EIT

B.Tech (IT)

Notes

Prepared By: TOPPERWorld

Unit-1

Introduction to Java & Principles of Object Oriented Programming

What Are OOPS Concepts

OOP's or Object Oriented Programming is an approach or a programming pattern where the programs are structured around objects rather than functions and logic. It makes the data partitioned into two memory areas, i.e., data and functions, and helps make the code flexible and modular.

Object-oriented programming mainly focuses on objects that are required to be manipulated. In OOPs, it can represent data as objects that have attributes and functions.

Why Do You Need Object-Oriented Programming?

The earlier approaches to programming were not that good, and there were several limitations as well. Like in procedural-oriented programming, you cannot reuse the code again in the program, and there was the problem of global data access, and the approach couldn't solve the real-world problems very well.

In object-oriented programming, it is easy to maintain the code with the help of classes and objects. Using inheritance, there is code reusability, i.e., you don't have to write the same code again and again, which increases the simplicity of the program. Concepts like encapsulation and abstraction provide data hiding as well.

Basic Object-Oriented Programming (OOPS) Concept

There are some basic concepts that act as the building blocks of OOPs.

- Classes & Objects

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

So now, you will understand each of these concepts in detail.

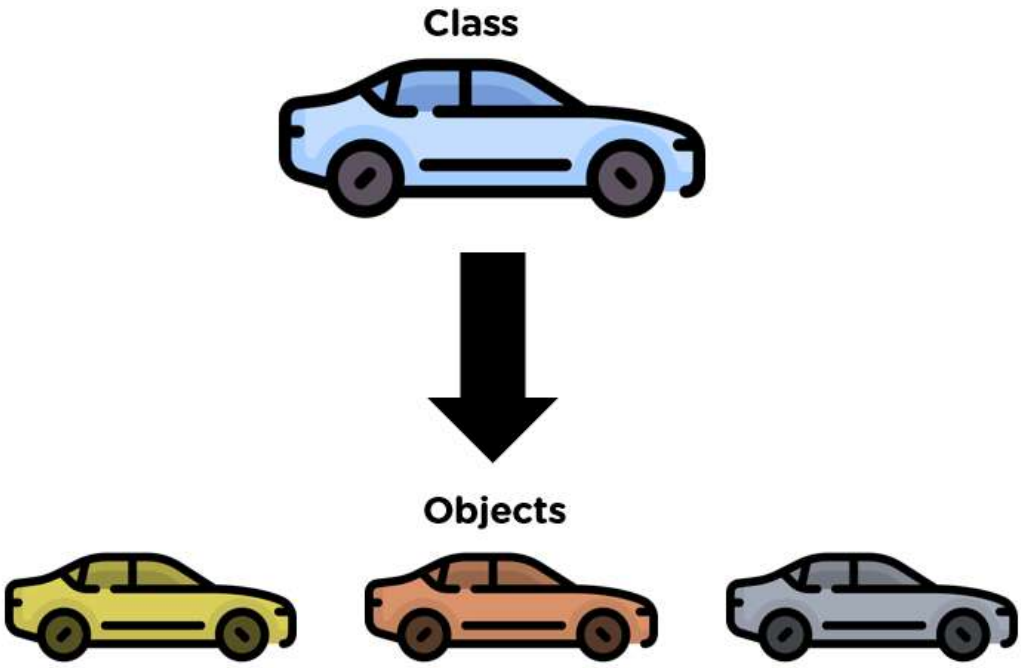
- **Object**

An Object can be defined as an entity that has a state and behavior, or in other words, anything that exists physically in the world is called an object. It can represent a dog, a person, a table, etc.

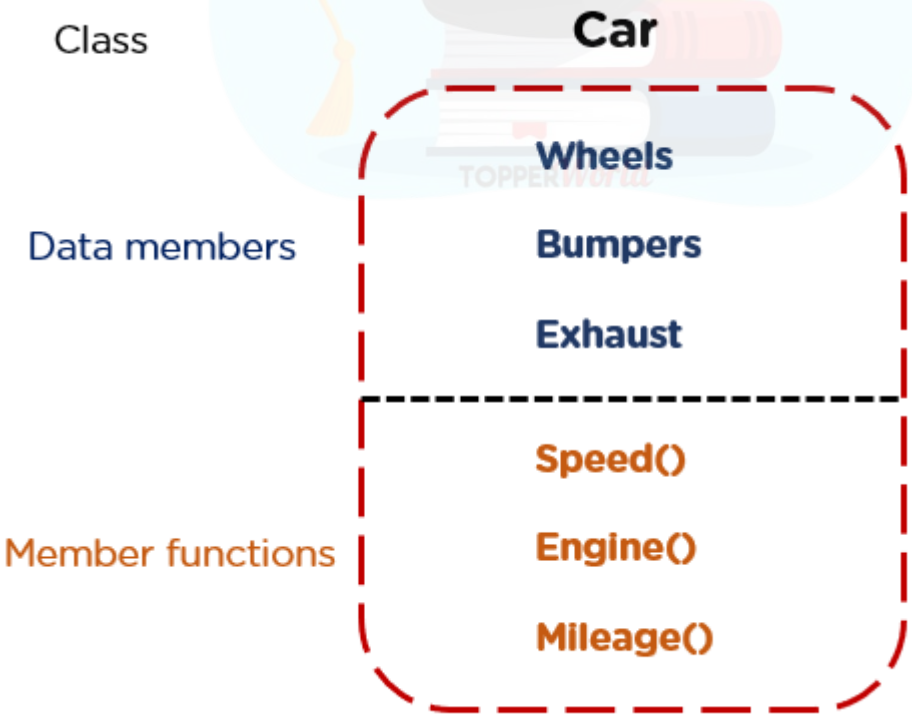
An object means the combination of data and programs, which further represent an entity.

- **Classes**

Class can be defined as a blueprint of the object. It is basically a collection of objects which act as building blocks.

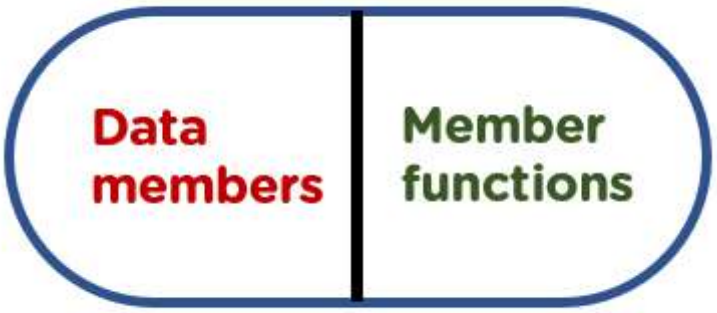


A class contains data members (variables) and member functions. These member functions are used to manipulate the data members inside the class.



Abstraction

Abstraction helps in the data hiding process. It helps in displaying the essential features without showing the details or the functionality to the user. It avoids unnecessary information or irrelevant details and shows only that specific part which the user wants to see.



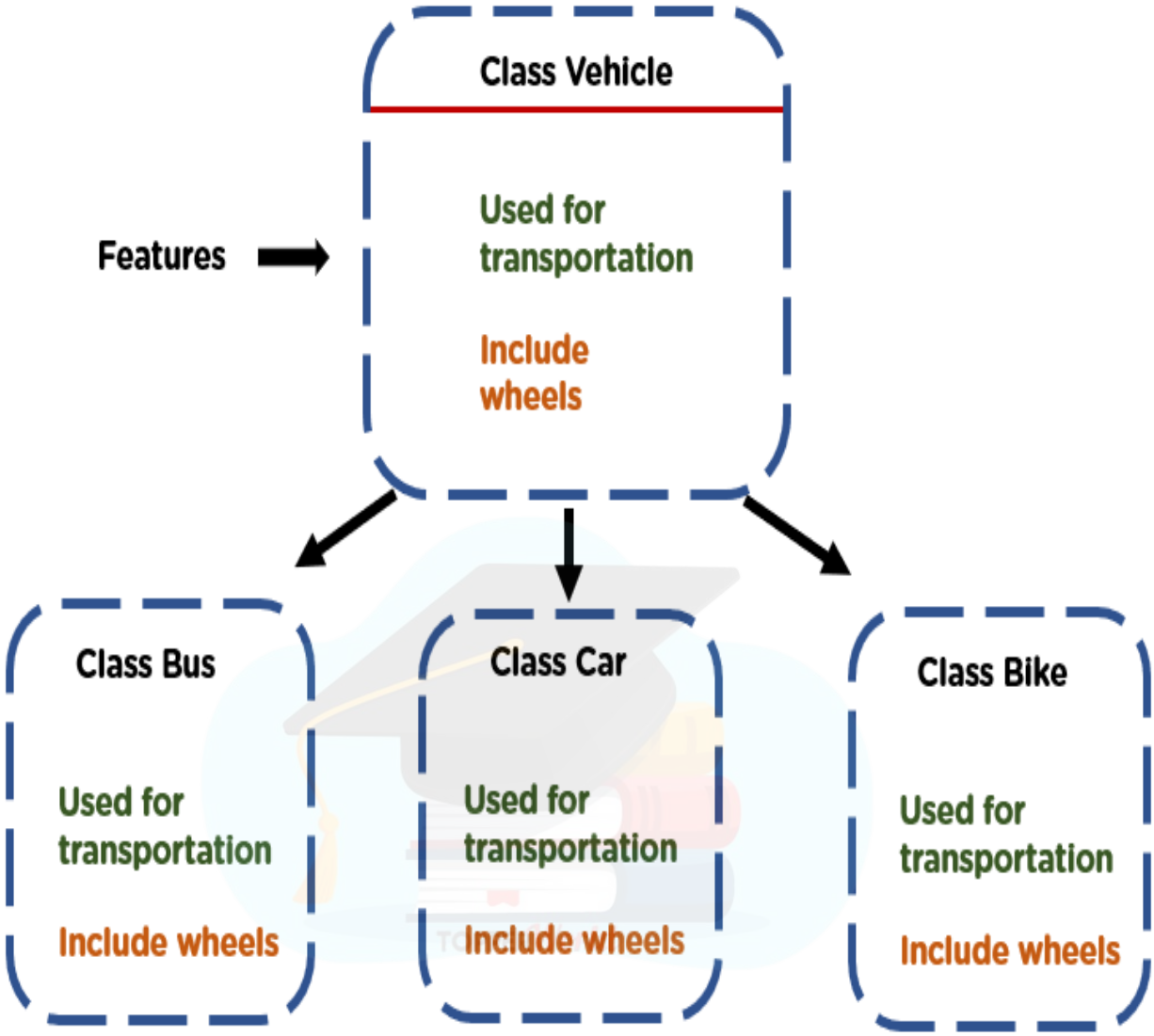
Encapsulation

The wrapping up of data and functions together in a single unit is known as encapsulation. It can be achieved by making the data members' scope private and the member function’s scope public to access these data members. Encapsulation makes the data non-accessible to the outside world.

Inheritance

Inheritance is the process in which two classes have an is-a relationship among each other and objects of one class acquire properties and features of the other class. The class which inherits the features is known as the child class, and the class whose features it inherited is

called the parent class. For example, Class Vehicle is the parent class, and Class Bus, Car, and Bike are child classes.



Let’s have a look at an example of inheritance.

```
2   #include <iostream>
3
4   using namespace std;
5
6   class Parent
7   {
8       public:
9       string name1="Harley";
10  };
11
12  class Child: public Parent
13  {
14      public:
15      string name2="Davidson";
16  };
17
18  int main()
19  {
20      Child ch;
21      cout << ch.name1 + " " + ch.name2;
22
23      return 0;
24  }
```

As we can see, there are two classes named parent and child. In line 12, the child class inherits the parent class. Initially, you created an instance of the child class, through which you are calling name1 and name2 variables; both of these string variables are holding Harley and Davidson, respectively.

Since child class is inheriting parent class, when you call using the object of the child class, you get the result name1 as Harley and name2 as Davidson.

Below is the output.

```
TERMINAL    PROBLEMS    OUTPUT
C:\Users\harsh>cd "c:\Users\
Harley Davidson
c:\Users\harsh\Desktop\Work>
```

Polymorphism

Polymorphism means many forms. It is the ability to take more than one form. It is a feature that provides a function or an operator with more than one definition. It can be implemented using function overloading, operator overload, function overloading , virtual function.

Let’s have a look at an example where we are implementing function overriding.

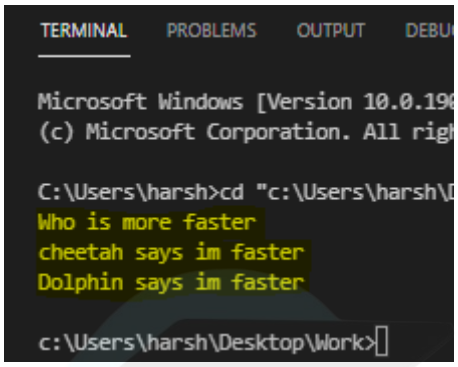
```
#include <iostream>
using namespace std;

class Animal {
  public:
    void speed() {
      cout << "Who is more faster\n" ;
    }
};

class Cheetah : public Animal {
  public:
    void speed() {
      cout << "cheetah says im faster \n" ;
    }
};

class Dolphin : public Animal {
  public:
    void speed() {
      cout << "Dolphin says im faster  \n" ;
    }
};
int main() {
  Animal a;
  Cheetah c;
  Dolphin d;

  a.speed();
  c.speed();
  d.speed();
  return 0;
}
```

As you can see in the above example, there are three classes. Class Animal is the parent class, Class Cheetah, which is the derived class, and Class Dolphin is again the derived class of Animal class.

All three classes have a function of the same name, i.e., speed, but the definition of this speed function is different in all three classes. Inside the main function, firstly,

you are invoking the speed function using the object of the parent class, then using the object of child class Cheetah, you are again invoking the function, and similarly, you are invoking the function using the object of dolphin class.

Below is the output of the above program. You can see that when you call the function using the object of the parent class, it invokes the function of the parent class. But when you call the function using the object of the child class, it overrides the parent class function and prints the function of the child class.



Now go through the advantages of C++ OOPs.

Advantages of OOPs

There are various advantages of object-oriented programming.

- OOPs provide reusability to the code and extend the use of existing classes.
- In OOPs, it is easy to maintain code as there are classes and objects, which helps in making it easy to maintain rather than restructuring.
- It also helps in data hiding, keeping the data and information safe from leaking or getting exposed.

Disadvantages of OOP

- The length of the programmes developed using OOP language is much larger than the procedural approach. Since the programme becomes larger in size, it requires more time to be executed that leads to slower execution of the programme.
- We can not apply OOP everywhere as it is not a universal language. It is applied only when it is required. It is not suitable for all types of problems.
- Programmers need to have brilliant designing skill and programming skill along with proper planning because using OOP is little bit tricky.
- OOPs take time to get used to it. The thought process involved in object-oriented programming may not be natural for some people.
- Everything is treated as object in OOP so before applying it we need to have excellent thinking in terms of objects.

Benifts of OOP's

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
  - OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
  - The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
  - OOP systems can be easily upgraded from small to large systems.
  - It is possible that multiple instances of objects co-exist without any interference,
  - It is very easy to partition the work in a project based on objects.
  - It is possible to map the objects in problem domain to those in the program.

- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.
- Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- The data-centered design approach enables us to capture more details of model in an implementable form.

Application of OOP

OOPs stands for Object Oriented Programme It is about creating objects that contain both data and functions. Object-Oriented programming has several advantages over procedural languages. As OOP is faster and easier to execute it becomes more powerful than procedural languages like C++. OOPs is the most important and flexible paradigm of modern programming. It is specifically useful in modeling real-world problems. Below are some applications of OOPs:

- **Real-Time System design:** Real-time system inherits complexities and makes it difficult to build them. OOP techniques make it easier to handle those complexities.
- **Hypertext and Hypermedia:** Hypertext is similar to regular text as it can be stored, searched, and edited easily. Hypermedia on the other hand is a superset of hypertext. OOP also helps in laying the framework for hypertext and hypermedia.
- **AI Expert System:** These are computer application that is developed to solve complex problems which are far beyond the human brain. OOP helps to develop such an AI expert System
- **Office automation System:** These include formal as well as informal electronic systems that primarily concerned with information sharing and communication to and from people inside and outside the organization. OOP also help in making office automation principle.
- **Neural networking and parallel programming:** It addresses the problem of prediction and approximation of complex-time varying systems. OOP simplifies the entire process by simplifying the approximation and prediction ability of the network.
- **Stimulation and modeling system:** It is difficult to model complex systems due to varying specifications of variables. Stimulating complex systems require modeling and understanding interaction

explicitly. OOP provides an appropriate approach for simplifying these complex models.

- **Object-oriented database:** The databases try to maintain a direct correspondence between the real world and database object in order to let the object retain it identity and integrity.
- **Client-server system**: Object-oriented client-server system provides the IT infrastructure creating object-oriented server internet(**OCSI**) applications.
- **CIM/CAD/CAM systems**: OOP can also be used in manufacturing and designing applications as it allows people to reduce the efforts involved. For instance, it can be used while designing blueprints and flowcharts. So it makes it possible to produce these flowcharts and blueprint accurately.

The Creation of Java

In this section, we will know the history of Java that is very interesting. In 1990, Sun Microsystems Inc. (US) imagined a project to develop software for consumer electronic devices that could be controlled by a remote.

Initially, this project was named Stealth Project but later its name was changed to Green Project.

In January of 1991, Bill Joy, James Gosling, Patrick Naughton, Mike Sheridan, and several others met in Aspen, Colorado to discuss this project.

The job of Mike Sheridan was to focus on business development. Patrick Naughton was to work on the graphics system.

James Gosling was to recognize the proper programming language for the project. He thought that C and C++ programming languages could be used to develop this project.

But the problem he faced in using these languages is that C and C++ were system-dependent programming languages. Due to which they could not be used on various computer systems or electronic devices.

So, he started to develop a new programming language that was completely platform-independent and could be run on any electronic device.

This programming language was initially named Oak but later it was changed to Java in 1995.

Thus, Java was developed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems in 1991. It took 18 months to develop the first working version of Java.

Importance of Java to the Internet

Java has had a profound effect on the Internet because it allows objects to move freely in Cyberspace. In a network there are two categories of objects that are transmitted between the Server and the Personal computer.

Passive information

Dynamic active programs

The Dynamic Self-executing programs cause serious problems in the areas of Security and probability. But Java addresses those concerns and by doing so has opened the door to an exciting new form of program called the Applet.

Java can be used to create two types of programs

Applications: An application is a program that runs on our Computer under the operating system of that computer. It is more or less like one creating using C or C++. Java's ability to create Applets makes it important.

Applet: An Applet is an application designed to be transmitted over the Internet and executed by a Java compatible web browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image. But the difference is it is an intelligent program, not just a media file. It can react to the user input and dynamically change.

Features of Java Security

Every time you that you download a program you are risking a viral infection. Prior to Java, most users did not download executable programs frequently and most users were worried about the possibility of infecting their systems with a virus. Java answers both these concerns by providing a "firewall" between a network application and your computer. When you use a Java-compatible Web browser, you can safely download Java applets without fear of virus infection.

Portability

For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed .As you will see, the same mechanism that helps ensure security also helps create portability. Indeed Java's solution to these two problems is both elegant and efficient.

Java Virtual Machine (JVM)

Beyond the language there is the Java virtual machine. The Java virtual machine is an important element of the Java technology. The virtual machine can be embedded within a web browser or an operating system. Once a piece of Java code is loaded onto a machine, it is verified. As part of the loading process, a class loader is invoked and does byte code verification makes sure that the code that's has been generated by the compiler will not corrupt the machine that it's loaded on. Byte code verification takes place at the end of the compilation process to make sure that is all accurate and correct

Java Architecture

Java architecture provides a portable, robust, high performing environment for development. Java provides portability by compiling the byte codes for the Java Virtual Machine, which is then interpreted on each platform by the run-time environment

Compilation of code

When you compile the code, the Java compiler creates machine code (called byte code) for a hypothetical machine called Java Virtual Machine (JVM). The JVM is supposed to execute the byte code. The JVM is created for overcoming the issue of portability. The code is written and compiled for one machine and interpreted on all machines. This machine is called Java Virtual Machine.

Simple

Java was designed to be easy for the Professional programmer to learn and to use effectively. If you are an experienced C++ programmer, learning Java will be even easier. Because Java inherits the C/C++ syntax and many of the objects oriented features of C++. Most of the confusing concepts from C++ are either left out of Java or implemented in a cleaner, more approachable manner

Object-Oriented

Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean usable, pragmatic approach to objects. The object model in Java is simple and easy to extend while simple types such as integers are kept as high-performance non-objects.

Robust

The multi-platform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. The ability to create robust programs was given a high priority in the design of Java. Java is strictly typed language; it checks your code at compile time and run time. Java virtually eliminates the problems of memory management and

de-allocation, which is completely automatic. In a well-written Java program, all run time errors can -and should -be managed by your program.

Java's Magic: The Bytecode

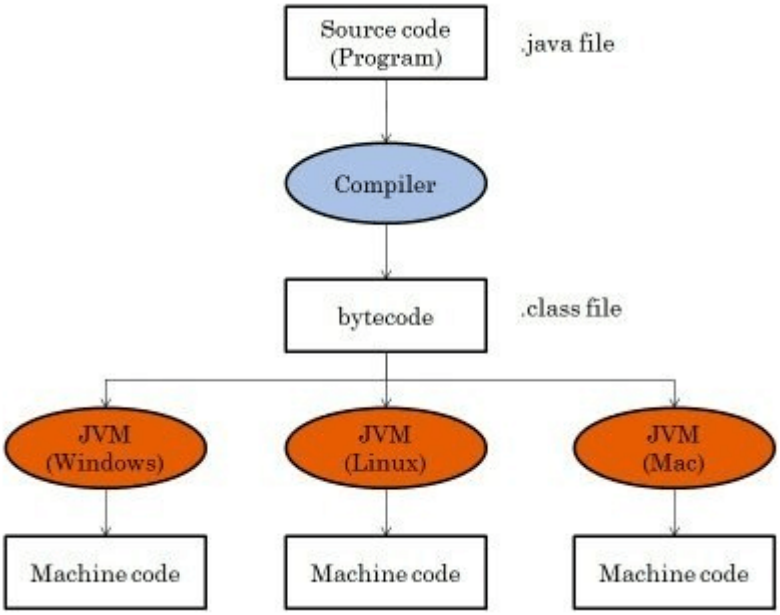
The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM).* In essence, the original JVM was designed as an *interpreter for bytecode.* This may come as a bit of a surprise because many modern languages are designed to be compiled into executable code due to performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs.

What is Java Bytecode?

Java bytecode is the instruction set for the Java Virtual Machine. It acts similar to an assembler which is an alias representation of a C++ code. As soon as a java program is compiled, java bytecode is generated. In more apt terms, java bytecode is the machine code in the form of a .class file. With the help of java bytecode we achieve platform independence in java.

How does it works?

When we write a program in Java, firstly, the compiler compiles that program and a bytecode is generated for that piece of code. When we wish to run this .class file on any other platform, we can do so. After the first compilation, the bytecode generated is now run by the Java Virtual Machine and not the processor in consideration. This essentially means that we only need to have basic java installation on any platforms that we want to run our code on. Resources required to run the bytecode are made available by theJava Virtual Machine, which calls the processor to allocate the required resources. JVM's are stack-based so they stack implementation to read the codes.



Advantage of Java Bytecode

Platform independence is one of the soul reasons for which James Gosling started the formation of java and it is this implementation of bytecode which helps us to achieve this. Hence bytecode is a very important component of any java program.The set of instructions for the JVM may differ from system to system but all can interpret the bytecode. A point to keep in mind is that bytecodes are non-runnable codes and rely on the availability of an interpreter to execute and thus the JVM comes into play.

Bytecode is essentially the machine level language which runs on the Java Virtual Machine. Whenever a class is loaded, it gets a stream of bytecode per method of the class. Whenever that method is called during the execution of a program, the bytecode for that method gets invoked.Javac not only compiles the program but also generates the bytecode for the program. Thus, we have realized that the bytecode implementation makes Java a **platform-independent** language. This helps to add portability to Java which is lacking in languages like C or C++. Portability ensures that Java can be implemented on a wide array of platforms like desktops, mobile devices, severs and many more. Supporting this, Sun Microsystems captioned JAVA as *"write once, read anywhere" or "WORA"* in resonance to the bytecode interpretation.

Object-Oriented Programming in Java

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object

- Class
- Inheritance

- Polymorphism

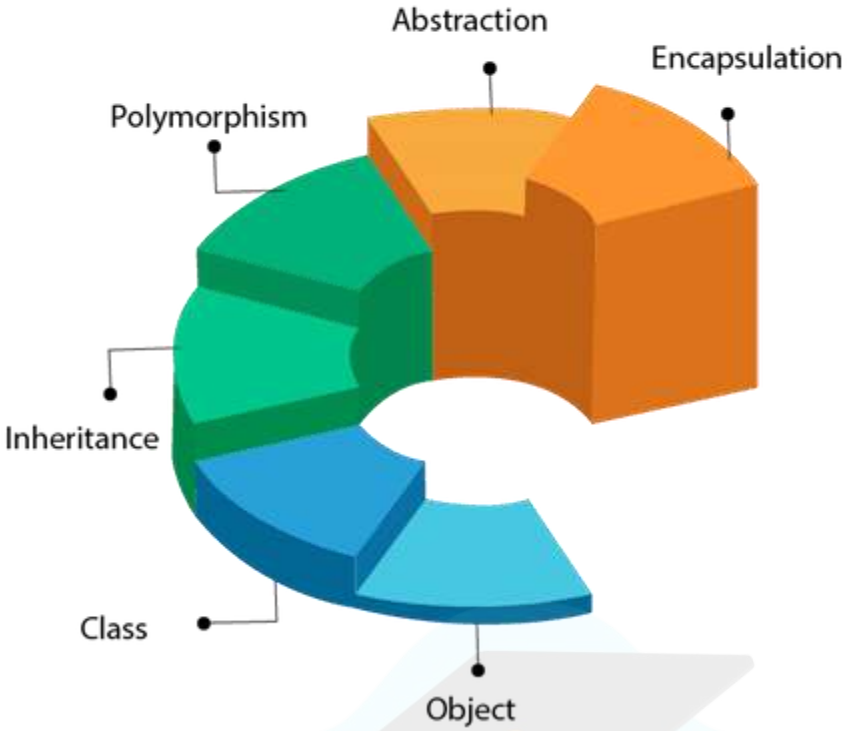
- Abstraction

- Encapsulation

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

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

OOPs (Object-Oriented Programming System)



Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

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

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

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

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

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

Cohesion

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

Association

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

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

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

Association can be undirectional or bidirectional.

Aggregation

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

Composition

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

Java Program Structure

Java is an object-oriented programming, **platform-independent,** and **secure** programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications. So, before diving in depth, it is necessary to understand the **basic structure of Java program** in detail. In this section, we have discussed the basic **structure of a Java program**. At the end of this section, you will able to develop the Hello world Java program, easily.

Structure of Java Program

Let's see which elements are included in the structure of a Java program. A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviors

Documentation Section

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler

ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

- **Single-line Comment:** It starts with a pair of forwarding slash **(//)**. For example:

```
1. //First Java Program
```

- **Multi-line Comment:** It starts with a **/*** and ends with ***/.** We write between these two symbols. For example:

```
1. /*It is an example of
2. multiline comment*/
```

- **Documentation Comment:** It starts with the delimiter **(/**)** and ends with ***/**. For example:

```
1. /**It is an example of documentation comment*/
```

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used. For all these classes package belongs to a single parent directory. We use the keyword **package** to declare the package name. For example:

```
1. package javatpoint; //where javatpoint is the package name
2. package com.javatpoint; //where com is the root directory and javatpoint is the subdirectory
```

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the **import** keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements. For example:

```
1. import java.util.Scanner; //it imports the Scanner class only
2. import java.util.*; //it imports all the class of the java.util package
```

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword. For example:

```
1. interface car
2. {
3. void start();
4. void stop();
5. }
```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method. For example:

```
1. class Student //class definition
2. {
3. }
```

Class Variables and Constants

In this section, we define variables and **constants** that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables. For example:

```
1. class Student //class definition
2. {
```

```
3. String sname;  //variable
4. int id;
5. double percentage;
6. }
```

Main Method Class

In this section, we define the **main() method.** It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the main() method:

```
1. public static void main(String args[])
2. {
3. }
```

For example:

```
1. public class Student //class definition
2. {
3. public static void main(String args[])
4. {
5. //statements
6. }
7. }
```

You can read more about the Java main() method here.

Methods and behavior

In this section, we define the functionality of the program by using the methods. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task. For example:

```
1. public class Demo //class definition
2. {
3. public static void main(String args[])
4. {
5. void display()
```

```
{
System.out.println("Welcome to javatpoint");
}
//statements
}
}
```

When we follow and use the above elements in a Java program, the program looks like the following.

CheckPalindromeNumber.java

```
/*Program name: Palindrome*/
//Author's name: Mathew
/*Palindrome is number or string that will remains the same
When we write that in reverse order. Some example of
palindrome is 393, 010, madam, etc.*/
//imports the Scanner class of the java.util package
import java.util.Scanner;
//class definition
public class CheckPalindromeNumber
{
//main method
public static void main(String args[])
{
//variables to be used in program
int r, s=0, temp;
int x; //It is the number variable to be checked for palindrome
Scanner sc=new Scanner(System.in);
System.out.println("Enter the number to check: ");
//reading a number from the user
x=sc.nextInt();
//logic to check if the number id palindrome or not
temp=x;
while(x>0)
{
r=x%10;  //finds remainder
s=(s*10)+r;
```

```
27. x=x/10;
28. }
29. if(temp==s)
30. System.out.println("The given number is palindrome.");
31. else
32. System.out.println("The given number is not palindrome.");
33. }
34. }
```

Output:

```
Enter the number to check: 121
The given number is palindrome.
```

What is Class in java :

Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

Example:

```
public class Dog {
  String breed;
  int age;
  String color;

  void barking() {
  }

  void hungry() {
  }

  void sleeping() {
  }
}
```

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor −

Example

```
public class Puppy {
  public Puppy() {
  }

  public Puppy(String name) {
    // This constructor has one parameter, name.
  }
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note − We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

Example

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

If we compile and run the above program, then it will produce the following result −

Output

```
Passed Name is :tommy
```

What is Method in java :

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

Example

Create a method inside Main:

```
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

Example Explained

- myMethod() is the name of the method
- static means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- void means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, myMethod() is used to print a text (the action), when it is called:

Example

Inside main, call the myMethod() method:

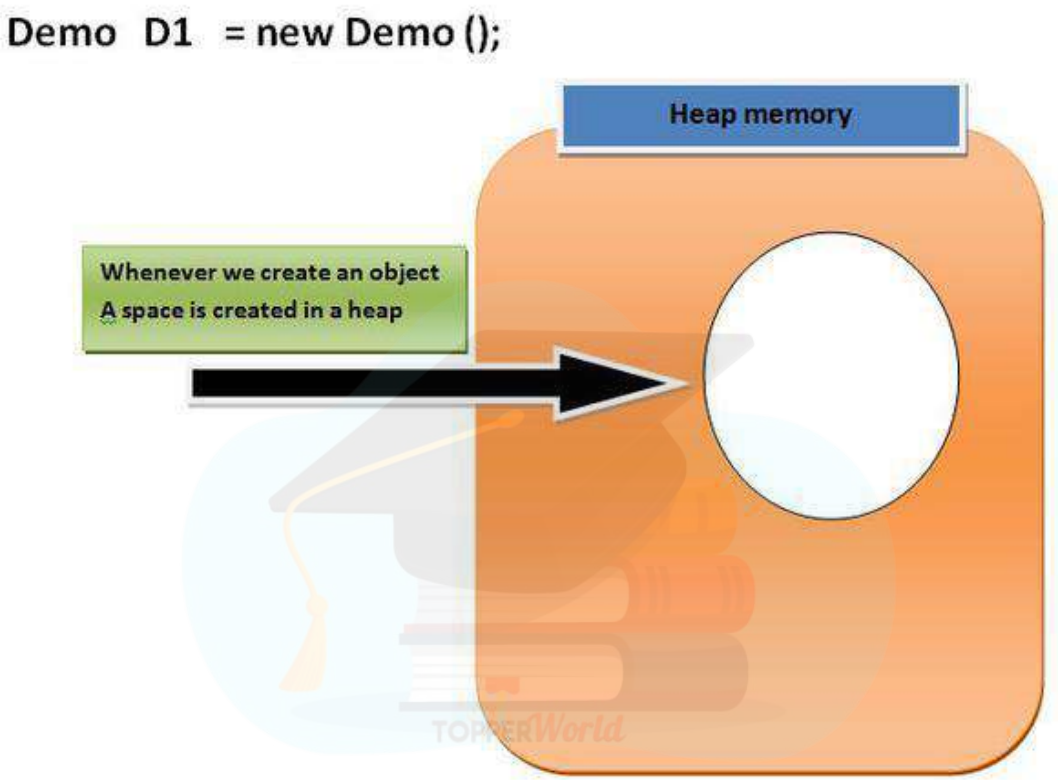
```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "I just got executed!"
```

Assigning Object Reference Variables:

Before We Started with the **Reference variable** we should know about the following facts.

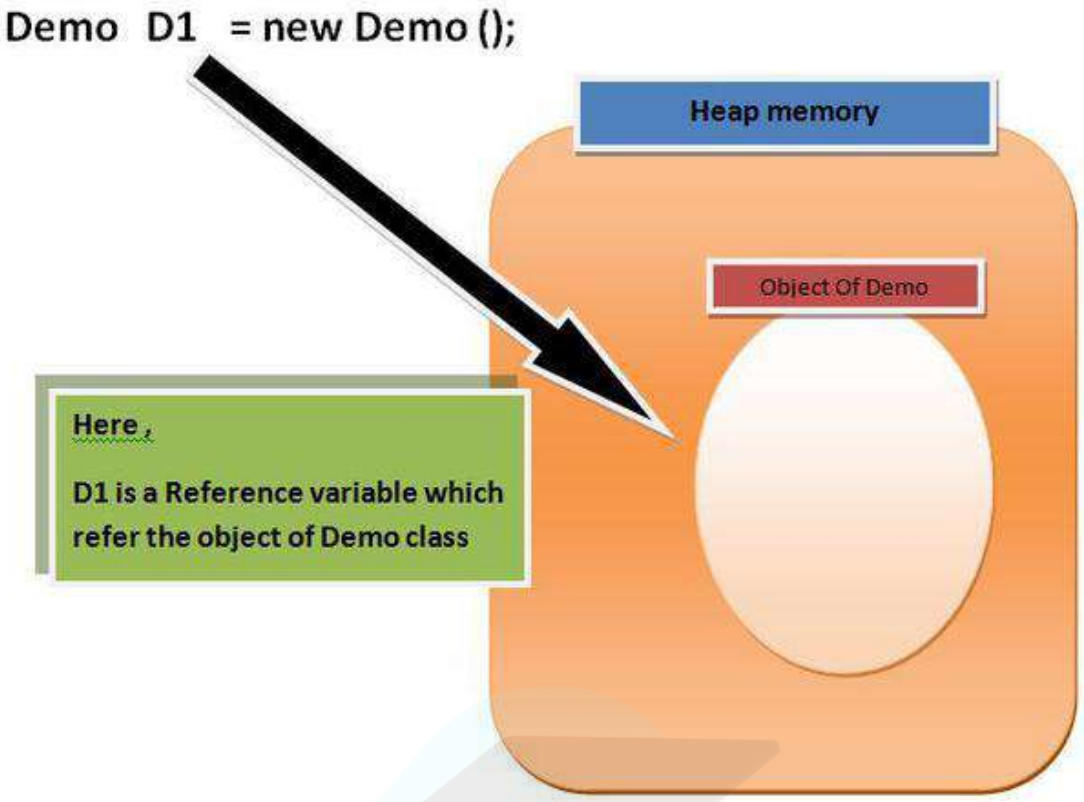
1. When we create an object (instance) of class then space is reserved in heap memory. Let's understand with the help of an example.

Demo D1 = new Demo();



Now, The space in the heap Memory is created but the question is **how to access that space?**.

Then, We create a Pointing element or simply called **Reference variable** which simply points out the Object(the created space in a Heap Memory).



Understanding Reference variable

1. Reference variable is used to point object/values.
2. Classes, interfaces, arrays, enumerations, and, annotations are reference types in Java. Reference variables hold the objects/values of reference types in Java.
3. Reference variable can also store **null** value. By default, if no object is passed to a reference variable then it will store a null value.
4. You can access object members using a reference variable using **dot** syntax.
<reference variable name >.<instance variable_name / method_name>

Example:

- Java

```
// Java program to demonstrate reference
// variable in java

import java.io.*;

class Demo {
    int x = 10;
    int display()
    {
```

```
        System.out.println("x = " + x);
        return 0;
    }
}

class Main {
    public static void main(String[] args)
    {
        Demo D1 = new Demo(); // point 1

        System.out.println(D1); // point 2

        System.out.println(D1.display()); // point 3
    }
}
```

Output

Demo@214c265e

x = 10

0

Let us see what is actually happening step by step.

1. When we create an object of demo class **new DEMO();**, the default constructor is called and returns a reference of the object, and simply this reference will be stored to the reference variable **D1** (As we know that associativity is Right-hand side to left-hand side).

2. The value of a reference variable is a reference. When we attempt to print the value of a reference variable, the output contains the type of the variable and the hash code created for it by Java: the string **Demo@214c265e** tells us that the given variable is of type Name and its hexadecimal format of hash code is 214c265e.

3. At this point we will access the methods **display()** of the class demo using our custom reference variable that we created.

***BINDING UP** : The constructor call returns a value that is a reference to the newly-created object. The equality sign tells the program that the value of the right-hand side expression is to be copied as the value of the variable on the left-hand side. The reference to the newly-created object, returned by the constructor call, is copied as the value of the variable.*

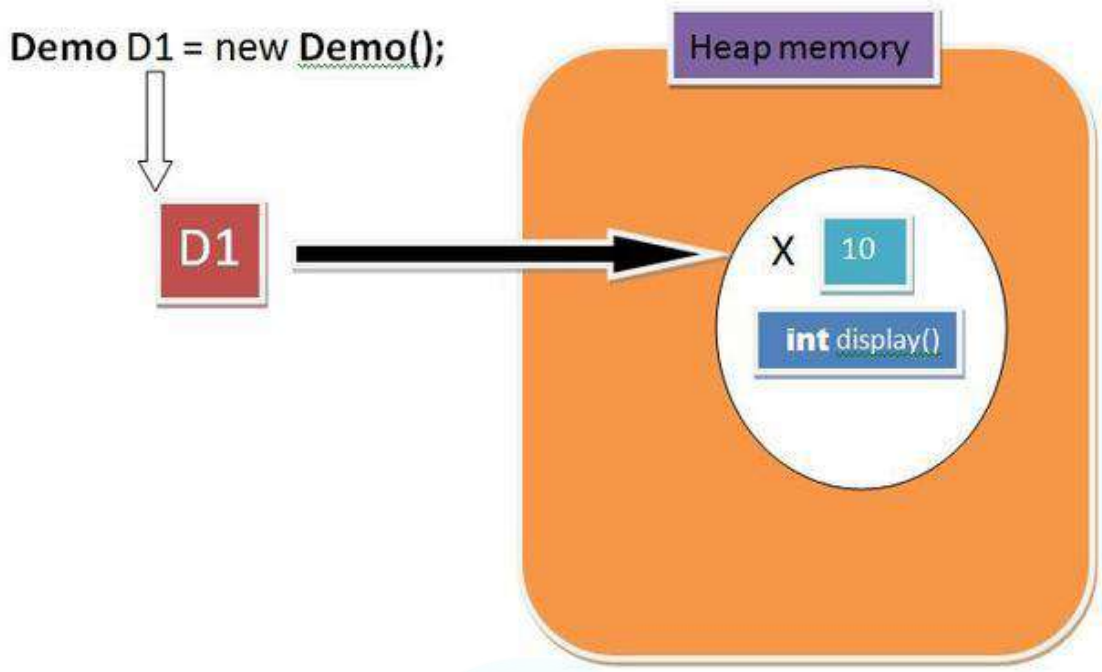
- Java

```
import java.io.*;
class Demo {
    int x = 10;

    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}

class Main {
    public static void main(String[] args)
    {
         // create instance
        Demo D1 = new Demo();

        // accessing instance(object) variable
        System.out.println(D1.x);

        // point 3
        // accessing instance(object) method
        D1.display();
    }
}
```

Output

10

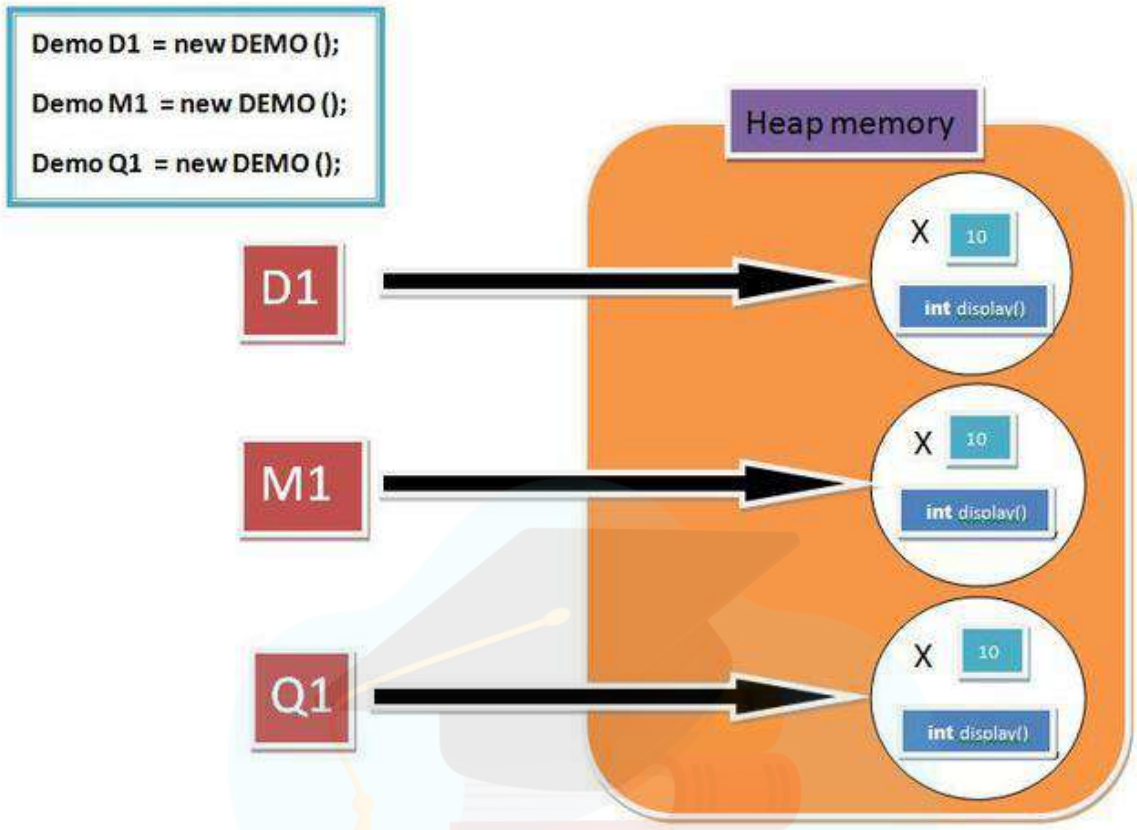
x = 10



- Java

```
// Accessing instance methods

import java.io.*;
class Demo {

    int x = 10;
    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
         // create instances
        Demo D1 = new Demo();

        Demo M1 = new Demo();
```

```
        Demo Q1 = new Demo();
    }
}
```



- Java

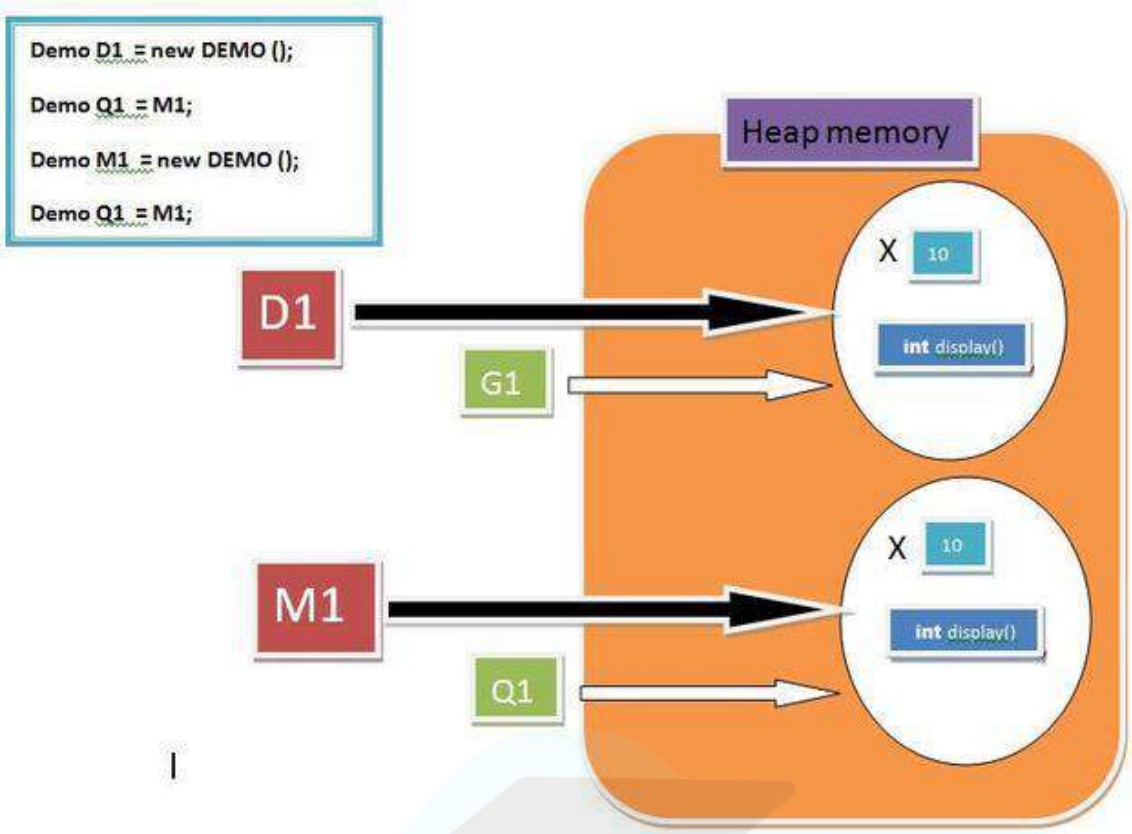
```
// Pointing to same instance memory

import java.io.*;
class Demo {
    int x = 10;
    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
```

```
        // create instance
        Demo D1 = new Demo();

        // point to same reference
        Demo G1 = D1;

        Demo M1 = new Demo();

        Demo Q1 = M1;

        // updating the value of x using G!
        // reference variable
        G1.x = 25;

        System.out.println(G1.x); // Point 1

        System.out.println(D1.x); // Point 2
    }
}
```

Output

25

25



Note:
Here we pass G1 and Q1 reference variable point out the same object respectively. Secondly ***At Point 1*** *we try to get the value of the object with G1 reference variable which shows it as* ***25*** *and* ***At Point 2*** *we try to get the value of an object with D1 reference variable which shows it as* ***25*** *as well. This will prove that the modification in the object can be done by using any reference variable but the condition is it should hold the same reference.*

More on Reference Variable

1. Reference Variable as Method Parameters:

As the value of a primitive variable is directly stored in the variable, whereas the value of a reference variable holds a reference to an object. We also mentioned that assigning a value with the equality sign copies the value (possibly of some variable) on the right-hand side and stores it as the value of the left-hand-side variable. A similar kind of copying occurs during a method call. Regardless of whether the variable is primitive or reference type, a copy of the value is passed to the method's argument and copied to that argument.

Note: *Java only supports pass by value.*

But we know that the reference variable holds the reference of an instance(OBJECT) so a copy of the reference is passed to the method's argument.

Example:

- Java

```
// Pass by reference and value

import java.io.*;
class Demo {
    int x = 10;
    int y = 20;

    int display(Demo A, Demo B)
    {
        //  Updating value using argument
        A.x = 95;

        System.out.println("x = " + x);

        System.out.println("y = " + y);

        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
        Demo C = new Demo();

        Demo D = new Demo();

        // updating value using primary reference
        // variable
        D.y = 55;

        C.display(C, D); // POINT 1
```

```
        D.display(C, D); // POINT 2
    }
}
```

Output

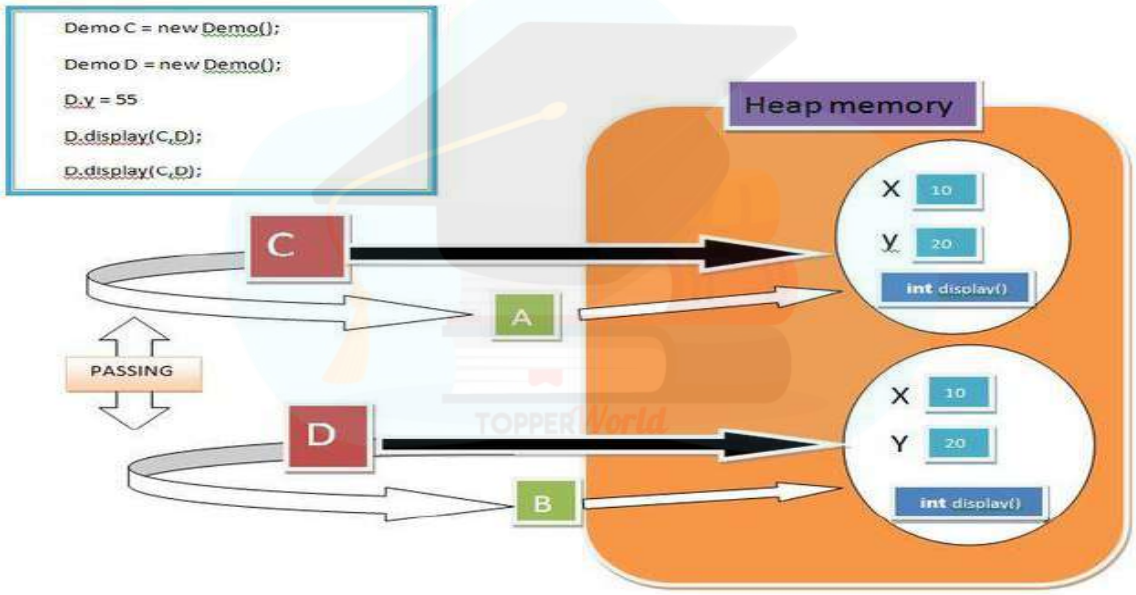
x = 95

y = 20

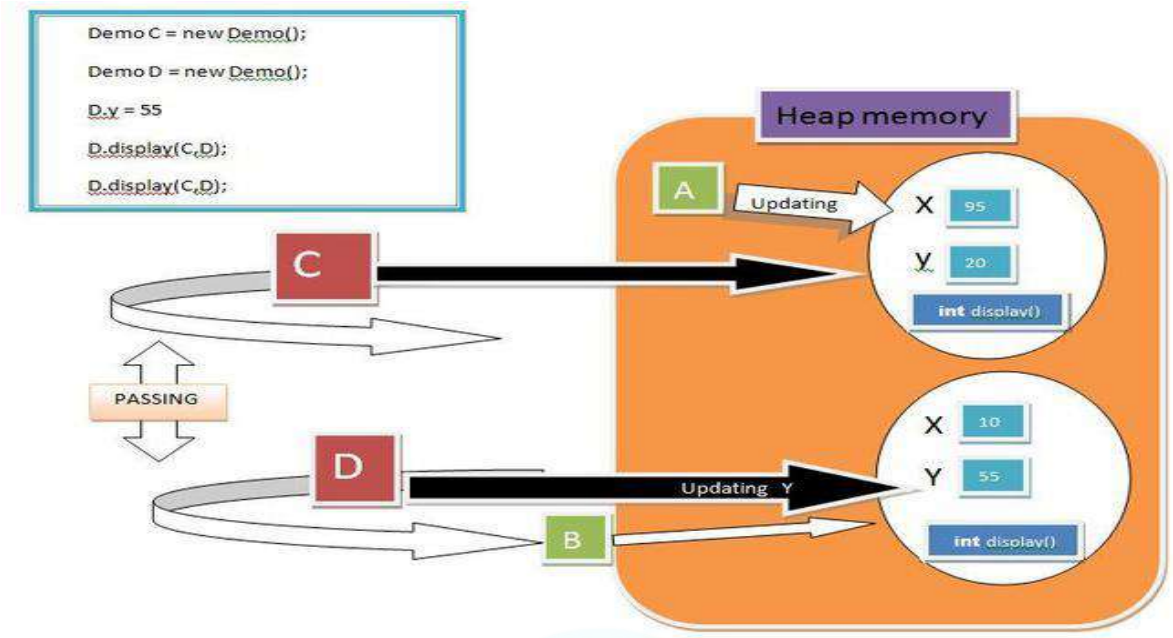
x = 10

y = 55

SCENE 1 :



SCENE 2:



Now, What is going on here, when we pass the reference to the method it will copy to the reference variable defined in the method signature and After that, they also have access to the object members. Here, We defined two instances named **C and D**. Afterwards we pass C and D to the method which further gives reference to **A and B**

At Point 1: A will update the value of x from 10 to 95, hence C.display() will show 95 20 but in another object D we update the value of x through D only from y =20 to 55, hence D, display() will show 10 and 55.

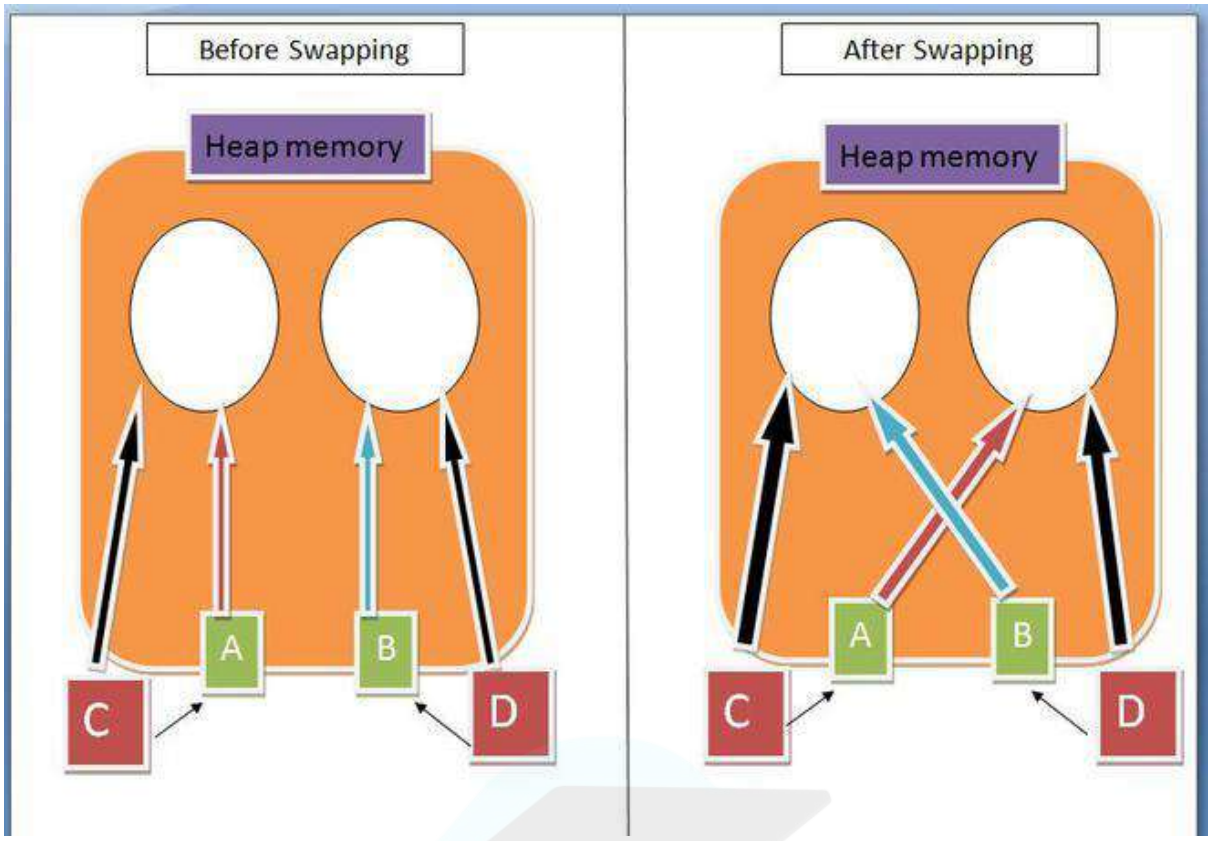
Note: *Any Object Updation will not affect the other object's member.*

2. What if we swap the reference variables with the help of the Swap Method?

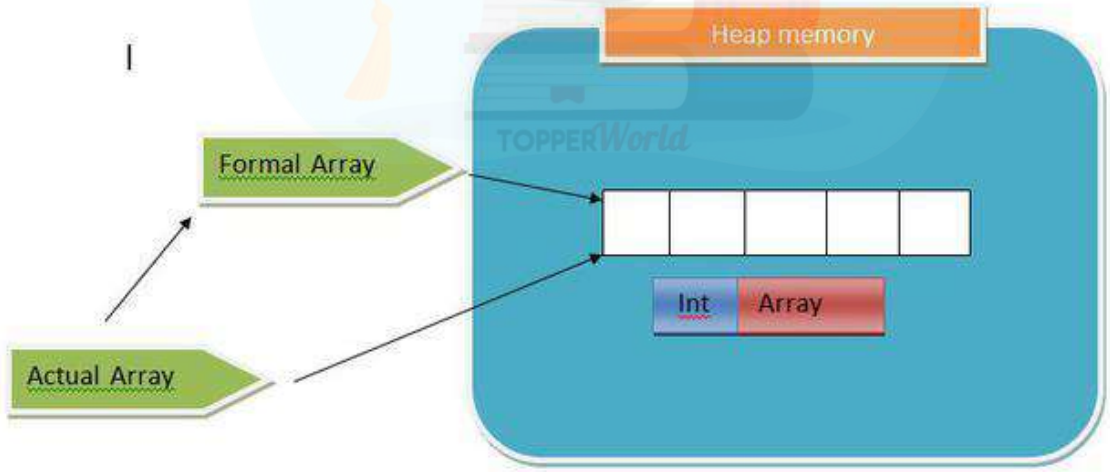
The fact is if we try to swap the reference variable, then they just swap their Pointing element there is no effect on the address of reference variable and object(Instance) Space. Let's Understand It with the help of an example:

```java
// Swapping object references

import java.io.*;
class Demo {

    // Swapping Method
    int Swap(Demo A, Demo B)
    {
        Demo temp = A;
        A = B;
        B = temp;
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
        Demo C = new Demo();

        Demo D = new Demo();

        // Passing C and reference variables
        // to Swap method
        C.Swap(C, D);
    }
}
```

Here we created, two instances of demo class and passes it to swap method, further those C and D will copy their references to A and B respectively.Before swapping A point to C's(Object) and B point to D's(Object). After we perform swapping on A and B, A will now point D's(Object) and B will Point C's Object. As described in the figure.



Note: *There is no swapping between Variables, They only change their References.*



Example:

- Java

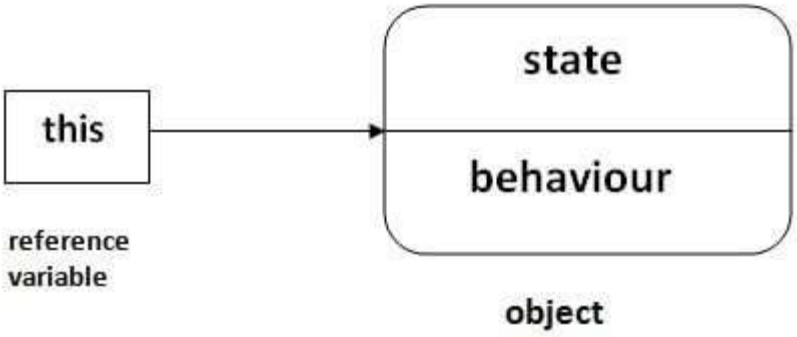
```java
import java.io.*;
class Demo {
    int arrayUpdate(int[] formalArray)
    {
        formalArray[2] = 99;
        formalArray[4] = 77;
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
        Demo d1 = new Demo();
        int[] actualArray = { 1, 2, 3, 4, 5 };

        for (int items : actualArray)
            System.out.print(items
                             + " , "); // printing array

        System.out.println();
        d1.arrayUpdate(actualArray);
        System.out.println();

        for (int items : actualArray)
            System.out.print(items
                             + " , "); // printing array
    }
}
```

Output

```
1 , 2 , 3 , 4 , 5 ,


1 , 2 , 99 , 4 , 77 ,
```

4. this and super keywords are also Pointing Elements.

this keyword. In java, **this** is a reference variable that refers to the current object.



super is used to refer immediate parent class instance variable. We can use the super keyword to access the data member or field of the parent class. It is used if parent class and child class have the same fields.

5. null value of a reference variable.

Demo obj = null ;

A. The **null** reference can be set as the value of any reference type variable.

B. The object whose name is **obj** is referred to by nobody. In other words, the object has become **garbage**. In the Java programming language, the programmer need not worry about the program's memory use. From time to time, the automatic garbage collector of the Java language cleans up the objects that have become garbage. If the garbage collection did not happen, the garbage objects would reserve a memory location until the end of the program execution.

- Java

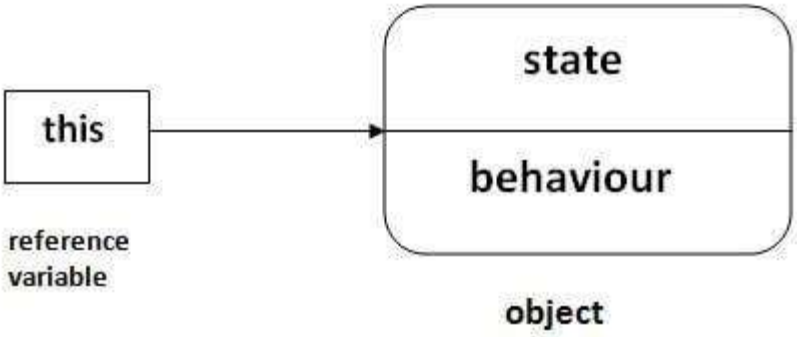
```java
// null in java

import java.io.*;
class Demo {
    int x = 10;
    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
        Demo obj = null;

        // accessing instance(object) method
        Kuchbhi.display();
    }
}
```

Output

```
Exception in thread "main" java.lang.NullPointerException
at Main.main(File.java:17)
Java Result: 1
```

The keyword “this”

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.

1. this can be used to refer current class instance variable.

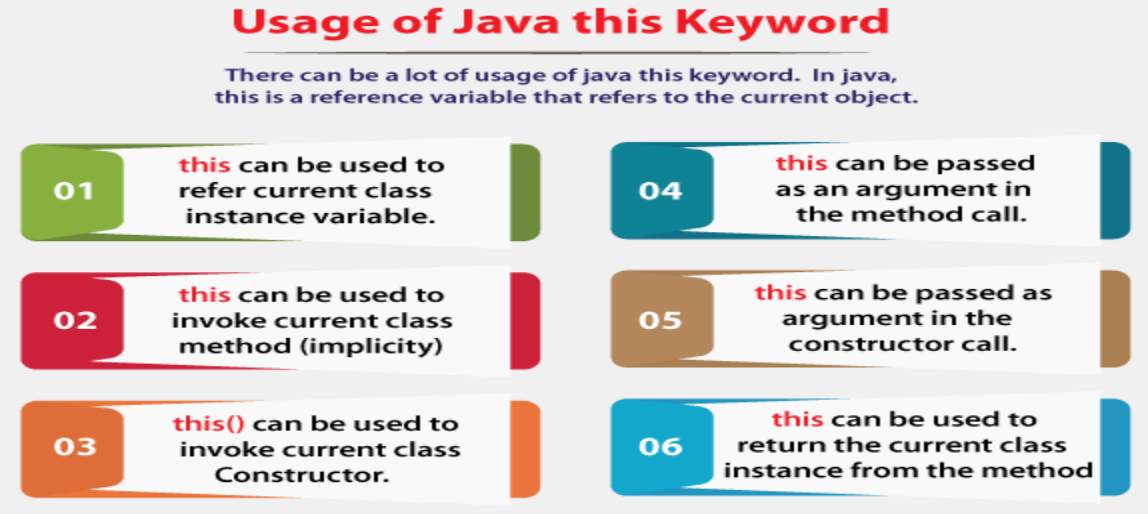
2. this can be used to invoke current class method (implicitly)

3. this() can be used to invoke current class constructor.

4. this can be passed as an argument in the method call.

5. this can be passed as argument in the constructor call.

6. 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:

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. 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);
```

```
17. s1.display();
18. s2.display();
19. }}
```

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:

Program where this keyword is not required

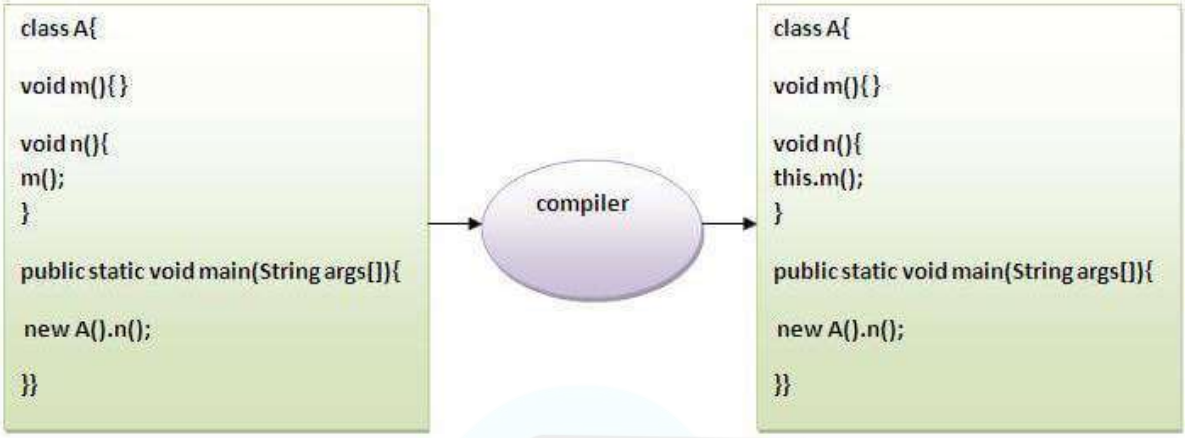
```
1.  class Student{
2.  int rollno;
3.  String name;
4.  float fee;
5.  Student(int r,String n,float f){
6.  rollno=r;
7.  name=n;
8.  fee=f;
9.  }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis3{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}
```

Output:

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

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example




```
1. class A{
2. void m(){System.out.println("hello m");}
3. void n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. this.m();
7. }
8. }
9. class TestThis4{
10. public static void main(String args[]){
11. A a=new A();
12. a.n();
13. }}
```

Test it Now

Output:

```
hello n
hello m
```

---

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor

```
1. class A{
2. A(){System.out.println("hello a");}
3. A(int x){
4. this();
5. System.out.println(x);
6. }
7. }
8. class TestThis5{
9. public static void main(String args[]){
10. A a=new A(10);
11. }}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor

```
1. class A{
2. A(){
3. this(5);
4. System.out.println("hello a");
5. }
6. A(int x){
7. System.out.println(x);
8. }
```

```
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
```

```
18. Student s1=new Student(111,"ankit","java");
19. Student s2=new Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}
```

Output:

```
111 ankit java 0.0
112 sumit java 6000.0
```

```
1.  class Student{
2.  int rollno;
3.  String name,course;
4.  float fee;
5.  Student(int rollno,String name,String course){
6.  this.rollno=rollno;
7.  this.name=name;
8.  this.course=course;
9.  }
10. Student(int rollno,String name,String course,float fee){
11. this.fee=fee;
12. this(rollno,name,course);//C.T.Error
13. }
14. void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis8{
17. public static void main(String args[]){
18. Student s1=new Student(111,"ankit","java");
19. Student s2=new Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}
```

Test it Now

Output:

```
Compile Time Error: Call to this must be first statement in constructor
```

---

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
1.  class S2{
2.    void m(S2 obj){
3.    System.out.println("method is invoked");
4.    }
5.    void p(){
6.    m(this);
7.    }
8.    public static void main(String args[]){
9.    S2 s1 = new S2();
10.   s1.p();
11.   }
12. }
```

Output:

```
method is invoked
```

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

---

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
1. class B{
2.   A4 obj;
3.   B(A4 obj){
4.     this.obj=obj;
5.   }
6.   void display(){
7.     System.out.println(obj.data);//using data member of A4 class
8.   }
9. }
10.
11. class A4{
12.  int data=10;
13.  A4(){
14.   B b=new B(this);
15.   b.display();
16.  }
17.  public static void main(String args[]){
18.   A4 a=new A4();
19.  }
20. }
```

```
Output:10
```

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1. return_type method_name(){
2. return this;
3. }
```

Example of this keyword that you return as a statement from the method

```
1. class A{
2. A getA(){
3. return this;
4. }
5. void msg(){System.out.println("Hello java");}
6. }
7. class Test1{
8. public static void main(String args[]){
9. new A().getA().msg();
10. }
11. }
```

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
1. class A5{
2. void m(){
3. System.out.println(this);//prints same reference ID
4. }
5. public static void main(String args[]){
6. A5 obj=new A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }
```

Output:

```
A5@22b3ea59
A5@22b3ea59
```

Automatic Garbage Collection:

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

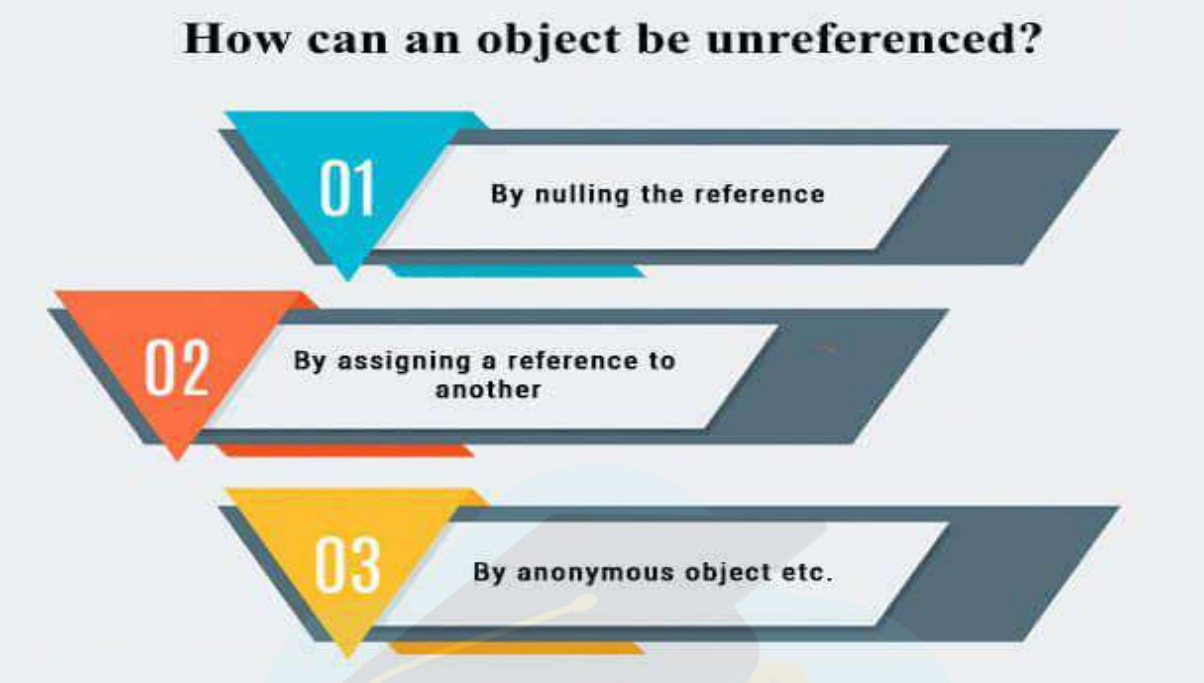
Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.



1) By nulling a reference:

```
1. Employee e=new Employee();
2. e=null;
```

2) By assigning a reference to another:

```
1. Employee e1=new Employee();
2. Employee e2=new Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
1. new Employee();
```

---

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
1. protected void finalize(){}
```

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
1. public static void gc(){}
```

Simple Example of garbage collection in java

```
1. public class TestGarbage1{
2.  public void finalize(){System.out.println("object is garbage collected");}
3.  public static void main(String args[]){
4.   TestGarbage1 s1=new TestGarbage1();
5.   TestGarbage1 s2=new TestGarbage1();
6.   s1=null;
7.   s2=null;
8.   System.gc();
9.  }
10. }
```

```
object is garbage collected
object is garbage collected
```

Arrays and Strings:

An array is a collection of similar data elements stored at contiguous memory locations. It is the simplest data structure where each data element can be accessed directly by only using its index number.

For instance, if we want to store the marks scored by a student in 5 subjects, then there's no need to define individual variables for each subject. Rather, we can define an array that will store the data elements at contiguous memory locations.

Array marks[5] define the marks scored by a student in 5 different subjects where each subject's marks are located at a particular location in the array, i.e., marks[0] denote the marks scored in the first subject, **marks[1]** denotes the marks scored in 2nd subject and so on.

The Complexity of Array Operations

Time Complexity

Algorithm	Average case
Access	O(1)
Search	O(n)
Insertion	O(n)
Deletion	O(n)

The space complexity of an array for the worst case is **O(n)**.

Advantage of Array

- Arrays represent multiple data elements of the same type using a single name.
- Accessing or searching an element in an array is easy by using the index number.
- An array can be traversed easily just by incrementing the index by 1.
- Arrays allocate memory in contiguous memory locations for all its data elements.

Disadvantage of Array

- The size of the array should be known in advance.
- The array is a static data structure with a fixed size so, the size of the array cannot be modified further and hence no modification can be done during runtime.
- Insertion and deletion operations are costly in arrays as elements are stored in contiguous memory.
- If the size of the declared array is more than the required size then, it can lead to memory wastage.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Output:

```
10
20
70
40
50
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

1. dataType[][] arrayRefVar; (or)

2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
1.  //Java Program to illustrate the jagged array
2.  class TestJaggedArray{
3.     public static void main(String[] args){
4.        //declaring a 2D array with odd columns
5.        int arr[][] = new int[3][];
6.        arr[0] = new int[3];
7.        arr[1] = new int[4];
8.        arr[2] = new int[2];
9.        //initializing a jagged array
10.       int count = 0;
11.       for (int i=0; i<arr.length; i++)
12.          for(int j=0; j<arr[i].length; j++)
13.             arr[i][j] = count++;
14.
15.       //printing the data of a jagged array
16.       for (int i=0; i<arr.length; i++){
17.          for (int j=0; j<arr[i].length; j++){
18.             System.out.print(arr[i][j]+" ");
19.          }
20.          System.out.println();//new line
21.       }
22.    }
23. }
```

Output:

```
0 1 2
3 4 5 6
7 8
```

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

```
1. //Java Program to demonstrate the addition of two matrices in Java
2. class Testarray5{
3. public static void main(String args[]){
4. //creating two matrices
5. int a[][]={{1,3,4},{3,4,5}};
6. int b[][]={{1,3,4},{3,4,5}};
7.
8. //creating another matrix to store the sum of two matrices
9. int c[][]=new int[2][3];
10.
11. //adding and printing addition of 2 matrices
12. for(int i=0;i<2;i++){
13. for(int j=0;j<3;j++){
14. c[i][j]=a[i][j]+b[i][j];
15. System.out.print(c[i][j]+" ");
16. }
17. System.out.println();//new line
18. }
19.
20. }}
```

Output:

```
2 6 8
6 8 10
```

Arrays of Characters

Character Array in Java is an Array that holds character data types values. In Java programming, unlike C, a character array is different from a string array, and neither a string nor a character array can be terminated by the NUL character.

The Java language uses UTF-16 representation in a character array, string, and StringBuffer classes.

The Character arrays are very advantageous in Java

. They are very efficient and faster. Also, the data can be manipulated without any allocations.

Java Strings are immutable means we can not change their internal state once they are created. However, char arrays allow us to manipulate after the creation. Even data structures List and Set are also acceptable.nisters

What is a character in Java

in Java, the characters are primitive data types. The char keyword

is used to declare the character types of variables and methods. The default value of a char data type **'\u0000'**. The character values are enclosed with a single quote. Its default size is 2 bytes.

The char data type can sore the following values:

- Any alphabet
- Numbers between 0 to 65,535 (Inclusive)
- special characters (@, #, $, %, ^, &, *, (,), ¢, £, ¥)
- 16-bit Unicode characters.

How to declare Character Arrays

We can declare the character array using the char keyword with square brackets. The character array can be declared as follows:

1. **char**[] JavaCharArray;

We can place the square bracket at the end of the statement as well:

```
1. char JavaCharArray[];
```

After the declaration, the next thing is initialization. Let's understand how to initialize the character array:

How to Initialize Character Array

We can initialize the character array with an initial capacity. For example, to assign an instance with size 5, initialize it as follows:

```
1. char[] JavaCharArray = new char[5];
```

The values will be assign to this array as follows:

```
1. char[] JavaCharArray = new char[5];
2. JavaCharArray[0] = 'a';
3. JavaCharArray[1] = 'b';
4. JavaCharArray[2] = 'c';
5. JavaCharArray[3] = 'd';
6. JavaCharArray[4] = 'e';
```

We can perform many useful operations such as sorting, looping, conversion to a string, and more on character array. Let's understand them:

Loops in Character Array

We can use for loop to iterate through the values in a character array.

Consider the below example:

CharArrayDemo.java:

```
1. public class CharArrayDemo {
2.     public static void main(String[] args) {
3.         char[] JavaCharArray = {'a', 'b', 'c', 'd', 'e'};
4.         for (char c:JavaCharArray) {
5.         System.out.println(c);
6.         }
7.     }
8. }
```

Output:

```
a
b
c
d
e
```

We can also iterate it as follows:

CharArrayDemo1.java:

```
1.  public class CharArrayDemo1 {
2.      public static void main(String[] args) {
3.          char[] JavaCharArray = {'a', 'b', 'c', 'd', 'e'};
4.          for (int i=0; i<JavaCharArray.length; i++) {
5.          System.out.println(JavaCharArray[i]);
6.          }
7.          }
8.  }
```

Output:

```
a
b
c
d
e
```

From the above examples, both programs are producing the same output. So we can iterate the character array using any of the above implementation methods.

Let's understand how to sort a character array:

Sorting a Character Array

The Arrays.sort() method is used to sort an array. Consider the below syntax of Array.sort() method:

```
1.  Arrays.sort(ArrayName)
```

Consider the below example:

CharArrayDemo2.java:

```
1. import java.util.Arrays;
2. public class CharArrayDemo2 {
3.     public static void main(String[] args) {
4.         char[] JavaCharArray = {'e', 'b', 'c', 'a', 'd'};
5.         Arrays.sort(JavaCharArray);
6.         System.out.println(Arrays.toString(JavaCharArray));
7.         }
8.         }
```

Output:

```
[a, b, c, d, e]
```

How to Convert a String Array into Character Array

We can easily convert a string array into a character array

using the toCharArray() method. It is the easiest method to convert a string field into a character field.

Consider the below example:

```
1. public class CharArrayDemo4 {
2.         public static void main(String[] args) {
3.          String value = "Raman"; //Enter String
4.         //Convert string to a char array.
5.          char[] array = value.toCharArray(); // Conversion to character from string

6.         for(char c : array) //Iterating array values
7.         {
8.         System.out.println(c);
9.         }
10.       }
11.        }
```

Output:

```
R
a
m
a
n
```

From the above example, a string array is converted into a character array.

String handling Using String Class

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

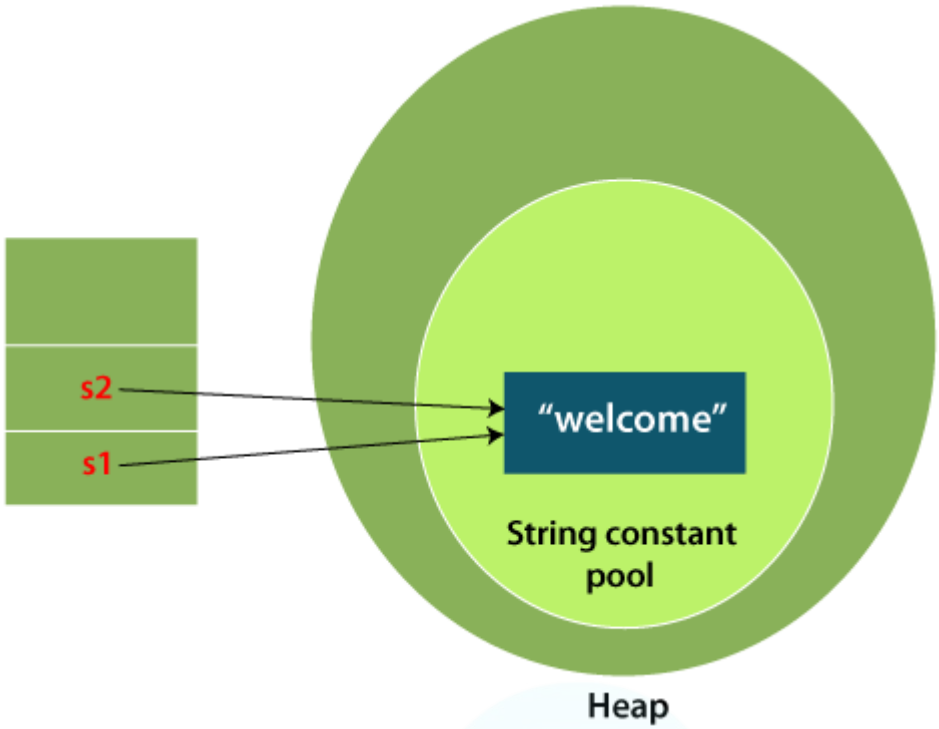
1) String Literal

Java String literal is created by using double quotes. For Example

1. String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM

will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
1. public class StringExample{
2. public static void main(String args[]){
```

```
3. String s1="java";//creating string by Java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating Java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}
```

Output:

```
java
strings
example
```

The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

Operations on String Handling Using String Buffer Class.

StringBuffer class is used to create a **mutable** string object. It means, it can be changed after it is created. It represents growable and writable character sequence.

It is similar to String class in Java both are used to create string, but stringbuffer object can be changed.

So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously. StringBuffer defines 4 constructors.

1. **StringBuffer**(): It creates an empty string buffer and reserves space for 16 characters.
2. **StringBuffer**(int size): It creates an empty string and takes an integer argument to set capacity of the buffer.
3. **StringBuffer**(String str): It creates a stringbuffer object from the specified string.

4. **StringBuffer**(charSequence []ch): It creates a stringbuffer object from the charsequence array.

Example: Creating a StringBuffer Object

In this example, we are creating string buffer object using StrigBuffer class and also testing its mutability.

```
public class Demo {



	public static void main(String[] args) {



		StringBuffer sb = new StringBuffer("study");

		System.out.println(sb);

		// modifying object

		sb.append("tonight");

		System.out.println(sb);    // Output: studytonight



	}

}
```

output

```
study

studytonight
```

Example: difference between String and StringBuffer

In this example, we are creating objects of String and StringBuffer class and modifying them, but only stringbuffer object get modified. See the below example.

```
class Test {

 public static void main(String args[])

 {

  String str = "study";

  str.concat("tonight");

  System.out.println(str);      // Output: study



  StringBuffer strB = new StringBuffer("study");

  strB.append("tonight");

  System.out.println(strB);    // Output: studytonight

 }

}
```

output

```
study

studytonight
```

Explanation:

Output is such because String objects are immutable objects. Hence, if we concatenate on the same String object, it won't be altered But StringBuffer creates mutable objects. Hence, it can be altered.

Important methods of StringBuffer class

The following methods are some most commonly used methods of StringBuffer class.

append()

This method will concatenate the string representation of any type of data to the end of the StringBuffer object. append() method has several overloaded forms.

```
StringBuffer append(String str)

StringBuffer append(int n)

StringBuffer append(Object obj)
```

The string representation of each parameter is appended to **StringBuffer** object.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer str = new StringBuffer("test");

		str.append(123);

		System.out.println(str);

	}

}
```

output

```
test123
```

insert()

This method inserts one string into another. Here are few forms of insert() method.

```
StringBuffer insert(int index, String str)

StringBuffer insert(int index, int num)

StringBuffer insert(int index, Object obj)
```

Here the first parameter gives the index at which position the string will be inserted and string representation of second parameter is inserted into **StringBuffer** object.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer str = new StringBuffer("test");

		str.insert(2, 123);

		System.out.println(str);

	}

}
```

output

```
test123
```

reverse()

This method reverses the characters within a **StringBuffer** object.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer str = new StringBuffer("Hello");

		str.reverse();

		System.out.println(str);

	}

}
```

output

```
olleH
```

replace()

This method replaces the string from specified start index to the end index.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer str = new StringBuffer("Hello World");

		str.replace( 6, 11, "java");

		System.out.println(str);

	}

}
```

output

```
Hello java
```

capacity()

This method returns the current capacity of **StringBuffer** object.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer str = new StringBuffer();

		System.out.println( str.capacity() );

	}
}
```

output

16

Note: Empty constructor reserves space for 16 characters. Therefore the output is 16.

---

ensureCapacity()

This method is used to ensure minimum capacity of **StringBuffer** object.

If the argument of the ensureCapacity() method is less than the existing capacity, then there will be no change in existing capacity.

If the argument of the ensureCapacity() method is greater than the existing capacity, then there will be change in the current capacity using following rule: **newCapacity = (oldCapacity*2) + 2**.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer str = new StringBuffer();
```

```
                System.out.println( str.capacity()); //output: 16 (since empty constructor reserves space for 16 characters)

                str.ensureCapacity(30); //greater than the existing capacity

                System.out.println( str.capacity()); //output: 34 (by following the rule - (oldcapacity*2) + 2.) i.e (16*2)+2 = 34.

        }

}
```

output

```
16

34
```

Extending Class and Inheritance:

Using Existing Classes

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs

(Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes

that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding

  (so runtime polymorphism

  can be achieved).

- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

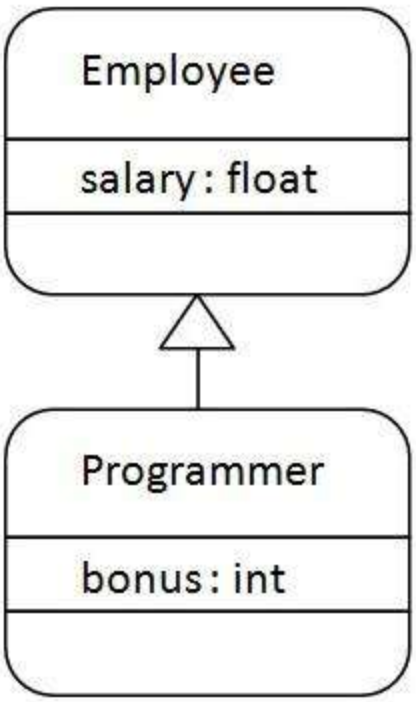
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1.  class Employee{
2.   float salary=40000;
3.  }
4.  class Programmer extends Employee{
5.   int bonus=10000;
6.   public static void main(String args[]){
7.     Programmer p=new Programmer();
8.     System.out.println("Programmer salary is:"+p.salary);
9.     System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

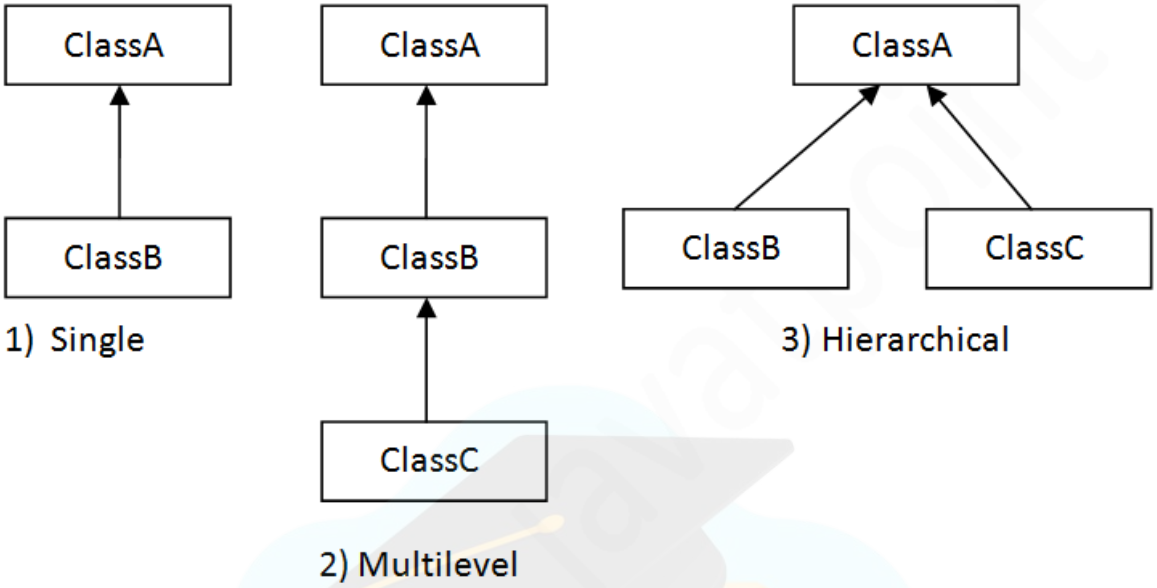
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

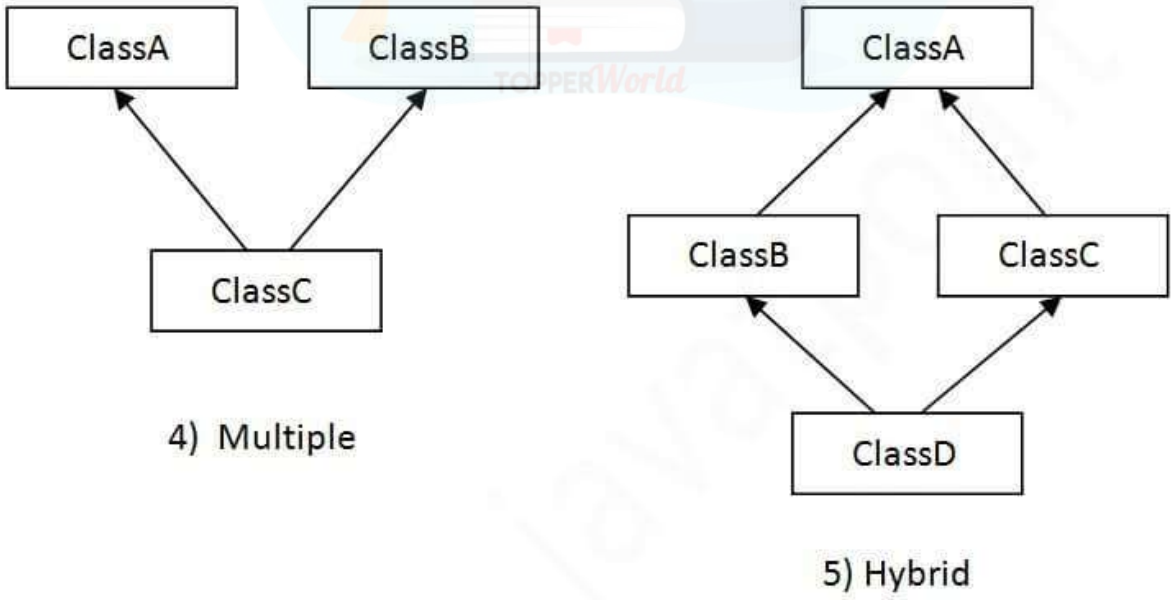
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
```

```
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
```

```
15. //c.bark();//C.T.Error
16. }}
```

Output:

```
meowing...
eating...
```

Choosing Base Class

A base class is a class, in an object-oriented programming language, from which other classes are derived. It facilitates the creation of other classes that can reuse the code implicitly inherited from the base class (except constructors and destructors). A programmer can extend base class functionality by adding or overriding members relevant to the derived class.

A base class may also be called parent class or superclass.

Access Attributes

An attribute is a property or characteristic of an entity. An entity may contain any number of attributes. One of the attributes is considered as the primary key. In an Entity-Relation model, attributes are represented in an elliptical shape.

Example: Student has attributes like name, age, roll number, and many more.

To uniquely identify the student, we use the primary key as a roll number as it is not repeated. Attributes can also be subdivided into another set of attributes. There are five such types of attributes: Simple, Composite, Single-valued, Multi-valued, and Derived attribute. One more attribute is their, i.e. Complex Attribute, this is the rarely used attribute.

Simple attribute :
An attribute that cannot be further subdivided into components is a simple attribute.
Example: The roll number of a student, the id number of an employee.

Composite attribute :
An attribute that can be split into components is a composite attribute.

Example: The address can be further split into house number, street number, city, state, country, and pin code, the name can also be split into first name middle name, and last name.

Single-valued attribute :
The attribute which takes up only a single value for each entity instance is a single-valued attribute.

Example: The age of a student.

Multi-valued attribute :
The attribute which takes up more than a single value for each entity instance is a multi-valued attribute.

Example: Phone number of a student: Landline and mobile.

Derived attribute :
An attribute that can be derived from other attributes is derived attributes.

Example: Total and average marks of a student.

Complex attribute :
Those attributes, which can be formed by the nesting of composite and multi-valued attributes, are called "*Complex Attributes*". These attributes are rarely used in DBMS(DataBase Management System). That's why they are not so popular.

Representation:
Complex attributes are the nesting of two or more composite and multi-valued attributes. Therefore, these multi-valued and composite attributes are called 'Components' of complex attributes.

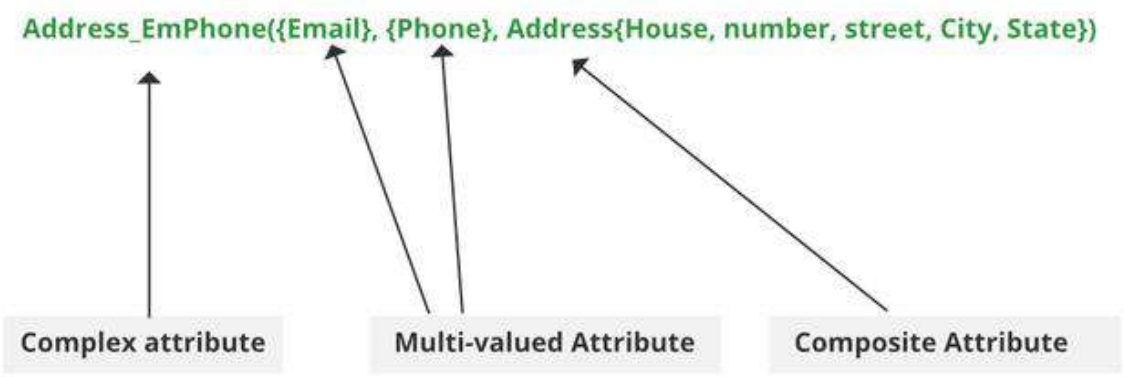
These components are grouped between parentheses '( )' and multi-valued attributes between curly braces '{ }', Components are separated by commas ', '.

Representation:
Complex attributes are the nesting of two or more composite and multi-valued attributes. Therefore, these multi-valued and composite attributes are called 'Components' of complex attributes.

These components are grouped between parentheses '( )' and multi-valued attributes between curly braces '{ }', Components are separated by commas ', '.

For example**:** let us consider a person having multiple phone numbers, emails, and an address.

Here, phone number and email are examples of multi-valued attributes and address is an example of the composite attribute, because it can be divided into house number, street, city, and state.



Complex attributes

Components:
Email, Phone number, Address(All are separated by commas and multi-valued components are represented between curly braces).

Complex Attribute: Address_EmPhone(You can choose any name).

Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

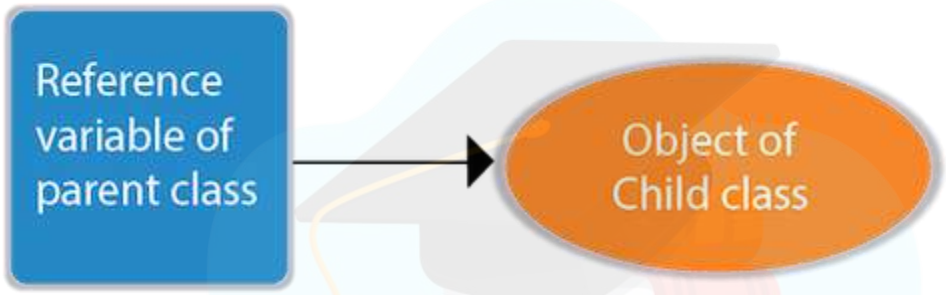
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
1. class A{}
2. class B extends A{}
1. A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
1. interface I{}
2. class A{}
3. class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1. class Bike{
2.   void run(){System.out.println("running");}
3. }
4. class Splendor extends Bike{
5.   void run(){System.out.println("running safely with 60km");}
6.
7.   public static void main(String args[]){
8.     Bike b = new Splendor();//upcasting
9.     b.run();
10.  }
11. }
```

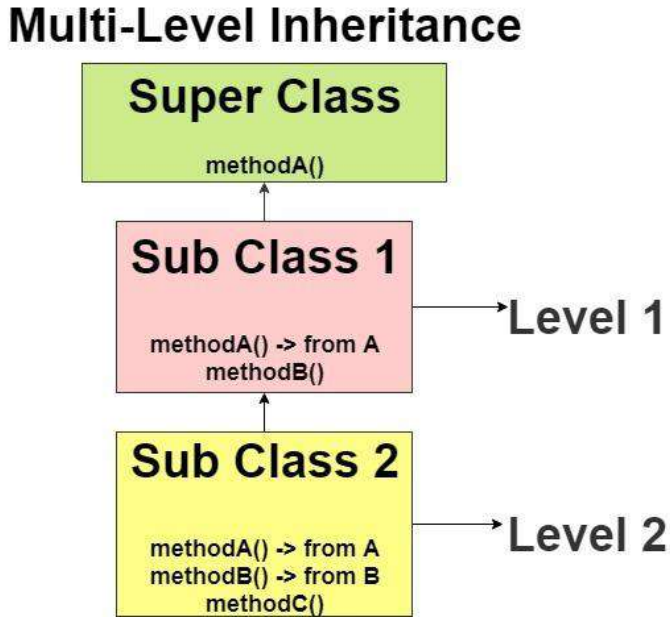
Output:

```
running safely with 60km.
```

Multiple Levels of Inheritance

When a class extends a class, which extends anther class then this is called **multilevel inheritance**. For example class C extends class B and class B extends class A then this **type of inheritance** is known as multilevel inheritance.

Lets see this in a diagram:



It's pretty clear with the diagram that in Multilevel inheritance there is a concept of grand parent class. If we take the example of this diagram, then class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and methods of class A along with class B that's what is called multilevel inheritance.

To learn the basics of inheritance refer this tutorial: **Inheritance in Java**

Multilevel Inheritance Example

In this example we have three classes – Car, Maruti and Maruti800. We have done a setup – class Maruti extends Car and class Maruti800 extends Maruti. With the help of this Multilevel hierarchy setup our Maruti800 class is able to use the methods of both the classes (Car and Maruti).

```
class Car{
  public Car()
  {
        System.out.println("Class Car");
  }
  public void vehicleType()
  {
        System.out.println("Vehicle Type: Car");
```

```
  }
}
class Maruti extends Car{
  public Maruti()
  {
        System.out.println("Class Maruti");
  }
  public void brand()
  {
        System.out.println("Brand: Maruti");
  }
  public void speed()
  {
        System.out.println("Max: 90Kmph");
  }
}
public class Maruti800 extends Maruti{

  public Maruti800()
  {
        System.out.println("Maruti Model: 800");
  }
  public void speed()
  {
        System.out.println("Max: 80Kmph");
  }
  public static void main(String args[])
  {
         Maruti800 obj=new Maruti800();
         obj.vehicleType();
         obj.brand();
         obj.speed();
  }
}
```

Output:

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max: 80Kmph
```

Abstraction through Abstract Classes

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

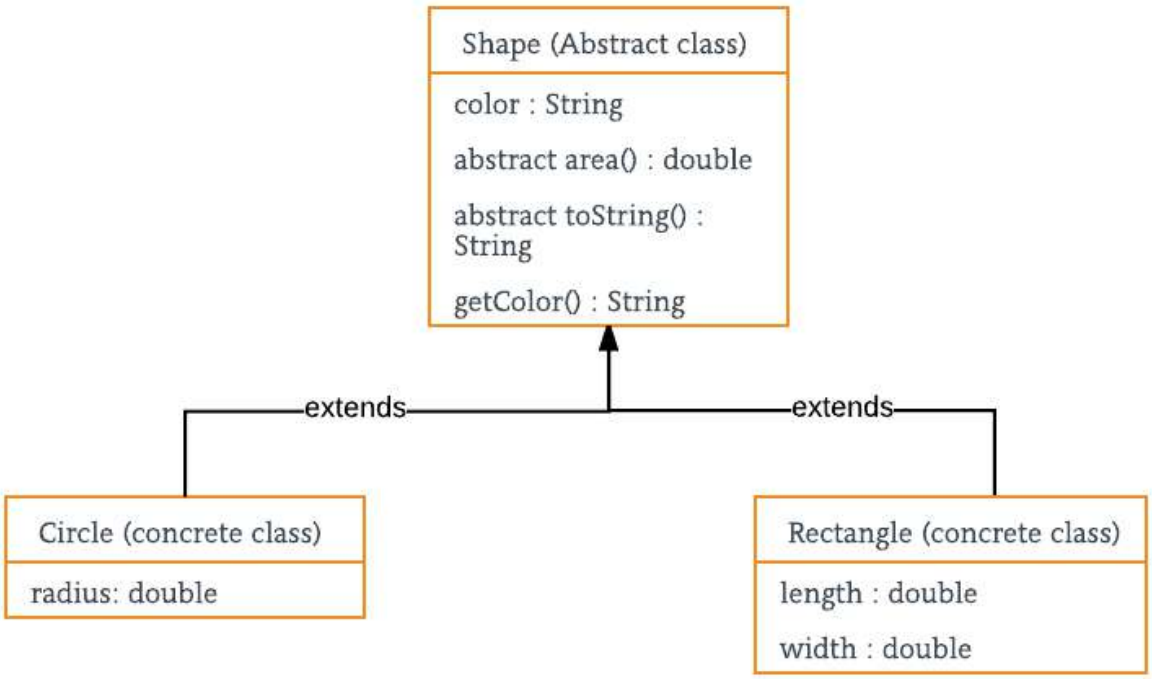
Abstract classes and Abstract methods :

1. An abstract class is a class that is declared with an abstract keyword.
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

When to use abstract classes and abstract methods with an example

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size, and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle, and so on — each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



- Java

```
// Java program to illustrate the
// concept of Abstraction
abstract class Shape {
   String color;

   // these are abstract methods
   abstract double area();
   public abstract String toString();

   // abstract class can have the constructor
   public Shape(String color)
   {
      System.out.println("Shape constructor called");
      this.color = color;
   }
```

```
    // this is a concrete method
    public String getColor() { return color; }
}
class Circle extends Shape {
    double radius;

    public Circle(String color, double radius)
    {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override public String toString()
    {
        return "Circle color is " + super.getColor()
            + "and area is : " + area();
    }
}
class Rectangle extends Shape {

    double length;
    double width;
```

```
    public Rectangle(String color, double length,
                     double width)
    {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override double area() { return length * width; }

    @Override public String toString()
    {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}
public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```


Output

Shape constructor called

Circle constructor called

Shape constructor called

Rectangle constructor called

Circle color is Redand area is : 15.205308443374602

Rectangle color is Yellowand area is : 8.0

Encapsulation vs Data Abstraction

1. Encapsulation is data hiding(information hiding) while Abstraction is detailed hiding(implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deal with exposing the interface to the user and hiding the details of implementation.
3. Encapsulated classes are java classes that follow data hiding and abstraction while We can implement abstraction by using abstract classes and interfaces.
4. Encapsulation is a procedure that takes place at the implementation level, while abstraction is a design-level process.

Advantages of Abstraction

1. It reduces the complexity of viewing things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only essential details are provided to the user.
4. It improves the maintainability of the application.
5. It improves the modularity of the application.
6. The enhancement will become very easy because without affecting end-users we can able to perform any type of changes in our internal system.

Using Final Modifier

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
```

```
7.    Bike9 obj=new  Bike9();
8.    obj.run();
9.    }
10. }//end of class
```

```
Output:Compile Time Error
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
1.  class Bike{
2.    final void run(){System.out.println("running");}
3.  }
4.
5.  class Honda extends Bike{
6.     void run(){System.out.println("running safely with 100kmph");}
7.
8.     public static void main(String args[]){
9.     Honda honda= new Honda();
10.    honda.run();
11.    }
12.}
```

```
Output:Compile Time Error
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
1.  final class Bike{}
2.
3.  class Honda1 extends Bike{
```

```
4.    void run(){System.out.println("running safely with 100kmph");}
5.
6.    public static void main(String args[]){
7.    Honda1 honda= new Honda1();
8.    honda.run();
9.    }
10.}
```

Output:Compile Time Error

Unit-2

Package & 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.

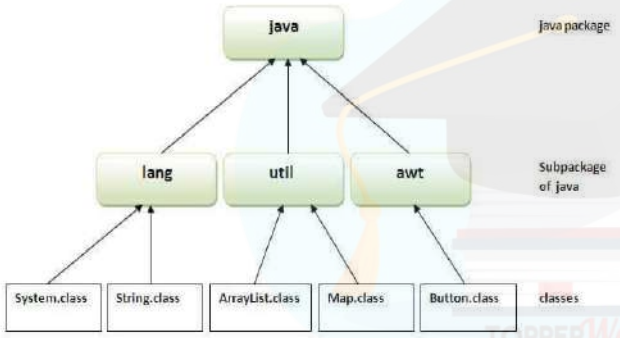
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7. }
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

```
Output:Welcome to package
```

Adding Classes from aPackage to your Program

The packages are used for categorization of the same type of classes and interface in a single unit. There is no core or in-built classes that belong to unnamed default package. To use any classes or interface in other class, we need to use it with their fully qualified type name. **But** some time, we've the need to use all or not all the classes or interface of a package then it's a tedious job to use in such a way discussed. Java supports *imports* statement to bring entire package, or certain classes into visibility. **It** provides flexibility to the programmer to save a lot of time just by importing the classes in his/her program, instead of rewriting them.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and must be before of any class definitions. This is the general form is as follows:

import *pkg1 (.pkg2). [classname|*l;*

For example

import java.util.Vector;

import java.io.*;

import java.awt.*;

import java.awt.event.*;

The star (*) increases the compilation time, if the package size is very large. A good programming methodology says that give the fully qualified type name explicitly, i.e. import only those classes or interface that you want to use instead of importing whole packages. Yet the start will not increase the overhead on run-time performance of code or size of classes.

Example:

//Student.java

package student;

class Student

{

private int rollno;

private String name;

private String address;

public Student(int rno, String sname, String sadd}

{

rollno = rno;

name = sname;

address = sadd;

}

public void show()

{

System.out.println(“Roll No :: ” + rollno);

System.out.println(“Name :: ” + name);

System.out.println(“Address :: ” + address);

}

```
}
// Test.java

package student;
class Test extends Student

{
protected int marksSubjecti;
protected int marksSubject2;
protected int marksSubject3;
protected int marksSubject4;
public Test(int rno, String sname, String sadd,int mi, int m2, int m3, int m4)
{
super(rno,sname,sadd);
marksSubjecti = mi;
marksSubject2 = m2;
marksSubject3 = m3;
marksSubject4 = m4;
}
public void show()
{
super.show();
System.out.println("Marks of Subject1 :: " + marksSubject1);
System.out.println("Marks of Subject2 :: " + marksSubject2);
System.out.println("Marks of Subject3 :: " + marksSubject3);
System.out.println("Marks of Subject4 :: " + marksSubject4);
}
}

//Result.java

package student;
public class Result ex..tends Test
```

```
{
private int totalMarks;
private float percentage;
private char grade;
public Result(int rno, String sname, String sadd,int mi, int m2, int m3, int m4)
{
super(rno,sname,sadd,ml,m2,m3,m4);

totalMarks = marksSubject1 + marksSubject2 + marksSubject3 + marksSubject4;
percentage = (totalMarks*100.00F/600.00F);
if (percentage >=50.00F)
grade='D';
else
if(percentage >=55.00F && percentage<=60.00F)
grade = 'C';
else
if (percentage >=6l.00F && percentage<=70.00F)
grade = 'B';
else
if(percentage >=7l.00F && percentage<=75.00F)
grade = 'A';
else
if (percentage >=76.00F && percentage<=85.00F)
grade = 'H';
else
grade = 'S';
}
public void show()
{
super.show();
System.out.println("Total Marks :: " + totalMarks);
System.out.println("Percentage :: " + percentage);
System.out.println("Grade :: " + grade);
```

```
}
}
//ImportPackageDemo.java
import student.Result;
public class ImportPackageDemo
{
public static void main(String ar[])
{
Result ob = new Result (1001, "Alice", "New York",135,130,132,138);
ob.show ();
}
}
```

In the source file ImportPackageDemo, *import student. Result* is the first statement; it will import the class Result from the student package.

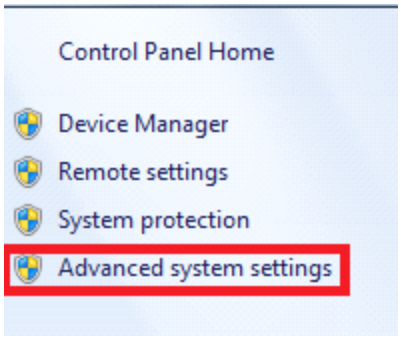
Understanding CLASSPATH

CLASSPATH: CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

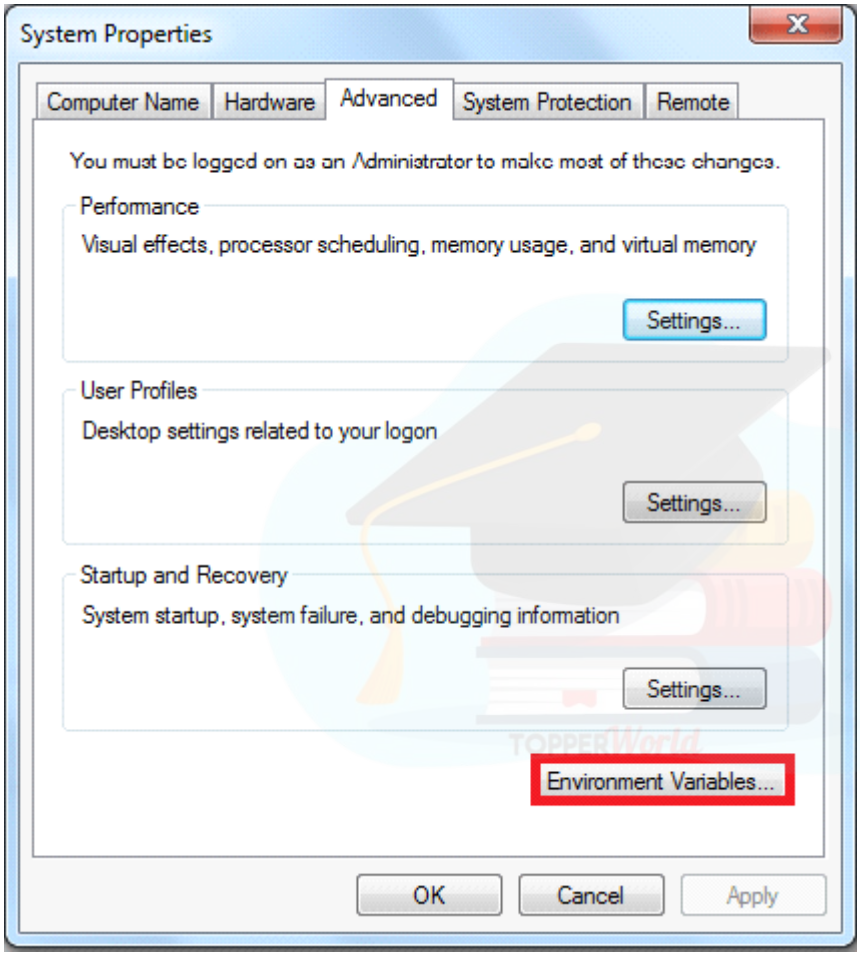
Let's see how to set CLASSPATH of MySQL database:

Step 1: Click on the Windows button and choose Control Panel. Select System.

Step 2: Click on **Advanced System Settings**.



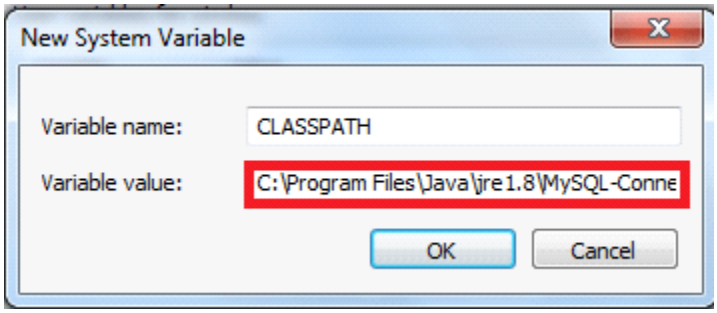
Step 3: A dialog box will open. Click on Environment Variables.



Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;.;*

Remember: Put ;.; at the end of the CLASSPATH.



Difference between PATH and CLASSPATH

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is also an environment variable.
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

Access Protection in Packages

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Access Modifier

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
8.    A obj=new A();
```

```
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12.}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1.  class A{
2.  private A(){}//private constructor
3.  void msg(){System.out.println("Hello java");}
4.  }
5.  public class Simple{
6.   public static void main(String args[]){
7.     A obj=new A();//Compile Time Error
8.   }
9.  }
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1.  //save by A.java
2.  package pack;
3.  class A{
4.    void msg(){System.out.println("Hello");}
5.  }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();//Compile Time Error
7.    obj.msg();//Compile Time Error
8.   }
9. }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

---

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
```

```
3. import pack.*;
4.
5. class B extends A{
6.   public static void main(String args[]){
7.    B obj = new B();
8.    obj.msg();
9.   }
10.}
```

Output:Hello

---

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }
```

```
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.   public static void main(String args[]){
8.    A obj = new A();
9.    obj.msg();
10.  }
11.}
```

Output:Hello

Concept of Interface

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- `Language` is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```
Run Code

Output

```
The area of the rectangle is 30
```

In the above example, we have created an interface named `Polygon`. The interface contains an abstract method `getArea()`.

Here, the `Rectangle` class implements `Polygon`. And, provides the implementation of the `getArea()` method

Exception Handling

The Idea behind Exceptions:

Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

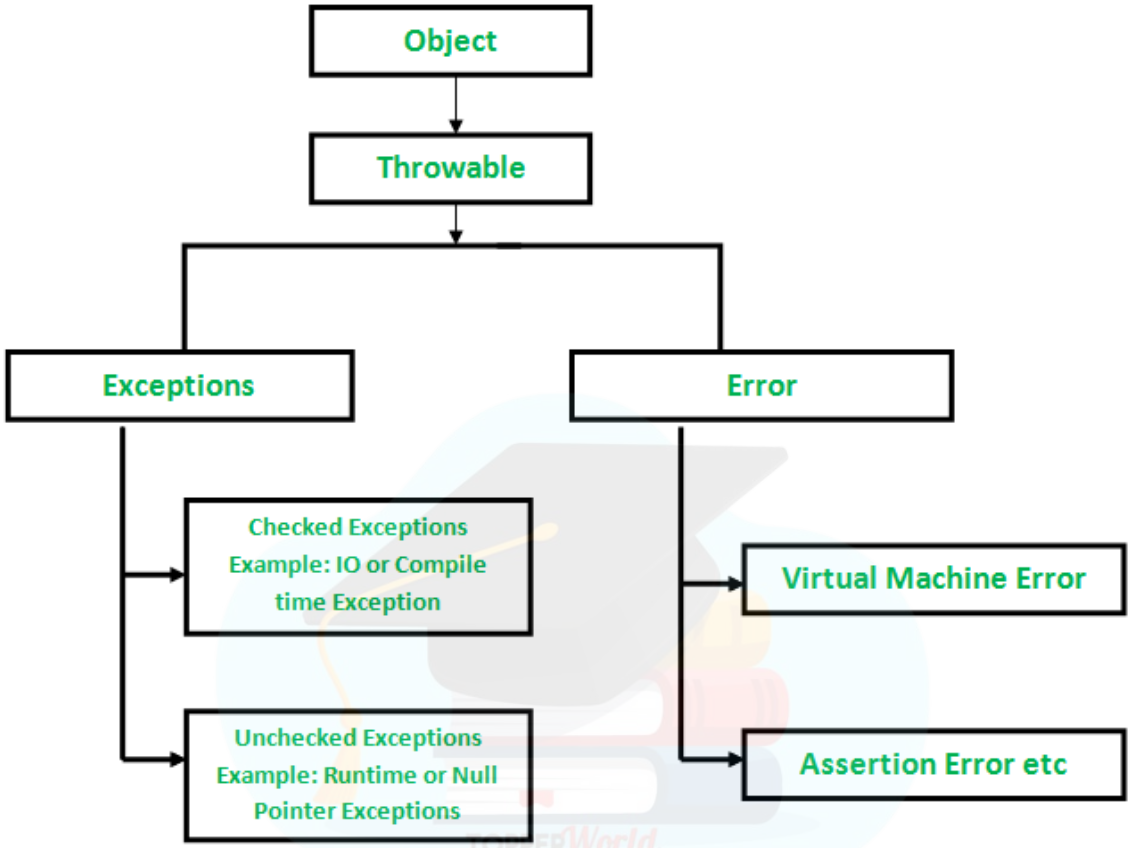
Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

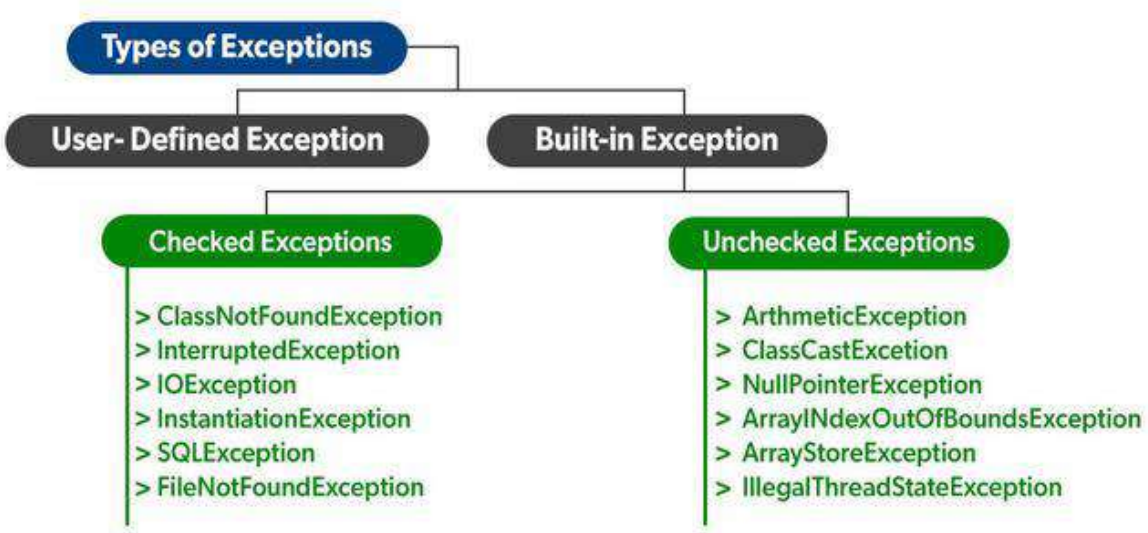
Exception Hierarchy

All exception and error types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
    - Checked Exception
    - Unchecked Exception
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

A. Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: *For checked vs unchecked exception, see Checked vs Unchecked Exceptions*

B. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

Dealing with Exceptions:

The `try-catch` is the simplest method of handling exceptions. Put the code you want to run in the `try` block, and any Java exceptions that the code throws are caught by one or more `catch` blocks.

This method will catch any type of Java exceptions that get thrown. This is the simplest mechanism for handling exceptions.

```
try {
  // block of code that can throw exceptions
} catch (Exception ex) {
  // Exception handler
}
```

Note: You can't use a `try` block alone. The try block should be immediately followed either by a `catch` or `finally` block.

Catching specific Java exceptions

You can also specify specific exceptions you would like to catch. This allows you to have dedicated code to recover from those errors. For example, some Java errors returned from a REST API are recoverable and others are not. This allows you to treat those conditions separately.

A `try` block can be followed by one or more `catch` blocks, each specifying a different type. The first catch block that handles the exception class or one of its superclasses will be executed. So, make sure to catch the most specific class first.

```
try {
  // block of code that can throw exceptions
} catch (ExceptionType1 ex1) {
  // exception handler for ExceptionType1
} catch (ExceptionType2 ex2) {
  // Exception handler for ExceptionType2
}
```

If an exception occurs in the try block, the exception is thrown to the first catch block. If not, the Java exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches.

The finally block

Any code that must be executed irrespective of occurrence of an exception is put in a finally block. In other words, a finally block is executed in all circumstances. For example, if you have an open connection to a database or file, use the finally block to close the connection even if the try block throws a Java exception.

```
try {
  // block of code that can throw exceptions
} catch (ExceptionType1 ex1) {
  // exception handler for ExceptionType1
} catch (ExceptionType2 ex2) {
  // Exception handler for ExceptionType2
} finally {
  // finally block always executes
}
```

Handling uncaught or runtime Java exceptions

Uncaught or runtime exceptions happen unexpectedly, so you may not have a try-catch block to protect your code and the compiler cannot give you warnings either. Thankfully, Java provides a powerful mechanism for handling runtime exceptions. Every Thread includes a hook that allows you to set an UncaughtExceptionHandler. Its interface declares the method uncaughtException(Thread t1, Throwable e1). It will be invoked when a given thread t1 terminates due to the given uncaught exception e1.

```
Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread t1, Throwable e1) {
  // Exception handling code
  }
});
```

Exception Objects

An exception object is an instance of an exception class. It gets created and handed to the Java runtime when an exceptional event occurred that disrupted the normal flow of the application. This is called “to throw an exception” because in Java you use the keyword “throw” to hand the exception to the runtime.

Checked And Unchecked Exceptions in Java:

Unchecked exceptions extend the *RuntimeException*. You should use them for internal errors that you can’t anticipate and that, most often, the application can’t recover from. Methods can but don’t need to handle or specify an unchecked exception. Typical examples that throw unchecked exceptions are:

- the missing initialization of a variable which results in a *NullPointerException* or
- the improper use of an API that causes an *IllegalArgumentException*

How to Handle an Exception

Java provides two different options to handle an exception. You can either use the try-catch-finally approach to handle all kinds of exceptions. Or you can use the try-with-resource approach which allows an easier cleanup process for resources.

Try-Catch-Finally

That is the classical approach to handle an exception in Java. It can consist of 3 steps:

- a *try* block that encloses the code section which might throw an exception,
- one or more *catch* blocks that handle the exception and
- a *finally* block which gets executed after the *try* block was successfully executed or a thrown exception was handled.

The *try* block is required, and you can use it with or without a *catch* or *finally* block.

The Try Block

Let's talk about the *try* block first. It encloses the part of your code that might throw the exception. If your code throws more than one exception, you can choose if you want to:

- use a separate *try* block for each statement that could throw an exception or
- use one *try* block for multiple statements that might throw multiple exceptions.

The following example shows a try block which encloses three method calls.

```
public void performBusinessOperation() {

          try {

                    doSomething("A message");

                    doSomethingElse();

                    doEvenMore();

          }

          // see following examples for catch and finally blocks

}



public void doSomething(String input) throws MyBusinessException {

          // do something useful ...

          throw new MyBusinessException("A message that describes the error.");

}



public void doSomethingElse() {

          // do something else ...

}



public void doEvenMore() throws NumberFormatException{
```

```
        // do even more ...

}
```

As you can see in the method definitions, only the first and the third method specify an exception. The first one might throw a *MyBusinessException*, and the *doEvenMore* method might throw a *NumberFormatException*.

In the next step, you can define one *catch* block for each exception class you want to handle and one *finally* block. All checked exceptions that are not handled by any of the catch blocks need to be specified.

The Catch Block

You can implement the handling for one or more exception types within a *catch* block. As you can see in the following code snippet, the catch clause gets the exception as a parameter. You can reference it within the catch block by the parameter name.

```
public void performBusinessOperation() {

        try {

                doSomething("A message");

                doSomethingElse();

                doEvenMore();

        } catch (MyBusinessException e) {

                e.printStackTrace();

        } catch (NumberFormatException e) {

                e.printStackTrace();

        }

}
```

The previous code sample shows two catch blocks. One to handle the *MyBusinessException* and one to handle the *NumberFormatException*. Both blocks handle the exceptions in the same way. Since Java 7, you can do the same with just one catch block.

```
public void performBusinessOperation() {
```

```
        try {

                doSomething("A message");

                doSomethingElse();

                doEvenMore();

        } catch (MyBusinessException|NumberFormatException e) {

                e.printStackTrace();

        }

}
```

The implementation of the catch blocks in the previous examples is very basic. I just call the *printStackTrace* method which writes the class, message and call stack of the exception to system out.

```
com.stackify.example.MyBusinessException: A message that describes the error.

        at com.stackify.example.TestExceptionHandling.doSomething(TestExceptionHandling.j
ava:84)

        at com.stackify.example.TestExceptionHandling.performBusinessOperation(TestExcept
ionHandling.java:25)

        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.j
ava:43)

        at java.lang.reflect.Method.invoke(Method.java:497)

        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.ja
va:47)

        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.jav
a:12)

        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java
:44)

        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:
17)

        at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)

        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:
70)
```

```
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:
50)

        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)

        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:63)

        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:236)

        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:53)

        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:229)

        at org.junit.runners.ParentRunner.run(ParentRunner.java:309)

        at org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestRefer
ence.java:50)

        at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)

        at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunn
er.java:459)

        at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunn
er.java:675)

        at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.ja
va:382)

        at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.j
ava:192)
```

In a real application, you might want to use a more advanced implementation. You can, for example, show an error message to the user and request a different input or you could write a record into the work log of your batch process. Sometimes, it might even be ok to catch and ignore the exception.

And in production, you also need to monitor your application and its exception handling. That's where Retrace and its error monitoring capabilities become very helpful.

The Finally Block

The *finally* block gets executed after the successful execution of the *try* block or after one of the *catch* blocks handled an exception. It is, therefore, a good place to implement any cleanup logic, like closing a connection or an *InputStream*.

You can see an example of such a cleanup operation in the following code snippet. The *finally* block will be executed, even if the instantiation of the *FileInputStream* throws a *FileNotFoundException* or the processing of the file content throws any other exception.

```
FileInputStream inputStream = null;

try {

          File file = new File("./tmp.txt");

          inputStream = new FileInputStream(file);



          // use the inputStream to read a file



} catch (FileNotFoundException e) {

          e.printStackTrace();

} finally {

          if (inputStream != null) {

                    try {

                              inputStream.close();

                    } catch (IOException e) {

                              e.printStackTrace();

                    }

          }

}
```

As you've seen, the *finally* block provides a good option to prevent any leaks. And before Java 7, it was a best practice to put all cleanup code into a *finally* block.

Try-With-Resource

That changed when Java 7 introduced the try-with-resource statement. It automatically closes all resources that implement the *AutoCloseable* interface. And that is the case for most Java objects that you need to close.

The only thing you need to do to use this feature is to instantiate the object within the try clause. You also need to handle or specify all exceptions that might be thrown while closing the resource.

The following code snippet shows the previous example with a try-with-resource statement instead of a try-catch-finally statement.

```
File file = new File("./tmp.txt");

try (FileInputStream inputStream = new FileInputStream(file);) {

          // use the inputStream to read a file

} catch (FileNotFoundException e) {

          e.printStackTrace();

} catch (IOException e) {

          e.printStackTrace();

}
```

As you can see, the try-with-resource statement is a lot easier to implement and read. And the handling of the *IOException*, which might be thrown while closing the *FileInputStream*, doesn't require a nested try-catch statement. It is now handled by a catch block of the try-with-resource statement.

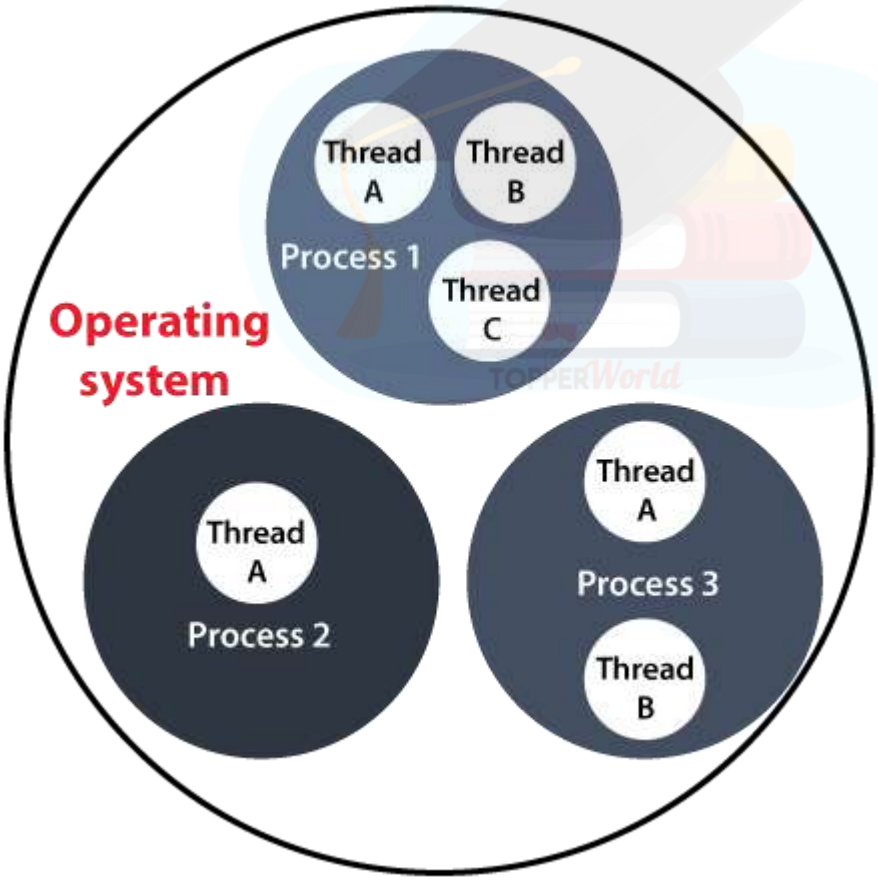
Multithreading Programming

The Java Thread Model

Before introducing the **thread concept**, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, **Thread Concept** was introduced.

A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the **Thread concept in Java**. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables

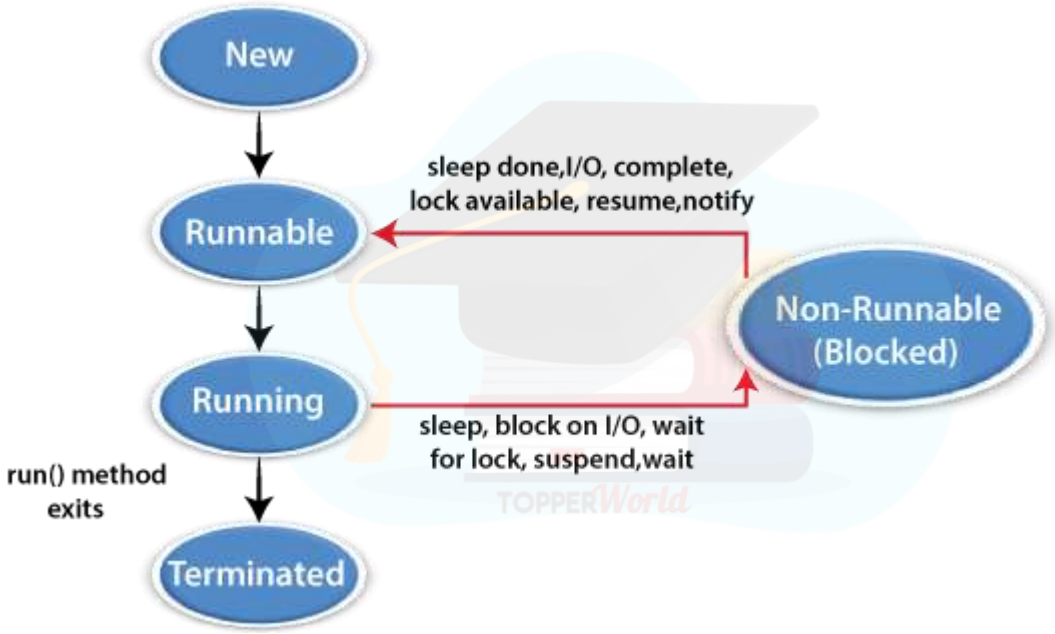
and program counter. When multiple threads are executed in parallel at the same time, this process is known as **Multithreading**.

In a simple way, a Thread is a:

- Feature through which we can perform multiple activities within a single process.
- Lightweight process.
- Series of executed statements.
- Nested sequence of method calls.

Thread Model

Just like a process, a thread exists in several states. These states are as follows:



1) New (Ready to run)

A thread is in **New** when it gets CPU time.

2) Running

A thread is in **a Running** state when it is under execution.

3) Suspended

A thread is in the **Suspended** state when it is temporarily inactive or under execution.

4) Blocked

A thread is in the **Blocked** state when it is waiting for resources.

5) Terminated

A thread comes in this state when at any given time, it halts its execution immediately.

Creating Thread

A thread is created either by "creating or implementing" the **Runnable Interface** or by extending the **Thread class**. These are the only two ways through which we can create a thread.

Let's dive into details of both these way of creating a thread:

Thread Class

A **Thread class** has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the **Object** class. The **Object** class implements the **Runnable** interface. The thread class has the following constructors that are used to perform various operations.

- **Thread()**
- **Thread(Runnable, String name)**
- **Thread(Runnable target)**
- **Thread(ThreadGroup group, Runnable target, String name)**
- **Thread(ThreadGroup group, Runnable target)**
- **Thread(ThreadGroup group, String name)**
- **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**

Runnable Interface(run() method)

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the **run()** method to perform an action for the thread.

start() method

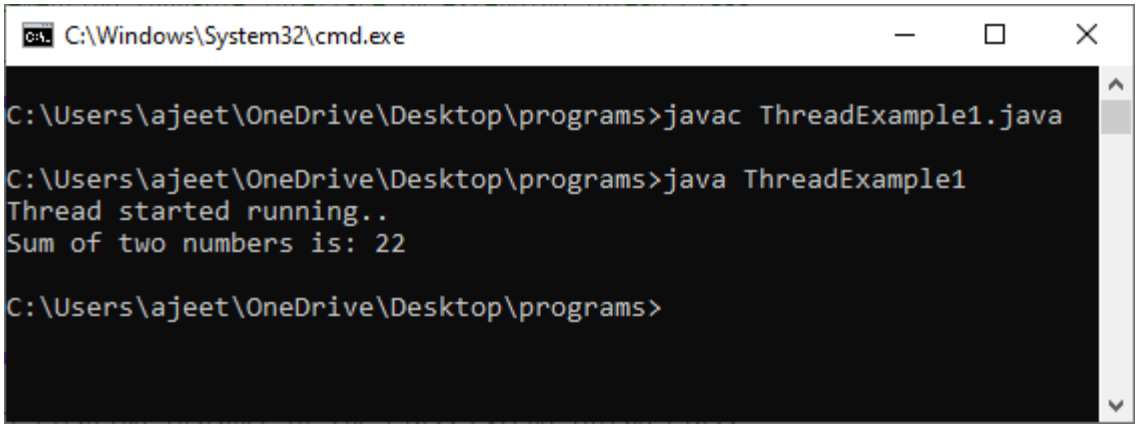
The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the **start()** method, the thread changes the state from New to Runnable. It executes the **run() method** when the thread gets the correct time to execute it.

Let's take an example to understand how we can create a Java thread by extending the Thread class:

ThreadExample1.java

```
1.  // Implementing runnable interface by extending Thread class
2.  public class ThreadExample1 extends Thread {
3.      // run() method to perform action for thread.
4.      public void run()
5.      {
6.        int a= 10;
7.        int b=12;
8.        int result = a+b;
9.        System.out.println("Thread started running..");
10.       System.out.println("Sum of two numbers is: "+ result);
11.     }
12.     public static void main( String args[] )
13.     {
14.      // Creating instance of the class extend Thread class
15.       ThreadExample1 t1 = new  ThreadExample1();
16.       //calling start method to execute the run() method of the Thread class

17.       t1.start();
18.     }
19.}
```

Output:



Creating thread by implementing the runnable interface

In Java, we can also create a thread by implementing the runnable interface. The runnable interface provides us both the run() method and the start() method.

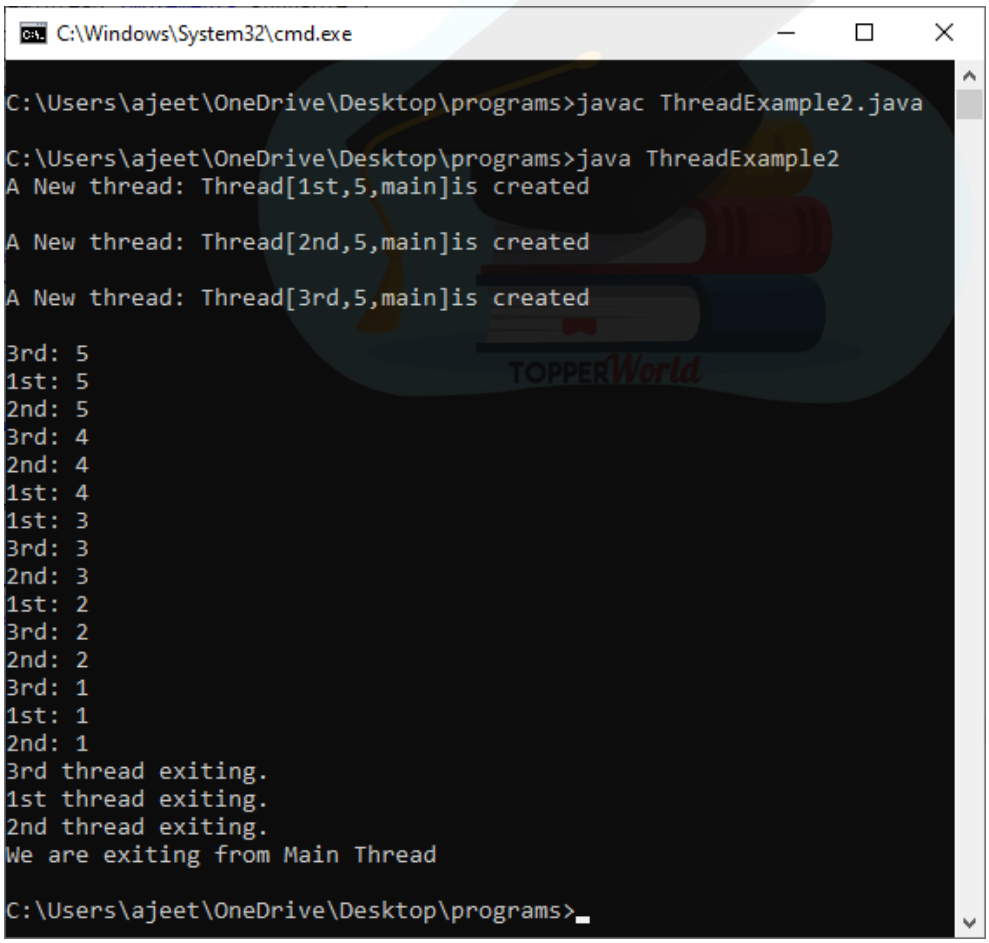
Let's takes an example to understand how we can create, start and run the thread using the runnable interface.

ThreadExample2.java

```
1.  class NewThread implements Runnable {
2.      String name;
3.      Thread thread;
4.      NewThread (String name){
5.          this.name = name;
6.          thread = new Thread(this, name);
7.          System.out.println( "A New thread: " + thread+ "is created\n" );
8.          thread.start();
9.      }
10.     public void run() {
11.     try {
12.         for(int j = 5; j > 0; j--) {
13.             System.out.println(name + ": " + j);
14.             Thread.sleep(1000);
15.         }
16.     }catch (InterruptedException e) {
17.         System.out.println(name + " thread Interrupted");
18.     }
19.      System.out.println(name + " thread exiting.");
20.     }
```

```
21.}
22.class ThreadExample2 {
23.    public static void main(String args[]) {
24.        new NewThread("1st");
25.        new NewThread("2nd");
26.        new NewThread("3rd");
27.        try {
28.            Thread.sleep(8000);
29.        } catch (InterruptedException excetion) {
30.            System.out.println("Inturruption occurs in Main Thread");
31.        }
32.        System.out.println("We are exiting from Main Thread");
33.    }
34.}
```

Output:



In the above example, we perform the Multithreading by implementing the runnable interface. To learn more about Multithreading,

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

FileName: ThreadPriorityExample.java

```
1. // Importing the required classes
2. import java.lang.*;
3.
4. public class ThreadPriorityExample extends Thread
5. {
6.
7. // Method 1
8. // Whenever the start() method is called by a thread
```

```
9.  // the run() method is invoked
10. public void run()
11. {
12. // the print statement
13. System.out.println("Inside the run() method");
14. }
15.
16. // the main method
17. public static void main(String argvs[])
18. {
19. // Creating threads with the help of ThreadPriorityExample class
20. ThreadPriorityExample th1 = new ThreadPriorityExample();
21. ThreadPriorityExample th2 = new ThreadPriorityExample();
22. ThreadPriorityExample th3 = new ThreadPriorityExample();
23.
24. // We did not mention the priority of the thread.
25. // Therefore, the priorities of the thread is 5, the default value
26.
27. // 1st Thread
28. // Displaying the priority of the thread
29. // using the getPriority() method
30. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
31.
32. // 2nd Thread
33. // Display the priority of the thread
34. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
35.
36. // 3rd Thread
37. // // Display the priority of the thread
38. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
39.
40. // Setting priorities of above threads by
41. // passing integer arguments
42. th1.setPriority(6);
43. th2.setPriority(3);
44. th3.setPriority(9);
45.
```

```
46. // 6
47. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
48.
49. // 3
50. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
51.
52. // 9
53. System.out.println("Priority of the thread th3 is : " + th3.getPriority());
54.
55. // Main thread
56.
57. // Displaying name of the currently executing thread
58. System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName())
    ;
59.
60. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
61.
62. // Priority of the main thread is 10 now
63. Thread.currentThread().setPriority(10);
64.
65. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
66. }
67. }
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

FileName: ThreadPriorityExample1.java

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
1.  class Table{
2.  void printTable(int n){//method not synchronized
3.    for(int i=1;i<=5;i++){
4.      System.out.println(n*i);
5.      try{
6.       Thread.sleep(400);
7.      }catch(Exception e){System.out.println(e);}
8.    }
9.
10.  }
11. }
12.
13. class MyThread1 extends Thread{
14. Table t;
15. MyThread1(Table t){
16. this.t=t;
17. }
18. public void run(){
19. t.printTable(5);
20. }
21.
22. }
23. class MyThread2 extends Thread{
24. Table t;
25. MyThread2(Table t){
26. this.t=t;
27. }
```

```
28. public void run(){
29. t.printTable(100);
30. }
31. }
32.
33. class TestSynchronization1{
34. public static void main(String args[]){
35. Table obj = new Table();//only one object
36. MyThread1 t1=new MyThread1(obj);
37. MyThread2 t2=new MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```
1.  //example of java synchronized method
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.  }
12. }
13.
14. class MyThread1 extends Thread{
15. Table t;
16. MyThread1(Table t){
17. this.t=t;
18. }
19. public void run(){
20. t.printTable(5);
21. }
22.
23. }
24. class MyThread2 extends Thread{
25. Table t;
26. MyThread2(Table t){
27. this.t=t;
28. }
29. public void run(){
30. t.printTable(100);
31. }
32. }
33.
34. public class TestSynchronization2{
35. public static void main(String args[]){
```

```
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }
```

Output:

```
5
 10
 15
 20
 25
 100
 200
 300
 400
 500
```

Deadlocks inter-thread communication:

Java provide benefits of avoiding thread pooling using inter-thread communication.
The wait(), notify(), and notifyAll() methods of Object class are used for this purpose. These method are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context

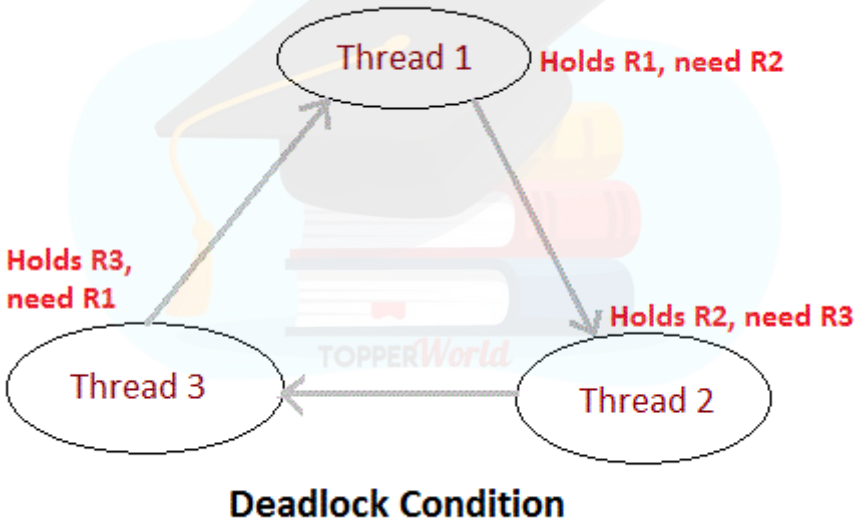
- wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- notify() wakes up a thread that called wait() on same object.
- notifyAll() wakes up all the thread that called wait() on same object.

Difference between wait() and sleep()

wait()	sleep()
called from synchronised block	no such requirement
monitor is released	monitor is not released
gets awake when notify() or notifyAll() method is called.	does not get awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition	sleep() method is simply used to put your thread on sleep.

Thread Pooling

Thread Deadlock in Java



Deadlock Condition

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

Example

In this example, multiple threads are accessing same method that leads to deadlock condition. When a thread holds the resource and does not release it then other thread will wait and in deadlock condition wait time is never ending.

```
class Pen{}

class Paper{}


public class Write {


  public static void main(String[] args)

  {

    final Pen pn =new Pen();

    final Paper pr =new Paper();


    Thread t1 = new Thread() {

      public void run()

      {

        synchronized(pn)

        {

          System.out.println("Thread1 is holding Pen");

          try{

            Thread.sleep(1000);

          }

          catch(InterruptedException e){

            // do something

          }

          synchronized(pr)

          {

            System.out.println("Requesting for Paper");

        }
```

```
    }
  };
  Thread t2 = new Thread() {
    public void run()
    {
      synchronized(pr)
      {
        System.out.println("Thread2 is holding Paper");
        try {
          Thread.sleep(1000);
        }
        catch(InterruptedException e){
          // do something
        }
        synchronized(pn)
        {
          System.out.println("requesting for Pen");
        }
      }
    }
  };
  t1.start();
 t2.start();
}}
```

```
OUTPUT:

Thread1 is holding Pen

Thread2 is holding Paper
```

Deadlocks:

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Example of Deadlock in Java

TestDeadlockExample1.java


```
1.  public class TestDeadlockExample1 {
2.    public static void main(String[] args) {
3.      final String resource1 = "ratan jaiswal";
4.      final String resource2 = "vimal jaiswal";
5.      // t1 tries to lock resource1 then resource2
6.      Thread t1 = new Thread() {
7.        public void run() {
8.          synchronized (resource1) {
9.           System.out.println("Thread 1: locked resource 1");
10.
11.          try { Thread.sleep(100);} catch (Exception e) {}
12.
```

```
13.            synchronized (resource2) {
14.             System.out.println("Thread 1: locked resource 2");
15.            }
16.          }
17.       }
18.     };
19.
20.     // t2 tries to lock resource2 then resource1
21.     Thread t2 = new Thread() {
22.       public void run() {
23.         synchronized (resource2) {
24.           System.out.println("Thread 2: locked resource 2");
25.
26.           try { Thread.sleep(100);} catch (Exception e) {}
27.
28.           synchronized (resource1) {
29.             System.out.println("Thread 2: locked resource 1");
30.           }
31.         }
32.       }
33.     };
34.
35.
36.     t1.start();
37.     t2.start();
38.   }
39. }
```

Output:

```
Thread 1: locked resource 1
      Thread 2: locked resource 2
```

Input/Output in Java:

I/O Basic:

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

```
1. System.out.println("simple message");
2. System.err.println("error message");
```

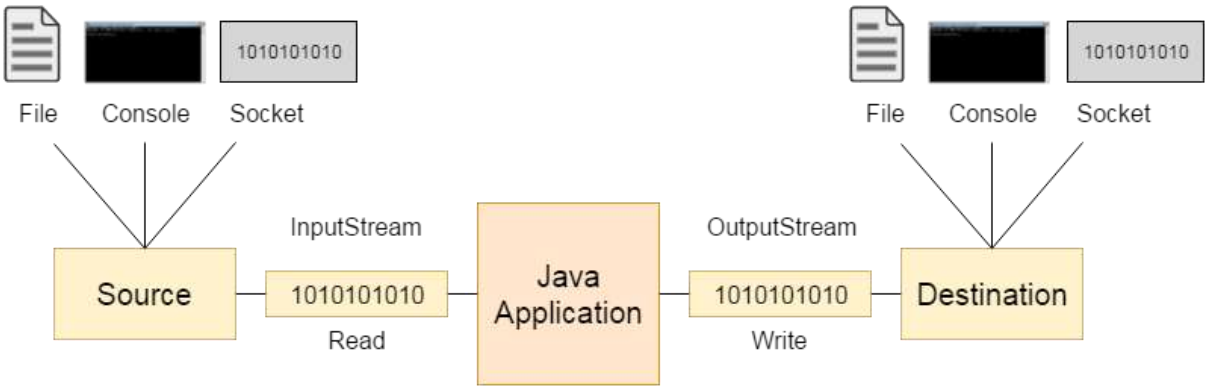
Let's see the code to get **input** from console.

```
1. int i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((char)i);//will print the character
```

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



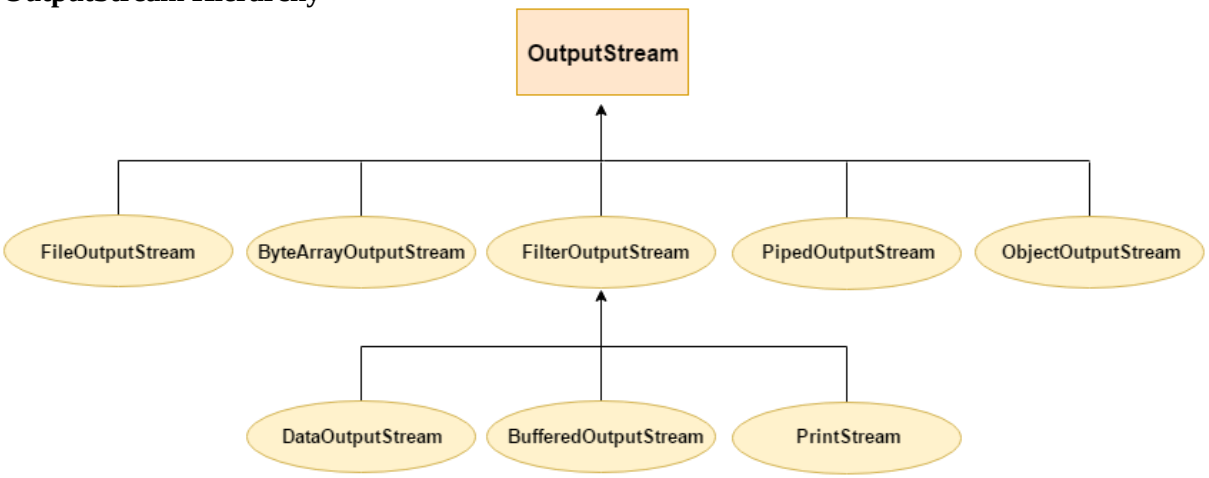
OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output s
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



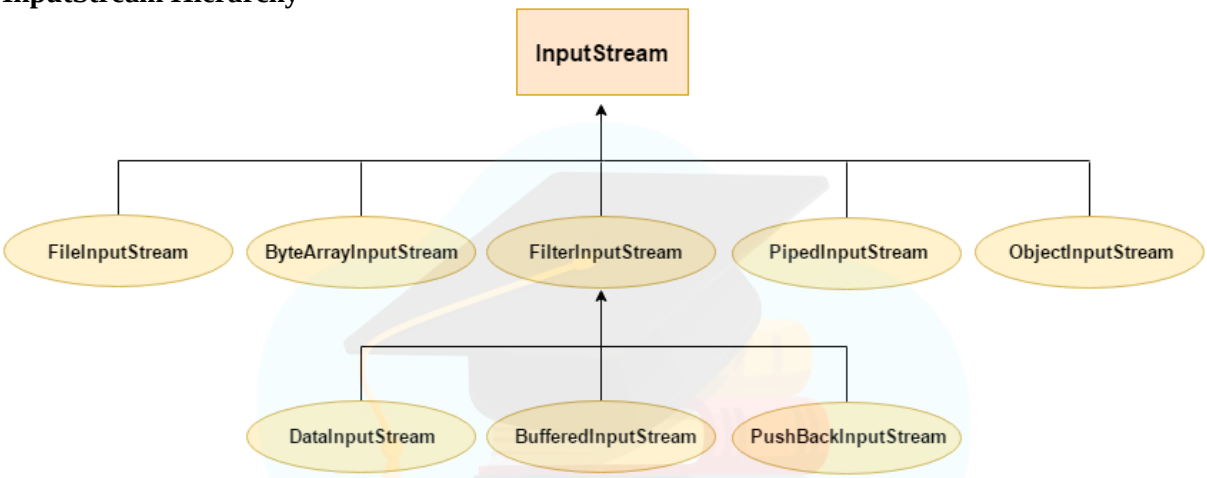
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



Byte and Character Structure:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file –

Example

```
import java.io.*;
public class CopyFile {

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

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is a test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating an output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit Unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter.** Though internally FileReader uses FileInputStream and

FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having Unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {

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

    try {
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating an output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java  $java CopyFile
```

Reading Console Input:

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Let's see a simple example to read text from console.

```
1. String text=System.console().readLine();
2. System.out.println("Text is: "+text);
```

Java Console class declaration

Let's see the declaration for Java.io.Console class:

```
1. public final class Console extends Object implements Flushable
```

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method console() that returns the singleton

instance of Console class

1. **public static** Console console(){}

Let's see the code to get the instance of Console class.

1. Console c=System.console();

Java Console Example

```
1. import java.io.Console;
2. class ReadStringTest{
3. public static void main(String args[]){
4. Console c=System.console();
5. System.out.println("Enter your name: ");
6. String n=c.readLine();
7. System.out.println("Welcome "+n);
8. }
9. }
```

Output

```
Enter your name: Nakul Jain
Welcome Nakul Jain
```

Reading and Writing on Files:

Let's start by using the `FileReader` and `FileWriter` classes:

```
String directory = System.getProperty("user.home");
String fileName = "sample.txt";
String absolutePath = directory + File.separator + fileName;

// Write the content in file
try(FileWriter fileWriter = new FileWriter(absolutePath)) {
    String fileContent = "This is a sample text.";
    fileWriter.write(fileContent);
    fileWriter.close();
} catch (IOException e) {
    // Cxception handling
}

// Read the content from file
try(FileReader fileReader = new FileReader(absolutePath)) {
    int ch = fileReader.read();
    while(ch != -1) {
        System.out.print((char)ch);
        fileReader.close();
    }
} catch (FileNotFoundException e) {
    // Exception handling
} catch (IOException e) {
    // Exception handling
}
```

Both classes accept a String, representing the path to the file in their constructors. You can also pass a `File` object as well as a `FileDescriptor`.

The `write()` method writes a valid character sequence - either a `String`, a `char[]`. Additionally, it can write a single `char` represented as an `int`.

The `read()` method reads and returns character-by-character, allowing us to use the read data in a `while` loop for example.

Don't forget to close both of these classes after use!

Reading and Writing with BufferedReader and BufferedWriter

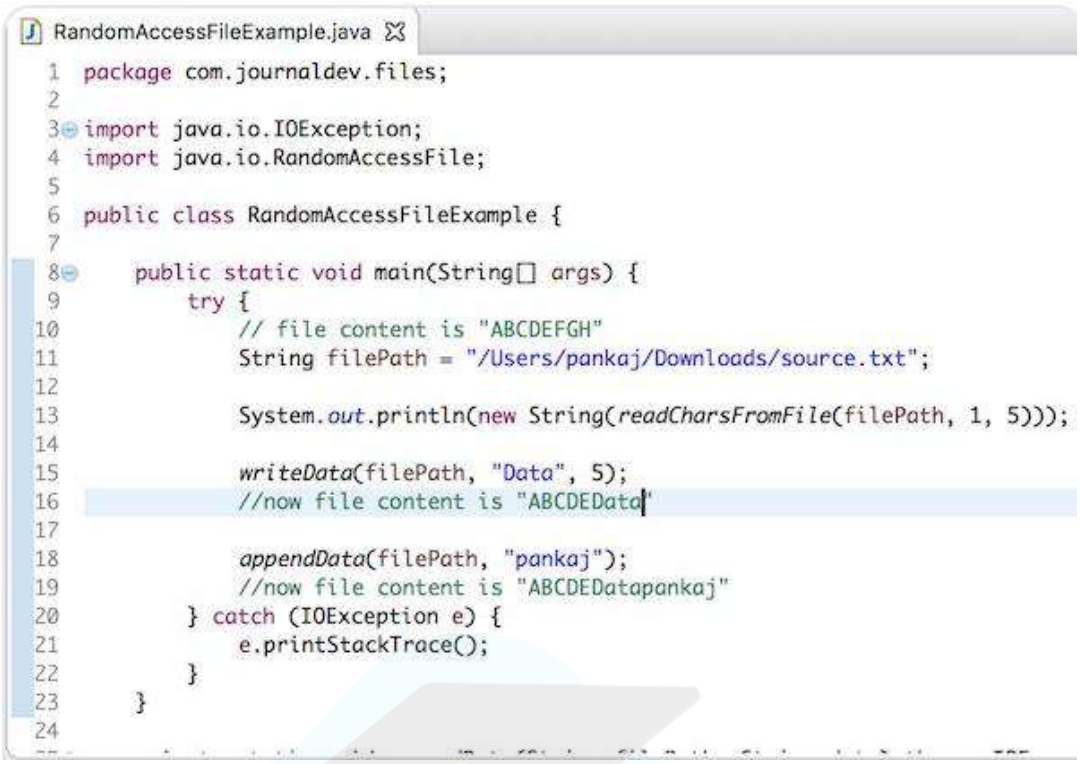
Using `BufferedReader` and `BufferedWriter` classes:

```
String directory = System.getProperty("user.home");
String fileName = "sample.txt";
String absolutePath = directory + File.separator + fileName;

// Write the content in file
try(BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(absolutePath))) {
    String fileContent = "This is a sample text.";
    bufferedWriter.write(fileContent);
} catch (IOException e) {
    // Exception handling
}

// Read the content from file
try(BufferedReader bufferedReader = new BufferedReader(new FileReader(absolutePath))) {
    String line = bufferedReader.readLine();
    while(line != null) {
        System.out.println(line);
        line = bufferedReader.readLine();
    }
} catch (FileNotFoundException e) {
    // Exception handling
} catch (IOException e) {
    // Exception handling
}
```

Random Access Files:

Java RandomAccessFile provides the facility to read and write data to a file. RandomAccessFile works with file as large array of bytes stored in the file system and a cursor using which we can move the file pointer position.

Java RandomAccessFile



RandomAccessFile class is part of Java IO. While creating the instance of RandomAccessFile in java, we need to provide the mode to open the file. For example, to open the file for read only mode we have to use "r" and for read-write operations we have to use "rw". Using file pointer, we can read or write data from random access file at any position. To get the current file pointer, you can call getFilePointer() method and to set the file pointer index, you can call seek(int i) method. If we write data at any index where data is already present, it will override it.

Stream Benefits:

There are a lot of benefits to using streams in Java, such as the ability to write functions at a more abstract level which can reduce code bugs, compact functions into fewer and more readable lines of code, and the ease they offer for parallelization.

An Example of How Java Streams Work

The simplest way to think of Java streams, and the way that helps me the most, is imagining a list of objects that are disconnected from each other, entering a pipeline one at a time. You can control how many of these objects enter the pipeline, what you do to these objects inside of the pipeline, and how you catch these objects as they exit the pipeline.

Streams are much easier to learn by looking at the big picture first and then breaking it down. To do so, let's make a program that takes a certain number of multiples of four, squares each of them, then takes the sum of all of the squares that are not divisible by ten:

```
protected static int example1(int numberOfMultiples){
    return Stream.iterate( seed: 4, multipleOfFour -> multipleOfFour + 4)
                .limit(numberOfMultiples)
                .map(multipleOfFour -> multipleOfFour * multipleOfFour)
                .filter(multipleOfFourSquared -> multipleOfFourSquared % 10 == 0)
                .reduce( identity: 0, Integer::sum);
}
```

Even if you don't know how streams work in Java, there's a very good chance you were able to quickly figure out what the code above is doing. Apart from the first line *(Stream.iterate(4, multipleOfFour -> multipleOfFour + 4))*, the rest of the code almost reads like a set of instructions.

Let's take this one piece at a time.

Lambda Expression

```
multipleOfFour -> multipleOfFour + 4
```

This is how Lambda expressions are defined in Java. Lambda expressions provide a more compact and simplified way to define a function. Essentially, you can imagine the multipleOfFour on the left is the input, the -> is the Lambda, and the multipleOfFour + 4 on the right is the return value. Imagine this reading as "Give me an input (multipleOfFour), and I will return to you that input + 4." Note that if a Lambda expression in Java takes more than one line, you need to enclose the right side (after the Lambda ->) with curly brackets "{}" and add a return statement.

Stream Creation

```
Stream.iterate( seed: 4, multipleOfFour -> multipleOfFour + 4 )
```

Stream.iterate() is a function that creates an infinite stream by taking a seed (starting point) and a UnaryOperator (for which you can pass in a Lambda expression). Starting at four, this particular iteration will return a stream of: 4, 8, 12, 16, 20, 24, 28, ...

Limit

```
.limit(numberOfMultiples)
```

Higher Order Functions

Higher order functions are functions that take functions as arguments and either act on the given function or return a function. When using Java's functional interface, higher order functions are extremely useful as they allow the developer to reach a higher level of abstraction.

Map

```
.map(multipleOfFour -> multipleOfFour * multipleOfFour)
```

Map is a higher order function. It takes a Lambda expression and converts each value in the stream it is acting on and writes it to a different stream as specified in the Lambda. In this case, it is taking every multiple of four in our original stream and creating a new stream that contains all of those multiples of four squared. Although in this scenario we are going Stream<Integer> -> Stream<Integer>, you can use map to completely convert your stream to a different type of stream!

For example, if I ran:

```
.map(multipleOfFour -> String.valueOf(multipleOfFour * multipleOfFour))
```

It would convert my *Stream<Integer> -> Stream<String>*. This is extremely powerful, and if you were to implement something like this object-oriented style, the code will not be as direct and elegant. You'll also need to unnecessarily create variables and objects with limited scope. Here's the object-oriented version:

```
List<String> multiplesOfFourStrings = new ArrayList<>();
for (Integer multipleOfFour : multiplesOfFour){
    multiplesOfFourStrings.add(String.valueOf(multipleOfFour * multipleOfFour));
}
```

Unit-3

Creating Applets in Java

Applet Basics:

An applet is a special kind of Java program that runs in a Java enabled browser. This is the first Java program that can run over the network using the browser. Applet is typically embedded inside a web page and runs in the browser.

In other words, we can say that Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as apart of a web document.

After a user receives an applet, the applet can produce a graphical user interface. It has limited access to resources so that it can run complex computations without introducing the risk of viruses or breaching data integrity.

To create an applet, a class must class extends **java.applet.Applet** class.

An Applet class does not have any main() method. It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.

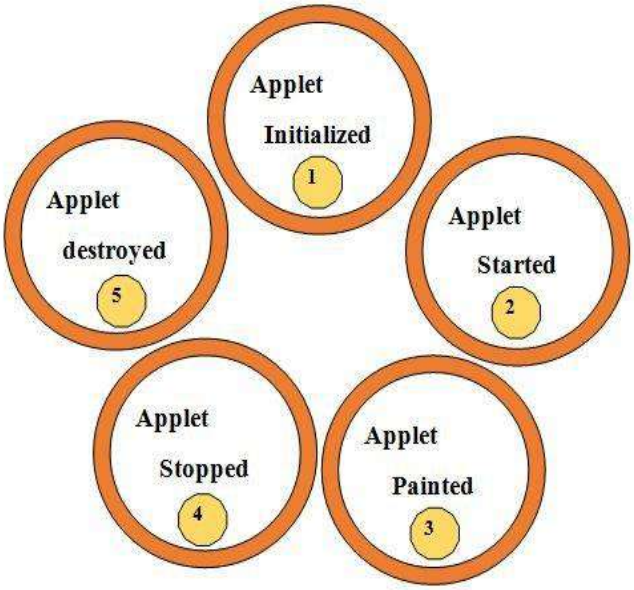
JVM creates an instance of the applet class and invokes init() method to initialize an Applet.

Note: Java Applet is deprecated since Java 9. It means Applet API is no longer considered important.

Lifecycle of Java Applet

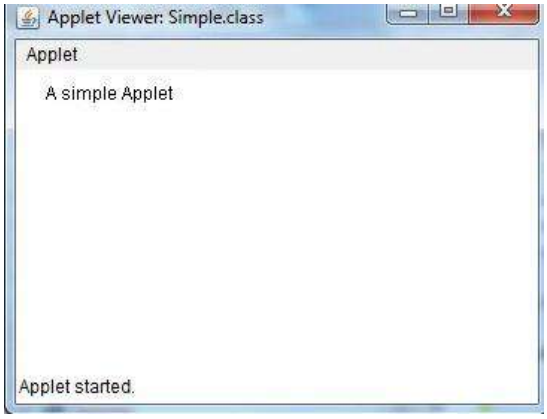
Following are the stages in Applet

1. Applet is initialized.
2. Applet is started
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.



A Simple Applet

```
import java.awt.*;
import java.applet.*;
public class Simple extends Applet
{
  public void paint(Graphics g)
    {
      g.drawString("A simple Applet", 20, 20);
    }
}
```



Every Applet application must import two packages - java.awt and java.applet.

java.awt.* imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT. The AWT contains support for a window-based, graphical user interface. java.applet.* imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet class.

The class in the program must be declared as public, because it will be accessed by code that is outside the program.Every Applet application must declare a paint() method. This method is defined by AWT class and must be overridden by the applet. The paint() method is called each time when an applet needs to redisplay its output. Another important thing to notice about applet application is that, execution of an applet does not begin at main() method. In fact an applet application does not have any main() method.

Advantages of Applets

1. It takes very less response time as it works on the client side.
2. It can be run on any browser which has JVM running in it.

Applet class

Applet class provides all necessary support for applet execution, such as initializing and destroying of applet. It also provide methods that load and display images and methods that load and play audio clips.

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.

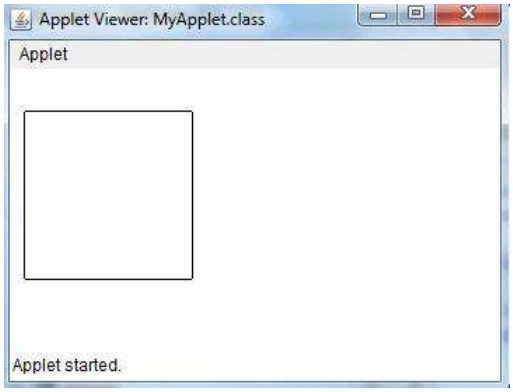
Note: The stop() method is always called before destroy() method.

Example of an Applet Skeleton:

```
import java.awt.*;
import java.applet.*;
public class AppletTest extends Applet
{
 public void init()
 {
  //initialization
 }
 public void start ()
 {
  //start or resume execution
 }
 public void stop()
 {
  //suspend execution
 {
 public void destroy()
 {
  //perform shutdown activity
 }
 public void paint (Graphics g)
 {
  //display the content of window
 }
}
```

Example of Applet

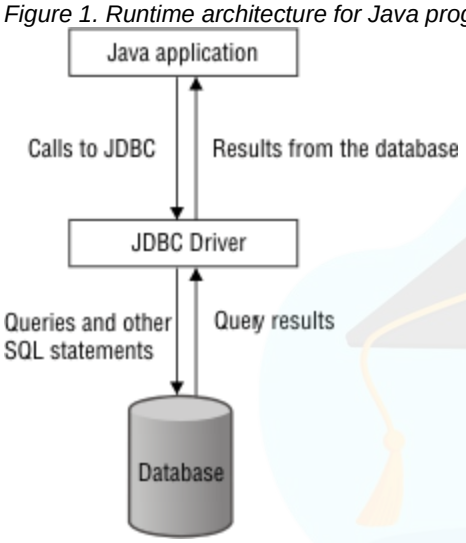
```
import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
 int height, width;
 public void init()
 {
  height = getSize().height;
  width = getSize().width;
  setName("MyApplet");
 }
 public void paint(Graphics g)
 {
  g.drawRoundRect(10, 30, 120, 120, 2, 3);
 }
}
```



Applets Architecture:

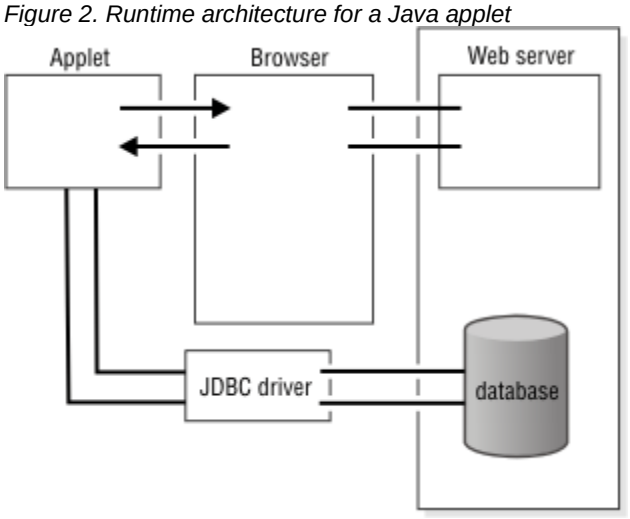
When you write a Java application for time series data, you use the JDBC Driver to connect to the database, as shown in the following figure.

Figure 1. Runtime architecture for Java programs that connect to a database



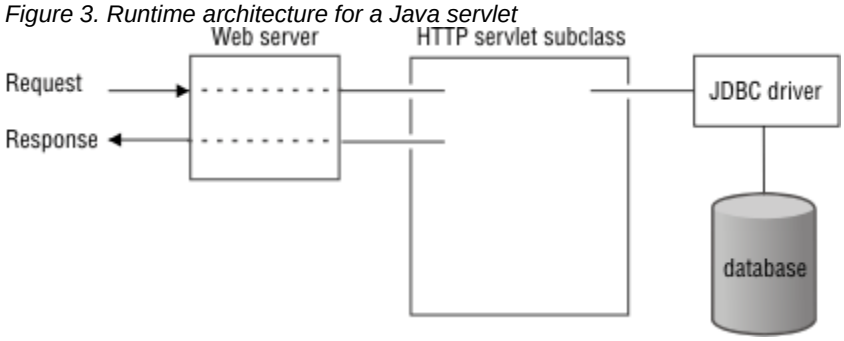
The Java application makes calls to the JDBC driver, which sends queries and other SQL statements to the database. The database sends query results to the JDBC driver, which sends them on to the Java application. You can also use the time series Java classes in Java applets and servlets, as shown in the following figures.

Figure 2. Runtime architecture for a Java applet



The database server is connected to the JDBC driver, which is connected to the applet. The applet is also connected to a browser, which is connected to a web server that communicates with the database.

Figure 3. Runtime architecture for a Java servlet



A request from an application goes through a web server, an HTTP servlet subclass, and the JDBC driver to the database. The database sends responses back along the same path.

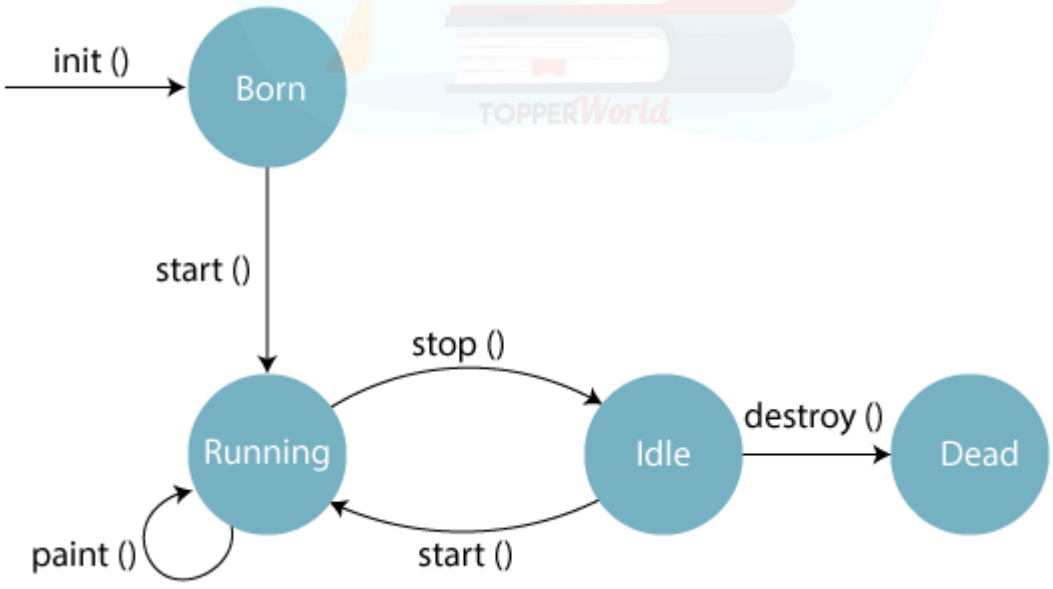
Applet Life Cycle:

In Java, an applet is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

Along with the browser, the applet also works on the client side, thus having less processing time.

Methods of Applet Life Cycle



There are five methods of an applet life cycle, and they are:

- **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.
- **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.
- **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
- **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
- **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

Sequence of method execution when an applet is executed:

1. init()
2. start()
3. paint()

Sequence of method execution when an applet is executed:

1. stop()
2. destroy()

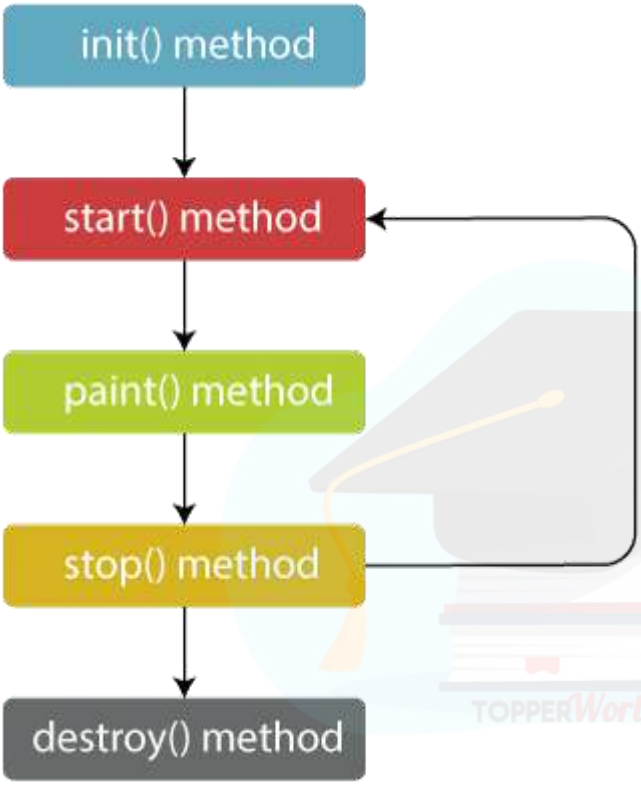
Applet Life Cycle Working

- The Java plug-in software is responsible for managing the life cycle of an applet.
- An applet is a Java application executed in any web browser and works on the client-side. It doesn't have the main() method because it runs in the browser. It is thus created to be placed on an HTML page.

- The init(), start(), stop() and destroy() methods belongs to the **applet.Applet** class.
- The paint() method belongs to the **awt.Component** class.
- In Java, if we want to make a class an Applet class, we need to extend the **Applet**
- Whenever we create an applet, we are creating the instance of the existing Applet class. And thus, we can use all the methods of that class.

Flow of Applet Life Cycle:

These methods are invoked by the browser automatically. There is no need to call them explicitly.



Syntax of entire Applet Life Cycle in Java

```
1. class TestAppletLifeCycle extends Applet {
2. public void init() {
3. // initialized objects
4. }
5. public void start() {
6. // code to start the applet
7. }
8. public void paint(Graphics graphics) {
9. // draw the shapes
```

```
10. }
11. public void stop() {
12. // code to stop the applet
13. }
14. public void destroy() {
15. // code to destroy the applet
16. }
17. }
```

Simple Applet Display Methods:

Applets are displayed in a window and they use the AWT to perform input and output functions. To output a string to an applet, use drawString( ), which is a member of the Graphics class. Typically, it is called from within either update() or paint( ). It has the following general form:

void drawString(String message, int x, int y)

Here, *message* is the string to be output beginning at *x, y*. In a Java window, the upper-left corner is location 0, 0. The drawString( ) method will not recognize newline characters. If we want to start a line of text on another line, we must do it manually, specifying the X,Y location where we want the line to begin.

To set the background color of an applet's window, we use setBackground( ). To set the foreground color, we use setForeground(). These methods are defined by Component, and have the general forms

void setBackground(Color *newColor)*

void setForeground(Color *newColor)*

Here, *newColor* specifies the new color. The class Color defines the following values that can be used to specify colors:

Color.black Color.magenta Color.blue Color.orange Color.cyan Color.pink Color.darkGray Color. red Color.gray Color.white Color. green Color.yellow Color.lightGray

For example, this sets the background color to blue and the text color to yellow:

```
setBackground(Color.blue);
setForeground(Color.yellow);
```

We can change these colors as and when required during the execution of the applet. The default foreground color is black. The default background color is light gray. We can obtain the current settings for the background and foreground colors by calling getBackground( ) and

getForeground( ), respectively. They are also defined by Component and bear the general form:

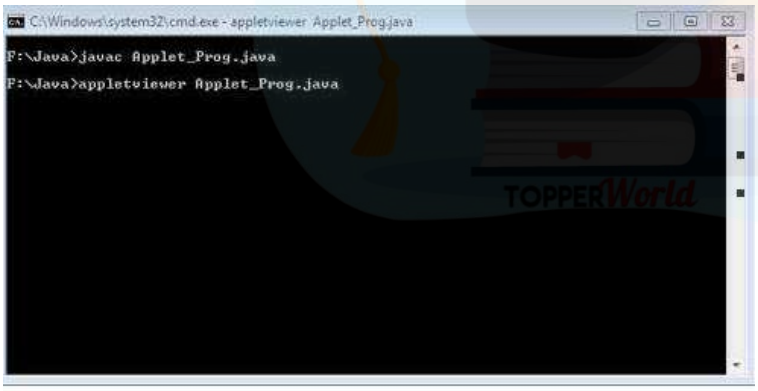
Color getBackground()

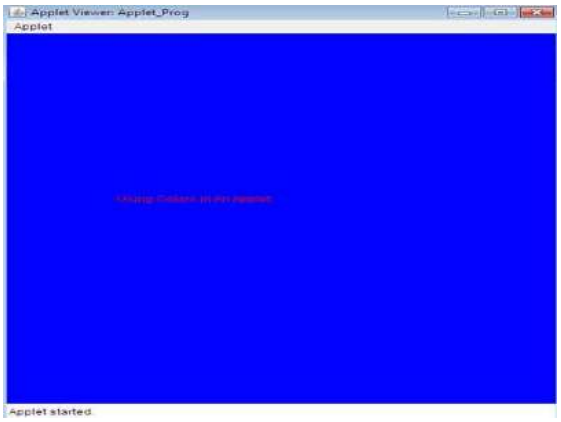
Color getForeground()

The example below shows a simple applet to change the color of the foreground and background.

```
import java.awt.*;
 import java.applet.*;
 /* <applet code="Applet_Prog" width=500 height=550> </applet>*/
 public class Applet_Prog extends Applet
 {
          public void paint (Graphics g)
        {
           setBackground(Color.BLUE);
           setForeground(Color.RED);
           g.drawString("Using Colors in An Applet" , 100,250);
         }
 }
```

The output of the Applet would look like:





Requesting Repaintin:

The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint( )**. The AWT will then execute a call to **paint( )**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint()**. Inside **paint( )**, you will output the string using **drawString( )**.

The **repaint( )** method has four forms. Let's look at each one, in turn. The simplest version of **repaint()** is shown here:

void repaint( )

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

void repaint(int *left*, int *top*, int *width*, int *height*)

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint( )** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

void repaint(long *maxDelay*)

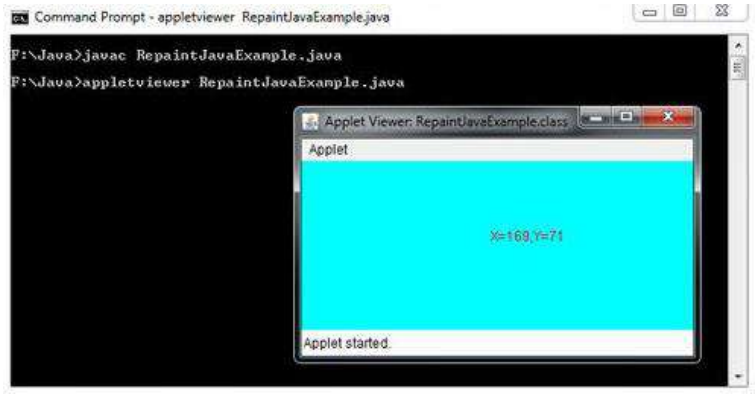
void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called. Beware, though. If the time elapses before **update()** can be called, it isn't called. There's no return value or exception thrown, so you must be careful.

Example :

```
import java.awt.*;
 import java.applet.Applet;
 import java.awt.event.*;
 /*<applet code="RepaintJavaExample.class" width="350" height="150"> </applet>*/
 public class RepaintJavaExample extends Applet implements MouseListener
 {
        private int mouseX, mouseY;
        private boolean mouseclicked = false;
        public void init()
        {
          setBackground(Color.CYAN);
          addMouseListener(this);
        }
        public void mouseClicked(MouseEvent e)
        {
          mouseX = e.getX();
          mouseY=e.getY();
          mouseclicked = true;
          repaint();
        }
        public void mouseEntered(MouseEvent e){};
        public void mousePressed(MouseEvent e){};
        public void mouseReleased(MouseEvent e){};
        public void mouseExited(MouseEvent e){};
        public void paint( Graphics g )
        {
           String str;
           g.setColor(Color.RED);
           if (mouseclicked)
           {
             str = "X="+ mouseX + "," + "Y="+ mouseY ;
             g.drawString(str,mouseX,mouseY);
             mouseclicked = false;
           }
        }
 }
```

output:



Using the Status Windows:

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates **showStatus( )**:

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet{
    public void init() {
        setBackground(Color.cyan);
    }

    // Display msg in applet window.
    public void paint(Graphics g) {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
    }
}
```

Sample output from this program is shown here:

The HTML APPLET Tag:

HTML <applet> tag was used to embed the Java applet in an HTML document. This element has been deprecated in HTML 4.0 and instead of it we can use <object> and newly added element <embed>.

The use of Java applet is also deprecated, and most browsers do not support the use of plugins.

Syntax

1. <applet code="URL" height="200" width="100">.....</applet>

Following are some specifications about <applet> tag

Display	Block
Start tag/End tag	Both Start tag and End tag
Usage	Embed Applets

Example :

```
<!DOCTYPE html>
<html>
<head>
 <title>Applet Tag</title>
</head>
<body>
  <p>Example of Applet Tag</p>
   <applet code="Shapes.class" align="right" height="200" width="300">
    <b>Sorry! you need Java to see this</b>
    </applet>
 </body>
</html>
```

Passing parameters to Applets:

- In the upcoming code, we are going to pass a few parameters like Name, Age, Sport, Food, Fruit, Destination to the applet using **param** attribute in <applet>
- Next, we will retrieve the values of these parameters using **getParameter()** method of Applet class.

```java
import java.awt.*;
import java.applet.*;


/*
<applet code="Applet8" width="400" height="200">
<param name="Name" value="Roger">
<param name="Age" value="26">
<param name="Sport" value="Tennis">
<param name="Food" value="Pasta">
<param name="Fruit" value="Apple">
<param name="Destination" value="California">
</applet>
*/


public class Applet8 extends Applet
{
String name;
String age;
String sport;
String food;
String fruit;
String destination;



public void init()
{
name = getParameter("Name");
age = getParameter("Age");
food = getParameter("Food");
fruit = getParameter("Fruit");
destination = getParameter("Destination");
sport = getParameter("Sport");
}



public void paint(Graphics g)
{
g.drawString("Reading parameters passed to this applet -", 20, 20);
g.drawString("Name -" + name, 20, 40);
g.drawString("Age -" + age, 20, 60);
g.drawString("Favorite fruit -" + fruit, 20, 80);
g.drawString("Favorite food -" + food, 20, 100);
g.drawString("Favorite destination -" + name, 20, 120);
g.drawString("Favorite sport -" + sport, 20, 140);
}

}
```

Output:

In order to run our applet using the **appletviewer,** type the following command at the command prompt-

```
appletviewer Applet8.java
```

Where Applet8.java is the name of java file that contains the code of an applet. Right after running the applet program using **appletviewer** a new applet window is displayed to us -

Java Data Base Connectivity (JDBC)

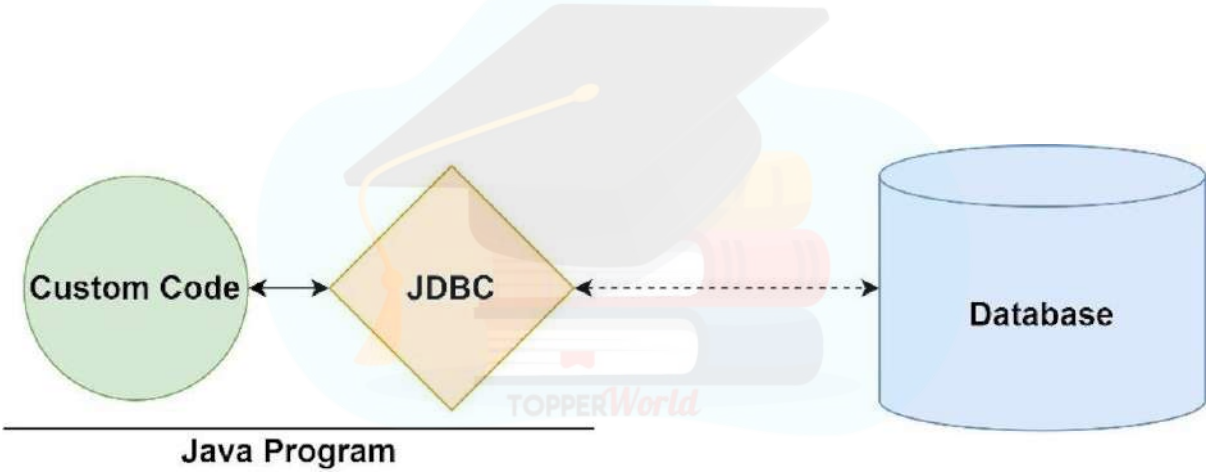
Database Connectivity- JDBC (Java Database Connectivity) is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database. Released as part of JDK 1.1 in 1997, JDBC was one of the earliest libraries developed for the Java language.

JDBC was initially conceived as a client-side API, enabling a Java client to interact with a data source. That changed with JDBC 2.0, which included an optional package supporting server-side JDBC connections. Every new JDBC release since then has featured updates to both the client-side package (java.sql) and the server-side package (javax.sql). JDBC 4.3, the most current version as of this writing, was released as part of Java SE 9 in September 2017 as JSR 221.

This article presents an overview of JDBC and JDBC drivers, followed by a hands-on introduction to using JDBC to connect a Java client to a lightweight relational database.

How JDBC works

As a developer, you can use JDBC to interact with a database from within a Java program. JDBC acts as a bridge from your code to the database, as shown in Figure 1.



IDG

Figure 1. JDBC connects Java programs to databases.

JDBC vs ODBC

Before JDBC, developers used Open Database Connectivity (ODBC), a language-agnostic standard approach to accessing a relational database management system, or RDBMS. In some ways, JDBC takes its inspiration from ODBC. The difference is that JDBC is Java-specific, offering a programming-level interface that handles the mechanics of Java applications communicating with a database.

JDBC's architecture

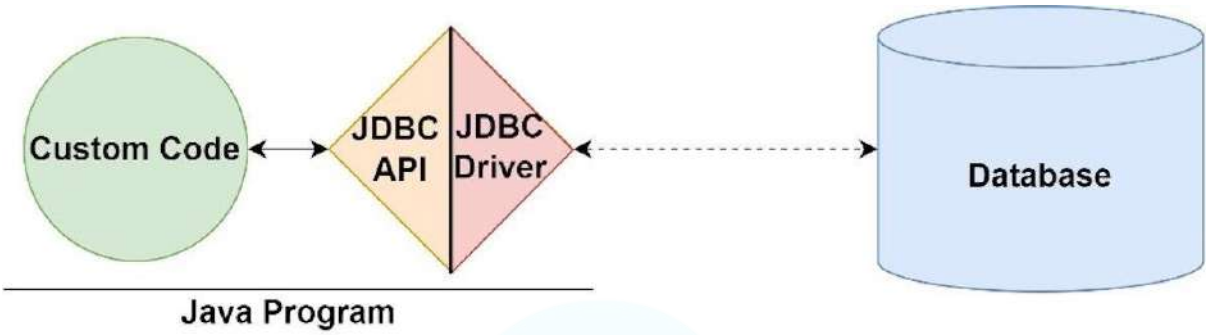
The JDBC interface consists of two layers:

1. The JDBC API supports communication between the Java application and the JDBC manager.
2. The JDBC driver supports communication between the JDBC manager and the database driver.

The JDBC API and JDBC driver have been refined extensively over the years, resulting in a feature-rich, performant, and reliable library.

JDBC is the common API that your application code interacts with. Beneath that is the JDBC-compliant driver for the database you are using.

Figure 2 illustrates the JDBC architecture.



IDG

Figure 2. JDBC's architecture consists of the JDBC API and JDBC drivers.

JDBC drivers

As an application programmer, you don't need to immediately be concerned with the implementation of the driver you use, so long as it is secure and official. However, it is useful to be aware that there are four JDBC driver types:

1. JDBC-ODBC bridge driver: A thin Java layer that uses an ODBC driver under the hood.
2. Native API driver: Provides an interface from Java to the native database client.
3. Middleware driver: A universal interface ("middleware") between Java and the RDBMS's vendor-specific protocol.
4. Pure Java driver: A driver that implements the vendor-specific protocol directly in Java.

When you start thinking about architecture and performance, it will be beneficial to consider the type of driver you are using.

Simple database connections and queries

One of the benefits of programming in the Java ecosystem is that you will likely find a stable JDBC database connector for whatever database you choose. In this tutorial, we'll use SQLite to get to know JDBC, mainly because it's so easy to use.

The steps for connecting to a database with JDBC are as follows:

1. Install or locate the database you want to access.
2. Include the JDBC library.
3. Ensure the JDBC driver you need is on your classpath.
4. Use the JDBC library to obtain a connection to the database.

5. Use the connection to issue SQL commands.
6. Close the connection when you are finished.

Relation Databases:

A relational database (RDB) is a way of structuring information in tables, rows, and columns. An RDB has the ability to establish links—or relationships–between information by joining tables, which makes it easy to understand and gain insights about the relationship between various data points.

The relational database model

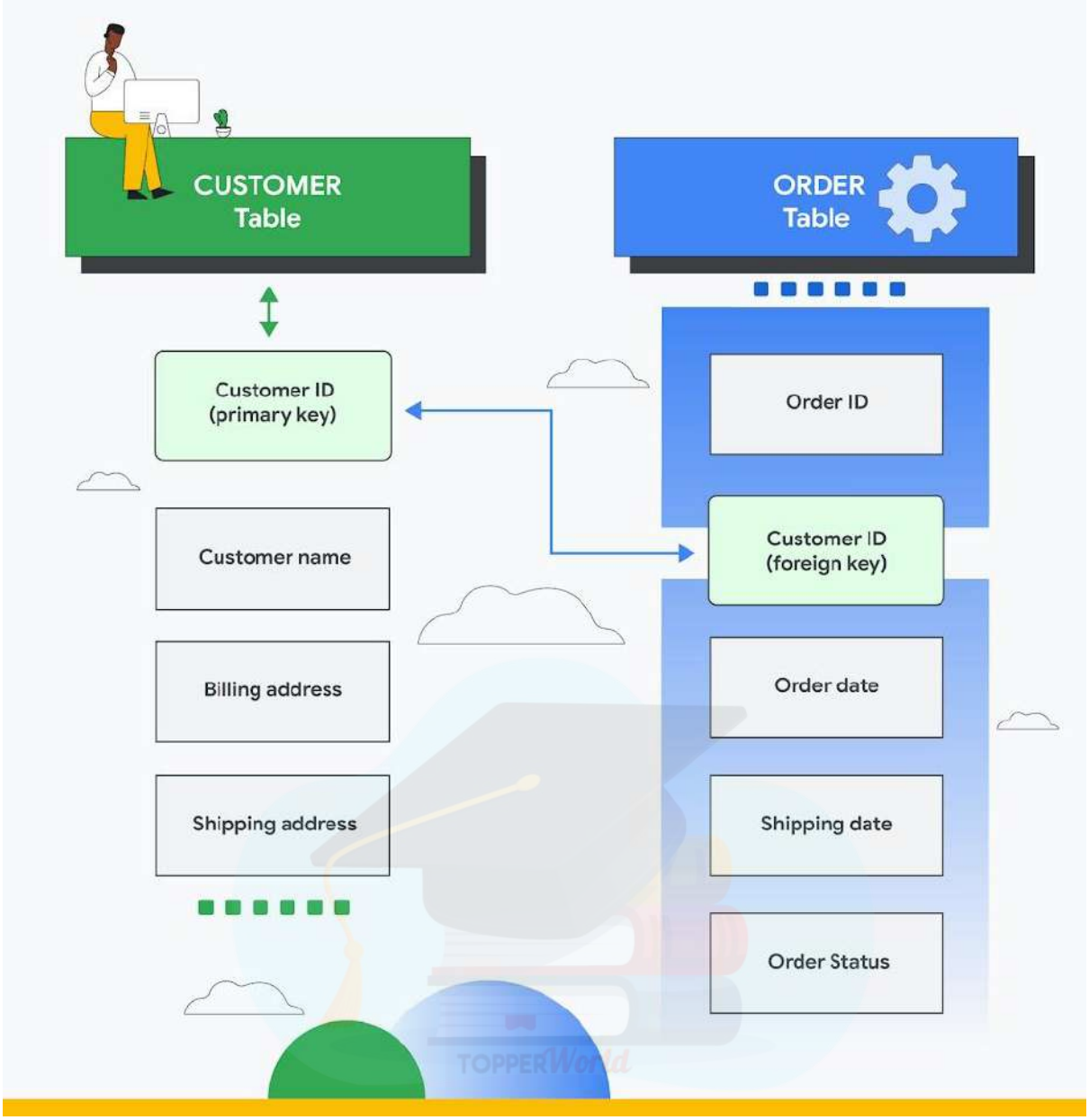
Developed by E.F. Codd from IBM in the 1970s, the relational database model allows any table to be related to another table using a common attribute. Instead of using hierarchical structures to organize data, Codd proposed a shift to using a data model where data is stored, accessed, and related in tables without reorganizing the tables that contain them.

Think of the relational database as a collection of spreadsheet files that help businesses organize, manage, and relate data. In the relational database model, each "spreadsheet" is a table that stores information, represented as columns (attributes) and rows (records or *tuples*).

Attributes (columns) specify a data type, and each record (or row) contains the value of that specific data type. All tables in a relational database have an attribute known as the **primary key**, which is a unique identifier of a row, and each row can be used to create a relationship between different tables using a **foreign key**—a reference to a primary key of another existing table.

Let's take a look at how the relational database model works in practice:

Say you have a **Customer** table and an **Order** table.



The **Customer** table contains data about the customer:

- Customer ID (primary key)
- Customer name
- Billing address
- Shipping address

In the **Customer** table, the customer ID is a primary key that uniquely identifies who the customer is in the relational database. No other customer would have the same Customer ID.

The **Order** table contains transactional information about an order:

- Order ID (primary key)
- Customer ID (foreign key)
- Order date
- Shipping date
- Order status

Here, the primary key to identify a specific order is the Order ID. You can connect a customer with an order by using a foreign key to link the customer ID from the **Customer** table.

The two tables are now related based on the shared customer ID, which means you can query both tables to create formal reports or use the data for other applications. For instance, a retail branch manager could generate a report about all customers who made a purchase on a specific date or figure out which customers had orders that had a delayed delivery date in the last month.

The above explanation is meant to be simple. But relational databases also excel at showing very complex relationships between data, allowing you to reference data in more tables as long as the data conforms to the predefined relational schema of your database.

As the data is organized as pre-defined relationships, you can query the data declaratively. A declarative query is a way to define what you want to extract from the system without expressing how the system should compute the result. This is at the heart of a relational system as opposed to other systems.

JDBC API:

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to 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.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects −

- Core JAVA Programming
- SQL or MySQL Database

JDBC Architecture

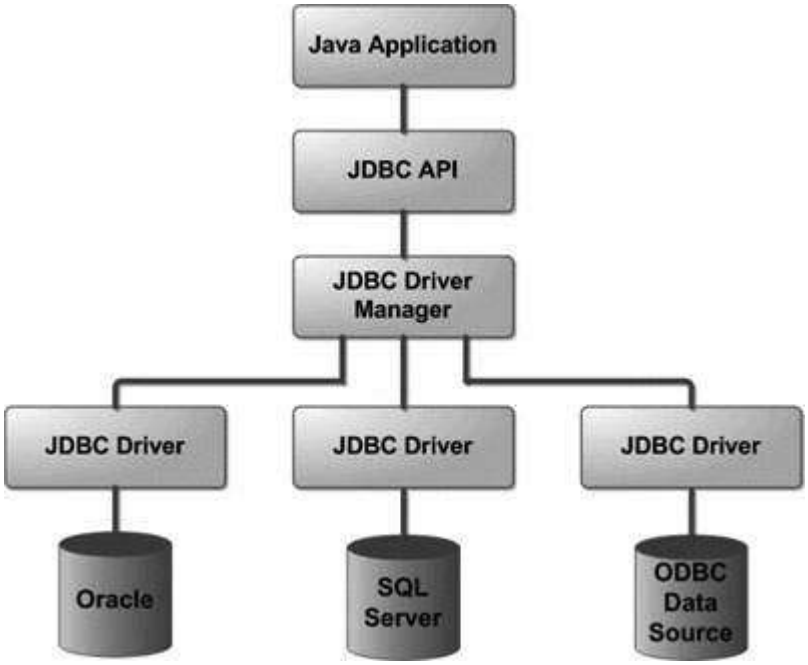
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API** − This provides the application-to-JDBC Manager connection.
- **JDBC Driver API** − This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager** − This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver** − This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection** − This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement** − You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet** − These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException** − This class handles any errors that occur in a database application.

Reusing DatabaseObjects:

In the introduction, we mentioned that the servlet life cycle allows for extremely fast database access. After you've used JDBC for a short time, it will become evident that the major performance bottleneck often comes right at the beginning, when you are opening a database connection. This is rarely a problem for most applications and applets because they can afford a few seconds to create a Connection that is used for the life of the program. With servlets this bottleneck is more serious because we are creating and tearing down a new Connection for every page request. Luckily, the servlet life cycle allows us to reuse the same connection for multiple requests, even concurrent requests, as Connection objects are required to be thread safe.

Reusing Database Connections

A servlet can create one or more Connection objects in its init() method and reuse them in its service(), doGet(), and doPost() methods. To demonstrate, shows the phone lookup servlet rewritten to create its Connection object in advance. It also uses the HtmlSQLResult class from to display the results.

Transactions:

Transactions in Java, as in general refer to a series of actions that must all complete successfully. Hence, **if one or more action fails, all other actions must back out leaving the state of the application unchanged**.

Resource Local Transactions

We'll first explore how can we use transactions in Java while working with individual resources. Here, we may have **multiple individual actions that we perform with a resource like a database**. But, we may want them to happen as a unified whole, as in an indivisible unit of work. In other words, we want these actions to happen under a single transaction.

Working with Windows

AWT Classes: The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe.

1. **Class : Description**
2. AWTEvent : Encapsulates AWT events.
3. AWTEventMulticaster : Dispatches events to multiple listeners.
4. BorderLayout : The border layout manager. Border layouts use five componentsNorth, South, East, West, and Center.
5. Button : Creates a push button control.
6. Canvas : A blank, semantics-free window.
7. CardLayout : The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
8. Checkbox : Creates a check box control.
9. CheckboxGroup : Creates a group of check box controls.
10. CheckboxMenuItem : Creates an on/off menu item.
11. Choice : Creates a pop-up list.
12. Color : Manages colors in a portable, platform-independent fashion.
13. Component : An abstract superclass for various AWT components.
14. Container : A subclass of Component that can hold other components.
15. Cursor : Encapsulates a bitmapped cursor.
16. Dialog : Creates a dialog window.
17. Dimension : Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.
18. EventQueue : Queues events.
19. FileDialog : Creates a window from which a file can be selected.
20. FlowLayout : The flow layout manager. Flow layout positions components left to right, top to bottom.
21. Font : Encapsulates a type font.
22. FontMetrics : Encapsulates various information related to a font. This information helps you display text in a window.
23. Frame : Creates a standard window that has a title bar, resize corners, and a menu bar.
24. Graphics : Encapsulates the graphics context. This context is used by the various output methods to display output in window.
25. GraphicsDevice : Describes a graphics device such as a screen or printer.
26. GraphicsEnvironment : Describes the collection of available Font and GraphicsDevice objects.
27. GridBagConstraints : Defines various constraints relating to the GridBagLayout class.
28. GridBagLayout : The grid bag layout manager. Grid bag layout displays components subject to the constraints specified l GridBagConstraints.
29. GridLayout : The grid layout manager. Grid layout displays components in a two-dimensional grid.
30. Image : Encapsulates graphical images.
31. Insets : Encapsulates the borders of a container.
32. Label : Creates a label that displays a string.
33. List : Creates a list from which the user can choose. Similar to the standard Windows list box.
34. MediaTracker : Manages media objects.

35. Menu : Creates a pull-down menu.
36. MenuBar : Creates a menu bar.
37. MenuComponent : An abstract class implemented by various menu classes.
38. MenuItem : Creates a menu item.
39. MenuShortcut : Encapsulates a keyboard shortcut for a menu item.
40. Panel : The simplest concrete subclass of Container.
41. Point : Encapsulates a Cartesian coordinate pair, stored in x and y.
42. Polygon : Encapsulates a polygon.
43. PopupMenu : Encapsulates a pop-up menu.
44. PrintJob : An abstract class that represents a print job.
45. Rectangle : Encapsulates a rectangle.
46. Robot : Supports automated testing of AWT-based applications.
47. Scrollbar : Creates a scroll bar control.
48. ScrollPane : A container that provides horizontal and/or vertical scroll bars for another component.
49. SystemColor : Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
50. TextArea : Creates a multiline edit control.
51. TextComponent : A superclass for TextArea and TextField.
52. TextField : Creates a single-line edit control.
53. Toolkit : Abstract class implemented by the AWT.
54. Window : Creates a window with no frame, no menu bar, and no title.

Window Fundamentals:

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard application window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 25-1 shows the class hierarchy for **Panel** and **Frame**. Let's look at each of these classes now.

Component

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are

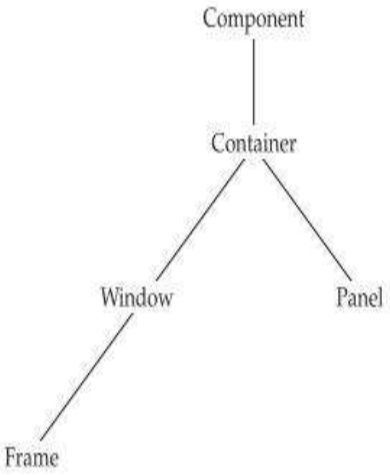


Figure 25-1 The class hierarchy for **Panel** and **Frame**

subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. (You already used many of these methods when you created applets in Chapters 23 and 24.) A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers, which you will learn about in Chapter 26.

Panel

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, **setPreferredSize( )**, or **setBounds( )** methods defined by **Component**.

Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. The precise look of a **Frame** will differ among environments. A number of environments are reflected in the screen captures shown throughout this book.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: Canvas. Derived from **Component**, **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

Working with Frame:

The java.awt.Frame component is a Windows graphics system which has a title bar and borders, behaving like a normal GUI window. How is a subclass of java.awt.Container can contain other components being that its primary purpose. The default alignment components added to a Java.awt.BorderLayout.

When working with Frame objects, the following steps are basically followed to get a window to appear on the screen.

1. Create an object of type Frame.

2. Give the Frame object a size using setSize () method.

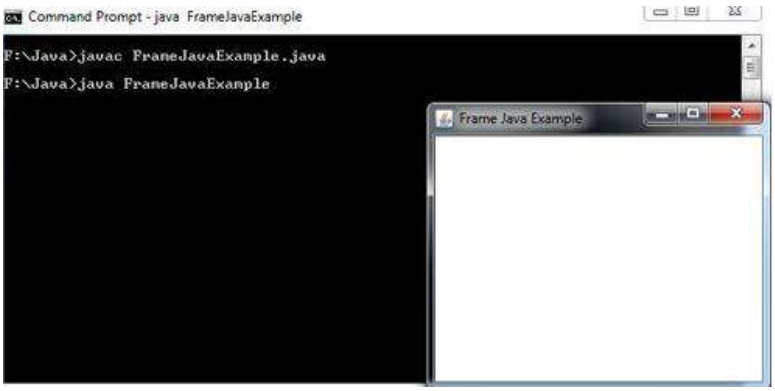
3. Make the Frame object appear on the screen by calling setVisible () method.

4. In order to close the window by clicking the close(X) button, you will have to insert the code for window closing event.

Following is a table that lists some of the methods of this class:

Method	Description
Frame ()	Constructs a new instance, invisible and without title.
Frame (String)	Constructs a new instance, invisible and entitled given
dispose ()	Releases the resources used by this component
getTitle ()	Gets the title of the window.
isResizable ()	Determines whether or not the window is sizable.
setMenuBa (MenuBar)	Adds the specified menu bar of the window.
setResizable (Boolean)	Specifies whether or not the window is sizable.
setTitle (String)	Specifies the window title.

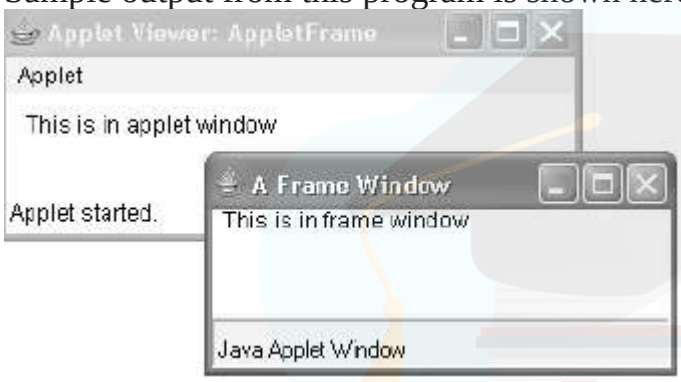
```
import java.awt.*;
 import java.awt.event.*;
 class FrameJavaExample
 {
    public static void main (String args[])
    {
          Frame frame = new Frame("Frame Java Example");
          //set the size of the frame
          frame.setSize(300,250);
          frame.addWindowListener(new WindowAdapter()
                {
                  public void windowClosing(WindowEvent e)
                  {
                    System.exit(0);
                  }
                });
                 frame.setVisible(true);
    }
 }
```



Creating a Frame Window in anApplet:

```
public class AppletFrame extends Applet {
Frame f;
public void init() {
f = new SampleFrame("A Frame Window");
f.setSize(250, 250);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
public void paint(Graphics g) {
g.drawString("This is in applet window", 10, 20);
}
}
```

Sample output from this program is shown here:



Displaying information within a Window:

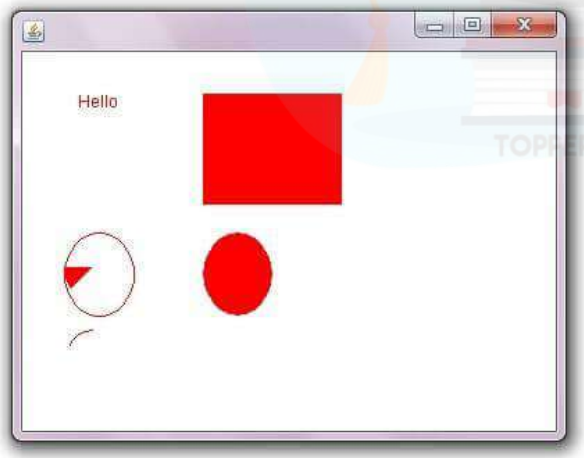
Displaying graphics in swing:

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of displaying graphics in swing:



```
1.  import java.awt.*;
2.  import javax.swing.JFrame;
3.
4.  public class DisplayGraphics extends Canvas{
5.
6.      public void paint(Graphics g) {
7.          g.drawString("Hello",40,40);
8.          setBackground(Color.WHITE);
9.          g.fillRect(130, 30,100, 80);
10.         g.drawOval(30,130,50, 60);
11.         setForeground(Color.RED);
12.         g.fillOval(130,130,50, 60);
13.         g.drawArc(30, 200, 40,50,90,60);
14.         g.fillArc(30, 130, 40,50,180,40);
15.
16.     }
17.         public static void main(String[] args) {
18.         DisplayGraphics m=new DisplayGraphics();
19.         JFrame f=new JFrame();
20.         f.add(m);
21.         f.setSize(400,400);
22.         //f.setLayout(null);
23.         f.setVisible(true);
24.     }
25.
26. }
```

Unit-4

Event Handling

Two Event Handling Mechanisms: Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces:

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}

- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```
1. import java.awt.*;
2. import java.awt.event.*;
3. class AEvent extends Frame implements ActionListener{
4. TextField tf;
5. AEvent(){
6.
7. //create components
8. tf=new TextField();
9. tf.setBounds(60,50,170,20);
10. Button b=new Button("click me");
```

```
11. b.setBounds(100,120,80,30);
12.
13. //register listener
14. b.addActionListener(this);//passing current instance
15.
16. //add components and set size, layout and visibility
17. add(b);add(tf);
18. setSize(300,300);
19. setLayout(null);
20. setVisible(true);
21. }
22. public void actionPerformed(ActionEvent e){
23. tf.setText("Welcome");
24. }
25. public static void main(String args[]){
26. new AEvent();
27. }
28. }
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

The Delegation Event Model: The Delegation Event model is defined to handle events in GUI programming languages

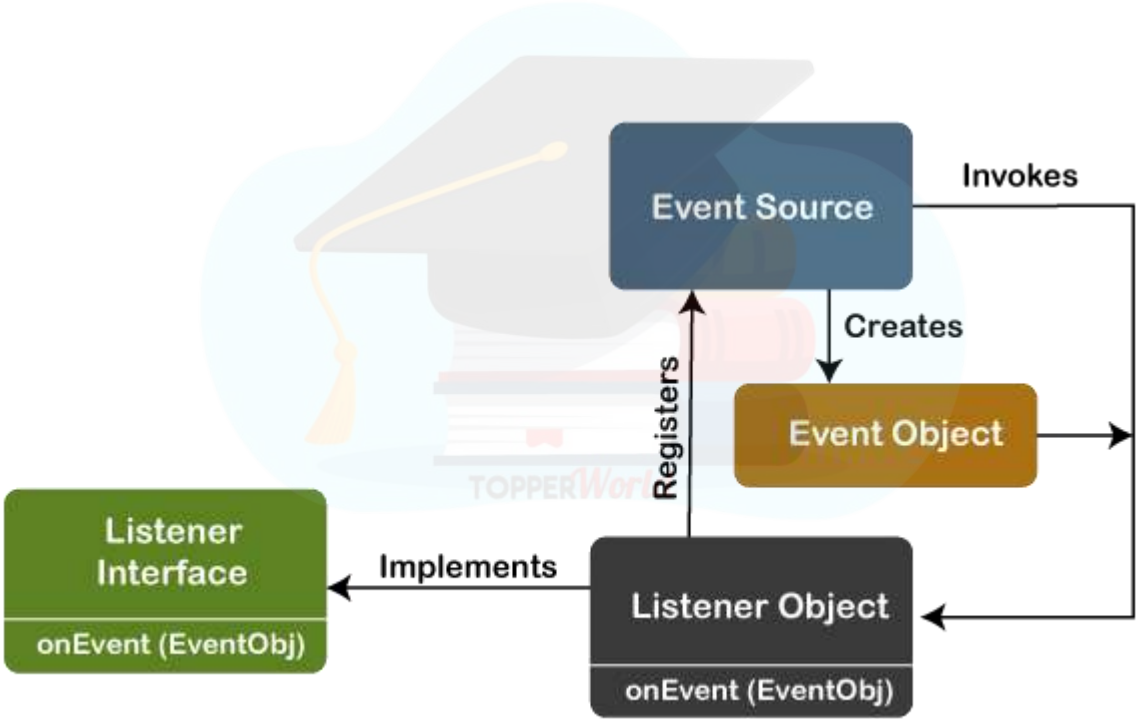
. The GUI
stands for Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

Event Processing in Java

Java support event processing since Java 1.0. It provides support for AWT (Abstract Window Toolkit)

, which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods. The below image demonstrates the event processing.



Basically, an Event Model is based on the following three components:

- Events
- Events Sources
- Events Listeners

Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

1. **public void** addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as **addKeyListener()**. For the mouse event listener, the method will be called as **addMouseMotionListener()**. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

Some listeners allow only one listener to register. Below is an example:

1. **public void** addTypeListener(TypeListener e2) **throws** java.util.TooManyListenersException

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

1. **public void** removeTypeListener(TypeListener e2?)

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the **MouseMotionListener** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

Types of Events

The events are categories into the following two categories:

The Foreground Events:

The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.

The Background Events :

The Background events are those events that result from the interaction of the end-user. For example, an Operating system interrupts system failure (Hardware or Software).

To handle these events, we need an event handling mechanism that provides control over the events and responses.

The Delegation Model

The Delegation Model is available in Java since Java 1.1. it provides a new delegation-based event model using AWT to resolve the event problems. It provides a convenient mechanism to support complex Java programs.

Design Goals

The design goals of the event delegation model are as following:

- It is easy to learn and implement
- It supports a clean separation between application and GUI code.
- It provides robust event handling program code which is less error-prone (strong compile-time checking)
- It is Flexible, can enable different types of application models for event flow and propagation.
- It enables run-time discovery of both the component-generated events as well as observable events.
- It provides support for the backward binary compatibility with the previous model.

Let's implement it with an example:

Java Program to Implement the Event Deligation Model

The below is a Java program to handle events implementing the event deligation model:

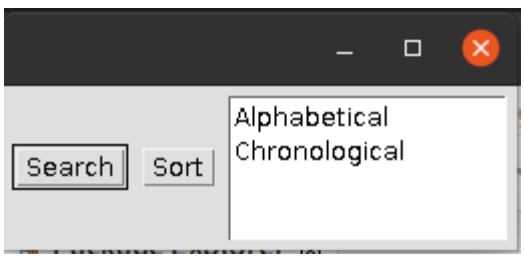
TestApp.java:

```
1. import java.awt.*;
2. import java.awt.event.*;
3.
4. public class TestApp {
```

```
5.      public void search() {
6.         // For searching
7.          System.out.println("Searching...");
8.      }
9.      public void sort() {
10.        // for sorting
11.         System.out.println("Sorting....");
12.     }
13.
14.     static public void main(String args[]) {
15.        TestApp app = new TestApp();
16.        GUI gui = new GUI(app);
17.     }
18. }
19.
20. class Command implements ActionListener  {
21.     static final int SEARCH = 0;
22.     static final int SORT = 1;
23.     int id;
24.     TestApp app;
25.
26.     public Command(int id, TestApp app) {
27.         this.id = id;
28.         this.app = app;
29.     }
30.
31.     public void actionPerformed(ActionEvent e) {
32.         switch(id) {
33.           case SEARCH:
34.             app.search();
35.             break;
36.           case SORT:
37.             app.sort();
38.             break;
39.         }
40.     }
41. }
```

```
42.
43. class GUI {
44.
45.     public GUI(TestApp app) {
46.         Frame f = new Frame();
47.         f.setLayout(new FlowLayout());
48.
49.         Command searchCmd = new Command(Command.SEARCH, app);
50.         Command sortCmd = new Command(Command.SORT, app);
51.
52.         Button b;
53.         f.add(b = new Button("Search"));
54.         b.addActionListener(searchCmd);
55.         f.add(b = new Button("Sort"));
56.         b.addActionListener(sortCmd);
57.
58.         List l;
59.         f.add(l = new List());
60.         l.add("Alphabetical");
61.         l.add("Chronological");
62.         l.addActionListener(sortCmd);
63.         f.pack();
64.
65.         f.show();
66.     }
67. }
```

Output:



Adapter Classes:

Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

Pros of using Adapter classes:

- It assists the unrelated classes to work combinedly.
- It provides ways to use classes in different ways.
- It increases the transparency of classes.
- It provides a way to include related patterns in the class.
- It provides a pluggable kit for developing an application.
- It increases the reusability of the class.

The adapter classes are found in **java.awt.event, java.awt.dnd** and **javax.swing.event** packages. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Java Servlet Programming:

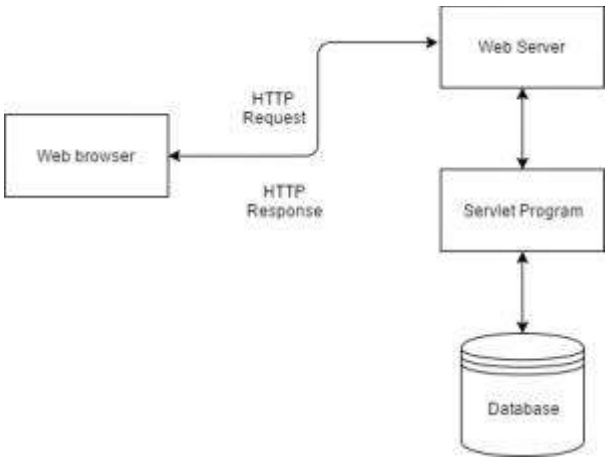
Role and Advantages of Java Servlets in Web application Development:

Servlets are the Java programs that run on the Java-enabled web server or application server. They are used to handle the request obtained from the webserver, process the request, produce the response, then send a response back to the webserver.

Properties of Servlets are as follows:

- Servlets work on the server-side.
- Servlets are capable of handling complex requests obtained from the webserver.

Servlet Architecture is can be depicted from the image itself as provided below as follows:



Execution of Servlets basically involves six basic steps:

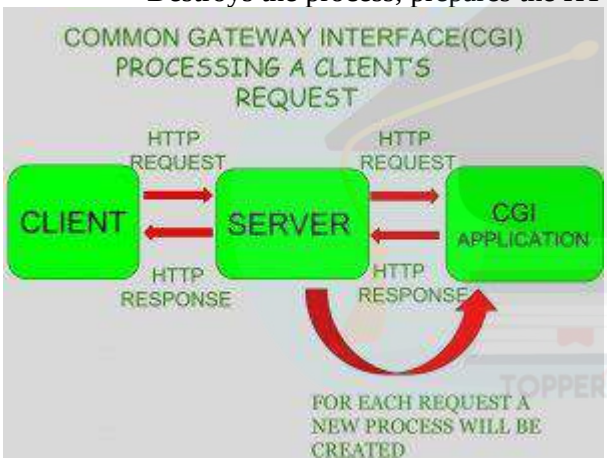
1. The clients send the request to the webserver.
2. The web server receives the request.
3. The web server passes the request to the corresponding servlet.
4. The servlet processes the request and generates the response in the form of output.
5. The servlet sends the response back to the webserver.
6. The web server sends the response back to the client and the client browser displays it on the screen.

What is CGI?

CGI is actually an external application that is written by using any of the programming languages like **C** or **C++** and this is responsible for processing client requests and generating dynamic content.

In CGI application, when a client makes a request to access dynamic Web pages, the Web server performs the following operations :

- It first locates the requested web page *i.e* the required CGI application using URL.
- It then creates a new process to service the client's request.
- Invokes the CGI application within the process and passes the request information to the application.
- Collects the response from the CGI application.
- Destroys the process, prepares the HTTP response, and sends it to the client.



So, in **CGI** server has to create and destroy the process for every request. It's easy to understand that this approach is applicable for handling few clients but as the number of clients increases, the workload on the server increases and so the time is taken to process requests increases.

Difference between Servlet and CGI

Servlet	CGI(Common Gateway Interface)
Servlets are portable and efficient.	CGI is not portable
In Servlets, sharing data is possible.	In CGI, sharing data is not possible.
Servlets can directly communicate with the webserver.	CGI cannot directly communicate with the webserver.
Servlets are less expensive than CGI.	CGI is more expensive than Servlets.
Servlets can handle the cookies.	CGI cannot handle the cookies.

Servlets API's:

Servlets are build from two packages:

- javax.servlet(Basic)
- javax.servlet.http(Advance)

Various classes and interfaces present in these packages are:

Component	Type	Package
Servlet	Interface	javax.servlet.*
ServletRequest	Interface	javax.servlet.*
ServletResponse	Interface	javax.servlet.*
GenericServlet	Class	javax.servlet.*
HttpServlet	Class	javax.servlet.http.*
HttpServletRequest	Interface	javax.servlet.http.*
HttpServletResponse	Interface	javax.servlet.http.*
Filter	Interface	javax.servlet.*
ServletConfig	Interface	javax.servlet.*

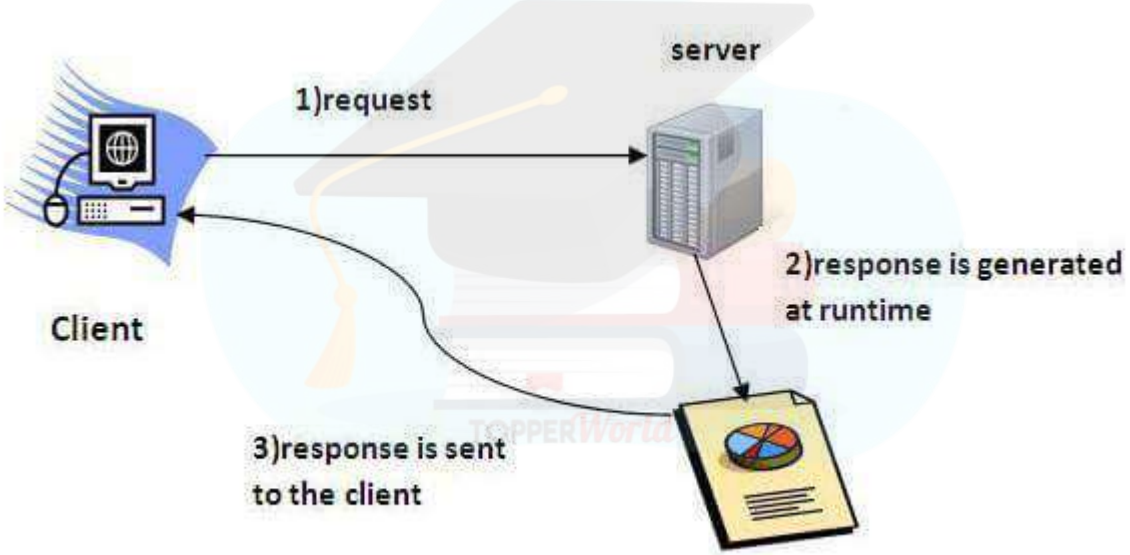
HTTP Servlets: **Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).

Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language.

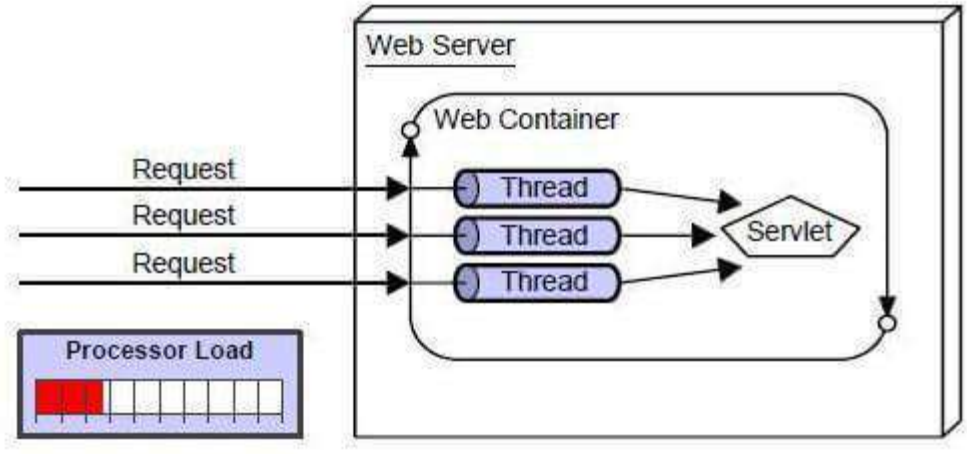
What is a Servlet?

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.



Advantages of Servlet



There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

page generation:

The most basic type of HTTP servlet generates a full HTML page. Such a servlet has access to the same information usually sent to a CGI script, plus a bit more. A servlet that generates an HTML page can be used for all the tasks where CGI is used currently, such as for processing HTML forms, producing reports from a database, taking orders, checking identities, and so forth.

Writing Hello World

shows an HTTP servlet that generates a complete HTML page. To keep things as simple as possible, this servlet just says " Hello World" every time it is accessed via a web browser.

```
import java.io.*;

import javax.servlet.*;
```

```
import javax.servlet.http.*;



public class HelloWorld extends HttpServlet {



 public void doGet(HttpServletRequest req, HttpServletResponse res)

                        throws ServletException, IOException {



   res.setContentType("text/html");

   PrintWriter out = res.getWriter();



   out.println("<HTML>");

   out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");

   out.println("<BODY>");

   out.println("<BIG>Hello World</BIG>");

   out.println("</BODY></HTML>");

 }

}
```

This servlet extends the HttpServlet class and overrides the doGet() method inherited from it. Each time the web server receives a GET request for this servlet, the server invokes this doGet() method, passing it an HttpServletRequest object and an HttpServletResponse object.

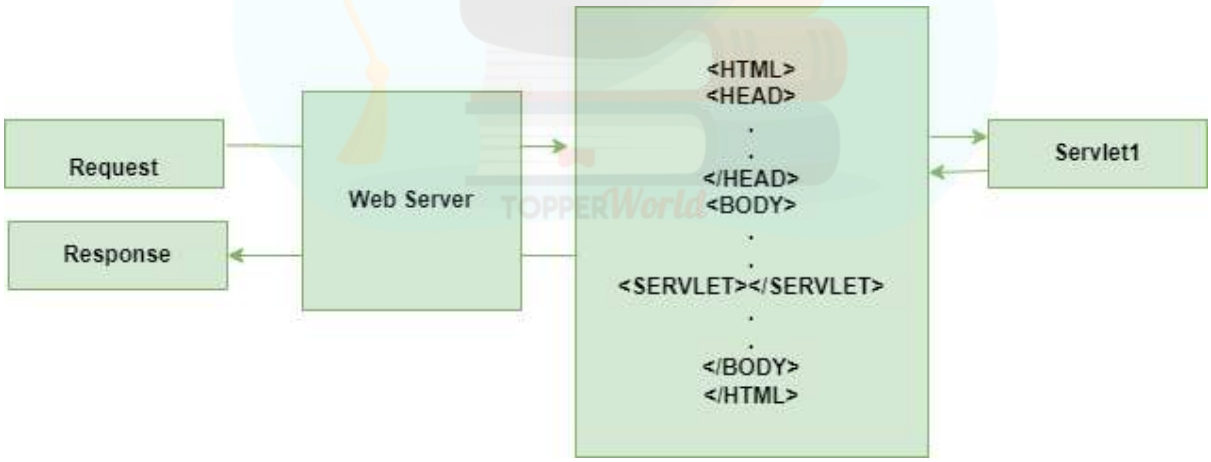
The HttpServletRequest represents the client's request. This object gives a servlet access to information ...

Server side includes:

Server-side includes are instructions and directives included in a web page that the web server may analyze when the page is provided. SSI refers to the servlet code that is embedded into the HTML code. Not all web servers can handle SSI. so you may read documents supported by a web server before utilizing SSI in your code.

Syntax:
<SERVLET CODE=MyGfgClassname CODEBASE=path initparam1=initparamvalue initparam2=initparam2value>
<PARAM NAME=name1 VALUE=value1>
<PARAM NAME=name2 VALUE=value2>
</SERVLET>
Here the path indicates the MyGfgClassname class name path in the server. you can set up the remote file path also. the remote file path syntax is,

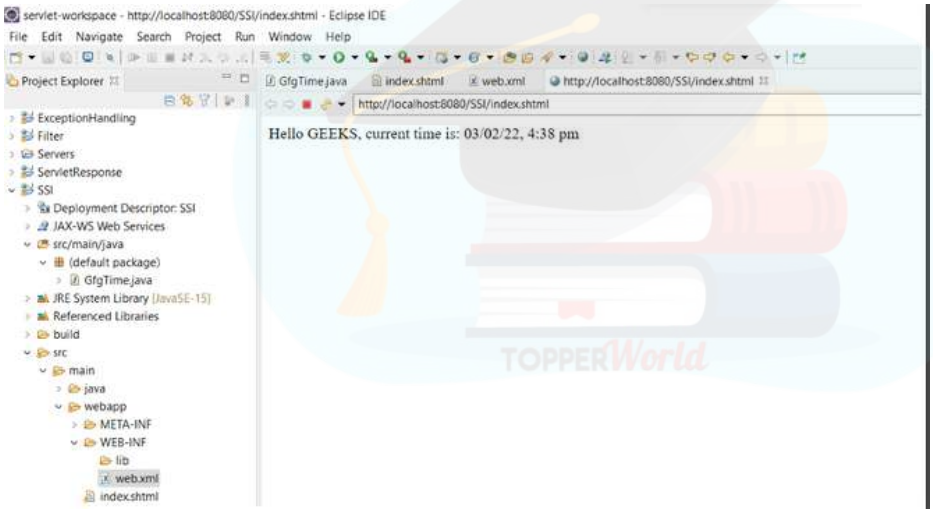
http://server:port/dir
When a server that does not support SSI sees the SERVLET> tag while returning the page, it replaces it with the servlet's output. The server does not parse all of the pages it returns; only those with a.shtml suffix are parsed. The class name or registered name of the servlet to invoke is specified by the code attributes. It's not required to use the CODEBASE property. The servlet is presumed to be local without the CODEBASE attribute. The PARAM> element may be used to send any number of parameters to the servlet. The getParameter() function of ServletRequest may be used by the servlet to obtain the parameter values.




```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GfgTime extends HttpServlet
{
```

```
private static final long serialVersionUID = 1L;
        public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException
        {
                PrintWriter out = res.getWriter();
                Date date = new Date();
                DateFormat df = DateFormat.getInstance();

                // Here write the response shtml file
                out.println("Hello GEEKS, current time is:");
                out.println(df.format(date));
        }
}
```

Output:



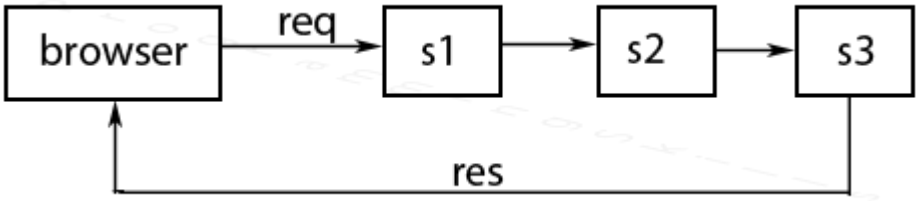
Servlet Chaining:

If a client request is processed by group of servlets, then that servlets are known as servlet chaining or if the group of servlets process a single client request then those servlets are known as servlet chaining.

In order to process a client request by many number of servlets then we have two models, they are forward model and include model.

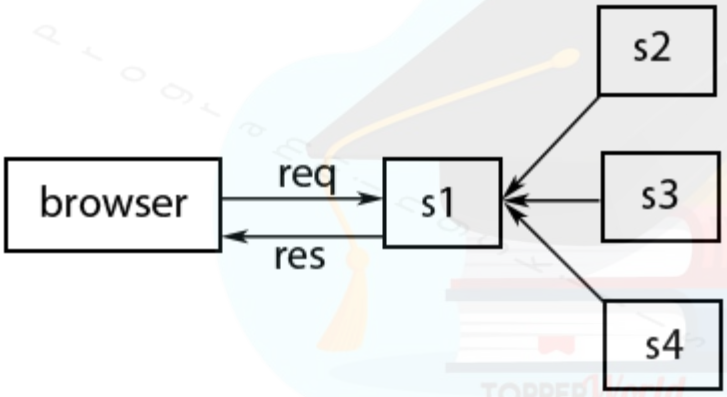
Forward model:

In this model when we forward a request to a group of servlets, finally we get the result of destination servlet as a response but not the result of intermediate servlets.



Include model:

If a single client request is passed to a servlet and that servlet makes use of other group of servlets to process a request by including the group of servlets into a single servlet.



In the above diagram client request goes to servlet s1 and s1 internally includes s2, s3 and s4 servlets and finally result of all these servlets given to the client by a source servlet s1.

Java Server pages:

- It stands for **Java Server Pages**.
- It is a server side technology.
- It is used for creating web application.
- It is used to create dynamic web content.
- In this JSP tags are used to insert JAVA code into HTML pages.
- It is an advanced version of Servlet Technology.
- It is a Web based technology helps us to create dynamic and platform independent web pages.
- In this, Java code can be inserted in HTML/ XML pages or both.
- JSP is first converted into servlet by JSP container before processing the client's request.

JSP pages are more advantageous than Servlet:

- They are easy to maintain.

- No recompilation or redeployment is required.
- JSP has access to entire API of JAVA .
- JSP are extended version of Servlet.

Features of JSP

- **Coding in JSP is easy** :- As it is just adding JAVA code to HTML/XML.
- **Reduction in the length of Code** :- In JSP we use action tags, custom tags etc.
- **Connection to Database is easier** :-It is easier to connect website to database and allows to read or write data easily to the database.
- **Make Interactive websites** :- In this we can create dynamic web pages which helps user to interact in real time environment.
- **Portable, Powerful, flexible and easy to maintain** :- as these are browser and server independent.
- **No Redeployment and No Re-Compilation** :- It is dynamic, secure and platform independent so no need to re-compilation.
- **Extension to Servlet** :- as it has all features of servlets, implicit objects and custom tags

JSP syntax

Syntax available in JSP are following

1. **Declaration Tag** :-It is used to declare variables.

Syntax:-

<%! Dec var %>

Example:-

<%! int var=10; %>

2. **Java Scriplets** :- It allows us to add any number of JAVA code, variables and expressions.

Syntax:-

<% java code %>

3. **JSP Expression** :- It evaluates and convert the expression to a string.

Syntax:-

<%= expression %>

Example:-

<% num1 = num1+num2 %>

4. **JAVA Comments** :- It contains the text that is added for information which has to be ignored.

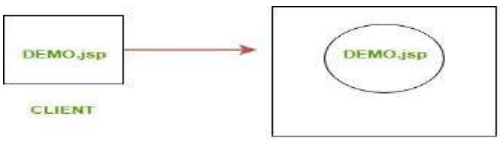
Syntax:-

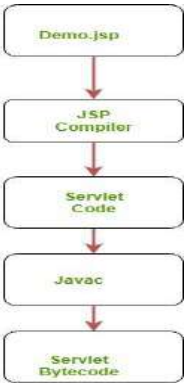
<% -- JSP Comments %>

Process of Execution

Steps for Execution of JSP are following:-

- Create html page from where request will be sent to server eg try.html.
- To handle to request of user next is to create .jsp file Eg. new.jsp
- Create project folder structure.
- Create XML file eg my.xml.
- Create WAR file.
- Start Tomcat
- Run Application





Example of Hello World

We will make one .html file and .jsp file

demo.jsp

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Hello World - JSP tutorial</title>
</head>
<body>
   <%= "Hello World!" %>
</body>
</html>
```

Advantages of using JSP

- It does not require advanced knowledge of JAVA
- It is capable of handling exceptions
- Easy to use and learn
- It can tags which are easy to use and understand
- Implicit objects are there which reduces the length of code
- It is suitable for both JAVA and non JAVA programmer
-

Disadvantages of using JSP

- Difficult to debug for errors.
- First time access leads to wastage of time
- It’s output is HTML which lacks features.

Server Life Cycle:

Servlet Alternative: Servlets are the Java programs that run on the Java-enabled web server or application server. They are used to handle the request obtained from the webserver, process the request, produce the response, then send the response back to the webserver. Below are some alternatives to servlets:

1. Common Gateway Interface (CGI)

It is the most typical server-side solution. CGI application is an independent program that receives requests from the online server and sends it back to the webserver. The use of CGI scripts was to process forms. it's the technology that permits web browsers to submit forms and connect with programs over an internet server. The common gateway interface provides an even way for data to be passed from the user's request to the appliance program and back to the user.

Example: When you fill-up the shape and submit the shape applying, click the submit button and it goes, what are the results from this level is CGI.
A common problem in this when a new process is created every time the web server receives a CGI request so it results in a delay of response time.

2. Proprietary API

Many proprietary web servers have built-in support for server-side programming. These also are called non-free software, or closed-source software, is computer software that the software's publisher or another person retains property right usual copyright of the ASCII text file, but sometimes patent rights. It's a software library interface "specific to at least one device or, more likely to a variety of devices within a specific manufacturer's product range". The motivation for employing a proprietary API is often vendor lock-in or because standard APIs don't support the device's functionality.

Examples: Netscape's NSAPI, Microsoft's ISAPI, and O'Reilly's WSAPI.
Drawback: Most of these are developed in C/C++ and hence can contain memory leaks and core dumps that can crash the webserver.

3. Active Server Pages (ASP)

Microsoft's ASP is another technology that supports server-side programming. Only Microsoft's Internet Information Server (IIS) supports this technology which isn't free. They work with simple HTML pages, the client (a web surfer) requests an internet page from a server. The server just sends the file to the client, and therefore the page is shown on the client's browser. ASP is now obsolete and replaced with ASP.NET. ASP.NET may be a compiled language and relies on the **.NET Framework**, while ASP is strictly an interpreted language.

4. Serverside JavaScript

It is another alternative to servlets. The sole known servers that support it are Netscape's Enterprise and FastTrack servers. This ties you to a specific vendor. Server-side JavaScript can be a JavaScript code that runs over a server local resources and it is similar to Java or C#, but the syntax is predicated on JavaScript.

Example: An ideal of this is often Node.JS. The advantage of server-side scripting is that the ability to highly customize the response supported the user's requirements, access rights, or queries into data stores.

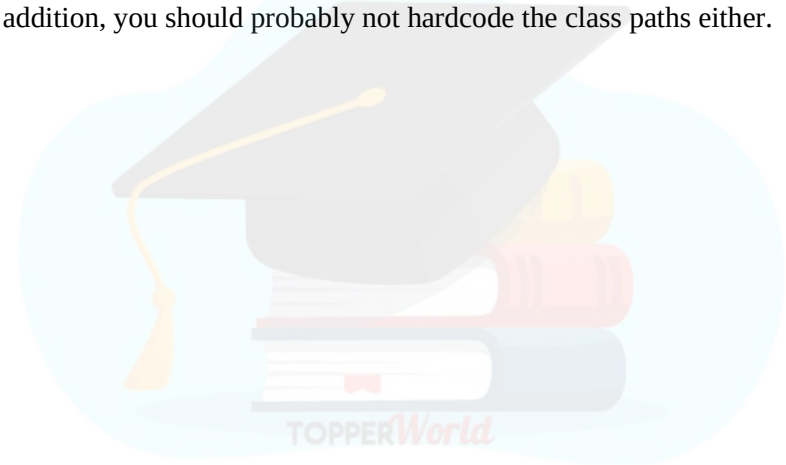
Reloading: Reloading the class is therefore not possible using Java's builtin class loaders. To reload a class you will have to implement your own ClassLoader subclass.

Even with a custom subclass of ClassLoader you have a challenge. Every loaded class needs to be linked.

ClassLoader Load / Reload Example

The text above has contained a lot of talk. Let's look at a simple example. Below is an example of a simple ClassLoader subclass. Notice how it delegates class loading to its parent except for the one class it is intended to be able to reload. If the loading of this class is delegated to the parent class loader, it cannot be reloaded later. Remember, a class can only be loaded once by the same ClassLoader instance.

As said earlier, this is just an example that serves to show you the basics of a ClassLoader's behaviour. It is not a production ready template for your own class loaders. Your own class loaders should probably not be limited to a single class, but a collection of classes that you know you will need to reload. In addition, you should probably not hardcode the class paths either.



```
public class MyClassLoader extends ClassLoader{

    public MyClassLoader(ClassLoader parent) {
        super(parent);
    }

    public Class loadClass(String name) throws ClassNotFoundException {
        if(!"reflection.MyObject".equals(name))
                return super.loadClass(name);

        try {
            String url = "file:C:/data/projects/tutorials/web/WEB-INF/" +
                         "classes/reflection/MyObject.class";
            URL myUrl = new URL(url);
            URLConnection connection = myUrl.openConnection();
            InputStream input = connection.getInputStream();
            ByteArrayOutputStream buffer = new ByteArrayOutputStream();
            int data = input.read();

            while(data != -1){
                buffer.write(data);
                data = input.read();
            }

            input.close();

            byte[] classData = buffer.toByteArray();

            return defineClass("reflection.MyObject",
                    classData, 0, classData.length);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }

}
```


Init and Destroy:

During the Spring Application Development, sometimes when the spring beans are created developers are required to execute the initialization operations and the cleanup operations before the bean is destroyed. In the spring framework, we can use the init-method and the destroy-method labels in the bean configuration.

init() Method

In the real-world application **init()** method is the one you will find it everywhere. init-method is the attribute of the spring <bean> tag. It is utilized to declare a custom method for the bean that will act as the bean initialization method. We can define it as follows.

<bean id="student" class="com.amiya.Student" init-method="myPostConstruct">

Here **myPostConstruct()** method is the bean initialization method in the Student class. This initialization method is called after initializing bean properties. We can use such an initialization method to validate the value of bean properties or initialize any task. The InitializingBean interface in spring performs the same task but it is highly coupled to spring, so we should prefer the init-method.

Why init()?

- You can add custom code/logic during bean initialization
- It can be used for setting up resources like database/socket/file etc.
-

destroy() Method

The destroy() method will be called before the bean is removed from the container. destroy-method is bean attribute using which we can assign a custom bean method that will be called just before the bean is destroyed. To use the destroy-method attribute, we do as follows.

<bean id="student" class="com.amiya.Student" destroy-method="myPreDestroy">

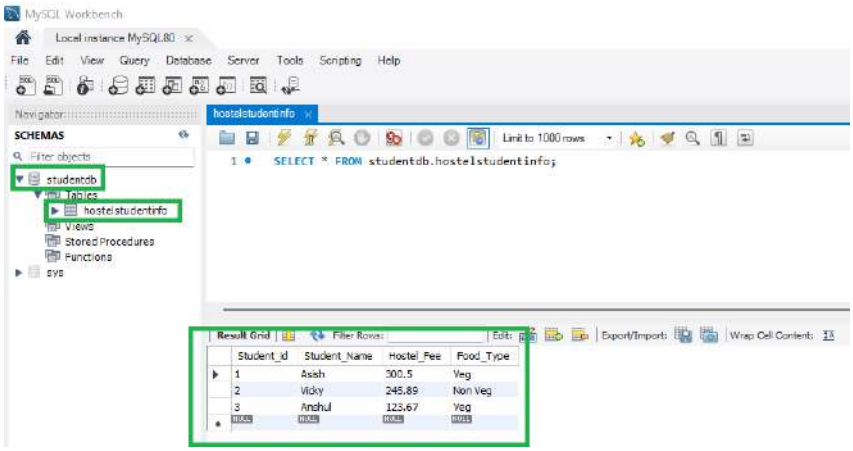
Here **myPreDestroy()** method will be defined in the Student class. Spring will call this method just before destroying the bean. destroy-method is used to release resources or perform some destruction task. DisposableBean interface in spring performs the same task but it is highly coupled to spring, so we should prefer destroy-method. So let's understand these two methods with a simple example.

Implementation:

We are going to explain init() and destroy() Methods through @PostConstruct and @PreDestroy Annotation by simply creating a Spring JDBC project. So let's create a Spring JDBC project first.

Step 1: Create a simple Java project in your preferred IDE (IntelliJ IDEA or Eclipse).

Step 2: Create some tables inside your database. In this article, we have used the MySQL database. And the following data has been present inside our MySQL Database



So here **studentdb** is our schema name and **hostelstudentinfo** is the table name. Similarly, you can create your own schema and table and put some data inside that table manually.

Single Thread Model:

Single Thread Model Interface was designed to guarantee that only one thread is executed at a time in a given servlet instance service method. It should be implemented to ensure that the servlet can handle only one request at a time. It is a marker interface and has no methods. Once the interface is implemented the system guarantees that there's never more than one request thread accessing a single instance servlet. This interface is currently **deprecated** because this doesn't solve all the thread safety issues such as static variable and session attributes can be accessed by multiple threads at the same time even if we implemented the **SingleThreadModel** interface. That's why to resolve the thread-safety issues it is recommended to use a synchronized block**.**

Syntax:

```
public class Myservlet  extends Httpservlet implements SingleThreadModel {

}
```

Implementation: SingleThreadModel interface

We created three files to make this application:

1. index.html
2. Myservlet.java
3. web.xml

The index.html file creates a link to invoke the servlet of URL-pattern "servlet1" and Myservlet class extends HttpServlet and implements the SingleThreadModel interface. Class Myservlet is a servlet that handles Single requests at a single time and sleep() is a static method in the Thread class used to suspend the execution of a thread for two thousand milliseconds. When another user will try to access the same servlet, the new instance is created instead of using the same instance for multiple threads.

```java
// Java Program to Illustrate MyServlet Class

// Importing required classes
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Class
public class MyServlet
    extends HttpServlet implements SingleThreadModel {

    // Method
    // Use doGet() when you want to intercept on
    // HTTP GET requests
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        // Sets the content type of the response being sent
        // to the client
        response.setContentType("text/html");

        // Returns a PrintWriter object that can send
        // character text to the client
        PrintWriter out = response.getWriter();

        out.print("welcome");

        // Try block to check for exceptions
        try {
            // Making thread to sleep for 2 seconds
            Thread.sleep(2000);
        }

        // Catch block to handle exceptions
        catch (Exception e) {
            // Display exception/s with line number
            e.printStackTrace();
        }

        out.print(" to servlet");

        // Closing the output stream
        // using close() method
        out.close();
    }
}
```

Background Processing LastModified time:

Java provides built-in methods such as lastModified() or getLastModifiedTime() to get the file last updation time. We can use either File class of **java.io** package or Files class of **java.nio** package. It's up to you which package you are using.

The lastModified() methods belong to File class and returns a time. Whereas the getLastModifiedTime() method belongs to Files class and returns time in a long type which further can be converted to datetime.

Time for an Example:

Let's create an example to get the file's last modified time. Here, we are using lastModified() method of the File class that returns a time in a long type so we need to convert it to time explicitly, as we did in the below example.

```
import java.io.File;
import java.io.IOException;
import java.time.Instant;
public class Main {
    public static void main(String[] args) throws IOException{
        try {
        File file = new File("abc.txt");
        long time  = file.lastModified();  // file modified time
        System.out.println(time);
        // Convert long time to date time
        System.out.println(Instant.ofEpochMilli(time));
        }catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

OUTPUT:

```
1592912186419

2020-06-23T11:36:26.419Z
```

Persistent state capabilities:

Definition of Persistence

Persistence denotes a process or an object that continues to exist even after its parent process or object ceases, or the system that runs it is turned off. When a created process needs persistence, non-volatile storage, including a hard disk, is used instead of volatile memory like RAM.

For example, imagine that you're using the Chrome browser on the Windows operating system and, due to some technical issues, the browser process was shut down or killed. The browser restarts the next time you open it and attempts to reopen any tabs that were open when it crashed. A persistent process thus exists even if it failed or was killed for some technical reasons. The persistent state is the state that remains even after a process shuts down. Persistent process states are stored in persistent storage (non-volatile storage like hard drives) before the system fails or shuts down. They're easily retrieved once the system is turned on. The core processes of the operating system must be persistent for maintaining the data, device, or workspace in a similar state when the system is switched on.

Types of Persistence

There are two types of persistence: object persistence and process persistence. In data terms, **object persistence** refers to an object that is not deleted until a need emerges to remove it from the memory. Some database models provide mechanisms for storing persistent data in the form of objects.
In **process persistence**, processes are not killed or shut down by other processes and exist until the user kills them. For example, all of the core processes of a computer system are persistent for enabling the proper functioning of the system. Persistent processes are stored in non-volatile memory. They do not need special databases like persistent objects.

Object Persistence in Databases

A persistent database stores persistent data in the form of **objects**, or records that are durable when changing devices and software. Persistent data is stable and recoverable. Traditional relational database management systems (RDBMS) store persistent data in the form of records and tables. However, they cannot store objects and their relationships. Objects have necessary features (like encapsulation, inheritance, persistence, and polymorphism) that do not translate well into records and tables. Thus, certain special databases like object-oriented database management systems (OODBMS) and object relational database management systems (ORDBMS) are needed for storing objects and preserving object persistence.

Persistent Data: Importance

Persistent data is important for several reasons. For example:

1. Persistent data is static and does not change with time (not dynamic).
2. Persistent data stores core information. For example, an organization's financial data must be persistent.
3. Persistent data cannot be deleted by external processes or objects until the user deletes it, meaning it's stable.
4. Persistent data is time independent data. The data created by an application or object continues to exist even after the parent application or object has been deleted and remains accessible beyond object boundaries, other processes, or transactions.
5. Persistent data remains in its original format. If data resides in volatile memory, it's deleted once the process is closed. However, persistent data stored in persistent databases, or non-volatile storage, continues to reside there even after closing the program.

6. Persistent data is non-volatile in nature and can withstand power outages.
7. Persistent data is recoverable data even after a system restarts or shuts down.
8. Persistent data changes cannot be lost and will always be available for access.

Persistent Databases: Examples

Data manipulated in a database can be persistent or transient. The transient data inside a program or transaction will be lost once its parent program or transaction terminates. Persistent data exists beyond the lifetime of its parent program or transaction and is capable of surviving transaction updates. Persistent databases are typically updated and accessed across several transactions.