main(String[] args) {
      // Open a connection
      try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
         Statement stmt = conn.createStatement();
       {
          String sql = "CREATE TABLE REGISTRATION " +
              "(id INTEGER not NULL, " +
              " first VARCHAR(255), " +
              " last VARCHAR(255), " +
              " age INTEGER, " +
              " PRIMARY KEY ( id ))";

         stmt.executeUpdate(sql);
         System.out.println("Created table in given database...");
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```

Now let us compile the above example as follows −

```
C:\>javac TestApplication.java
C:\>
```

When you run **TestApplication**, it produces the following result −

```
C:\>java TestApplication
Created table in given database...
C:\>
```

**DEPARTMENT OF INFORMATION TECHNOLOGY**

# Java Programming (R22CSI2215)

**UNIT – V**

**GUI Programming with Swing** – Introduction, limitations of AWT, MVC architecture, components, containers, Layout Manager Classes, Simple Applications using AWT and Swing.

**Event Handling-** The Delegation event model- Events, Event sources, Event Listeners, Event classes, Handling mouse and keyboard events, Adapter classes.

## Introduction to Java GUI Programming with Swing

- Java Swing is a Java GUI (Graphical User Interface) toolkit used for creating desktop-based applications.
- It provides a wide range of components, including buttons, text fields, menus, and more, allowing developers to build visually appealing and functional user interfaces.
- Swing is platform-independent, ensuring that applications look consistent across different operating systems.
- Swing components are built on top of the Abstract Window Toolkit (AWT), but they provide a number of advantages over AWT components.
- For example, Swing components are more lightweight and efficient
- Here's a breakdown of key concepts and elements related to Java Swing:

### 1. What is Swing?

- Swing is a part of the Java Foundation Classes (JFC) and is built upon the AWT
- (Abstract Windowing Toolkit).
- Swing is an extension of the Abstract Window Toolkit [AWT].
- Java Swing offers much-improved functionality over AWT, new components, expanded components features, and excellent event handling with drag-and-drop support.
- It's entirely written in Java, making it platform-independent.
- Swing components are lightweight and more powerful than AWT components, offering features like tabbed panes, scroll panes, and tables.
- Swing is a Set of API (API- Set of Classes and Interfaces)
- Swing is Provided to Design Graphical User Interfaces
- Swing is an Extension library to the AWT (Abstract Window Toolkit)
- Includes New and improved Components that have been enhancing the looks and Functionality of GUIs'
- Swing can be used to build (Develop) The Standalone swing GUI Apps as Servlets and Applets
- It Employs model/view design architecture.
- Swing is more portable and more flexible than AWT, the Swing is built on top of the AWT.
- Swing is Entirely written in Java.
- Java Swing Components are Platform-independent, and The Swing Components are lightweight.
- Swing Supports a Pluggable look and feel and Swing provides more powerful components.such as tables, lists, Scrollpanes, Colourchooser, tabbed pane, etc.
- Further Swing Follows MVC.

**2. Key Components and Concepts:**

- **JFrame:** Represents the main window of a Swing application.
- **JPanel:** A container that can hold other components, such as buttons or labels.
- **JLabel:** Displays text or images.
- **JTextField:** Allows users to input text.
- **JButton:** A button that can be clicked.
- **Layout Managers:** Used to organize and position components within containers.
- **Event Handling:** Allows applications to respond to user interactions, such as button clicks or mouse movements.

**3. Creating a Simple Swing GUI:**

1. **Create a JFrame:** Establish the main window of your application.
2. **Add Components:** Place components like buttons, labels, and text fields within the JFrame.
3. **Set Layout Manager:** Choose a layout manager to arrange the components (e.g., FlowLayout, BorderLayout, GridLayout).
4. **Handle Events:** Implement event listeners to respond to user interactions with the components.
5. **Set Visible:** Make the JFrame visible to the user.

## Limitations of AWT

Here are the limitations of the Abstract Window Toolkit (AWT) in Java:

- **Limited UI Components:**

  AWT provides a basic set of components, lacking advanced elements like tables and trees commonly used in desktop applications.
- **Lack of Customization:**

  AWT components offer limited flexibility for customization, restricting developers from creating visually rich and unique user interfaces.
- **Platform Dependency:**

  AWT relies on the underlying operating system's native GUI components, leading to inconsistencies in appearance and behavior across different platforms. This dependency also restricts extensibility and adaptability.
- **Heavyweight Components:**

  AWT components are considered "heavyweight" because they utilize native system resources. This can lead to performance issues, especially when dealing with complex UIs or a large number of components.
- **Size and Location Restrictions:**
  The maximum size and location of AWT components are limited by integer values, which can be further restricted by the platform. This can pose challenges in arranging components within containers, especially if their bounds exceed platform limits.

## Difference Between AWT and Swing

- AWT (Abstract Window Toolkit) and Swing are both Java toolkits used for creating GUI applications, but they differ significantly in their approach and features.

| AWT | Swing |
|---|---|
| The full form of AWT is the Abstract Window Toolkit. | Swing is a member of Java Foundation Classes (JFC). |
| The AWT components are provided by the java.awt package. | The Swing components are included in the javax.swing package. |
| The AWT components are dependent on the operating system resources. | The components in Swing are not dependent on the operating systems |
| They are not very consistent but integrate with the system on which it is working. It can easily be integrated into various cross-platforms. | They are more consistent and provide smooth interfaces on various platforms such as Linux, MacOS, Windows, etc. |
| The components of AWT are heavily weighted as they use operating system resources. | The components of the swing are lightweight. They do not require the resources of the operating system they are working on. |
| It consists of fewer functionalities than swing toolkits. | It consists of a greater number of functionalities when compared to AWT packages. |
| The AWT component's execution time is higher than the swing. Hence, it is less efficient and produces delays. | Swing execution times are quite fast and are more efficient than AWT packages. |
| It consists of a smaller number of components when compared to Swings in Java. | It consists of a large number of component options to choose from. |
| It requires a large amount of memory to complete execution. | It requires a very small amount of memory space to complete execution as compared to AWT. |
| AWT in Java are slower than swings when | Swings in Java are faster than the AWT |

| | |
|---|---|
| compared on the basis of performance. | toolkits in Java. |
| AWT does not support the MVC pattern. | Swings support the MVC pattern. |

- Swing is built on top of AWT, extending its capabilities and addressing some of its limitations.
- While AWT relies on the underlying operating system for its components, Swing components are rendered independently, providing a consistent look and feel across different platforms.
- Swing also offers a wider range of components and features, making it more suitable for complex GUI applications.

## MVC Architecture In Java

- Model-View-Controller (MVC) is a design pattern that separates an application into three interconnected components: Model, View, and Controller.
- This separation helps in organizing code, making it more modular, and enhancing the reusability and scalability of the application.
- MVC architecture is widely used in Java applications, especially in web development, as it provides a clear structure for building interactive applications with a clean separation of concerns.

### What is MVC Architecture in Java?

The Model-View-Controller (MVC) architecture in Java is a design pattern that provides a structured approach for developing applications. It separates the application's concerns into three main components: the model, the view, and the controller. Each component has a specific role and responsibility within the architecture.

- **Model:**

The model represents the data and business logic of the application. It encapsulates the application's data and provides methods for accessing, manipulating, and updating that data. The model component is independent of the user interface and focuses solely on the application's functionality.

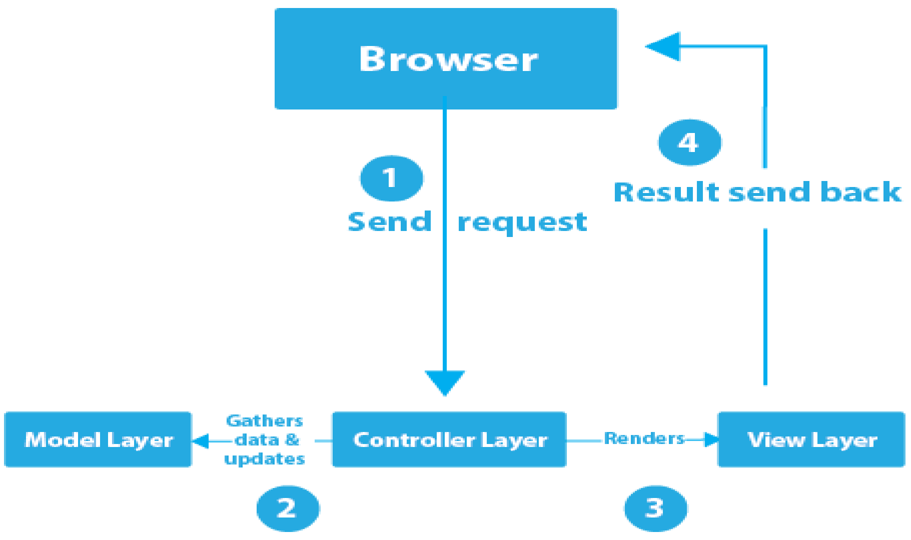
- **View:**

The view is responsible for rendering the user interface and displaying the data to the user. It presents the data from the model to the user in a visually appealing and understandable way. The view component does not contain any business logic but instead relies on the model for data.

- **Controller:**

The controller acts as an intermediary between the model and the view. It handles user input, processes user actions, and updates the model or view accordingly. The controller interprets user actions and triggers the appropriate methods in the model or view. It ensures the separation of concerns by keeping the view and model independent of each other.

In Java programming, the Model comprises basic Java classes that encapsulate data and business logic. The View is responsible for presenting the data to the user interface, while the Controller consists of servlets that handle user requests. This clear separation of components enables the following processing flow for user requests:



**MVC Architecture**

In the context of the server-client architecture, the process of handling a page request can be described as follows:

- A client, typically a web browser, initiates a request and sends it to the server-side controller.
- The controller receives the request and interacts with the model component. It retrieves the necessary data from the model, which may involve processing and manipulating the data as required.
- Once the controller has gathered the requested data, it transfers this data to the view layer.
- The view layer, utilizing the provided data, generates the appropriate output or representation of the requested page. Finally, the generated result is sent back to the client's browser, completing the request-response cycle.

## Advantages of MVC Architecture in Java

The MVC (Model-View-Controller) architecture offers several advantages in Java development:

- **Separation of Concerns:** MVC promotes a clear separation of concerns between the model, view, and controller components. This separation allows for better code organization, improved modularity, and easier maintenance. Developers can focus on specific aspects of the application without impacting other components.
- **Code Reusability:** By separating the concerns into distinct components, code reuse becomes more feasible. The model can be reused across different views, and multiple views can be created for a single model. This reusability reduces duplication of code and improves development efficiency.
- **Simultaneous Development:** MVC allows multiple developers to work simultaneously on different components. The model, view, and controller can be developed independently as long as they adhere to the defined interfaces and communication protocols. This parallel development approach accelerates the overall development process.

- **Flexibility and Extensibility:** MVC provides flexibility by allowing changes in one component without affecting others. For example, modifying the view does not require altering the model or controller. This flexibility also enables the easy addition of new views or controllers to enhance the application’s functionality.

- **Testability:** The separation of concerns in MVC makes unit testing and debugging more manageable. Each component can be independently tested, as they have well-defined responsibilities and interfaces. This promotes comprehensive testing, reduces dependencies, and improves the overall quality of the application.

- **Enhanced User Experience:** With MVC, the view layer handles the presentation of data to the user. This separation allows for greater control over the user interface and enables the use of different views for different platforms or devices. Developers can create responsive and user-friendly interfaces tailored to specific user needs.

- **Support for Maintainability:** MVC simplifies the maintenance of applications over time. Changes can be made to individual components without requiring extensive modifications to the entire system. This modularity enhances maintainability and reduces the risk of introducing bugs or breaking existing functionality.

## Implementation of MVC using Java

To adhere to the MVC pattern in Java implementation, it is necessary to create the following three classes:

- **Employee Class**, will act as model layer
- **EmployeeView Class**, will act as a view layer
- **EmployeeContoller Class**, will act a controller layer

### MVC Architecture Layers

MVC Architecture Layers are:

**1. Model Layer:**

In the MVC design pattern, the Model component serves as the data layer of the application. It encompasses the business logic and maintains the state of the application. The Model object is responsible for retrieving and storing the application’s data in a database. It applies rules and validations to the data, reflecting the core concepts and functionality of the application. Through the Model layer, the application’s data is managed and processed, ensuring adherence to the defined rules and behaviors.

**Let’s consider the following code snippet that creates a which is also the first step to implement MVC pattern.**

```
// class that represents model
public class Employee {
     // declaring the variables
      private String EmployeeName;
      private String EmployeeId;
      private String EmployeeDepartment;
     // defining getter and setter methods
      public String getId() {
         return EmployeeId;
      }

      public void setId(String id) {
         this.EmployeeId = id;
      }

      public String getName() {
         return EmployeeName;
      }

      public void setName(String name) {
         this.EmployeeName = name;
      }

      public String getDepartment() {
            return EmployeeDepartment;
          }

      public void setDepartment(String Department) {
            this.EmployeeDepartment = Department;
          }

   }
```

The above code simply consists of getter and setter methods to the Employee class.

**2. View Layer**

As the name implies, the View component in the MVC pattern is responsible for presenting the data obtained from the Model. It represents the visual representation or user interface of the application. The View layer generates the output of the application and communicates it to the client. The requested data is fetched from the Model layer by the Controller and passed to the View for rendering and display to the user or client. The View component ensures that the data is presented in a format that is suitable for the user interface or output requirements of the application.

**Let’s take an example where we create a view using the EmployeeView class.**

```
 // class which represents the view

 public class EmployeeView {

      // method to display the Employee details

 public void printEmployeeDetails (String EmployeeName, String EmployeeId, String
EmployeeDepartment){

          System.out.println("Employee Details: ");
          System.out.println("Name: " + EmployeeName);
          System.out.println("Employee ID: " + EmployeeId);
```

```
        System.out.println("Employee Department: " + EmployeeDepartment);
      }
    }
```

**3. Controller Layer**

In the MVC architecture, the Controller layer receives user requests from the View layer and handles them, including necessary validations and logic. It serves as the intermediary between the Model and View components. The Controller processes the user requests and forwards them to the Model layer for data processing. Once the requested data is processed by the Model, it is returned to the Controller. The Controller then transfers the data to the appropriate View, where it is displayed to the user. The Controller orchestrates the flow of data between the Model and View, ensuring the proper handling and presentation of the application’s functionality.

Let’s consider the following code snippet that creates the controller using the EmployeeController class.

```
// class which represents the controller
public class EmployeeController {

     // declaring the variables model and view
      private Employee model;
      private EmployeeView view;
     // constructor to initialize
      public EmployeeController(Employee model, EmployeeView view) {
         this.model = model;
         this.view = view;
      }
     // getter and setter methods
      public void setEmployeeName(String name){
         model.setName(name);
      }
      public String getEmployeeName(){
         return model.getName();
      }
      public void setEmployeeId(String id){
         model.setId(id);
      }
      public String getEmployeeId(){
         return model.getId();
      }

      public void setEmployeeDepartment(String Department){
            model.setDepartment(Department);
      }
         public String getEmployeeDepartment(){
            return model.getDepartment();
      }
      // method to update view
      public void updateView() {
         view.printEmployeeDetails(model.getName(), model.getId(), model.getDepartment());
```

```
        }
    }
```

The Model-View-Controller (MVC) architecture in Java provides a structured and modular approach to developing applications. It separates concerns into three components: the Model, View, and Controller. The Model represents the data and business logic, the View handles the user interface and presentation of data, and the Controller acts as the intermediary, processing user input and managing the interaction between the Model and View. MVC offers advantages such as separation of concerns, code reusability, flexibility, and testability, ultimately leading to more maintainable and scalable Java applications.

## Applet Basics

- An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.
- There are some important differences between an applet and a standalone Java application, including the following –
- An applet is a **Java class** that extends the **java.applet.Applet class**.
- **A main()** method is not invoked on an applet, and an applet class will not define **main().**
- Applets are designed to be embedded within an **HTML** page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A **JVM** is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

### 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.

### Life cycle of an Applet / An Applet Skeleton

Most applets override these four methods. These four methods forms Applet lifecycle.

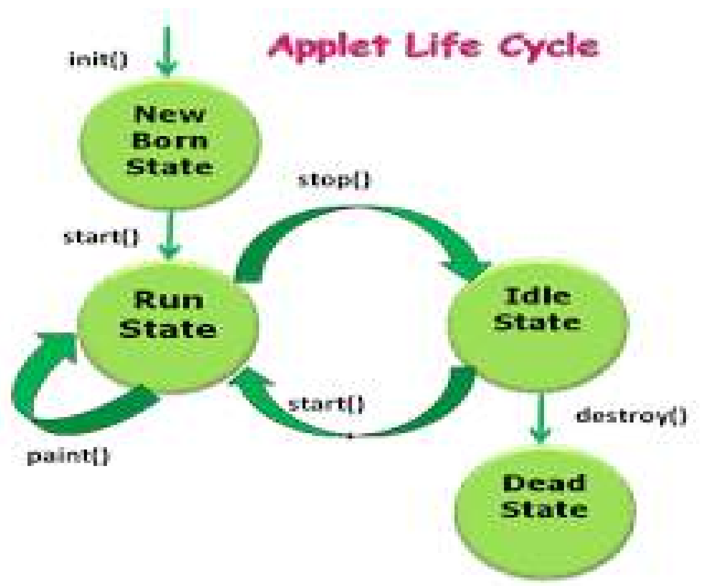
- **init() :**
  - init() is the first method to be called.
  - This is where variable are initialized.
  - This method is called only once during the runtime of applet.

- **start() :**
  - start() method is called after init().
  - This method is called to restart an applet after it has been stopped.

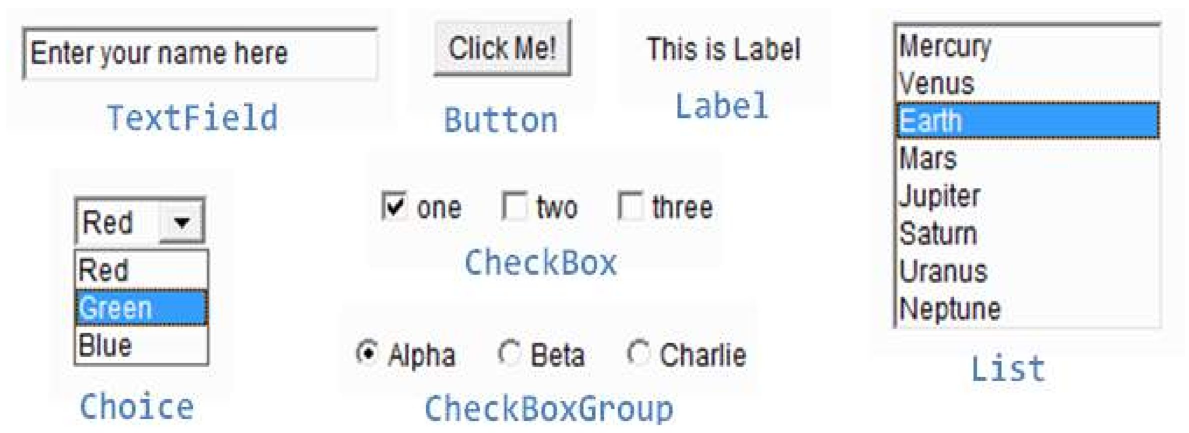
- **stop() :**
  - stop() method is called to suspend thread that does not need to run when applet is not visible.

- **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:



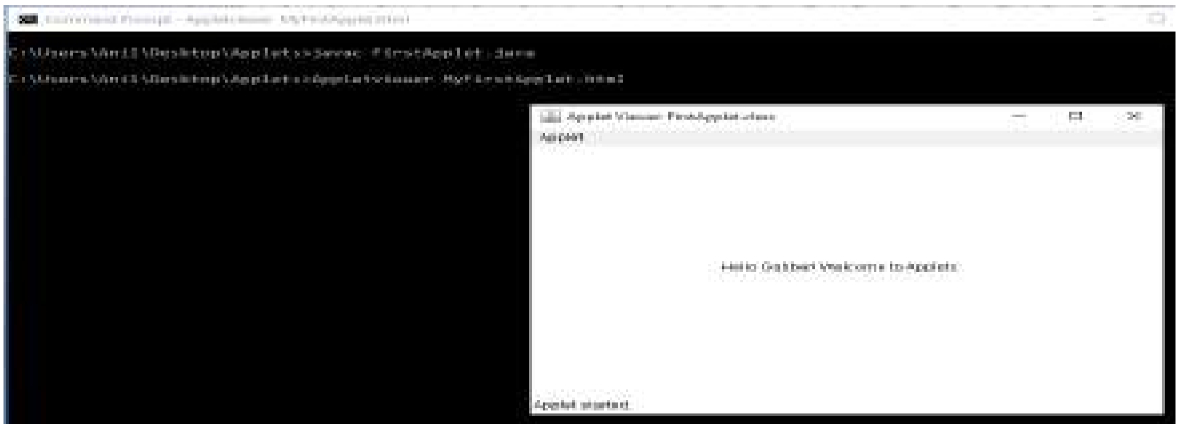
To create an applet first write the program and Save the file with name FirstApplet.java

```
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>
```

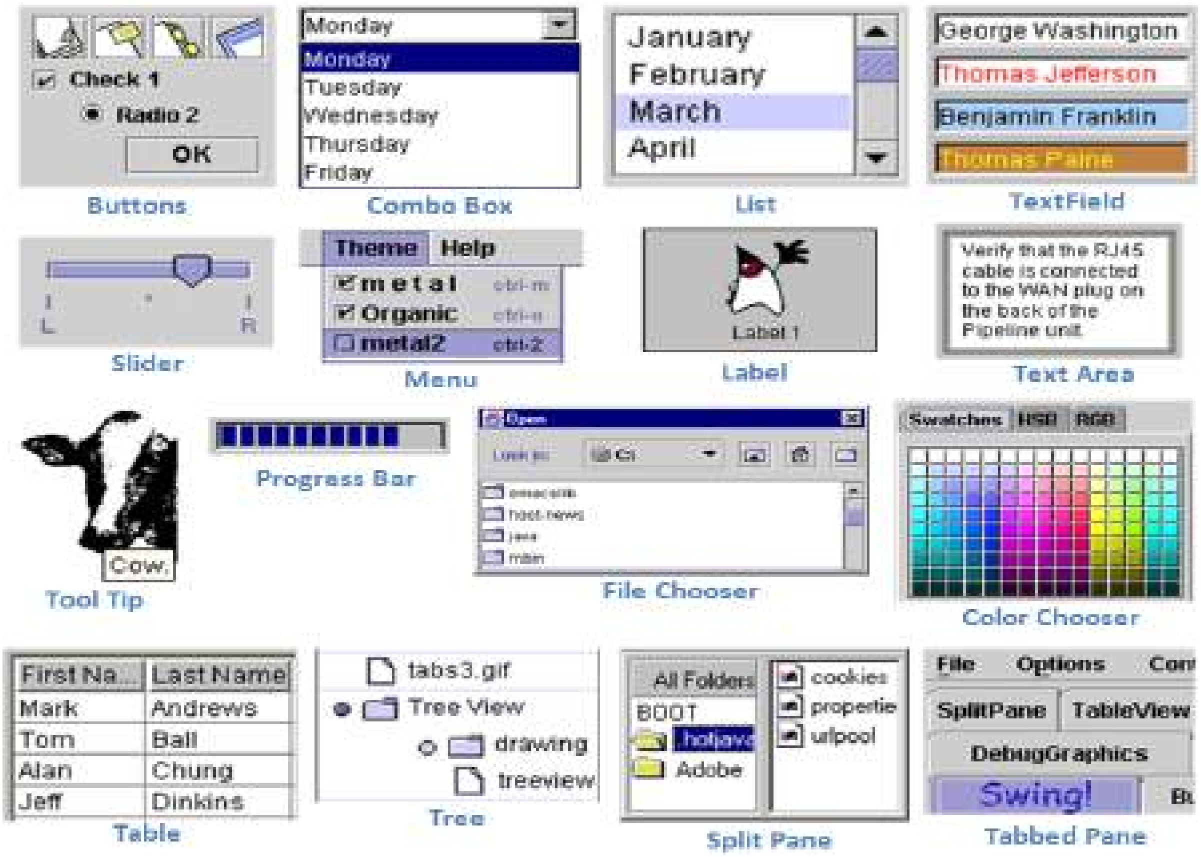
To run applet programs



```
C:\Users\Anil\Desktop\Applets>javac FirstApplet.java

C:\Users\Anil\Desktop\Applets>appletviewer MyFirstApplet.html
```

```
Applet Viewer: FirstApplet.class
Applet

Hello Gabbar Welcome to Applets

Applet started.
```

## A Simple Swing Application, Applets



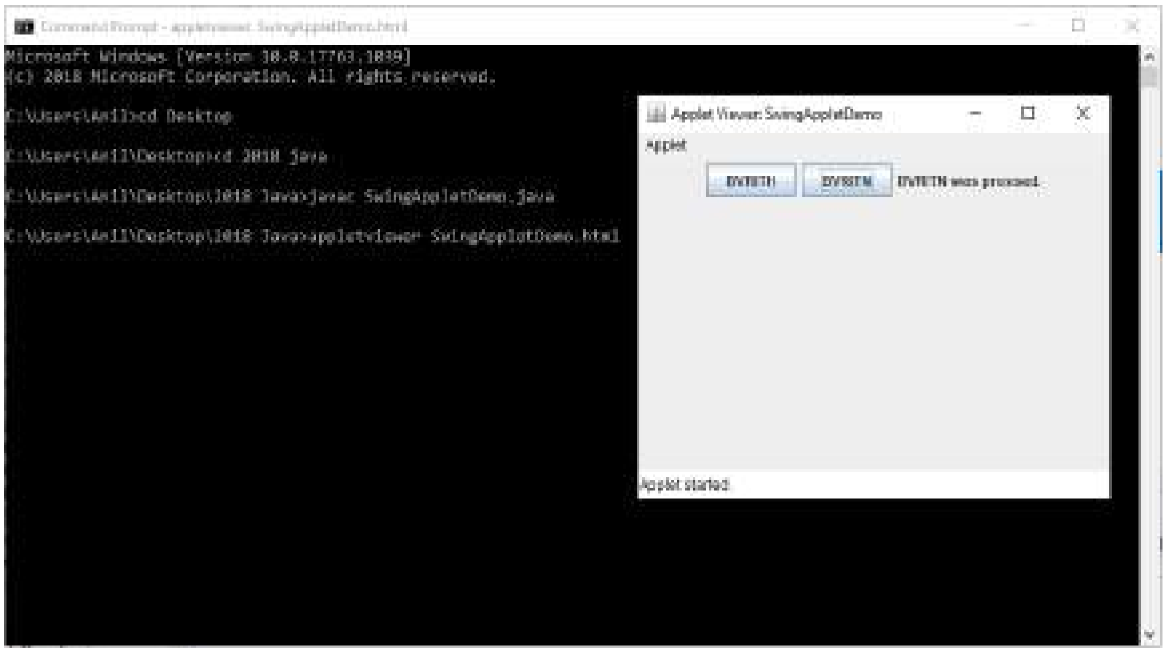
## Create a Swing Applet

- 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.invokeAndWait(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

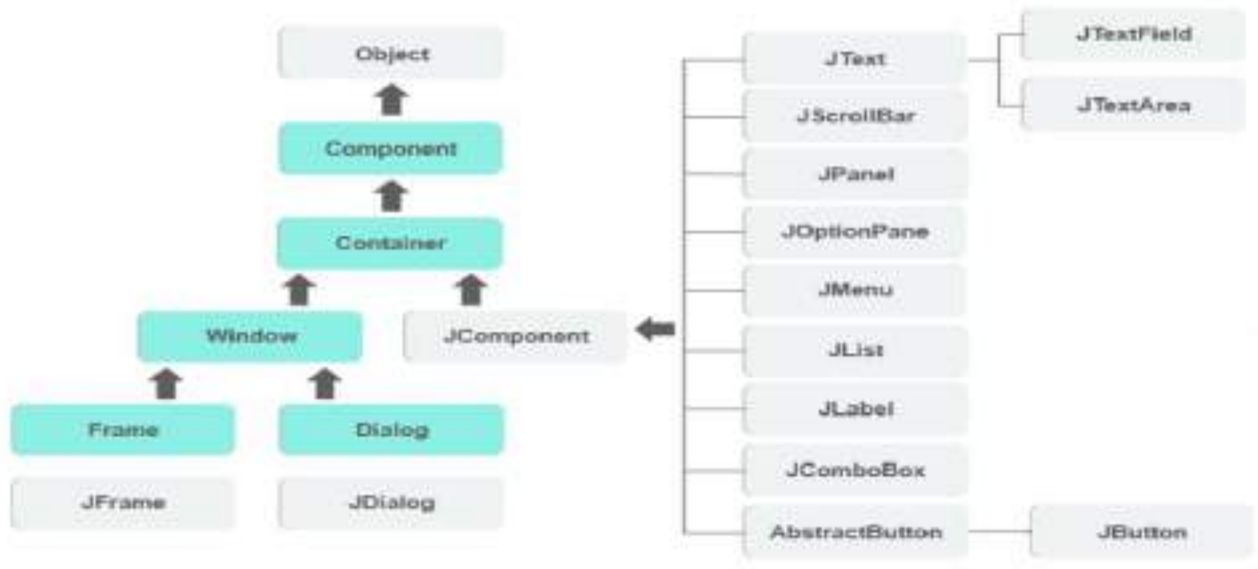


```
Microsoft Windows [Version 10.0.17763.1039]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Anil>cd Desktop

C:\Users\Anil\Desktop>cd 2018 java

C:\Users\Anil\Desktop\2018 Java>javac SwingAppletDemo.java

C:\Users\Anil\Desktop\2018 Java>appletviewer SwingAppletDemo.html
```

Command Prompt - appletviewer SwingAppletDemo.html
Applet Viewer: SwingAppletDemo
Applet
BVRITH
BVRITN
BVRITN was pressed.
Applet started.

## Swing Components and Containers

- Swing in Java provides a set of components for building graphical user interfaces (GUIs).
- These components can be broadly categorized into containers and components.



**Creating GUI using Java Swing**

### Containers

- Containers are special components that can hold other components.
- They provide the structure for organizing and managing the layout of the GUI.
- A container provides a space where a component can be located.
- A Container in AWT is a component itself and it provides the capability to add a component to itself.
- Swing offers several types of containers:

**Top-level containers:**

These are the main windows of an application.

- JFrame: A window with a title bar and border, commonly used as the main window for applications.
- JDialog: A sub-window used for temporary interactions, such as displaying error messages or prompts.
- JWindow: A simple window without a title bar or border.
- JApplet: A container designed to run within a web browser (deprecated in newer Java versions).

**Content pane containers:**

These are used to group and organize components within a top-level container.

- JPanel: A generic container for grouping components, often used for layout management.
- JLayeredPane: A container that allows components to be stacked on top of each other.

**Menu containers:**
These are used to create menus and menu items.

- JMenuBar: A bar that holds menus.
- JMenu: A menu that appears in the menu bar.
- JMenuItem: An item within a menu that can be selected.

## Components

Components are the individual elements that make up the GUI, such as buttons, labels, and text fields. Swing provides a wide range of components:

- JLabel: Displays text or images.
- JButton: A clickable button.
- JTextField: Allows the user to enter a single line of text.
- JTextArea: Allows the user to enter multiple lines of text.
- JCheckBox: A checkable box for selecting options.
- JRadioButton: A selectable button within a group.
- JComboBox: A drop-down list of options.
- JList: A list of items that can be selected.
- JTable: Displays data in a tabular format.
- JScrollPane: Provides scroll bars for components that exceed the visible area.

Containers and components are used together to create the structure and functionality of a Swing GUI. Containers provide the framework for organizing components, while components provide the interactive elements that the user interacts with.

## Layout Manager in Java

- The layout will specify the format or the order in which the components have to be placed on the container.
- Layout Manager may be a class or component that's responsible for rearranging the components on the container consistent with the required layout.
- A layout manager automatically arranges your controls within a window by using some algorithm.
- Each Container object features a layout manager related to it.
- A layout manager is an instance of any class implementing the LayoutManager interface.
- The layout manager is about by the setLayout( ) method.
- If no call to setLayout( ) is formed, the default layout manager is employed.
- Whenever a container is resized (or sized for the primary time), the layout manager is employed to position each of the components within it.

- The setLayout( ) method has the subsequent general form:

  **void setLayout(LayoutManager layoutObj)**

- Here, layoutObj may be a regard to the specified layout manager.

- If you want to manually disable the layout manager and position components, pass null for layoutObj.
- If you do this, you'll get to determine the form and position of every component manually, using the setBounds() method defined by Component.

## Types of Layout Managers

AWT package provides the following types of Layout Managers:

1. Flow Layout
2. Border Layout
3. Card Layout
4. Grid Layout
5. GridBag Layout
6. Box Layout
7. Spring Layout

### Flow Layout

- This layout will display the components from left to right, from top to bottom.
- The components will always be displayed in the first line and if the first line is filled, these components are displayed on the next line automatically.
- In this Layout Manager, initially, the container assumes 1 row and 1 column of the window.
- Depending on the number of components and size of the window, the number of rows and columns count is decided dynamically.

**Note:** If the row contains only one component, then the component is aligned in the center position of that row.

**Creation of Flow Layout**

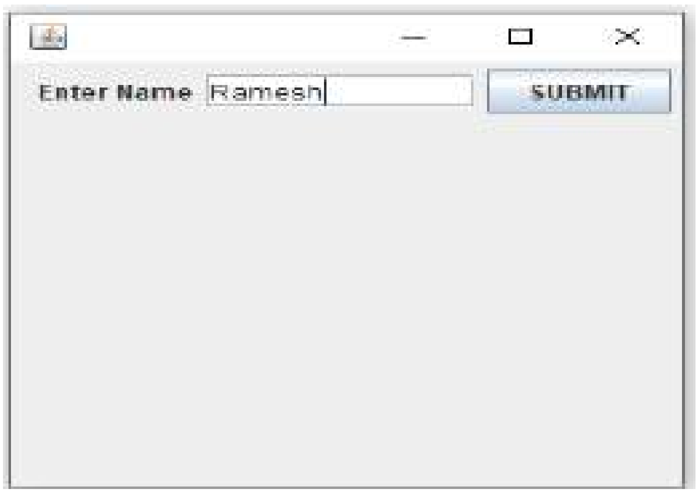
```
FlowLayout f1 = new FlowLayout();
FlowLayout f1 = new FlowLayout(int align);
FlowLayout f1 = new FlowLayout(int align, int hgap, int vgap);
```

Example to demonstrate Flow Layout in Java

```
import java.awt.*;
import javax.swing.*;
public class FlowLayoutDemo
{
JFrame f;
FlowLayoutDemo ()
{
f = new JFrame ();
JLabel l1 = new JLabel ("Enter Name");
JTextField tf1 = new JTextField (10);
JButton b1 = new JButton ("SUBMIT");
f.add (l1);
f.add (tf1);
```

```
    f.add (b1);
    f.setLayout (new FlowLayout (FlowLayout.RIGHT));
    //setting flow layout of right alignment
    f.setSize (300, 300);
    f.setVisible (true);
    }
    public static void main (String[]args)
    {
    new FlowLayoutDemo ();
    }
    }
```

Output:



## Border Layout

- This layout will display the components along the border of the container.
- This layout contains five locations where the component can be displayed.
- Locations are North, South, East, west, and Center.
- The default region is the center.
- The above regions are the predefined static constants belonging to the BorderLayout class. Whenever other regions' spaces are not in use, automatically container is selected as a center region default, and the component occupies the surrounding region's spaces of the window, which damages the look and feel of the user interface.

**Creation of BorderLayout**

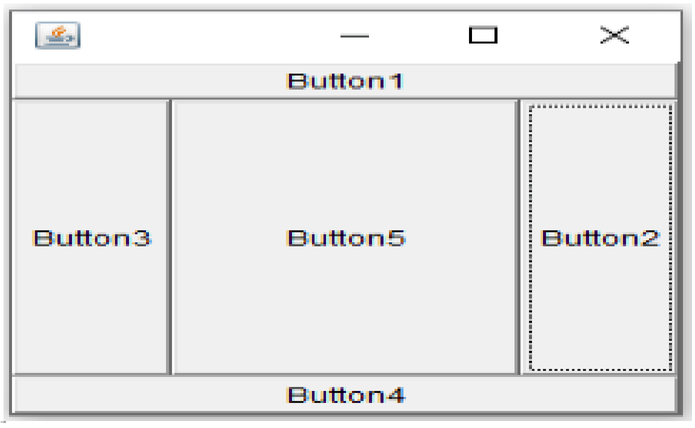
**BorderLayout bl = new BorderLayout();**
**BorderLayout bl = new BorderLayout(int vgap, int hgap);**

Example to demonstrate Border Layout in Java

```
import java.awt.*;
public class BorderLayoutDemo
{
public static void main (String[]args)
{
```

```
Frame f1 = new Frame ();
f1.setSize (250, 250);
Button b1 = new Button ("Button1");
Button b2 = new Button ("Button2");
Button b3 = new Button ("Button3");
Button b4 = new Button ("Button4");
Button b5 = new Button ("Button5");
f1.add (b1, BorderLayout.NORTH);
f1.add (b2, BorderLayout.EAST);
f1.add (b3, BorderLayout.WEST);
f1.add (b4, BorderLayout.SOUTH);
f1.add (b5);
f1.setVisible (true);
}
}
```

Output:



**In the above application, the frame class contains two types of add methods:**

1. **add(component):** This method default aligns components in the center region.
2. **add(component, region name):** Internally add method handovers a given component (i.e., object) to the container and container user peer class of that component to communicate with the OS library, and then the created component is aligned on the window.

## Card Layout

- A card layout represents a stack of cards displayed on a container.
- At a time, only one card can be visible, each containing only one component.

**Creation of Card Layout in Java**

**CardLayout cl = new CardLayout();**
**CardLayout cl = new CardLayout(int hgap, int vgap);**

**To add the components in CardLayout, we use the add method:**
**add("Cardname", Component);**

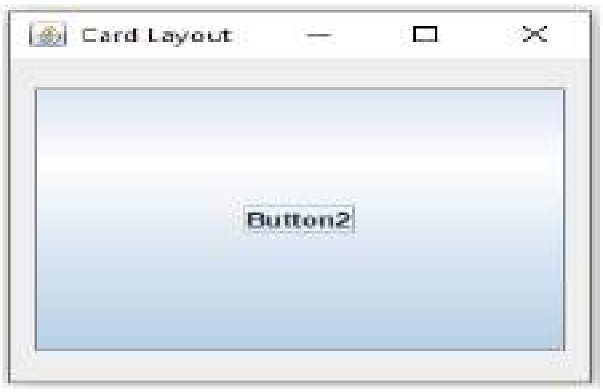
**Methods of CardLayout in Java**

1. **first(Container):** It is used to flip to the first card of the given container.
2. **last(Container):** It is used to flip to the last card of the given container.
3. **next(Container):** It is used to flip to the next card of the given container.
4. **previous(Container):** It is used to flip to the previous card of the given container.
5. **show(Container, cardname):** It is used to flip to the specified card with the given name.

Example to demonstrate Card Layout in Java

```
import java.awt.*;
import javax.swing.*;
import javax.swing.JButton;
import java.awt.event.*;
public class CardLayoutDemo extends JFrame implements ActionListener
{
JButton b1, b2, b3, b4, b5;
CardLayout cl;
Container c;
CardLayoutDemo ()
{
b1 = new JButton ("Button1");
b2 = new JButton ("Button2");
b3 = new JButton ("Button3");
b4 = new JButton ("Button4");
b5 = new JButton ("Button5");
c = this.getContentPane ();
cl = new CardLayout (10, 20);
c.setLayout (cl);
c.add ("Card1", b1);
c.add ("Card2", b2);
c.add ("Card3", b3);
b1.addActionListener (this);
b2.addActionListener (this);
b3.addActionListener (this);
setVisible (true);
setSize (400, 400);
setTitle ("Card Layout");
setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
}
public void actionPerformed (ActionEvent ae)
{
cl.next (c);
}
public static void main (String[]args)
{
new CardLayoutDemo ();
}
}
```

Output:



## Grid Layout

- The layout will display the components in the format of rows and columns statically.
- The container will be divided into a table of rows and columns.
- The intersection of a row and column cell and every cell contains only one component, and all the cells are of equal size.
- According to Grid Layout Manager, the grid cannot be empty.

**Creation of Grid Layout Manager in Java**

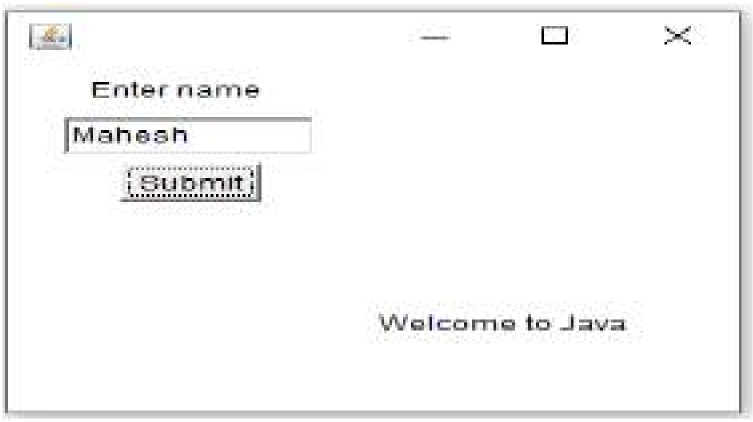
**GridLayout gl = new GridLayout(int rows, int cols);**
**GridLayout gl = new GridLayout(int rows, int cols, int vgap, int hgap);**

Example to demonstrate Grid Layout in Java

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutDemo
{
public static void main (String[]args)
{
Frame f1 = new Frame ();
f1.setSize (250, 250);
GridLayout ob = new GridLayout (2, 2);
f1.setLayout (ob);
Panel p1 = new Panel ();
Label l1 = new Label ("Enter name");
TextField tf = new TextField (10);
```

```
    Button b1 = new Button ("Submit");
    p1.add (l1);
    p1.add (tf);
    p1.add (b1);
    f1.add (p1);
    Panel p2 = new Panel ();
    f1.add (p2);
    Panel p3 = new Panel ();
    f1.add (p3);
    Label l2 = new Label ("Welcome to Java");
    f1.add (l2);
    f1.setVisible (true);
    }
    }
```

Output:



## Grid Bag Layout

- In GridLayout manager, there is no grid control, i.e., inside a grid, we cannot align the component in a specific position.
- To overcome this problem, we have an advanced Layout Manager, i.e., Grid Bag Layout Manager.
- This layout is the most efficient layout that can be used for displaying components.
- In this layout, we can specify the location, size, etc. In this Layout manager, we need to define grid properties or constraints for each grid.
- Based on grid properties, the layout manager aligns a component on the grid, and we can also span multiple grids as per the requirement.
- Grid properties are defined using the GridBagConstraints class.

**Creation of GridBagLayout:**

**GridBagLayout gbl = new GridBagLayout();**

**Note:** We can specify the location (or) the size with the help of GridBagConstraints.

**Properties of GridBagConstraints**:

1. **gridx, gridy:** For defining x and y coordinate values, i.e., specifying grid location.
2. **gridwidth, grid height:** For defining the number of grids to span a document.
3. **fill:** Used whenever component size is greater than the area (i.e., VERTICAL or HORIZONTAL).
4. **ipadx, ipady:** For defining the width and height of the components, i.e., for increasing component size.
5. **insets:** For defining the surrounding space of the component, i.e., top, left, right, bottom.
6. **anchor:** Used whenever component size is smaller than area, i.e., where to place in a grid.

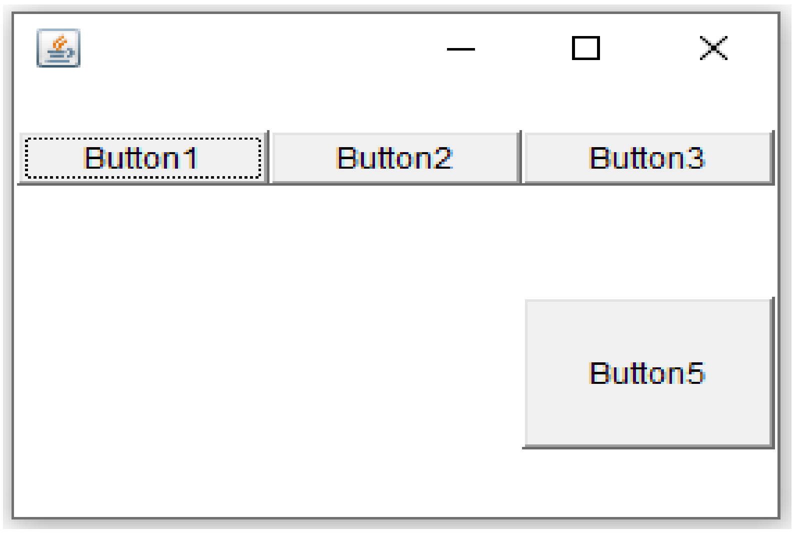
   **FIRST_LINE_START PAGE_START FIRST_LINE_END**
   **LINE_START CENTER LINE_END**
   **LAST_LINE_START PAGE_END LAST_LINE_END**

   The above format is equal to one grid, so we can specify components in any grid position.
7. **weightx, weighty:** These are used to determine how to distribute space among columns(weightx) and among rows(weighty), which is important for specifying resizing behavior.

Example to demonstrate GridBag Layout Manager in Java

```
import java.awt.*;
public class GridBagLayoutDemo
{
public static void main (String[]args)
{
Frame f1 = new Frame ();
f1.setSize (250, 250);
GridBagLayout gb = new GridBagLayout ();
f1.setLayout (gb);
GridBagConstraints gc = new GridBagConstraints ();
Button b1 = new Button ("Button1");
Button b2 = new Button ("Button2");
Button b3 = new Button ("Button3");
gc.fill = GridBagConstraints.HORIZONTAL;
gc.weightx = 0.5;
gc.weighty = 0.5;
gc.gridx = 0;
gc.gridy = 0;
f1.add (b1, gc);
gc.gridx = 1;
gc.gridy = 0;
f1.add (b2, gc);
gc.gridx = 2;
gc.gridy = 0;
f1.add (b3, gc);
Button b4 = new Button ("Button4");
gc.gridx = 0;
gc.gridy = 1;
gc.gridwidth = 3;
gc.ipady = 40;
Button b5 = new Button ("Button5");
gc.gridx = 2;
gc.gridy = 3;
gc.insets = new Insets (10, 0, 10, 0);
f1.add (b5, gc);
f1.pack ();
f1.setVisible (true);
}
}
```

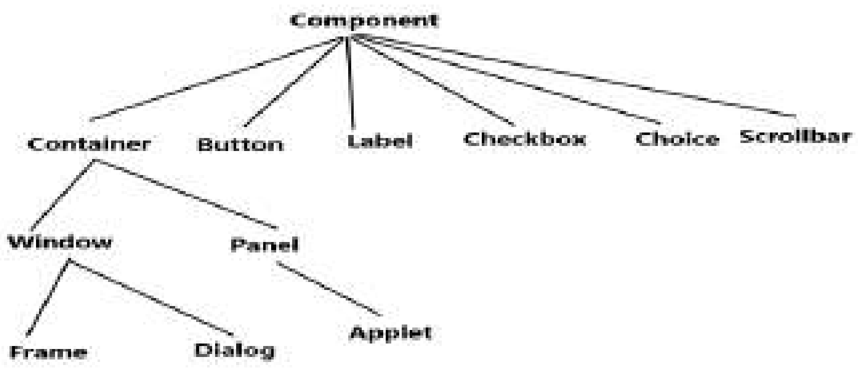
Output:



## Simple Application using AWT :

**AWT** stands for **Abstract Window Toolkit**. It is a platform dependent API for creating Graphical User Interface (GUI) for java programs.

**Why AWT is platform dependent?** Java AWT calls native platform (Operating systems) subroutine for creating components such as textbox, checkbox, button etc. For example an AWT GUI having a button would have a different look and feel across platforms like windows, Mac OS & Unix, this is because these platforms have different look and feel for their native buttons and AWT directly calls their native subroutine that creates the button. In simple, an application build on AWT would look like a windows application when it runs on Windows, but the same application would look like a Mac application when runs on Mac OS.



**AWT Component Hierarchy**

## Example:

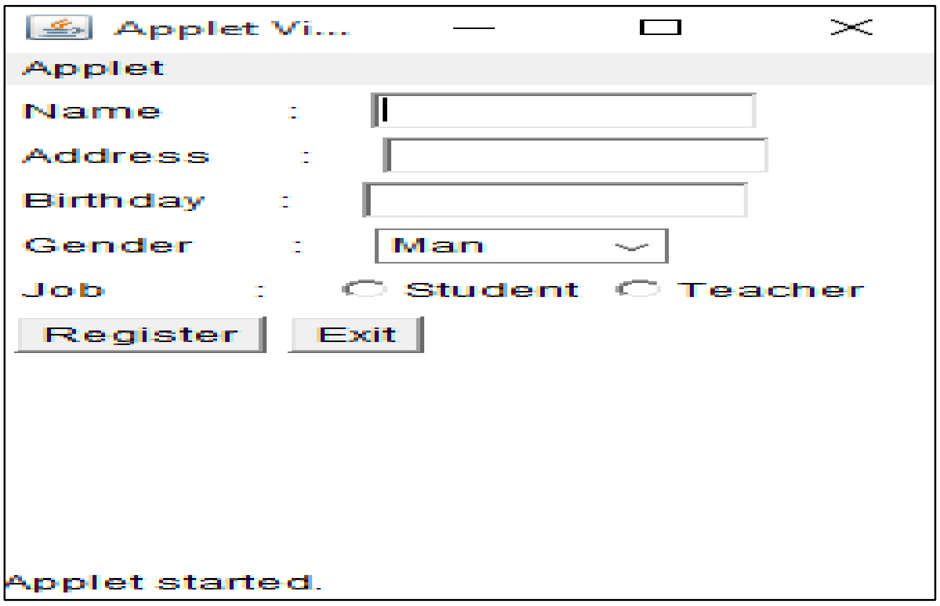
### File Name : Tes.java

```
import java.awt.*;
public class Tes extends java.applet.Applet
{
   public void init()
   {

      setLayout(new FlowLayout(FlowLayout.LEFT));

      add(new Label("Name          :"));
      add(new TextField(10));

      add(new Label("Address       :"));
      add(new TextField(10));

      add(new Label("Birthday      :"));
      add(new TextField(10));


      add(new Label("Gender        :"));
      Choice gender = new Choice();
      gender.addItem("Man");
      gender.addItem("Woman");
      Component add = add(gender);

      add(new Label("Job           :"));
      CheckboxGroup job = new CheckboxGroup();
      add(new Checkbox("Student", job, false));
      add(new Checkbox("Teacher", job, false));

      add(new Button("Register"));
      add(new Button("Exit"));
      }
}
```

**File Name : Tes.html**

```
<html>
<head><title>Register</title></head>
<body>
<applet code="Tes.class" width=230 height=300></applet>
</body>
</html>
```

**Output:**



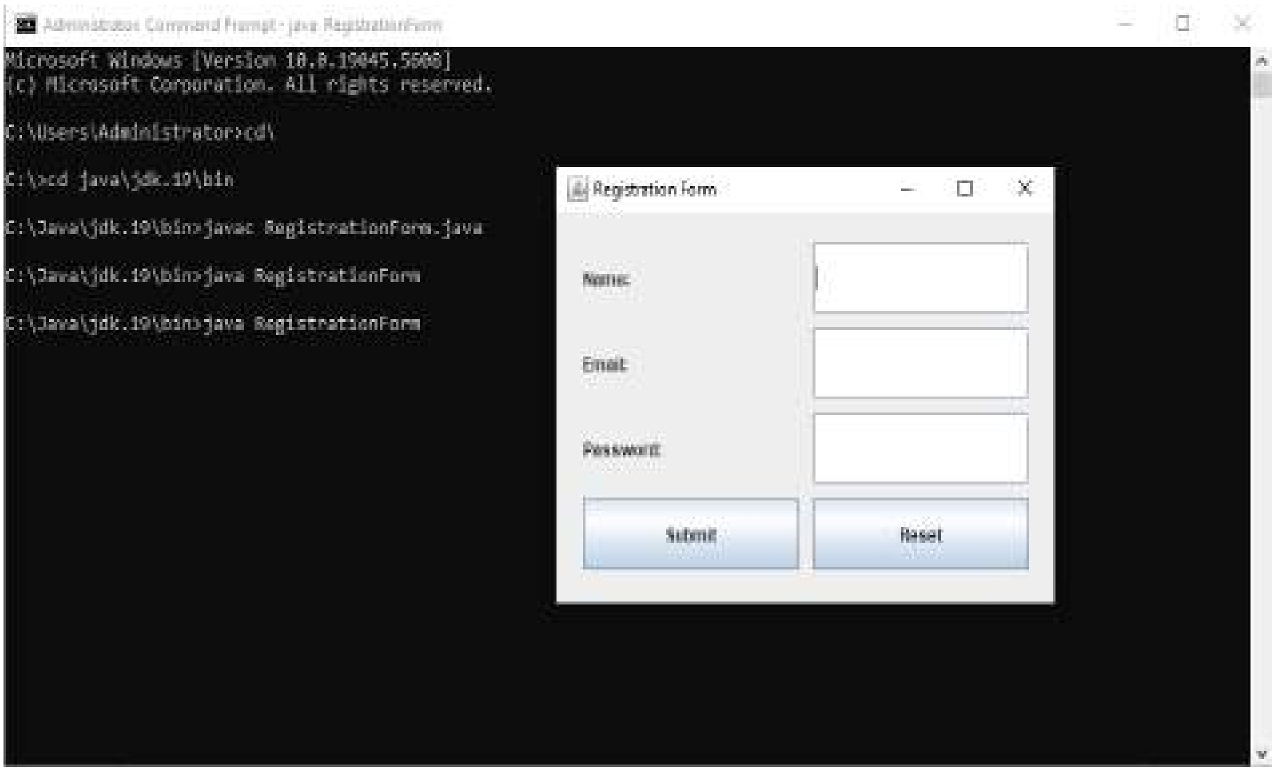
## Simple Application using Swing:

**File Name : RegistrationForm.java**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class RegistrationForm extends JFrame implements ActionListener {

   private final JLabel nameLabel, emailLabel, passwordLabel;
   private final JTextField nameField, emailField;
   private final JPasswordField passwordField;
   private final JButton submitButton, resetButton;

   public RegistrationForm() {
      setTitle("Registration Form");
      setSize(400, 300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLocationRelativeTo(null); // Center the frame
```

```
        // Panel to hold components
        JPanel panel = new JPanel(new GridLayout(4, 2, 10, 10));
        panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));

        // Labels
        nameLabel = new JLabel("Name:");
        emailLabel = new JLabel("Email:");
        passwordLabel = new JLabel("Password:");

        // Text fields
        nameField = new JTextField();
        emailField = new JTextField();
        passwordField = new JPasswordField();

        // Buttons
        submitButton = new JButton("Submit");
        resetButton = new JButton("Reset");

        // Add action listeners to buttons
        submitButton.addActionListener(this);
        resetButton.addActionListener(this);

        // Add components to panel
        panel.add(nameLabel);
        panel.add(nameField);
        panel.add(emailLabel);
        panel.add(emailField);
        panel.add(passwordLabel);
        panel.add(passwordField);
        panel.add(submitButton);
        panel.add(resetButton);

        // Add panel to frame
        add(panel);

        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == submitButton) {
            String name = nameField.getText();
            String email = emailField.getText();
            String password = new String(passwordField.getPassword());

            // Basic validation
            if (name.isEmpty() || email.isEmpty() || password.isEmpty()) {
                JOptionPane.showMessageDialog(this, "Please fill in all fields.", "Error",
JOptionPane.ERROR_MESSAGE);
            } else {
                JOptionPane.showMessageDialog(this, "Registration Successful!\nName: " + name
+ "\nEmail: " + email, "Success", JOptionPane.INFORMATION_MESSAGE);
            }
        } else if (e.getSource() == resetButton) {
            nameField.setText("");
            emailField.setText("");
            passwordField.setText("");
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(RegistrationForm::new);
    }
}
```

**Output:**



```
Administrator: Command Prompt - java RegistrationForm
Microsoft Windows [Version 10.0.19045.5608]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>cd\

C:\>cd java\jdk.19\bin

C:\Java\jdk.19\bin>javac RegistrationForm.java

C:\Java\jdk.19\bin>java RegistrationForm

C:\Java\jdk.19\bin>java RegistrationForm
```

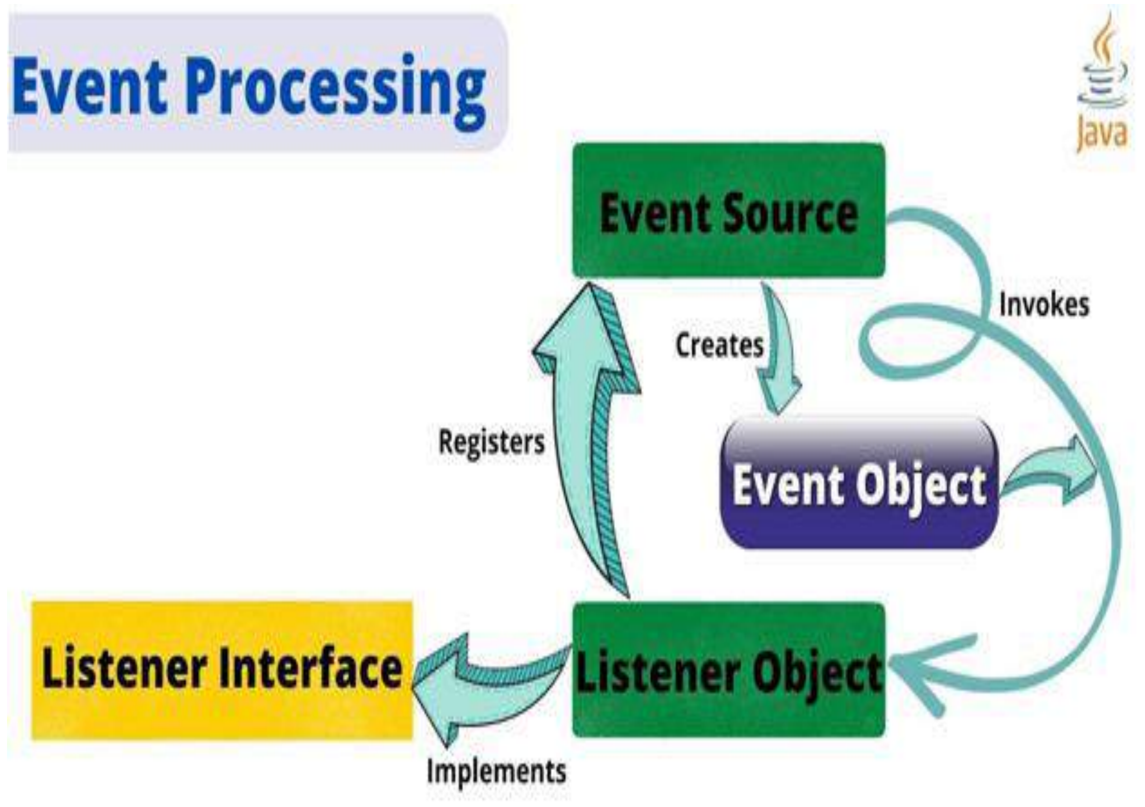
## Event Handling

### Introduction

- In Java, event handling is crucial for developing interactive applications, especially in GUI programming.
- The **Delegation Event Model** is a powerful mechanism that enables efficient event-driven programming.
- It follows a **producer-consumer** approach, where an event source generates events, and listeners handle them.
- This model improves code organization, enhances reusability, and simplifies event processing. In this blog, we will explore the **Delegation Event Model.**

## Event Processing in Java

- Java supports event processing and supports the API used to develop the Desktop application, i.e., Abstract Window Toolkit.
- Let's understand it graphically.



The delegation Model is the modern event processing approach, which has been available in Java since Java 1.1. It defines a standard and convenient mechanism to generate and process events.

## What is the Delegation Model?

- The delegation model is a programming design pattern that is used to handle events and event-driven programming in graphical user interfaces (GUIs).
- The delegation model works by separating the concerns of event generation and event handling, allowing for greater modularity and flexibility in GUI programming.
- In the delegation model, the objects responsible for generating events, such as buttons or menu items, are referred to as event sources.
- When an event occurs on an event source, the source creates an event object that encapsulates information about the event, such as its type and any associated data.
- The delegation model has several advantages over other event-handling models.
- It allows for greater modularity and flexibility in GUI programming, as the event generation and event handling are separated into distinct objects.
- This makes it easier to change the behavior of a GUI component without affecting other parts of the program.
- Additionally, the delegation model supports multiple event listeners for a single event source, allowing for greater customization and flexibility in handling events.
- Overall, the delegation model is a powerful and flexible approach to handling events in GUI programming.

**Delegation Event Model**

It has Sources and Listeners.

- **Source:** Event sources are objects that generate events. They are responsible for detecting specific occurrences or changes in the application and notifying interested listeners about these events. An event source could be a GUI component like a button, text field, or menu item, or it could be any object that generates events based on user interactions or other application-specific triggers.

- **Listeners:** Event listeners are objects that "listen" to specific events generated by event sources. When an event occurs, the corresponding listener's method(s) are invoked to handle that event.

Event Model includes the following three components:

- **Events**
- **Events Sources**
- **Events Listeners**

**1. Events**

- An event refers to an action that occurs within a graphical user interface (GUI), such as a button click, a key press, or a mouse movement.
- When an event is triggered, it is typically handled by an event listener or handler, which is registered to the graphical component that generates the event.

**2. Event Sources**

- An event source is an object that generates an event. It is typically a graphical user interface (GUI) component, such as a button, a text field, or a menu item, but it can also be other types of objects that generate events, such as timers or sockets.
- When an event is triggered on an event source, the event is encapsulated in an event object and passed to all the registered event listeners or handlers.
- The event source is responsible for notifying the event listeners or handlers of the event and providing them with the information they need to handle the event.

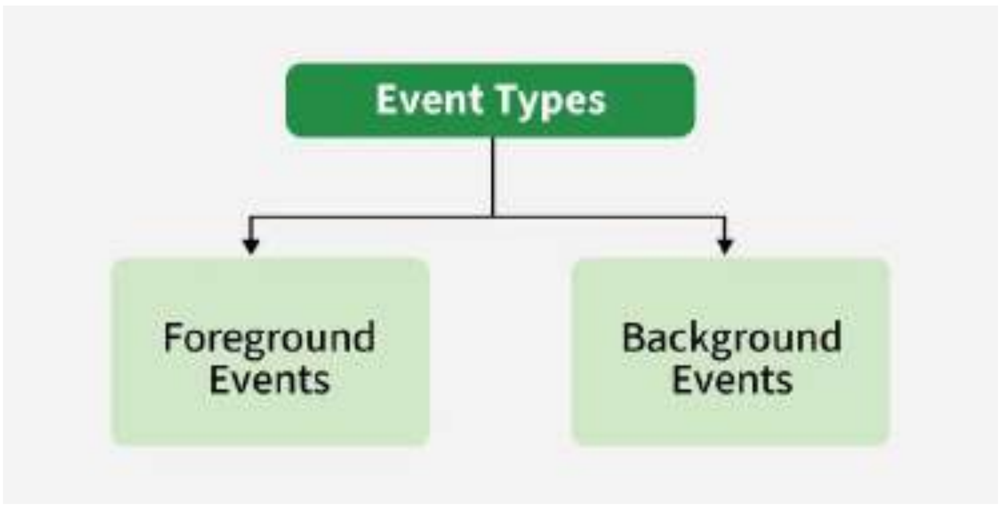
**3. Event Listeners**

- An event listener is an object that is registered to an event source and is responsible for handling events that are generated by that source.
- When an event occurs on the event source, the event listener is notified and performs the necessary actions to respond to the event.
- An event listener in Java is typically implemented as a class that implements a specific event listener interface, such as ActionListener, MouseListener, or KeyListener.
- Each interface specifies a set of methods that the event listener must implement to handle the corresponding type of event.
- For example, the ActionListener interface requires the implementation of a single method called actionPerformed, which is called when an action event occurs on the event source.

## Types of Events

- There are several types of events in the delegation event model, each with its own corresponding listener interfaces and methods for handling the events.

- Some common types of events in Java are:



**1. The Foreground Events:**

- Foreground events typically refer to events that require immediate attention and are directly related to the user interaction or user interface.
- These events are important for the applications functioning and generally require user input or trigger visible changes in the interface of the application.
- Examples of foreground events include mouse movements, key presses, or button clicks.

**2. The Background Events :**

- Background events occur in the background or behind the scenes, usually without requiring direct user interaction.
- These events may include tasks like network communication, data processing, or automated system operations.
- Background events are typically handled independently of the user interface.
- They may not have immediate visible effects but play a significant role in the overall functionality of an application.

**Registering the Source With Listener**

Registering the source with a listener is a crucial step in the Java Delegation Event Model. It establishes a connection between an event source and a listener, allowing the listener to receive and handle events generated by the source.

The following are the steps for registering the source with a listener:-

1. **Create the Event Listener:** Define a class that implements the appropriate listener interface for the type of event you want to handle. For example, if you're dealing with GUI events, you might implement the ActionListener interface.

2. **Implement Listener Methods:** Implement the methods specified by the listener interface within the listener class. These methods will be called when the associated event occurs.

3. **Create the Event Source:** Instantiate the event source object. This could be a GUI component like a button, text field, or menu item, depending on the type of event you're handling.

4. **Instantiate the Listener:** Create an instance of the listener class you implemented in step 1.

5. **Register the Listener with the Event Source:** Use a method provided by the event source to register the listener. This method varies depending on the event source and the type of event you're handling.

## Event Classes and Listener Interfaces

Java provides a variety of event classes and corresponding listener interfaces. Below table demonstrates the most commonly used event classes and their associated listener interfaces:

| Event Class | Listener Interface | Description |
|---|---|---|
| ActionEvent | ActionListener | An event that indicates that a component-defined action occurred like a button click or selecting an item from the menu-item list. |
| AdjustmentEvent | AdjustmentListener | The adjustment event is emitted by an Adjustable object like Scrollbar. |
| ComponentEvent | ComponentListener | An event that indicates that a component moved, the size changed or changed its visibility. |
| ContainerEvent | ContainerListener | When a component is added to a container (or) removed from it, then this event is generated by a container object. |
| FocusEvent | FocusListener | These are focus-related events, which include focus, focusin, focusout, and blur. |
| ItemEvent | ItemListener | An event that indicates whether an item was selected or not. |
| KeyEvent | KeyListener | An event that occurs due to a sequence of keypresses on the keyboard. |
| MouseEvent | MouseListener & MouseMotionListener | The events that occur due to the user interaction with the mouse (Pointing Device). |
| MouseWheelEvent | MouseWheelListener | An event that specifies that the mouse wheel was rotated in a component. |
| TextEvent | TextListener | An event that occurs when an object's text changes. |
| WindowEvent | WindowListener | An event which indicates whether a window has changed its status or not. |

**Note:** As Interfaces contains abstract methods which need to implemented by the registered class to handle events.

## Methods in Listener Interfaces

Each listener interface contains specific methods that must be implemented to handle events. Below table demonstrates the key methods for each interface:

| Listener Interface | Methods |
|---|---|
| ActionListener | actionPerformed() |
| AdjustmentListener | adjustmentValueChanged() |
| ComponentListener | componentResized()<br>componentShown()<br>componentMoved()<br>componentHidden() |
| ContainerListener | componentAdded()<br>componentRemoved() |
| FocusListener | focusGained()<br>focusLost() |
| ItemListener | itemStateChanged() |
| KeyListener | keyTyped()<br>keyPressed()<br>keyReleased() |
| MouseListener | mousePressed()<br>mouseClicked()<br>mouseEntered()<br>mouseExited()<br>mouseReleased() |
| MouseMotionListener | mouseMoved()<br>mouseDragged() |
| MouseWheelListener | mouseWheelMoved() |
| TextListener | textChanged() |
| WindowListener | windowActivated()<br>windowDeactivated()<br>windowOpened() |

| Listener Interface | Methods |
|---|---|
| | windowClosed()<br>windowClosing()<br>windowIconified()<br>windowDeiconified() |

## Flow of Event Handling

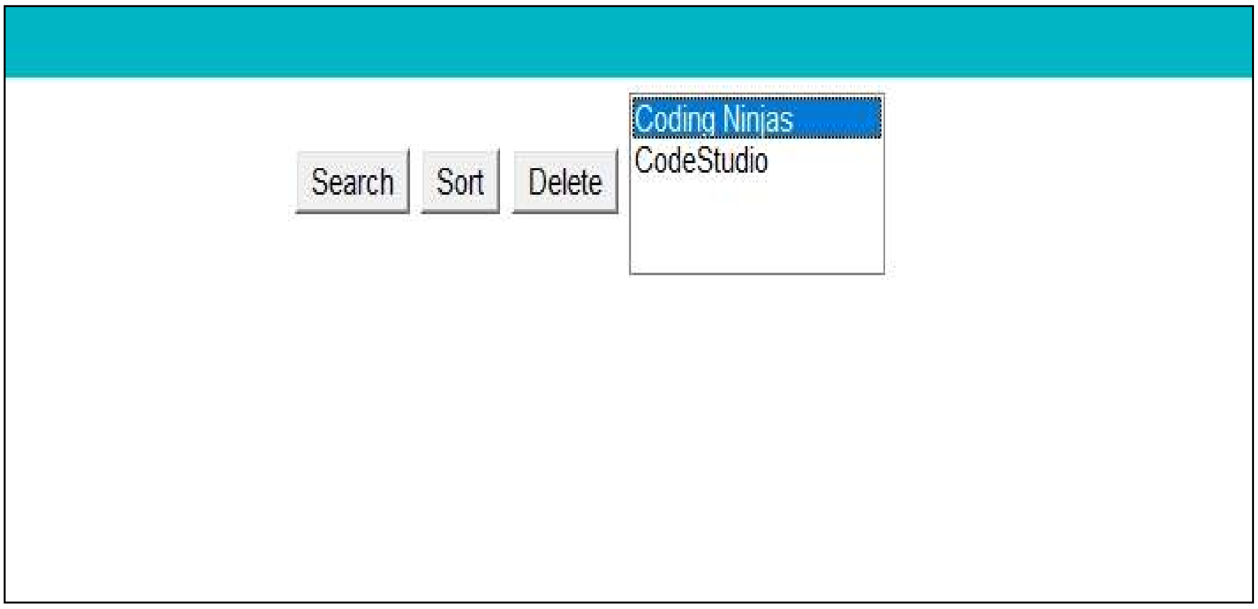
The event handling process in Java follows these steps:

1. User Interaction with a component is required to generate an event.
2. The object of the respective event class is created automatically after event generation, and it holds all information of the event source.
3. The newly created object is passed to the methods of the registered listener.
4. The method executes and returns the result.

## Example Java Program to Implement the Event Delegation Model

```
import java.awt.*;
import java.awt.event.*;
public class TestApp {
   public void search() {
      System.out.println("Searching...");
   }
   public void sort() {
      System.out.println("Sorting....");
   }
   public void delete(){
      System.out.println("Deleting....");
   }
   static public void main(String args[]) {
      TestApp app = new TestApp();
      GUI gui = new GUI(app);
   }
}
class Command implements ActionListener  {
   static final int SEARCH = 0;
   static final int SORT = 1;
   static final int DELETE =2;
   int id;
   TestApp app;
   public Command(int id, TestApp app) {
      this.id = id;
      this.app = app;
   }
   public void actionPerformed(ActionEvent e) {
      switch(id) {
         case SEARCH:
            app.search();
            break;
         case SORT:
                app.sort();
                break;
            case DELETE:
                app.delete();
        }
    }
}
class GUI {
    public GUI(TestApp app) {
        Frame f = new Frame();
        f.setLayout(new FlowLayout());
        Command searchCmd = new Command(Command.SEARCH, app);
        Command sortCmd = new Command(Command.SORT, app);
        Command deleteCmd = new Command(Command.DELETE, app);
        Button b;
        f.add(b = new Button("Search"));
        b.addActionListener(searchCmd);
        f.add(b = new Button("Sort"));
        b.addActionListener(sortCmd);
        f.add(b = new Button("Delete"));
        b.addActionListener(deleteCmd);
        List l;
        f.add(l = new List());
        l.add("Coding Ninjas");
        l.add("Coding Ninjas Studio");
        l.addActionListener(sortCmd);
        f.pack();
        f.show();
    }
}
```

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**.

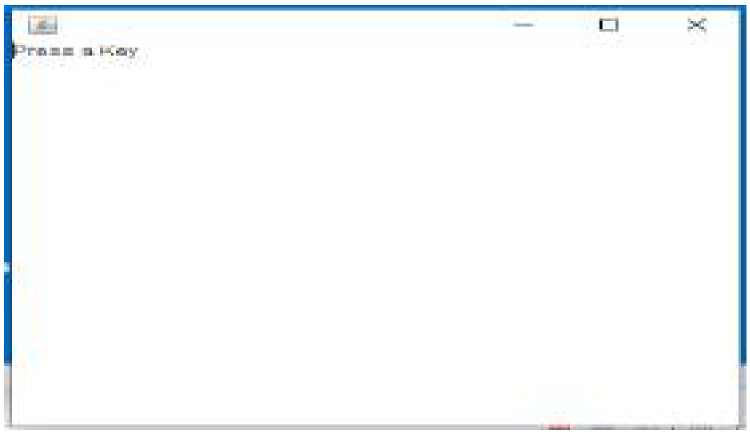
- To represent keys from a to z : VK_A to VK_Z.
- To represent keys from 1 to 9: VK_0 to VK_9.
- To represent keys from F1 to F12: VK_F1 to VK_F12.
- To represent home, end : VK_HOME, VK_END.
- To represent PageUp, PageDown: VK_PAGE_UP, VK_PAGE_DOWN
- To represent Insert, Delete: VK_INSERT, VK_DELETE
- 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:



```
Press a Key
```

## Handling Mouse Events

The user may click, release, drag, or move a mouse while interacting with athe 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 use.

### 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 mouseRelease(MouseEvent e):** this method is invoked when a mouse button has been released ona 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.

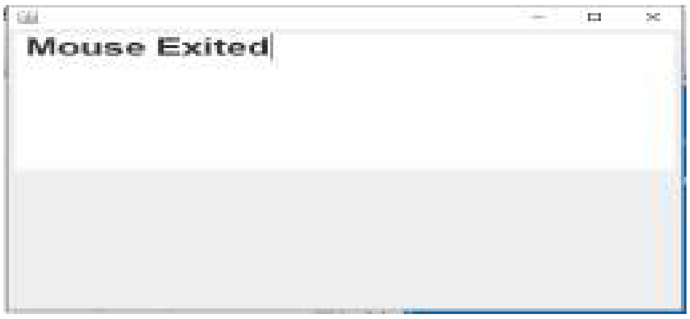
**The MouseEVent class has the following methods**
1. **int getButton():** this method returns a value representing a mouse button, when it is clicked it retursn 1 if left button is clicked, 2 if middle button, and 3 if right button is clicked.
2. **int getX();** this method returns the horizontal x position of the event relative to the source component.
3. **int getY();** this method returns the vertical y postion 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 click, 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.setTe
                xt(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:**



```
Mouse Exited
```

## Adapter Classes:

- Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations.
- An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

| Adapter Class | Listener Interface |
|---|---|
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| FocusAdapter | FocusListener |
| KeyAdapter | KeyListener |
| MouseAdapter | MouseListener |
| MouseMotionAdapter | MouseMotionListener |
| WindowAdapter | WindowListener |

**Table 20-4.** *Commonly Used Listener Interfaces Implemented by Adapter Classes*

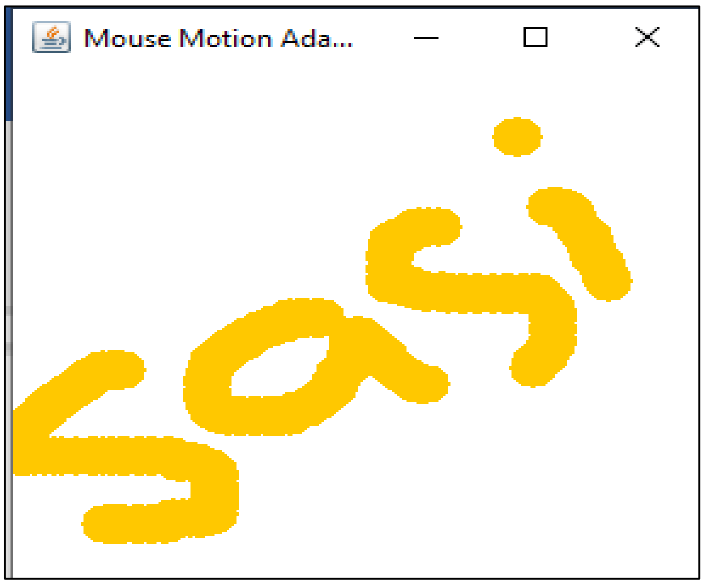
- For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**.
- The signatures of these empty methods are exactly as defined in the **MouseMotionListener** interface.
- If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and implement **mouseDragged( )**.

**// Demonstrate an adapter class. import**

```
// importing the necessary libraries
import java.awt.*;
import java.awt.event.*;

// class which inherits the MouseMotionAdapter class
public class MouseMotionAdapterExample extends MouseMotionAdapter {

// object of Frame class
   Frame f;

// class constructor
   MouseMotionAdapterExample() {
// creating the frame with the title
      f = new Frame ("Mouse Motion Adapter");
// adding MouseMotionListener to the Frame
      f.addMouseMotionListener (this);
 // setting the size, layout and visibility of the frame
```

```
        f.setSize (300, 300);
        f.setLayout (null);
        f.setVisible (true);
    }

// overriding the mouseDragged() method
public void mouseDragged (MouseEvent e) {
// creating the Graphics object and fetching them from the Frame object using getGraphics()
method
    Graphics g = f.getGraphics();
// setting the color of graphics object
    g.setColor (Color.ORANGE);
// setting the shape of graphics object
    g.fillOval (e.getX(), e.getY(), 20, 20);
}

public static void main(String[] args) {
    new MouseMotionAdapterExample();
}
}
```

**Output:**


Mouse Motion Ada...