

UNIT- I

INTRODUCTION PART OF JAVA PROGRAMMING

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. Java is a versatile and widely-used programming language known for its portability, readability, and robustness. Java was created by James Gosling and his team at Sun Microsystems (which later became part of Oracle Corporation) in the early 1990s. Initially, Java was developed as a programming language for consumer electronics, specifically for interactive television. It was originally named "Oak" after an oak tree that stood outside Gosling's office. The project later shifted its focus to develop a programming language for the emerging "Internet of Things" and interactive television market. In 1995, Sun Microsystems officially released the language under the name "Java," accompanied by the tagline "Write Once, Run Anywhere." Oracle Corporation acquired Sun Microsystems in 2010, becoming the new steward of the Java platform.

The release of Java marked a significant turning point as it was positioned as a platform-independent language, capable of running on any system with a compatible Java Virtual Machine (JVM).

Key Features of Java:

Platform Independence: One of Java's most significant advantages is its ability to run on multiple platforms without modification. This is achieved through the use of the Java Virtual Machine (JVM), which allows Java programs to be executed on any platform that has a compatible JVM installed.

Object-Oriented: Java is a fully object-oriented programming language, emphasizing the creation and manipulation of objects. This makes it easier to write modular and maintainable code.

Robust and Secure: Java's strict compile-time and runtime checks help catch errors early in the development process, enhancing code reliability. Additionally, Java's security features, such as the sandboxing of applets, make it a preferred choice for building secure applications.

Large Standard Library: Java provides a comprehensive standard library (known as the Java Standard Library or Java API) that simplifies common tasks such as file I/O, networking, and data manipulation.

Multi-threading Support: Java has built-in support for multi-threading, making it suitable for developing concurrent and scalable applications.

High Performance: While Java is often praised for its platform independence and ease of use, it also offers high performance through Just-In-Time (JIT) compilation.

Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages. Oracle offers its own HotSpot Java Virtual Machine, however the official reference implementation is the OpenJDK JVM which is free open-source software and used by most developers and is the default JVM for almost all Linux distributions.

As of March 2023, Java 20 is the latest version, while Java 17, 11 and 8 are the current long-term support (LTS) versions.

Major release versions of Java, along with their release dates:

Version	Date
JDK <u>Beta</u>	1995
JDK 1.0	January 23, 1996
JDK 1.1	February 19, 1997
J2SE 1.2	December 8, 1998
J2SE 1.3	May 8, 2000
J2SE 1.4	February 6, 2002
J2SE 5.0	September 30, 2004
Java SE 6	December 11, 2006
Java SE 7	July 28, 2011
Java SE 8 (LTS)	March 18, 2014
Java SE 9	September 21, 2017

Java SE 10	March 20, 2018
Java SE 11 (LTS)	September 25, 2018
Java SE 12	March 19, 2019
Java SE 13	September 17, 2019
Java SE 14	March 17, 2020
Java SE 15	September 15, 2020
Java SE 16	March 16, 2021
Java SE 17 (LTS)	September 14, 2021
Java SE 18	March 22, 2022
Java SE 19	September 20, 2022
Java SE 20	March 21, 2023

IDE FOR JAVA PROGRAMMING

Let's discuss some of the essential features and components of an Integrated Development Environment (IDE) and how they work in the context of Java development:

1. Code Editor:

- **Working:** The code editor is the central component of an IDE where you write, edit, and format your Java code. It provides features like syntax highlighting, code autocompletion, and code formatting to improve code readability and productivity.

2. Code Debugging:

- **Working:** Debugging tools allow developers to identify and fix errors in their code. IDEs provide features like breakpoints, variable inspection, and step-by-step execution to help developers locate and resolve issues efficiently.

3. Code Navigation:

- **Working:** Code navigation features enable developers to quickly move between different parts of their codebase. IDEs provide tools like "Go to Definition" to jump to variable or method definitions, making it easier to understand and modify code.

4. Project Management:

- **Working:** IDEs offer project management capabilities to organize and manage Java projects. Developers can create, open, and configure projects, manage dependencies, and organize source code files within a project structure.

5. Version Control Integration:

- **Working:** IDEs often integrate with version control systems like Git, allowing developers to commit, push, pull, and merge code changes directly from the IDE. This simplifies collaboration and code management.

6. Build and Compilation:

- **Working:** IDEs provide tools for building and compiling Java applications. They automate the process of generating bytecode from source code and can highlight compilation errors and warnings in real-time.

7. Code Refactoring:

- **Working:** Code refactoring tools help developers improve code quality and maintainability by suggesting and performing automated code transformations. Examples include renaming variables, extracting methods, and optimizing imports.

8. Code Templates and Snippets:

- **Working:** IDEs offer code templates and snippets that allow developers to quickly insert common code patterns. For Java, this could include creating classes, methods, loops, or exception handlers with just a few keystrokes.

9. Code Analysis and Suggestions:

- **Working:** IDEs provide static code analysis tools that identify potential issues, code smells, and coding style violations. They offer suggestions and warnings to help developers write cleaner and more efficient code.

10. Integrated Terminal:

- **Working:** Some IDEs include an integrated terminal or command-line interface, allowing developers to execute shell commands, run scripts, or perform various tasks without leaving the IDE.

11. Plugins and Extensions:

- **Working:** Many IDEs support plugins or extensions that enhance functionality. Developers can install and configure plugins to add support for additional languages, frameworks, and tools.

12. Code Collaboration:

- **Working:** Some IDEs offer features for real-time code collaboration, enabling multiple developers to work on the same codebase simultaneously. They may include features like pair programming and code sharing.

These features collectively create a productive and efficient development environment for Java programmers. IDEs streamline the development process, reduce the likelihood of errors, and provide a range of tools to improve code quality and maintainability. The choice of IDE often depends on individual preferences and project requirements, but a well-chosen IDE can significantly boost a developer's productivity and code quality.

There are several Integrated Development Environments (IDEs) available for Java development, each offering its own set of features and capabilities. Here are some popular Java IDEs:

1. **Eclipse:** Eclipse is a widely-used open-source IDE for Java development. It offers a rich set of features, including code assistance, debugging, version control integration, and a wide range of plugins for various programming languages and technologies.
2. **IntelliJ IDEA:** IntelliJ IDEA, developed by JetBrains, is a highly regarded commercial IDE for Java. It provides a user-friendly interface, powerful code analysis tools, intelligent code completion, and excellent support for Java frameworks.
3. **NetBeans:** NetBeans is an open-source IDE that supports multiple programming languages, including Java. It offers features such as code templates, visual design tools, and seamless integration with project management and version control systems.
4. **Visual Studio Code:** Visual Studio Code (VS Code) is a free and open-source code editor from Microsoft. It has gained popularity in recent years for its extensibility through plugins, including those for Java development. With the right extensions, it can function as a powerful Java IDE.
5. **Android Studio:** If you're specifically interested in Android app development using Java (or Kotlin), Android Studio is the official IDE provided by Google. It includes features tailored for Android app development and offers a seamless development experience for Android projects.
6. **BlueJ:** BlueJ is an educational IDE designed for teaching and learning Java programming. It provides a simplified interface and is often used in introductory Java courses.

7. **JDeveloper:** Oracle's JDeveloper is an IDE primarily focused on Java EE (Enterprise Edition) development. It offers tools for building enterprise-level Java applications, including support for Oracle databases and middleware.
8. **DrJava:** DrJava is a lightweight and open-source IDE designed for simplicity and ease of use, making it a good choice for beginners or small projects.
9. **JGrasp:** JGrasp is another lightweight IDE suitable for educational purposes. It provides visual representations of code structures and supports multiple programming languages, including Java.

The choice of IDE depends on your specific needs, preferences, and project requirements. Many developers prefer IntelliJ IDEA for its comprehensive features and excellent Java support, while Eclipse remains a popular choice, especially for open-source and community-driven projects. Experiment with a few IDEs to find the one that best suits your workflow and coding style.

JRE

JRE (Java Runtime Environment) is a critical component in the Java ecosystem. It is an essential part of the Java platform, and it plays a crucial role in running Java applications. Here's an explanation of what JRE is and how it works:

1. What is JRE?

- The Java Runtime Environment (JRE) is a software package that contains everything needed to run compiled Java applications and applets. It provides the runtime environment for executing Java bytecode.

2. Components of JRE:

- The JRE includes several key components:
 - **Java Virtual Machine (JVM):** The JVM is a crucial component of the JRE responsible for executing Java bytecode. It interprets the bytecode and translates it into machine-specific instructions that the underlying hardware can understand. The JVM also manages memory, handles garbage collection, and enforces security.
 - **Java Class Library:** The JRE contains a set of standard class libraries and APIs (Application Programming Interfaces). These libraries provide pre-written classes and methods that Java applications can use for various tasks, such as file I/O, networking, and user interface development. The Java Class Library is part of the Java Standard Library (Java API).

3. How JRE Works:

- When you develop a Java application, you write the source code and then use a Java compiler (e.g., `javac`) to compile it into bytecode. This bytecode is a platform-independent representation of your code.
- To run the compiled Java application or applet, you need the JRE installed on the target machine. Here's how it works:
 1. You execute the Java application by invoking the `java` command followed by the name of the class that contains the main method (the entry point of the application).

2. The java command starts the JVM included in the JRE.
3. The JVM loads the bytecode of your application and executes it.
4. The Java Class Library is used by your application to perform various tasks, such as reading files, creating user interfaces, and making network requests.

4. Importance of JRE:

- JRE is critical for Java's "Write Once, Run Anywhere" philosophy. Since Java bytecode is platform-independent, the same compiled Java application can run on any platform with a compatible JRE. This feature is a significant advantage for Java's cross-platform compatibility.

5. JRE vs. JDK:

- It's essential to distinguish between JRE and JDK (Java Development Kit). The JDK includes the JRE along with additional tools and libraries for Java development, such as the Java compiler (javac) and debugging tools. Developers use the JDK to create, compile, and package Java applications, while end-users only need the JRE to run them.

In summary, the Java Runtime Environment (JRE) is a critical component that allows Java applications to run on various platforms by providing the necessary runtime environment, including the Java Virtual Machine (JVM) and the Java Class Library. End-users require JRE to execute Java applications, while developers use the Java Development Kit (JDK), which includes the JRE, to create and compile Java applications.

JVM

The JVM, or **Java Virtual Machine**, is a crucial component of the Java platform. It plays a central role in executing Java bytecode, making it possible for Java programs to run on different hardware and operating systems without modification. Here's a detailed explanation of what the JVM is and how it works:

1. What is JVM (Java Virtual Machine)?

- The JVM is a software-based emulation of a physical computer that executes Java bytecode. It's a part of the Java Runtime Environment (JRE), which also includes the Java Class Library and other runtime components.

2. Key Components of JVM:

- The JVM consists of several important components:
 - **Class Loader:** The Class Loader is responsible for loading classes and interfaces as needed during program execution. It loads classes from the classpath, including system classes and user-defined classes.
 - **Class Area:** The Class Area is a part of memory that stores class metadata, such as class names, method names, and field names. It also stores the bytecode for loaded classes.
 - **Heap:** The Heap is the memory area where objects are allocated and deallocated. It's used for dynamic memory allocation, and garbage collection takes place in this area.

- **Method Area:** The Method Area, also known as the Permanent Generation (PermGen), stores class-level data, such as static variables, method code, and constant pool data. In newer versions of Java (Java 8 and later), PermGen has been replaced by the Metaspace.
- **Execution Engine:** The Execution Engine interprets and executes Java bytecode. It can use different methods, such as interpretation, Just-In-Time (JIT) compilation, or a combination of both, to execute code efficiently.
- **Native Interface:** The Native Interface allows Java applications to interact with native code written in languages like C or C++. It's essential for platform-specific operations and system calls.

3. How JVM Works:

- Here's a simplified overview of how the JVM works during the execution of a Java program:
 1. You compile your Java source code using a Java compiler (e.g., **javac**). This compilation step generates bytecode files (**.class** files).
 2. To run your Java application, you use the **java** command, specifying the name of the class that contains the **main** method (the entry point of the program).
 3. The JVM is invoked by the **java** command. It starts by loading the necessary class files and verifying their bytecode for correctness and security.
 4. The Execution Engine of the JVM executes the bytecode, interpreting it or using JIT compilation to translate it into native machine code.
 5. The program runs within the JVM, using memory areas like the heap for object storage and the method area for storing class-level information.
 6. Garbage collection in the JVM periodically identifies and frees up memory occupied by objects that are no longer reachable.

4. Platform Independence:

- One of the key advantages of the JVM is its ability to provide platform independence. Java bytecode is machine-independent, meaning you can write a Java program once and run it on any platform that has a compatible JVM.

5. Different JVM Implementations:

- There are multiple implementations of the JVM available, including Oracle HotSpot, OpenJ9, GraalVM, and others. These implementations may optimize the execution of bytecode differently and offer various features and performance improvements.

In summary, the Java Virtual Machine (JVM) is the runtime environment that executes Java bytecode. It abstracts the underlying hardware and operating system, providing platform independence for Java applications. The JVM's components work together to load, verify, and execute Java programs efficiently while managing memory and providing essential runtime services.

OBJECT ORIENTED PROGRAMMING CONCEPTS

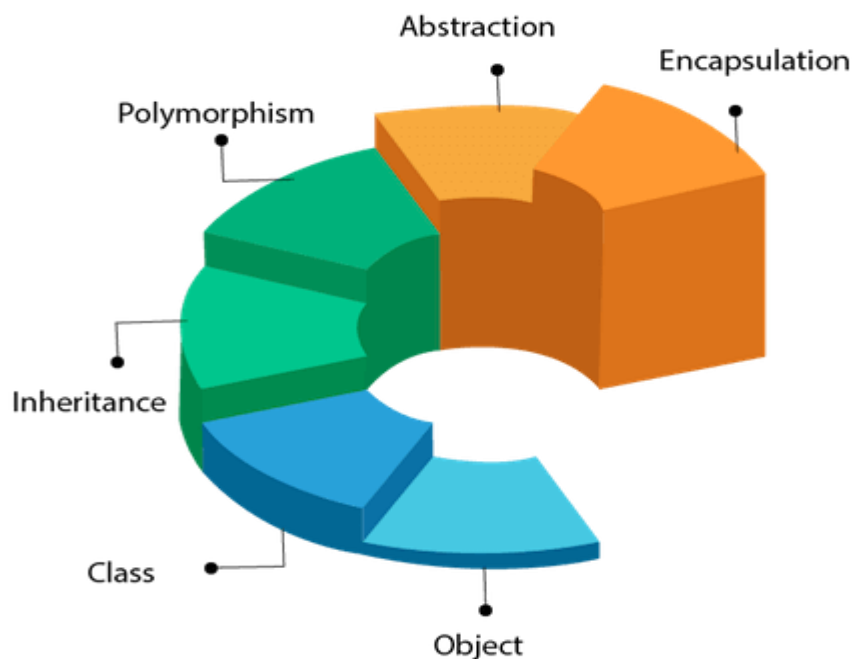
Object means a real-world entity such as a pen, chair, table, computer, watch, etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- 1) Object
- 2) Class
- 3) Inheritance
- 4) Polymorphism
- 5) Abstraction
- 6) Encapsulation

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

- a) Coupling
- b) Cohesion
- c) Association
- d) Aggregation
- e) Composition

OOPs (Object-Oriented Programming System)



1) 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.

Java is a popular object-oriented programming (OOP) language, and it embraces several core OOP concepts. These concepts form the foundation for designing and building Java applications. Here are the fundamental OOP concepts in Java:



An object has three characteristics:

- a. **State:** represents the data (value) of an object.
- b. **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- c. **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

2) 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.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Syntax to declare a class:

```
class <class_name>{  
    field;  
    method;  
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

New keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Classes and Objects:

- **Class:** A class is a blueprint or template for creating objects. It defines the structure and behavior that objects of that class will have. Classes in Java can contain fields (variables) and methods (functions).
- **Object:** An object is an instance of a class. It represents a real-world entity and can interact with other objects.

```
class Car {
    String brand;
    int year;

    void startEngine() {
        // Method implementation
    }
}

// Creating an object of the Car class
Car myCar = new Car();
```

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
    //defining fields
```

```

int id;//field or data member or instance variable

String name;

//creating main method inside the Student class

public static void main(String args[]){

    //Creating an object or instance

    Student s1=new Student();//creating an object of Student

    //Printing values of the object

    System.out.println(s1.id);//accessing member through reference variable

    System.out.println(s1.name);

}

}

```

Output:

```

0
null

```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```

//Java Program to demonstrate having the main method in

//another class

//Creating Student class.

class Student{

    int id;

    String name;

}

//Creating another class TestStudent1 which contains the main method

class TestStudent1{

    public static void main(String args[]){

```

```
Student s1=new Student();  
System.out.println(s1.id);  
System.out.println(s1.name);  
}  
}
```

Output:

```
0  
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
class Student{  
    int id;  
    String name;  
}  
class TestStudent2{  
    public static void main(String args[]){  
        Student s1=new Student();  
        s1.id=101;  
        s1.name="Sonu";  
        System.out.println(s1.id+" "+s1.name);//printing members with a white space  
    }  
}
```

Output:

101 Sonu

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
class Student{
    int id;
    String name;
}

class TestStudent3{
    public static void main(String args[]){
        //Creating objects
        Student s1=new Student();
        Student s2=new Student();
        //Initializing objects
        s1.id=101;
        s1.name="Sonu";
        s2.id=102;
        s2.name="Amrita";
        //Printing data
        System.out.println(s1.id+" "+s1.name);
        System.out.println(s2.id+" "+s2.name);
    }
}
```

Output:

```
101 Sonu
102 Amrita
```

2) Object and Class Example: Initialization through method

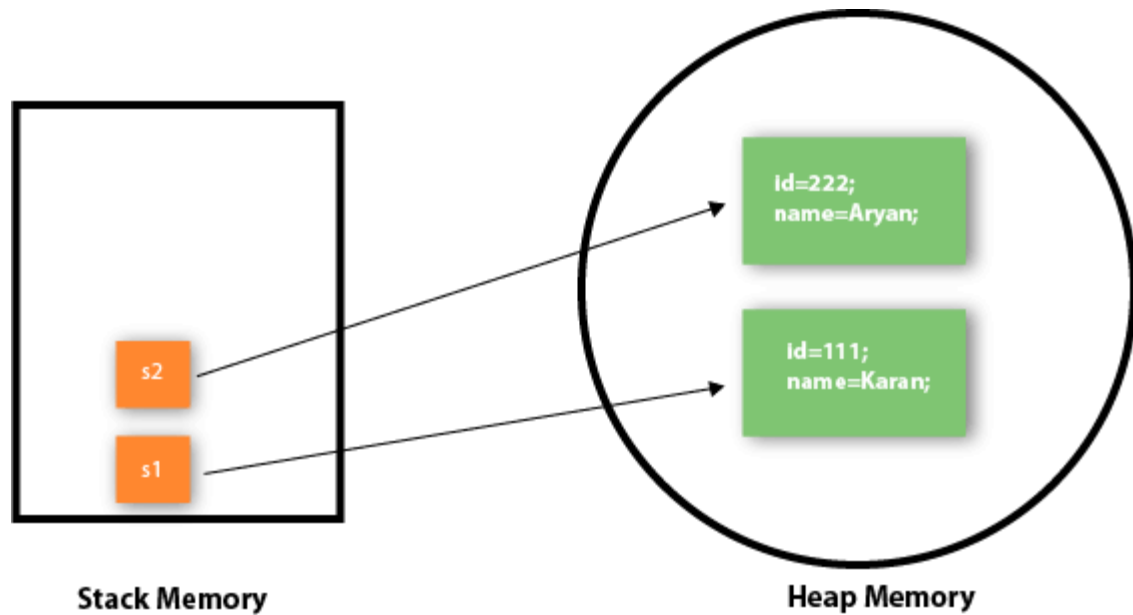
In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
class Student{  
    int rollno;  
    String name;  
    void insertRecord(int r, String n){  
        rollno=r;  
        name=n;  
    }  
    void displayInformation(){System.out.println(rollno+" "+name);}  
}  
class TestStudent4{  
    public static void main(String args[]){  
        Student s1=new Student();  
        Student s2=new Student();  
        s1.insertRecord(111,"Karan");  
        s2.insertRecord(222,"Aryan");  
        s1.displayInformation();  
        s2.displayInformation();  
    }  
}
```

Output:

```
111 Karan  
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
```



```

public class TestEmployee {
    public static void main(String[] args) {
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
        e1.insert(101,"ajeet",45000);
        e2.insert(102,"irfan",25000);
        e3.insert(103,"nakul",55000);
        e1.display();
        e2.display();
        e3.display();
    }
}

```

Output:

```

101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0

```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```

class Rectangle{
    int length;
    int width;
    void insert(int l, int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}

class TestRectangle1{

```

```
public static void main(String args[]){  
    Rectangle r1=new Rectangle();  
    Rectangle r2=new Rectangle();  
    r1.insert(11,5);  
    r2.insert(3,15);  
    r1.calculateArea();  
    r2.calculateArea();  
}  
}
```

Output:

```
55  
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation();//anonymous object
```

Calling method through a reference:

```
Calculation c=new Calculation();  
c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
class Calculation{
    void fact(int n){
        int fact=1;
        for(int i=1;i<=n;i++){
            fact=fact*i;
        }
        System.out.println("factorial is "+fact);
    }
    public static void main(String args[]){
        new Calculation().fact(5);//calling method with anonymous object
    }
}
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of reference variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
//Java Program to illustrate the use of Rectangle class which
```

```
//has length and width data members
```

```
class Rectangle{
    int length;
    int width;
    void insert(int l,int w){
        length=l;
```

```

    width=w;
}
void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
    public static void main(String args[]){
        Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
        r2.calculateArea();
    }
}

```

Output:

```

55
45

```

Real World Example: Account

File: TestAccount.java

```

//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
class Account{
    int acc_no;
    String name;
    float amount;
    //Method to initialize object
    void insert(int a,String n,float amt){
        acc_no=a;
        name=n;
    }
}

```

```

amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
}
}

```

```
a1.withdraw(15000);  
a1.checkBalance();  
}  
}
```

Output:

```
832345 Ankit 1000.0  
Balance is: 1000.0  
40000.0 deposited  
Balance is: 41000.0  
15000.0 withdrawn  
Balance is: 26000.0
```

3) Inheritance:

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

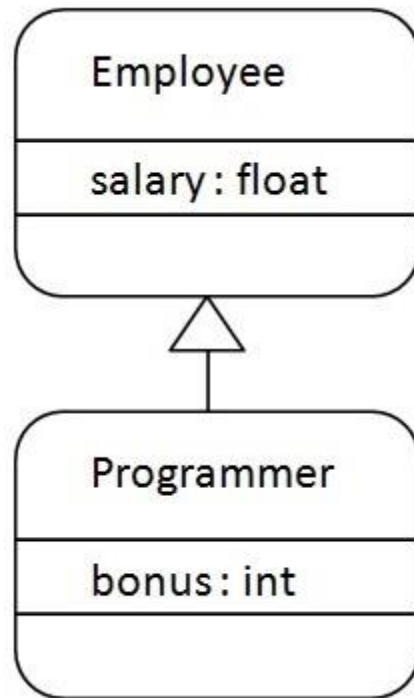
```
class Subclass-name extends Superclass-name  
{
```

```
//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.

```
class Employee{  
    float salary=40000;  
}  
  
class Programmer extends Employee{  
    int bonus=10000;  
  
    public static void main(String args[]){  
        Programmer p=new Programmer();  
        System.out.println("Programmer salary is:"+p.salary);  
    }  
}
```

```
System.out.println("Bonus of Programmer is:"+p.bonus);  
}  
}
```

Output:

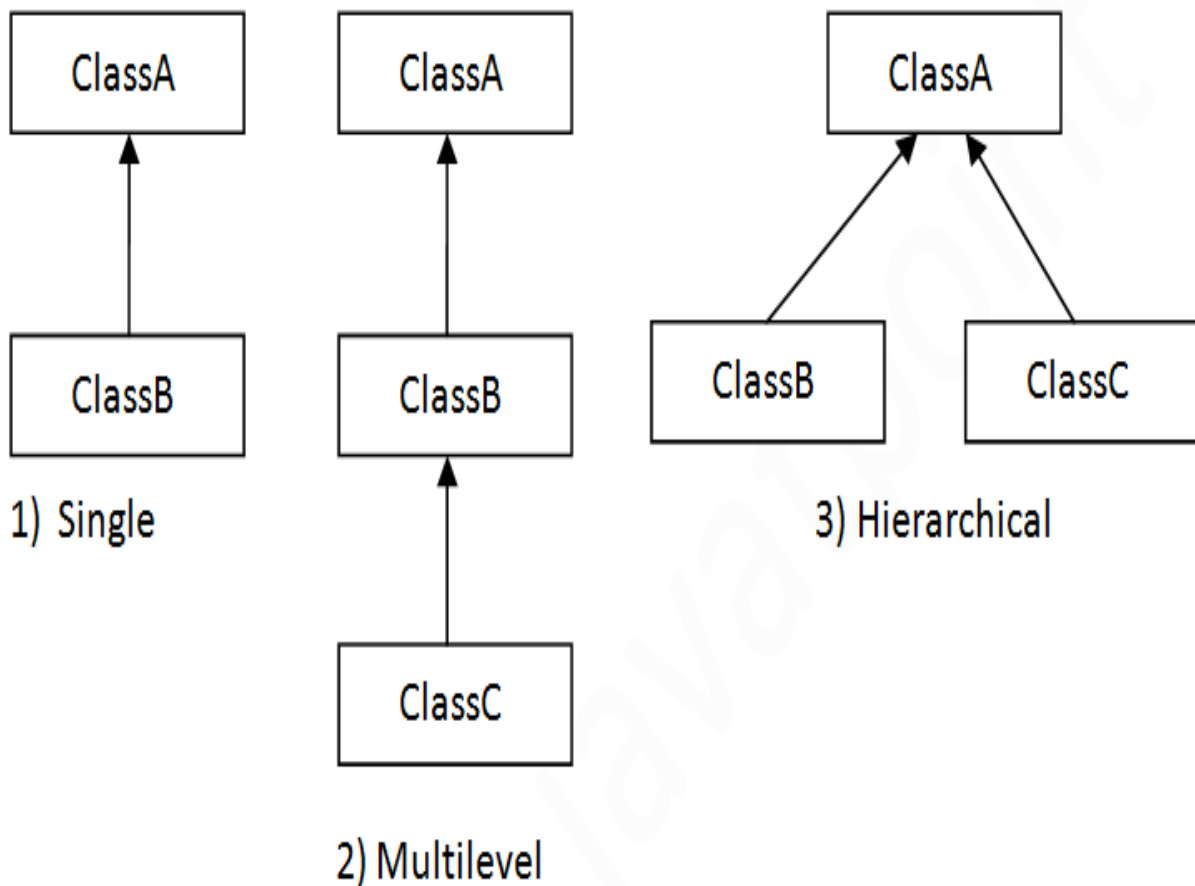
```
Programmer salary is:40000.0  
Bonus of programmer is:10000
```

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

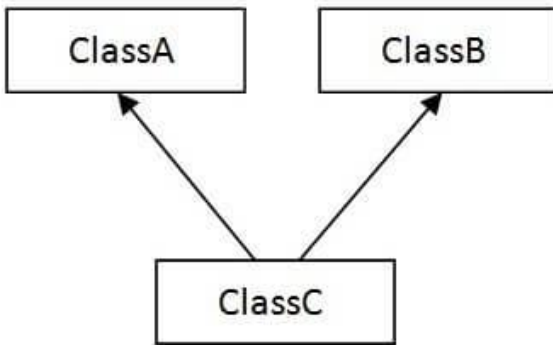
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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

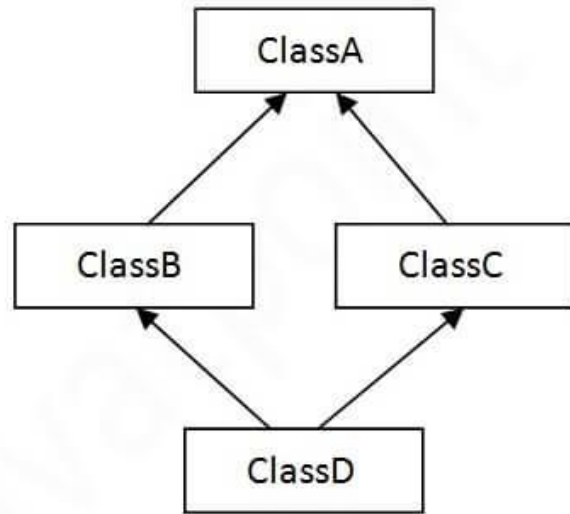


Note: Multiple inheritance is not supported in Java through class.

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



4) Multiple



5) Hybrid

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

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}
```

```
}
```

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

```
class Animal{  
    void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
    void bark(){System.out.println("barking...");}  
}  
class BabyDog extends Dog{  
    void weep(){System.out.println("weeping...");}  
}  
class TestInheritance2{  
    public static void main(String args[]){  
        BabyDog d=new BabyDog();  
        d.weep();  
        d.bark();  
        d.eat();  
    }  
}
```

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

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
    void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
    public static void main(String args[]){
        Cat c=new Cat();
        c.meow();
        c.eat();
        //c.bark();//C.T.Error
    }
}
```

Output:

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

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```

class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

public static void main(String args[]){
    C obj=new C();
    obj.msg();//Now which msg() method would be invoked?
}
}

```

OUTPUT:

Compile Time Error

4) Polymorphism in Java

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.

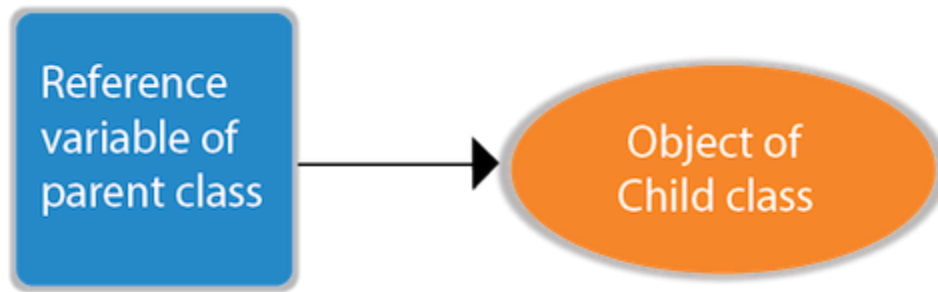
Runtime Polymorphism in Java

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:



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

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

```
interface I{}  
class A{}  
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.

```
class Bike{  
    void run(){System.out.println("running");}  
}  
class Splendor extends Bike{  
    void run(){System.out.println("running safely with 60km");}
```

```

public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
}

```

Output:

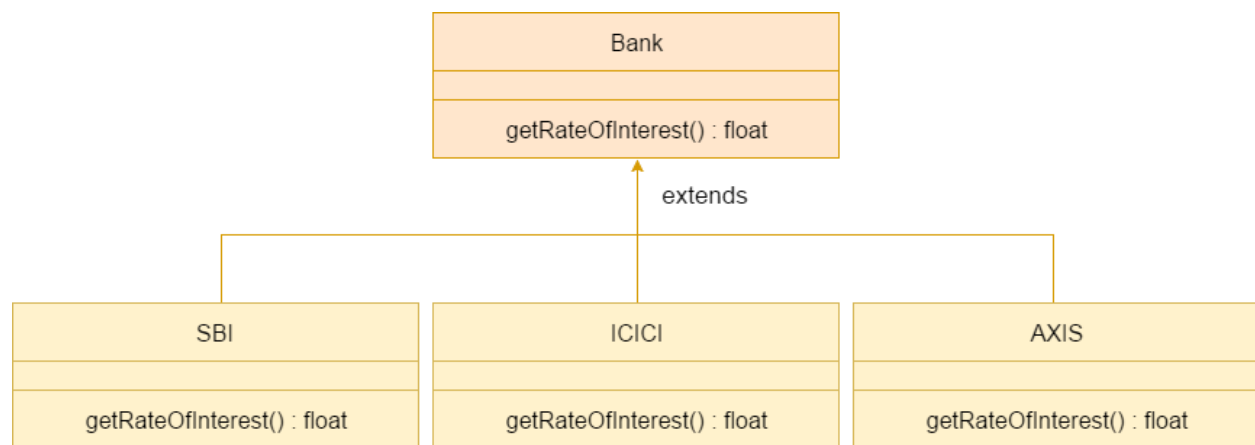
```

running safely with 60km.

```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

```

class Bank{
    float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
    float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
    float getRateOfInterest(){return 7.3f;}
}

```

```

class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}

```

Output:

```

SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7

```

Java Runtime Polymorphism Example: Shape

```

class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2{

```

```
public static void main(String args[]){  
    Shape s;  
    s=new Rectangle();  
    s.draw();  
    s=new Circle();  
    s.draw();  
    s=new Triangle();  
    s.draw();  
}  
}
```

Output:

```
drawing rectangle...  
drawing circle...  
drawing triangle...
```

Java Runtime Polymorphism Example: Animal

```
class Animal{  
    void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
    void eat(){System.out.println("eating bread...");}  
}  
class Cat extends Animal{  
    void eat(){System.out.println("eating rat...");}  
}  
class Lion extends Animal{  
    void eat(){System.out.println("eating meat...");}  
}  
class TestPolymorphism3{  
    public static void main(String[] args){  
        Animal a;  
        a=new Dog();  
        a.eat();  
    }  
}
```



```

a=new Cat();
a.eat();
a=new Lion();
a.eat();
}
}

```

Output:

```

eating bread...
eating rat...
eating meat...

```

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```

class Bike{
    int speedlimit=90;
}
class Honda3 extends Bike{
    int speedlimit=150;

    public static void main(String args[]){
        Bike obj=new Honda3();
        System.out.println(obj.speedlimit);//90
    }
}

```

Output:

```

90

```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
class Animal{
    void eat(){System.out.println("eating");}
}
class Dog extends Animal{
    void eat(){System.out.println("eating fruits");}
}
class BabyDog extends Dog{
    void eat(){System.out.println("drinking milk");}
    public static void main(String args[]){
        Animal a1,a2,a3;
        a1=new Animal();
        a2=new Dog();
        a3=new BabyDog();
        a1.eat();
        a2.eat();
        a3.eat();
    }
}
```

Output:

```
eating
eating fruits
drinking Milk
```

Try for Output

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}
class Dog extends Animal{
    void eat(){System.out.println("dog is eating...");}
```

```
}  
class BabyDog1 extends Dog{  
public static void main(String args[]){  
    Animal a=new BabyDog1();  
    a.eat();  
}  
}
```

Output:

```
Dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

5) Abstraction:

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

```
abstract class A{ }
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{  
    abstract void run();  
}  
class Honda4 extends Bike{  
    void run(){System.out.println("running safely");}  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
    }  
}
```

```
running safely
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
abstract class Shape{
    abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1 {
    public static void main(String args[]){
        Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
        s.draw();
    }
}
```

```
drawing circle
```

Another example of Abstract class in java

File: TestBank.java

```
abstract class Bank{
    abstract int getRateOfInterest();
}
class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
    int getRateOfInterest(){return 8;}
}

class TestBank{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
        b=new PNB();
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
    }
}
```

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
//Example of an abstract class that has abstract and non-abstract methods
abstract class Bike{
    Bike(){System.out.println("bike is created");}
```

```

    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}
//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}
//Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}

```

```

    bike is created
    running safely..
    gear changed

```

Rule: *If there is an abstract method in a class, that class must be abstract.*

```

class Bike12{
    abstract void run();
}

```

compile time error

Rule: *If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.*

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```

interface A{
    void a();
}

```

```
void b();  
void c();  
void d();  
}
```

```
abstract class B implements A{  
public void c(){System.out.println("I am c");}  
}
```

```
class M extends B{  
public void a(){System.out.println("I am a");}  
public void b(){System.out.println("I am b");}  
public void d(){System.out.println("I am d");}  
}
```

```
class Test5{  
public static void main(String args[]){  
A a=new M();  
a.a();  
a.b();  
a.c();  
a.d();  
}  
}
```

Output:

```
I am a  
I am b  
I am c  
I am d
```

6) Encapsulation:

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

File: Student.java

```
//A Java class which is a fully encapsulated class.  
//It has a private data member and getter and setter methods.  
package com.javatpoint;  
public class Student{  
    //private data member  
    private String name;  
    //getter method for name  
    public String getName(){  
        return name;  
    }  
}
```

```
//setter method for name
public void setName(String name){
    this.name=name
}
}
```

File: Test.java

```
//A Java class to test the encapsulated class.
package com.javatpoint;
class Test{
    public static void main(String[] args){
        //creating instance of the encapsulated class
        Student s=new Student();
        //setting value in the name member
        s.setName("vijay");
        //getting value of the name member
        System.out.println(s.getName());
    }
}
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Output:

```
vijay
```

Read-Only class

```
//A Java class which has only getter methods.
public class Student{
    //private data member
    private String college="AKG";
    //getter method for college
    public String getCollege(){
        return college;
    }
}
```

```
}  
}
```

Now, you can't change the value of the college data member which is "AKG".

```
s.setCollege("KITE");//will render compile time error
```

Write-Only class

```
//A Java class which has only setter methods.
```

```
public class Student{  
    //private data member  
    private String college;  
    //getter method for college  
    public void setCollege(String college){  
        this.college=college;  
    }  
}
```

Now, you can't get the value of the college, you can only change the value of college data member.

```
System.out.println(s.getCollege());//Compile Time Error, because there is no such method  
System.out.println(s.college);//Compile Time Error, because the college data member is private.  
//So, it can't be accessed from outside the class
```

Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

File: Account.java

```
//A Account class which is a fully encapsulated class.  
//It has a private data member and getter and setter methods.  
class Account {  
    //private data members  
    private long acc_no;  
    private String name,email;  
    private float amount;
```

```

//public getter and setter methods
public long getAcc_no() {
    return acc_no;
}
public void setAcc_no(long acc_no) {
    this.acc_no = acc_no;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public float getAmount() {
    return amount;
}
public void setAmount(float amount) {
    this.amount = amount;
}
}

```

File: TestAccount.java

```

//A Java class to test the encapsulated class Account.
public class TestEncapsulation {
public static void main(String[] args) {
    //creating instance of Account class
    Account acc=new Account();
    //setting values through setter methods

```

```
acc.setAcc_no(7560504000L);
acc.setName("Sonu");
acc.setEmail("sonu@google.com");
acc.setAmount(500000f);
//getting values through getter methods
System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
;
}
}
```

Output:

```
7560504000 Sonu sonu@google.com 500000.0
```

a) 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.

b) 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.

c) Association

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

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

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

Association can be unidirectional or bidirectional.

d) 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.

e) 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.

Advantage of OOPs over Procedure-oriented programming language

- 1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.
- 2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

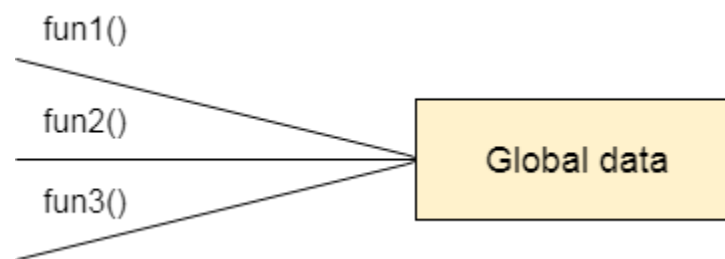


Figure: Data Representation in Procedure-Oriented Programming

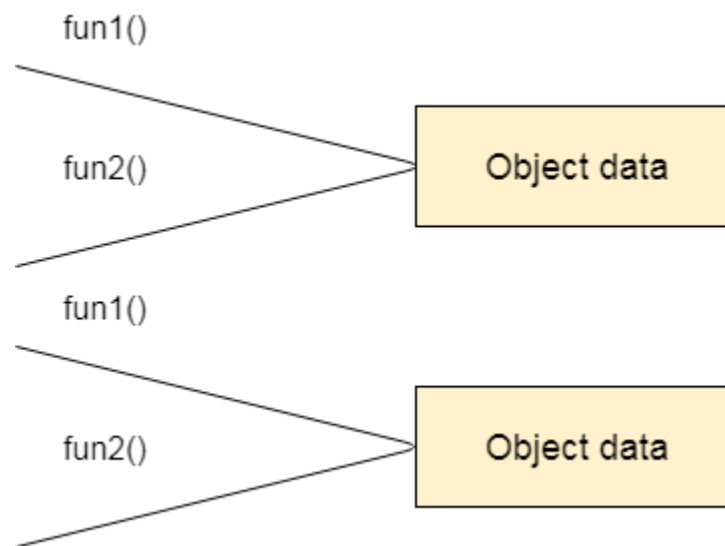


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

CONSTRUCTOR IN JAVA

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type

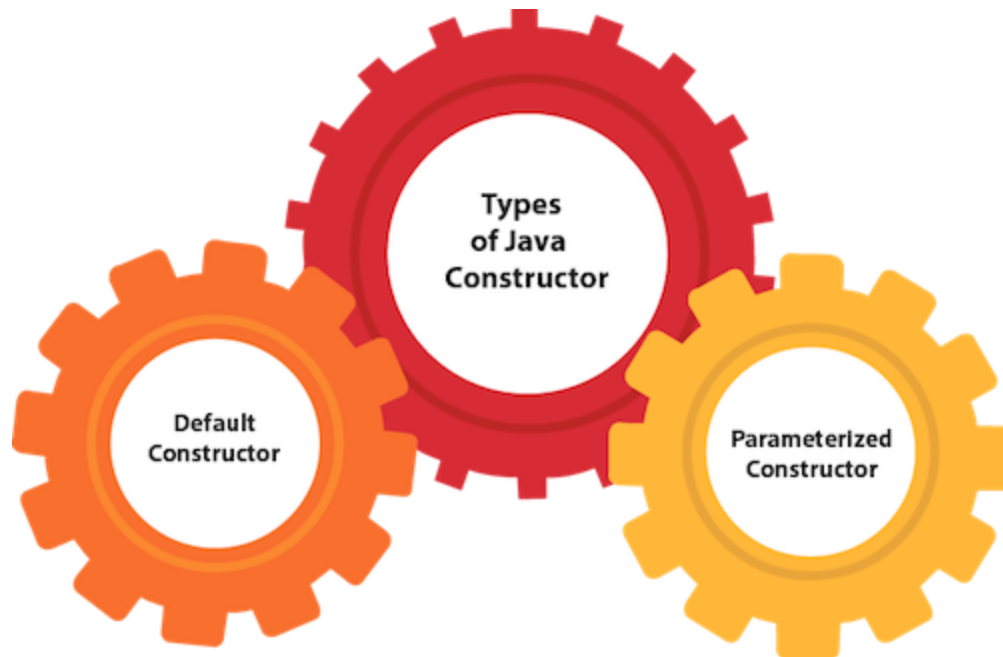
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have *private, protected, public* or *default* constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
<class_name>(){ }
```

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

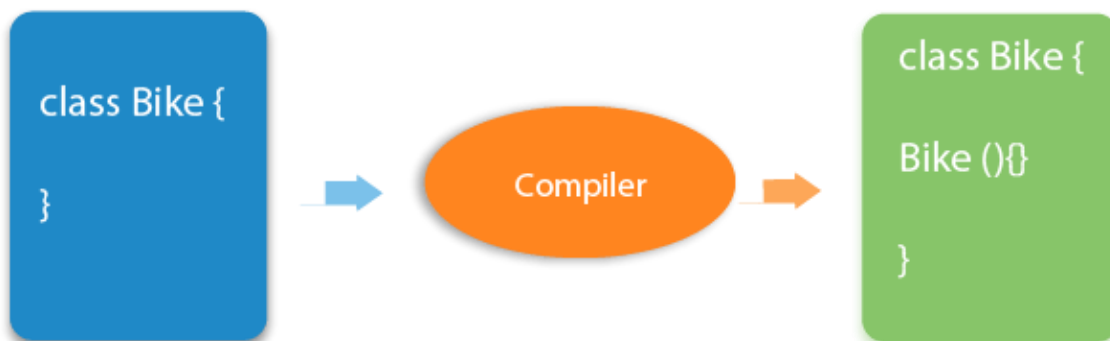
//Java Program to create and call a default constructor

```
class Bike1 {  
    //creating a default constructor  
    Bike1(){System.out.println("Bike is created");}  
    //main method  
    public static void main(String args[]){  
        //calling a default constructor  
        Bike1 b=new Bike1();  
    }  
}
```

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

//Let us see another example of default constructor

//which displays the default values

```
class Student3{  
    int id;  
    String name;
```

```
//method to display the value of id and name
void display(){System.out.println(id+" "+name);}
```

```
public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}
```

Output:

```
0 null
0 null
```

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
```

```

Student4(int i,String n){
    id = i;
    name = n;
}
//method to display the values
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
}
}

```

Output:

```

111 Karan
222 Aryan

```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```

//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor

```

```
Student5(int i,String n){
    id = i;
    name = n;
}
//creating three arg constructor
Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
}
void display(){System.out.println(id+" "+name+" "+age);}

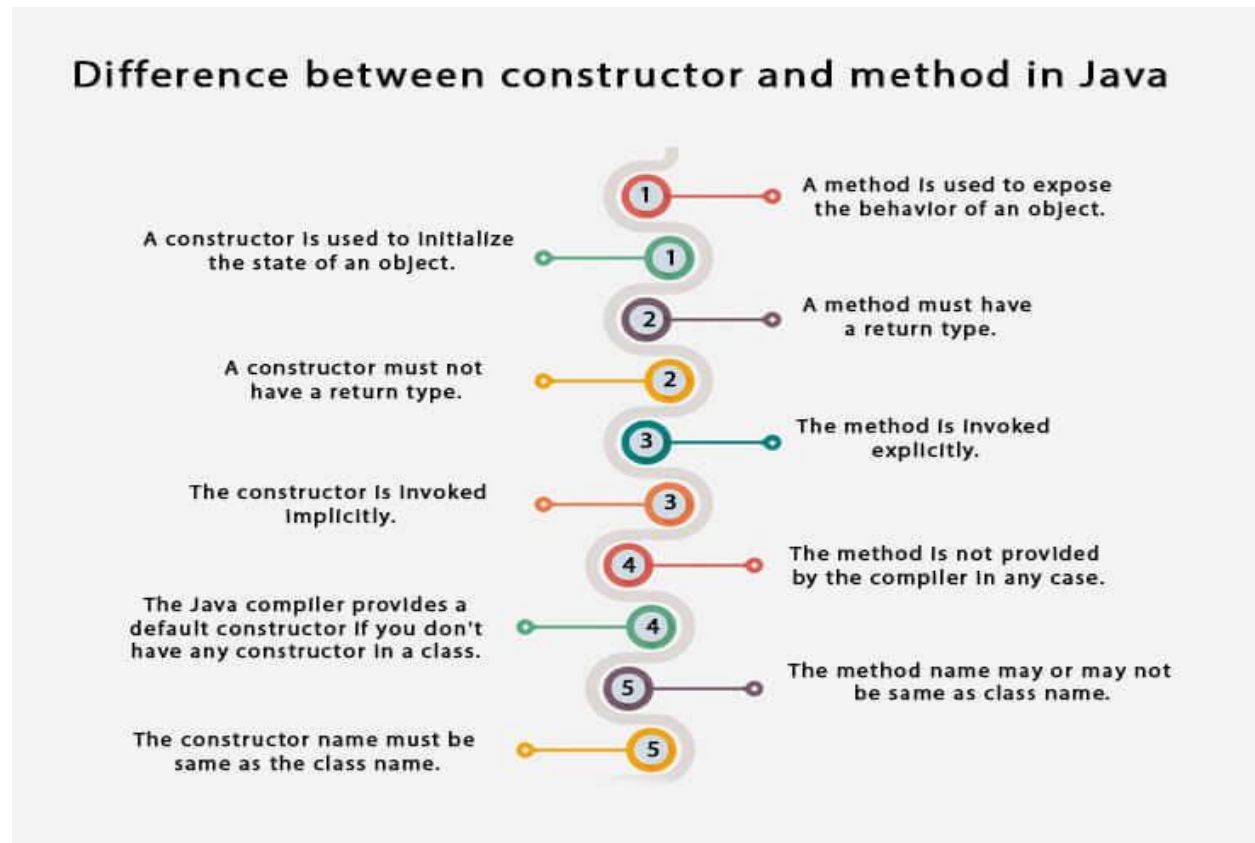
public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
}
}
```

Output:

```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.



Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

//Java program to initialize the values from one object to another object.

```
class Student6{
```

```

    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
        id = i;
        name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
        id = s.id;
        name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student6 s1 = new Student6(111,"Karan");
        Student6 s2 = new Student6(s1);
        s1.display();
        s2.display();
    }
}

```

Output:

```

111 Karan
111 Karan

```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```

class Student7{
    int id;
    String name;
    Student7(int i,String n){
        id = i;
    }
}

```

```

    name = n;
}
Student7(){
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student7 s1 = new Student7(111,"Karan");
        Student7 s2 = new Student7();
        s2.id=s1.id;
        s2.name=s1.name;
        s1.display();
        s2.display();
    }
}

```

Output:

```

111 Karan
111 Karan

```

What is Access Specifiers in Java?

In Java, access specifiers are used to specify the access level of a class or its members (data and methods). There are four access specifiers in Java:

- **public:** When we declare class members as public, they are accessible from outside the class.
- **private:** When we declare class members as private, they are only accessible within the class and are not accessible from outside the class.
- **default:** When we declare class members with no access specifier is considered as default, they are only accessible within the package and are not accessible from outside the package.
- **protected:** When we declare class members as protected, they are only accessible by any class within the same package or by any subclasses of the parent class in which the class members are declared as protected, regardless of whether the subclass is in the same package or a different package.

Note: A package is a container that contains classes in java.

We already know from the previous lessons how to use private and public access specifiers in a Java program. Now, we will see how to use a default and protected access specifier in a Java program.

Example

```
import java.util.Scanner;
```

```
class Student
```

```
{  
    int roll;           // default access specifiers  
    protected int eng;  
    protected int math;  
};
```

```
// Inherit the class Student into Result
```

```
class Result extends Student
```

```
{  
    private int total;  
    private float per;
```

```
public void input()
```

```
{  
    Scanner sc=new Scanner(System.in);  
    System.out.print("Enter Roll: ");  
    roll=sc.nextInt();  
    sc.nextLine();           // consume the new line character \n left by the previous input  
    System.out.print("Enter English: ");  
    eng=sc.nextInt();  
    System.out.print("Enter Math: ");  
    math=sc.nextInt();  
}
```

```
// Creating an instance method to calculate and store total and percentage
```



```

public void calculate()
{
    total=eng+math;
    per=(total/200.0f)*100;
}

void show()
{
    System.out.println("Roll: " + roll);
    System.out.println("English: " + eng);
    System.out.println("Math: " + math);
    System.out.println("Total Marks: " + total);
    System.out.println("Percentage: " + per);
}
}

public class Example
{
    public static void main(String args [])
    {
        Result x=new Result();
        x.input();
        x.calculate();
        x.show();
    }
}

```

Output:

```

Enter Roll: 1
Enter English: 74
Enter Math: 88
Roll: 1

```

English: 74
Math: 88
Total Marks: 162
Percentage: 81.0

In the above example, we can see that the default and protected data members of the Student class are accessible from the Result class after inheriting the Student class in it.

STATIC MEMBERS IN JAVA

Static method in Java is a method which belongs to the class and not to the object. A static method can access only static data. It is a method which belongs to the class and not to the object(instance).

- A static method can access only static data. It cannot access non-static data (instance variables) .
- A static method can call only other static methods and cannot call a non-static method from it .
- A static method can be accessed directly by the class name and doesn't need any object .

The program demonstrates the usage of static variables and methods in Java. In the staticTest class, there are two variables defined - a which is declared as static and b which is an instance variable. There are two methods defined in the class - show() which is an instance method and display() which is a static method.

In the main method of the stat_vari_methods class, two objects of the staticTest class are created - o1 and o2. The show() method is called on o1 which prints the values of a and b (which are 10 and 20 respectively). Then, the value of b in o2 is changed to 100 and the value of a is changed to 200. The show() method is called on o2 which prints the updated values of a and b. Finally, the show() method is called on o1 again which prints the original values of a and b (since a is a static variable and was updated to 200, the new value is reflected in all objects of the class).

Thus, the program demonstrates how static variables are shared among all instances of a class and how static methods can be called without creating an object of the class.

Source Code

//Static Variables and Static Methods

```
class staticTest
{
    static int a=10;

    int b=20;

    void show()
    {
        System.out.println("A : "+a+" B : "+b);
    }
}
```

```

    static void display()
    {
        System.out.println("A : "+a);
    }
}

public class stat_vari_methods {

    public static void main(String args[])
    {
        staticTest o1=new staticTest();

        o1.show();

        staticTest o2=new staticTest();

        o2.b=100;

        staticTest.a=200;

        o2.show();

        o1.show();

    }
}

```

Output

A : 10 B : 20

A : 200 B : 100

A : 200 B : 20

What is the Finalize Method in Java?

The finalize() method in Java is a protected method defined in the Object class, which is the superclass of all Java classes. It allows an object to perform any necessary cleanup operations before the garbage collector destroys it. When an object is no longer used or referenced, it becomes eligible for garbage collection. Before it is removed from memory, the garbage collector calls the finalize() method on the object. If the object has overridden the finalize() method, any custom cleanup code specified in that method will be executed.

To use the finalize() method in a class, you must override it with your implementation. In your implementation, you can include any necessary cleanup code, such as closing open files or releasing other resources.

However, there are some pitfalls to using the `finalize()` method. For example, it is not guaranteed to be called, and there is no way to know when it will be called. The `finalize()` method can also cause performance issues if it takes too long to execute.

Garbage Collector in Java

In Java, the [Garbage Collector](#) (GC) is a program that automatically manages the memory used by Java programs. The GC tracks object no longer being used or referenced by the program and free up memory by removing those objects. The GC works by periodically scanning the heap memory for objects that are no longer referenced by the program. If it finds any such objects, it frees up the memory occupied by those objects. This process is called garbage collection.

Java provides several different algorithms for garbage collection, each with its strengths and weaknesses. The most common algorithm the JVM uses is the mark-and-sweep algorithm, which works by identifying all objects that are no longer reachable by the program and then freeing up the memory occupied by those objects. One of the advantages of using a GC is that it eliminates the need for manual memory management, which can be error-prone and time-consuming. However, the GC does add some overhead to the program, which can affect its performance.

How does the Finalize Method in Java Work with Garbage Collection?

The Garbage Collector (GC) automatically manages memory by freeing up memory occupied by unused objects. The GC calls the `finalize()` method before an object is removed from memory, allowing it to perform any necessary cleanup operations.

However, the `finalize()` method can have performance implications and should be used cautiously.

Example code for overriding the `finalize()` method:

```
public class MyObject {
    private File file;

    public MyObject(String filename) {
        this.file = new File(filename);
    }

    @Override
    protected void finalize() throws Throwable {
        try {
            if (file != null) {
                file.close();
            }
        } finally {
            super.finalize();
        }
    }

    // Other methods here
}
```

In this example, the `MyObject` class represents an object that holds a reference to a `File` object. The `finalize()` method is overridden to close the file before the GC removes the object from memory.

Syntax of Finalize Method in Java

Here's the syntax for the finalize() method:

```
protected void finalize() throws Throwable {  
    // Cleanup code here  
}
```

Parameters

The finalize() method takes no arguments and has a void return type. It is declared protected to ensure that it can only be called by the garbage collector or by a subclass of the class that defines the finalize() method.

Return Value

The finalize() method does not return any value.

Why finalize() method in Java is used?

The finalize() method in Java is used for tasks such as closing external resources like files or network connections held by an object before it's garbage collected, releasing native resources to prevent leaks, and implementing custom cleanup logic. However, it's worth noting that as of Java 9, finalize() has been deprecated in favour of more modern resource management techniques like try-with-resources, considered more reliable beyond Java 8.

When to Use finalize() Method in Java?

The garbage collector calls the finalize () method before an object is reclaimed or destroyed. It can be used when you need to tidy up or release resources associated with an object. However, it is essential to note that the finalize() method has been deprecated, and its usage is discouraged. It is generally advised to employ alternative mechanisms for resource cleanup, like implementing the AutoCloseable interface and utilizing the try-with-resources statement. These alternative approaches provide improved control and ensure a predictable and reliable cleanup of resources, making them the preferred and recommended approach.

How to Override finalize() method?

To override the finalize() method in Java, and you can define your own implementation of the method in your class. This allows you to provide custom cleanup behavior for objects of that class.

When you override finalize(), you should include any necessary cleanup code within a try block and call **super.finalize()** in a finally block to ensure that any necessary cleanup is performed before the object is garbage collected.

Here's an example of how to override the finalize() method in Java:

- Java

```
public class MyClass {  
    private int id;  
  
    public MyClass(int id) {  
        this.id = id;  
    }  
  
    @Override  
    protected void finalize() throws Throwable {  
        System.out.println("Object with id " + id + " is being destroyed");  
        super.finalize();  
    }  
}
```

```

public static void main(String[] args) {
    MyClass obj1 = new MyClass(1);
    MyClass obj2 = new MyClass(2);
    MyClass obj3 = new MyClass(3);

    obj1 = null;
    obj2 = null;

    // calling garbage collector explicitly
    System.gc();

    obj3 = null;

    // calling garbage collector explicitly
    Runtime.getRuntime().gc();
}
}

```

Output

Object with id 1 is being destroyed
Object with id 2 is being destroyed

In this example, we create three instances of MyClass and then set obj1 and obj2 to null, so they are eligible for garbage collection. We then call the garbage collector explicitly using System.gc() and Runtime.getRuntime().gc().

When the garbage collector runs, it will call the finalize() method on any objects that are eligible for garbage collection. In this case, it will call finalize() on obj1 and obj2. The program's output will be:

Working of Finalize() Method in Different Scenarios

Here are a few different cases to illustrate how the finalize() method works:

Case 1: Object is garbage collected, but no finalize() method is defined

In this case, the garbage collector will destroy the object without calling any finalize() method.

- Java

```

class MyClass {
    // No finalize() method defined
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null;
        System.gc(); // Explicitly call the garbage collector
    }
}

```

Output

// No output

Case 2: Object is garbage collected, and a finalize() method is defined

In this case, the garbage collector will call the finalize() method before destroying the object.

- Java

```

class MyClass {
    protected void finalize() throws Throwable {
        System.out.println("Object destroyed");
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null;
        System.gc(); // Explicitly call the garbage collector
    }
}

```

Output

Object destroyed

Case 3: Exception in finalize() method

In this case, the garbage collector will continue with the destruction of the object, but the exception will be logged to the console.

- Java

```

class MyClass {
    protected void finalize() throws Throwable {
        throw new Exception("Exception in finalize() method");
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null;
        System.gc(); // Explicitly call the garbage collector
    }
}

```

Output

```

Exception in thread "Finalizer" java.lang.Exception: Exception in finalize() method
at MyClass.finalize(MyClass.java:3)
at java.base/java.lang.ref.Finalizer.invokeFinalizeMethod(Native Method)
at java.base/java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:170)

```

As you can see, the exception is logged to the console with details of where it occurred.

Handling Exceptions in Finalize () Method

When the garbage collector calls the finalize() method, it has a specific way of handling exceptions. If an exception occurs while the finalize() method is running, it is usually caught and logged somewhere to keep a record of it. However, the exception itself is not thrown again. This is because the garbage collector calls

the `finalize()` method automatically, and throwing an exception from there can cause some unexpected issues.

The Lifetime of the Finalized Object in Java

In Java, the lifetime of an object with a `finalize()` method goes through several phases. Here are the phases through which a finalized object passes through its lifetime:

1. Object is created

When an object is created, it is in the "live" phase. The object is allocated memory and can be accessed by the program.

2. Object becomes eligible for garbage collection

When any part of the program no longer references an object, it becomes eligible for garbage collection. At this point, the object enters the "finalizable" phase.

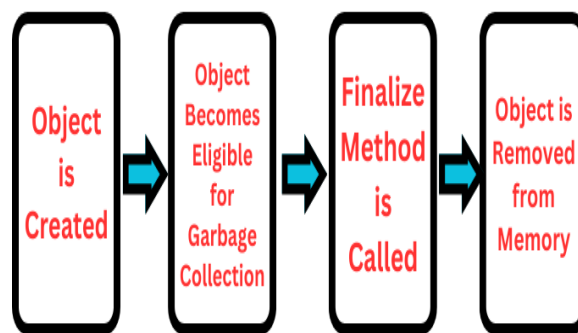
3. Finalize method is called

When the garbage collector determines that an object is eligible for garbage collection, it calls the object's `finalize()` method before reclaiming its memory. The object is in the "finalized" phase during this process.

4. Object is removed from memory

After the `finalize()` method completes, the object is removed from memory and enters the "unreachable" phase. The object is no longer accessible, and its memory is eligible for reuse by the JVM.

Lifetime of the Finalized Object



If the `finalize()` method resurrects the object by creating a new reference to it, the object will remain in the "finalizable" phase until the `finalize()` method completes. The object will then re-enter the "live" phase and continue to exist until it becomes eligible for garbage collection again.

Final vs Finally vs finalize() in Java

Below is the table providing the Key differences between the three methods:

Feature	Final	Finally	finalize()
Type	Keyword	Keyword	Method
Purpose	To restrict the modification of a variable, method, or class.	To ensure that a section of code is always executed, even if an exception is thrown.	To perform cleanup processing on an object before it is garbage collected.
Scope	Can be used with classes, methods, and variables.	Can only be used with try/catch blocks.	Can only be used with objects.
Execution	Checked at compile time.	Executed after the try/catch blocks, but before control transfers back to its origin.	Executed just before an object is garbage collected.

Advantages of Finalize Method in Java

The finalize method in Java has some potential advantages, including

- It can be used to perform cleanup operations on an object before it is garbage collected, such as releasing resources like file handles or network connections.
- It provides a safety net for situations where a programmer may forget to manually clean up an object's resources, ensuring that the resources are eventually released even if the object is no longer referenced.
- It can be used in cases where objects need to be "revived" or restored to a previous state after being garbage collected, although this usage is relatively rare.
- It can be useful for debugging or profiling, as it allows you to monitor when objects are being garbage collected and potentially identify memory leaks or other issues.

Disadvantages of Finalize Method in Java

The finalize method has some potential drawbacks, including

- It is not guaranteed to be called at any particular time or at all, so relying on it for critical cleanup operations can be risky.
- It can introduce performance issues, as the JVM must perform additional work to manage finalization and resurrected objects.
- It can make it more difficult to reason about the lifecycle of objects in a program, as the behavior of the finalize() method may not be immediately apparent.

- It is generally recommended to use other cleanup mechanisms, such as try-with-resources blocks or explicit close() method calls, instead of relying on the finalize() method.

Alternatives to Finalize() Method in Java

Here are some alternatives to using the finalize() method:

- **Use the try-with-resources statement:** This statement was introduced in Java 7 and makes it easy to manage resources that need to be closed when they are no longer needed. Resources that implement the AutoCloseable interface can be used in the try-with-resources statement, automatically calling the close() method on the resource
- **Use a shutdown hook:** A shutdown hook is a thread executed when the JVM is shutting down. This can be used to perform any necessary cleanup before the JVM exits. To use a shutdown hook, you can define a class that extends the Thread class and override the run() method
- **Use a reference queue:** A reference queue is a data structure that holds references to objects that have been garbage collected. By using a reference queue, you can perform cleanup actions on objects that are no longer needed
- **Use the PhantomReference class:** PhantomReference is a special type of reference that gets added to a reference queue when the object it refers to is garbage collected. This can be useful for performing cleanup actions on objects that are no longer needed

Best Practices to Use Finalize() Method in Java Correctly

Best Practices for Correct Usage of the finalize() Method in Java are:

- 1. Be cautious:** It is crucial to keep in mind that the finalize() method is not reliable in terms of when or even if it will be called by the garbage collector. This means relying on finalize() for critical operations or resource cleanup can be risky. Exploring alternative approaches like try-with-resources or implementing the Closeable or AutoCloseable interfaces is strongly recommended
- 2. Keep it simple:** When implementing the finalize() method, it's best to keep it simple and lightweight. Avoid including complex operations, resource-intensive tasks, or any code that could throw exceptions. Remember, the main purpose of finalize() is to handle basic cleanup operations, so it is important to focus on that
- 3. Avoid object resurrection:** It's important to avoid using the finalize() method to bring objects back to life. In other words, we shouldn't try to make objects accessible again by resurrecting them through finalize(). Attempting to do so can cause confusion, make our program behave strangely, and lead to unpredictable results
- 4. Use super.finalize():** In the finalize() method implementation, It's important to remember to include super.finalize(). This helps ensure the superclass's finalization process is carried out correctly. By doing so, we maintain the expected behaviour of the class hierarchy and allow any essential cleanup tasks defined in the superclass to be performed
- 5. Consider alternatives:** Instead of relying on the finalize() method, it's better to use explicit resource cleanup mechanisms like try-with-resources or implementing the Closeable or AutoCloseable interfaces. These newer approaches offer improved control and predictability when releasing resources

6. Prefer try-finally for cleanup: To ensure that resources are properly cleaned up. It is usually better to use a try-finally block or one of the alternative mechanisms mentioned earlier. These approaches ensure that cleanup operations are performed reliably, regardless of whether the finalize() method is called or not

7. Document intent: If you choose to use the finalize() method in your code, make sure to document its usage and limitations clearly. Explain why it is necessary and any specific precautions or requirements for it to work correctly

Frequently Asked Questions

What is the purpose of the finalize() method in Java?

The purpose of the finalize() method in Java is to perform cleanup tasks or release resources associated with an object before it gets reclaimed by the garbage collector, ensuring that necessary cleanup operations are carried out.

When is the finalize() method called?

The garbage collector calls the finalize () method when an object is ready to be reclaimed or destroyed. It offers a chance to tidy up or release any resources connected to the object. It ensures proper cleanup before removing it from memory.

What is the difference between final & finalize in Java?

In Java, the keyword final indicates that a variable, method, or class cannot be modified or extended. On the other hand, the garbage collector calls the finalize () method before discarding an object. It allows it to perform necessary cleanup or resource release operations.

How many times can finalize method be called in Java?

In Java, the finalize() method is called only once for an object during its lifetime. Once invoked by the garbage collector, it is removed from the finalization queue and will not be called again for that specific object.

INTRODUCTION TO ARRAY IN JAVA

In this digital world, storing information is a very crucial task to do as a programmer.

Every little piece of information needs to be saved in the memory location for future use. Instead of storing hundreds of values in different hundreds of variables, using an array, we can **store all these values in just a single variable**. To use that data frequently, we must assign a name, which is done by variable.

An array is a **homogenous non-primitive** data type used to save multiple elements (having the same data type) in a particular variable.

Arrays in Java can hold primitive data types (Integer, Character, Float, etc.) and non-primitive data types(Object). The values of primitive data types are stored in a memory location, whereas in the case of objects, it's stored in heap memory.

Assume you have to write a code that takes the marks of five students. Instead of creating five different variables, you can just create an array of lengths of five. As a result, it is so easy to **store and access data from only one place**, and also, it is **memory efficient**.

How do Arrays in Java work?

As discussed above, an array is a data structure or container that stores the same typed elements in a sequential format.

Let's use the above example to know how arrays work in java. This is how we can implement a program to store five subject marks of a student:

```
int mark1 = 78;
```

```
int mark2 = 84;
```

```
int mark3 = 63;
```

```
int mark4 = 91;
```

```
int mark5 = 75;
```

But what if we store all these five variables in one single variable?

```
marks[0] = 78;
```

```
marks[1] = 84;
```

```
marks[2] = 63;
```

```
marks[3] = 91;
```

```
marks[4] = 75;
```

All above mentioned five variables could be accessed by the index given in the square bracket, which is known as index-based variables.

The **index always starts from 0** by default. So, if we have an array of 5 elements, it will have an index range from 0 to 4 by default.

In the above example, the first element is stored at index 0 and the last one at the 4th index. The below-mentioned diagram shows how the array got placed in computer memory.

How to Create and Initialize Array in Java?

In every program, to use an array, we need to create it before it is used. We require two things to create an array in Java - data type and length.

As we know, an array is a homogenous container that can only store the same type of data. Hence, it's required to specify the data type while creating it. Moreover, we also need to put the number of elements the array will store.

Syntax:

Unlike other programming languages, while declaring an array in java, we can put [] both before and after the variable name.

The **new** keyword is crucial to creating an array. Without using it, we cannot make array reference variables.

```
datatype[] arrayName = new datatype[length]; OR  
datatype arrayName[] = new datatype[length];  
//[length] = Length of the array
```

Example:

// Both will create an int array with a length of 5.

```
int[] myArray = new int[5];  
int myArray1[] = new int[5];
```

We can Initialize Arrays in Java in two Different ways -

1. **Initialize using index** - Once an array is created successfully, we can assign the value to any specific element using its index. Un-assigned elements automatically take the default value of the array's data type. In this case, 0 for the integer data type. On the other hand, if we try to enter more values than the size of the array, it will raise an `ArrayIndexOutOfBoundsException`.

Example:

```
int[] myArray = new int[5]; // array declared  
myArray[0] = 10;           // assigned 10 at index 0  
myArray[2] = 37;           // assigned 37 at index 0
```

Array will be -

10 0 37 0 0

2. **Initialize while declaring** - We can also initialize the whole array while declaring it just by using curly brackets `{ }`. The string must be in quotes, whereas `int`, `float`, etc., do not require that.

Example:

```
int[] myArray = { 1, 2, 3, 4, 5 }; // For integer type  
String[] myArray1 = { "A", "B", "C", "D" }; // For string type
```

How to Change an Array Element?

To change a value at a specific position in Java arrays, refer to an index. By providing index value, we tell the Java compiler where we want to change the value.

For this, we need to use a Square bracket `[]` with an index number inside it. If somehow we entered the index number bigger than the array length, then an exception occurs of type `ArrayIndexOutOfBoundsException` (We will learn more about it in this article).

Example:

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

```
myArray[1] = 5;
```

// change the value 2 to 5 at index 1

Let's understand clearly with a Java program.

```
public class Demo {  
    public static void main(String[] args) {  
        // declaring and initializing an array  
        int[] myArray = { 2, 3, 4, 5, 6 };  
  
        // print array element at index 0  
        System.out.println("Before change myArray[0]: " + myArray[0]);  
  
        // changing the array element  
        myArray[0] = 1;  
  
        // print array element at index 0  
        System.out.println("After change myArray[0]: " + myArray[0]);  
    }  
}
```

Output:

Before change myArray[0]: 2

After change myArray[0]: 1

We have created an int type array in the above code example by initializing five values. After that, we re-assigned the array element at index [0] from 2 to 1 just by using an array-defined index.

Looping Through Array Elements

For performing any operation or displaying any elements, we used an index. But by using an index, we can only implement one operation at a time. If we are required to modify all elements in an array, we must write an operation for each index.

To overcome this situation, we have a looping feature in Java that follows the principle **write once and execute multiple times**.

There are two ways to loop through the array elements.

1. **Using for Loop** - In this way of looping through the array elements, we need the array's length first. This can be done by java in-built property length. If we start the for loop from 0, then the total iteration will be the array's length - 1, and if we start the for loop from 1, then the total iteration will be the length of an array.

As shown in the above image, there is a total of 9 value in the array, and we start looping from 0 to 8.

Example:

```
public class Demo {  
    public static void main (String[] args) {  
        // declaring and initializing an array  
        String strArray[] = {"Python", "Java", "C++", "C", "PHP"};  
  
        // Find the length of an array  
        int lengthOfArray = strArray.length;  
  
        // using for loop  
        for(int i=0;i<lengthOfArray;i++) {  
            System.out.println(strArray[i]);  
        }  
    }  
}
```

Output:

```
Python  
Java  
C++  
C  
PHP
```

In the above code, we have just used a for loop to go through the array element. For the number of iterations, we find the length of the array using the array length property.

2. **Using for-each loop** - In this way, we use a for loop, but we do not need the length of the array. The for-each method is much easier to loop through the array element than looping for-loop. While using the for-each loop, we need to define a variable of the same type as the data type of the defined array. If not done, the compiler will throw compile time error saying- incompatible types: String cannot be converted to YourDefinedDataType

Example:

```
public class Demo {  
  
    public static void main (String[] args) {  
  
        // declaring and initializing an array  
  
        String strArray[] = {"Python", "Java", "C++", "C", "PHP"};  
  
  
        // using for-each loop  
  
        for(String i : strArray) {  
            System.out.println(i);  
        }  
    }  
}
```

Output:

```
Python  
Java  
C++  
C  
PHP
```

The above for-each code can be read - *for each String element in strArray, print the value of i.*

Apart from the above two looping methods, we can also use a while loop for traversing.

Types of Array in Java

In Java, there are two types of arrays:

1. Single-Dimensional Array
2. Multi-Dimensional Array

1. Single Dimensional Array

An array that has **only one subscript** or one dimension is known as a single-dimensional array. It is just a list of the same data type variables.

One dimensional array can be of either one row and multiple columns or multiple rows and one column. For instance, a student's marks in five subjects indicate a single-dimensional array.

Example:

```
int marks[] = {56, 98, 77, 89, 99};
```

During the execution of the above line, JVM (Java Virtual Machine) will create five blocks at five different memory locations representing marks[0], marks[1], marks[2], marks[3], and marks[4].

We can also define a one-dimensional array by first declaring an array and then performing memory allocation using the new keyword.

Example:

```
int marks[];    // declared an array

marks = new int[5]; // allocating memory
```

```
// initializing array using index
```

```
marks[0] = 56;
marks[1] = 98;
marks[2] = 77;
marks[3] = 89;
marks[4] = 99;
```

Let's look at how we can implement a single-dimensional array in Java.

```
public class Demo {

    public static void main (String[] args) {

        // declaring and initializing an array

        String strArray[] = {"Python", "Java", "C++", "C", "PHP"};

        // using a for-each loop for printing the array

        for(String i : strArray) {

            System.out.print(i + " ");

        }

        // find the length of an array
```

```

        System.out.println("\nLength of array: "+strArray.length);

    }
}

```

Output:

Python Java C++ C PHP

Length of array: 5

In the above example code, we created a single-dimensional array called strArray, which has only one row and multiple columns.

2. Multi-Dimensional Array

In not all cases, we implement a single-dimensional array. Sometimes, we are required to create an array within an array.

Suppose we need to write a program that stores five different subjects and the marks of sixty students. In this case, we just need to implement 60 one-dimensional arrays with a length of five and then implement all these arrays in one array. Hence, we can make the 2D array with five columns and 60 rows.

A multi-dimensional array is just an array of arrays that represents multiple rows and columns. These arrays include 2D, 3D, and nD types. 2D data like tables and matrices can be represented using this type of array.

To implement a multi-dimensional array, one needs to add two square brackets instead of one.

Example:

```

int marks[][] = {
    {77,85,68,99,87},
    {98,56,79,90,92},
    {78,88,56,70,99}
};

```

OR

```

int marks[][] = new int[3][5];

```

// both will create a 2D array with 3 rows and 5 columns.

If we want to change any specific value, we need to use two square brackets, the first representing the rows and the second representing the columns.

```

marks[1][2] = 84    // change value at 1st row and 2nd column from 79 to 84

```

//array will look like this -

77,85,68,99,87

98,56,84,90,92

78,88,56,70,99

Now let's understand a multi-dimensional array with a famous Matrix multiplication example.

```
public class Demo {  
    public static void main (String[] args) {  
        // declaring and initializing arrays  
        int arr1[][] = {{1,2,3},{4,5,6},{7,8,9}};  
        int arr2[][] = {{2,2,2},{2,2,2},{2,2,2}};  
  
        // Printing Array1 in matrix format  
        System.out.println("Array1 -");  
        for(int i=0;i<3;i++) { //for-loop for number of rows  
            for(int j=0;j<3;j++) { //for-loop for number of columns  
                System.out.print(arr1[i][j] + " ");  
            }  
            System.out.println();  
        }  
  
        // Printing Array2 in matrix format  
        System.out.println("Array2 -");  
        for(int i=0;i<3;i++) { //for-loop for number of rows  
            for(int j=0;j<3;j++) { //for-loop for number of columns  
                System.out.print(arr2[i][j] + " ");  
            }  
            System.out.println();  
        }  
  
        // creating another array with the same dimensions to store the result
```

```

int arr3[][] = new int[3][3];

// Multiplying arr1 and arr2, storing results in arr3
System.out.println("Multiplication of Array1 and Array2 - ");
for(int i=0;i<arr1.length;i++) {
    for(int j=0;j<arr2.length;j++) {
        arr3[i][j] = 0;
        for(int k=0;k<arr3.length;k++) {
            arr3[i][j] += arr1[i][k] * arr2[k][j];
        }
        System.out.print(arr3[i][j] + " ");
    }
    System.out.println();
}
}
}

```

Output:

Array1 -

1 2 3

4 5 6

7 8 9

Array2 -

2 2 2

2 2 2

2 2 2

Multiplication of Array1 and Array2 -

12 12 12

30 30 30

48 48 48

Matrix multiplication is a very famous example of a 2D array. In the above code, we declared two arrays of 3x3 dimensions. Then we created one more array with the same dimensions to store the result. Later we implemented a logic of matrix multiplication.

Arrays of Objects

As the name suggests, an array of objects is nothing but a list of objects stored in the array. Notice that it does not store objects in an array but stores the reference variable of that object. The syntax will be the same as above.

Example:

```
Student studentObj[] = new Student[3];
```

After the above statement executes, it will create an array of objects of the Student class with a length of 3 studentObj references. For each object reference, we need to implement an object using new.

Let's understand it more clearly with a Java program.

```
class Student {  
    Student(int id, String name) {  
        System.out.println("Student ID is " + id + " and name is " + name );  
    }  
}  
  
public class Test {  
    public static void main (String[] args) {  
        // declaring an array of Object  
        Student obj[] = new Student[3];  
  
        obj[0] = new Student(1,"Bharat");  
        obj[1] = new Student(5,"Vivaan");  
        obj[2] = new Student(6,"Smith");  
  
    }  
}
```

Output:

Student ID is 1 and name is Bharat

Student ID is 5 and name is Vivaan

Student ID is 6 and name is Smith

As you can see in the above code, we have first declared the array of objects with a length of three, which create three object reference variable obj[0], obj[1], and obj[2]. And then initialized each object reference by using a new keyword.

Passing Arrays to Methods

Likewise, we pass variables as a parameter and can also pass the array to methods. While passing, we do not require adding a square bracket with an array name, but we do require that in formal parameters.

Passing an array while calling methods is not required to mention the size of the array. If you do so, then it will throw the compile-time error.

Note that it is not required to use the same name of the array variable in the parameter of a function as the array name. We can use any name we want. But make sure to define the parameter variable with the same data type defined in an array in the main function, or else you will see a compilation error.

Let's understand it more clearly with a small Java program.

```
public class Test {  
  
    // defined function to find max number from an array  
    public static int findMax(int[] arr) {  
        int max = arr[0];  
        for(int i=1;i<arr.length;i++) {  
            if(max<arr[i]) {  
                max = arr[i];  
            }  
        }  
        return max;  
    }  
  
    public static void main (String[] args) {  
        // declaring and initializing an array  
        int myArray[] = {45,33,98,65,76,43,99,23,68};  
  
        // call function by passing array in it  
        int maxNumber = findMax(myArray);  
  
        System.out.println("The max number of array is " + maxNumber);  
    }  
}
```

```
}  
}
```

Output:

The max number of array is 99

In the above code, we have created a static function in which we passed the array myArray from the main function. Notice that while passing an array, it is not required to use [] brackets, but in function parameters, we need to use [] brackets to define the array.

Returning Arrays from Methods

Sometimes, we need to pass an array to a method to perform some operations and want that modified array as a return value of the function.

We already learned how we could pass arrays in methods. Let's now see how to send arrays back from the function using the return keyword.

Let's understand it more clearly with a small Java program.

```
public class Test {  
    // defined function to find max number from an array  
    public static int[] doMultiplication(int[] arr) {  
        for(int i=0;i<arr.length;i++) {  
            arr[i] = arr[i] * 2;  
        }  
        return arr;  
    }  
    public static void main (String[] args) {  
        // declaring and initializing an array  
        int myArray[] = { 1,2,3,4,5};  
  
        // call function by passing array in it  
        int[] multiplyArr = doMultiplication(myArray);  
  
        // print array  
        System.out.println("Array multiply by 2 is - " );  
    }  
}
```

```

        for(int i : multiplyArr) {
            System.out.print(i + " ");
        }

    }
}

```

Output:

Array multiply by 2 is -
2 4 6 8 10

Like we defined the data type of a function according to the return type in an array, we also need to define a function with an array data type. We have done the same in the above code, defined function doMultiplication() of type int[], and stored the result of this method in int type array multiplyArr.

Anonymous Array in Java

As its name suggests, arrays having no name are called Anonymous arrays in java. This type of array is used only when we require an array instantly. We can also pass this array to any method without any reference variable.

As anonymous arrays do not have reference variables, we can not access its element, though we can use it entirely to meet our logic.

The primary reason to create an anonymous array is to implement and use an array instantly.

Syntax:

```
new datatype[] {values separated by comma}
```

Example:

//One-dimensional integer type anonymous array

```
new int[] {2,4,6,8};
```

// Multi-dimensional integer type anonymous array

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

Note: In an anonymous array, we do not need to mention the size inside [] but the number of values we put inside {} becomes the size of the array.

ArrayIndexOutOfBoundsException

Till now, we saw how to access a specific element of an array and loop through all the array elements. But what if, by mistake, we try to access an index value that is either greater than the array length or negative?

In this situation, JVM throws the runtime exception of **ArrayIndexOutOfBoundsException**, which points to an array accessed by an illegal index.

If in the above example we try to access `currencies[5]` then the system will give us `ArrayIndexOutOfBoundsException`.

Example:

```
public class Test {  
    public static void main (String[] args) {  
        // declaring and initializing an array  
        int myArray[] = new int[5];  
  
        myArray[0] = 3;  
        myArray[2] = 6;  
        myArray[4] = 9;  
        myArray[6] = 12; // index 6 not exists  
  
        for(int i : myArray) {  
            System.out.println(i);  
        }  
    }  
}
```

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 6 out of bounds for length 5

at Test.main(Test.java:9)

In the above code, we can see that all work fine till initializing index [4]. But, when we tried to initialize the array value at index [6], the compiler threw an exception, because we only have the array with a length of five.

Cloning of Arrays in Java

Cloning an array in Java is nothing but creating a new array, where data is copied from the existing one using the `clone()` property.

Syntax:

```
datatype Array2[] = Array1.clone();
```

Example:

```
public class Test{  
    public static void main (String[] args) {  
        // declaring and initializing an array  
        int myArray[] = {1,2,3,4,5};  
  
        // clone myArray by using clone property  
        int cloneArray[] = myArray.clone();  
  
        // checking whether both arrays are the same or not  
        System.out.println(myArray == cloneArray);  
    }  
}
```

Output:

false

When we are cloning on the multi-dimensional array, a **shallow copy** is performed by creating a new single array with each element referring to an original array.

Example:

```
public class Test {  
    public static void main (String[] args) {  
        // declaring and initializing an array  
        int myArray[][] = { {1,2,3}, {4,5,6}};  
  
        // clone myArray by using clone property  
        int cloneArray[][] = myArray.clone();  
  
        // checking whether both arrays are the same or not  
        System.out.println(myArray == cloneArray);  
  
        // checking whether both arrays have the same elements
```

```
System.out.println(myArray[1] == cloneArray[1]);  
}  
}
```

Output:

false

true

As we can see, we cloned the 2D array cloneArray from myArray. Although both arrays are not the same, their elements point to the same value. Use this [Java Compiler](#) to compile your code

Difference Between Java Array and C++ Array

Apart from the behavior of arrays, many things differ between C++ and Java talking about arrays.

- First and foremost, when we declare an array in C++, memory for the array is allocated. On the other hand, when we declare an array in Java, we only declare the reference variable, which will point to the array element once the array is initialized using a new keyword.

// In C++

```
int arr[5]; // arr of length 5 is created
```

```
arr[0] = 1; // initializing element
```

// In Java

```
int[] arr; // arr is just a reference variable
```

```
arr = new int[5]; // now array will allocate memory
```

```
arr[0] = 1; //initializing element
```

- Whenever we try to access an element using an index less than 0 or greater than the array length, JVM aborts the program by raising an exception of `IndexOutOfBoundsException`. At the same time, C++ does not have a feature to detect such errors. Hence, the program goes into an infinity loop until the max buffer length is not exceeded.
- As far as an array of objects is concerned, C++ will likely implement the constructed array of objects in a single operation. On the other hand, Java needs multiple steps to perform an array of objects.
- Like arrays in Java, C++ does not have a length property to calculate the array length every time we want. We need to store the length of an array in the C++ program.

Advantages and Disadvantages of Arrays in Java

Advantages

- We can **access an array element value randomly** just by using the index indicated by the array.
- At a time, we can **store lots of values** in arrays.
- It becomes **easier to create and work** with the multi-dimensional arrays.

Disadvantages

- Arrays in Java do not have in-built remove or add methods.
- In Java arrays, we need to specify the size of the array. So there might be a change of **memory wastage**. So, it is recommended to use ArrayList.
- Arrays in Java are **strongly typed**.

What is a String in Java?

There exists a primitive data type char in Java which is used to declare character-type variables. It represents or stores a single character. But what if we need to store any name or a sequence of characters?

There can be two ways(or methods) of doing this:

Using char[] array in Java

```
public class Main {

    public static void main(String args[]) {
        char[] ch = { 's', 'u', 'f', 'i', 'y', 'a', 'n' };
        System.out.println(ch); // sufiyan
    }
}
```

Output:

sufiyan

Using String class in Java

```
// student name

public class Main {

    public static void main(String args[]) {
        String stdName = "sufiyan";
        System.out.println(stdName); // sufiyan
    }
}
```

```
}
```

Output:

sufiyan

As you can see in picture above, String is an array of characters. Let us see how to create string objects in Java.

How to Create a String Object?

String object can be created using two ways:

1. Using **String Literal**.
2. Using **new keyword**.

String Literal and String Constant Pool

A literal, in computer science, is a notation used for representing a value. Java String literal can be created and represented using the double-quotes. All of the content/characters can be added in between the double quotes.

For Example:

```
public class Main {  
  
    public static void main(String args[]) {  
        // A string object  
        String scalerAcad = "Scaler Academy";  
    }  
}
```

In the above-given example, we have declared a string variable called scalerAcad and initialized it with the string value "Scaler Academy".

Now we will see the memory management of Strings in Java. The Strings in Java are stored in a special place in heap called "String Constant Pool" or "String Pool".

String Constant Pool

The string constant pool in Java is a storage area in the heap memory that stores string literals. When a string is created, the JVM checks if the same value exists in the string pool. If it does, the reference to that existing object is returned. Otherwise, a new string object is created and added to the string pool, and its reference is returned.

```
String str1 = "Scaler";
```

// New String is not created.

// str2 is pointing to the old string value only.

```
String str2 = "Scaler";
```

Why is only one string object created in the string constant pool when two strings are created using the string literal method in the above example?

- When str1 is initialized, the JVM checks the string constant pool for the string value "Scaler". If it's not present, a new string object is created and stored in the pool.
- When str2 is initialized, the JVM checks the string constant pool for the string value "Scaler". Since it's already present (due to str1), a new string value is not created. Instead, str2 points to the existing value.

By New Keyword

Strings can indeed be created using the new keyword in Java. When a string is created with new, a new object of the String class is created in the heap memory, outside the string constant pool.

Unlike string literals, these objects are allocated separate memory space in the heap, regardless of whether the same value already exists in the heap or not.

Syntax:

```
String stringName = new String("string_value");
```

Example: Creating Java Strings using the new keyword

```
String str = new String("Program");
```

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

Example of Strings in Java

Now we are going to see some more examples of string in Java.

```
String val1 = "Personal "; // using string literal
```

```
System.out.println(val1); // Personal
```

```
String val2 = new String("Computer"); // using new keyword
```

```
System.out.println(val2); // Computer
```

```
String val3 = val1 + val2; // concatenation of strings
```

```
System.out.println(val3); // Personal Computer
```

In the above-given examples, val1 is created using the string literal, val2 is created using the new keyword and val3 is created by concatenating/joining val1 and val2.

Methods of Java Strings

1. int length() Method

The length() method of string in Java is used to get the length of the string. In some cases, programs act according to the length of the string, and hence this method comes in very handy at that time.

Syntax:

```
stringName.length()
```

Example:

```
String genre = "action";  
int genreLength = genre.length();  
System.out.println(genreLength); // 6
```

Output:

```
6
```

2. char charAt(int index) Method

The charAt() method of string in Java accepts an index number as an argument and returns the character at the given index number.

The indexing is from 0 to length-1, where length is the length of the string. In case the passed index number is not present in the string(if it's equal to or greater than the length of the string or if it is a negative number), StringIndexOutOfBoundsException is thrown.

Syntax:

```
stringName.charAt(index)
```

Example :

```
String str = "scaler academy";  
char ch = str.charAt(7);  
System.out.println(ch); // a
```

Output:

```
a
```

Exception Example:

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

```
String str = "scaler academy";  
char ch = str.charAt(14);  
System.out.println(ch);  
}  
}
```

Output:

Exception: java.lang.StringIndexOutOfBoundsException: String index out of range: 14

Explanation: In the above-given example, when we passed 14 as an argument index number, we got StringIndexOutOfBoundsException. It is because the length of the string = 14, hence highest index in the string is 13.

3. String concat(String string1) Method

Details: What if we want to concat(or connect) two strings?

We have a method for that too! The concat() method of string in Java is used for concatenating two strings.

Syntax:

```
string.concat(anotherString)
```

Example:

```
String str1 = "scaler";  
str1 = str1.concat(" academy");  
System.out.println(str1); // scaler academy
```

Output:

```
scaler academy
```

4. String substring(int beginIndex, int endIndex[optional]) method

Details: The substring() method in Java returns a specified part of a string. It takes two parameters: the starting index (inclusive) and the ending index (exclusive). The substring will include characters starting from the beginIndex up to endIndex - 1.

If we do not provide the endIndex, it is assumed to be the length of the string.

Syntax:

```
string.substring(startIdx, endIdx)
```

Example:

```
String str = "scaler academy";  
String str1 = str.substring(0, 11);
```



```
String str2 = str.substring(3);  
System.out.println(str1); // scaler acad  
System.out.println(str2); // ler academy (from index 3 to 13 both inclusive)
```

Output:

```
scaler acad  
ler academy
```

5. String equals(String anotherString) method

The equals() method of string in Java is used to verify if both the strings are equal or not. The equals method accepts another string as an argument and then checks for the equality of both the strings. If both the strings are equal, true is returned else false is returned.

Syntax:

```
string.equals(anotherString)
```

Example:

```
String str = "scaler";  
String str1 = "SCALER";  
String str2 = "scaler";  
String str3 = "topics";  
System.out.println(str.equals(str1)); // false  
System.out.println(str.equals(str2)); // true  
System.out.println(str.equals(str3)); // false
```

Output:

```
false  
true  
false
```

6. String contains(String substring) Method

The contains() method in Java is used to check if a string contains a specified substring. It returns true if the substring is found and false otherwise.

Syntax:

```
string.contains(string)
```

Example:

```
String str = "Follow scaler on LinkedIn";
```

// 1st print statement

```
System.out.println(str.contains("on Linkedin")); // true
```

// 3rd print statement

```
System.out.println(str.contains("Follow")); // true
```

// 4th print statement

```
System.out.println(str.contains("follow")); // false
```

Output:

true

true

false

7. String join()

Details: The join() method of string in Java as the name suggests is used to join a group of strings using the joiner between them. The joiner variable can be any character, string or a sequence of characters.

Syntax:

```
string.join(joiner, str1, str2, str3,...)
```

Example:

```
String strJoin = String.join(" ", "Have", "a", "Nice", "day");
```

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

```
String str1Join = String.join("-", "Have", "a", "Nice", "day");
```

```
System.out.println(str1Join);
```

```
String str2Join = String.join("/", "19", "02", "2022");
```

```
System.out.println(str2Join);
```

```
String str3Join = String.join("::", "12", "18", "55");
```

```
System.out.println(str3Join);
```

Output:

Have a Nice day

Have-a-Nice-day

19/02/2022

12::18::55

8. int compareTo(String string1, String string2) Method

Details: The compareTo() method of string in Java compares the given strings in the order of occurrence in the dictionary. The comparison is solely based on the Unicode value of the characters of the strings.

Syntax:

```
string.compareTo(str1, str2)
```

Example:

```
String str1 = "Scaler Academy";
```

```
String str2 = "Scaler Academy";
```

```
String str3 = "Academy Scaler";
```

```
int result1 = str1.compareTo(str2);
```

```
System.out.println(result1); // 0
```

```
// comparing str1 with str3
```

```
int result2 = str1.compareTo(str3);
```

```
System.out.println(result2); // 18
```

```
// comparing str3 with str1
```

```
int result3 = str3.compareTo(str1);
```

```
System.out.println(result3); // -18
```

Output:

0

18

-18

9. String toUpperCase() Method

Details: The toUpperCase() method of string in Java is used to convert the lowercase characters of the string to the uppercase(or capital) characters.

Syntax:

```
string.toUpperCase()
```

Example:

```
String str = "keyboard";  
String strUpper = str.toUpperCase();  
System.out.println(strUpper);
```

Output:

KEYBOARD

It should be noted that there's no change in the original string str.

10. String toLowerCase() Method

Details: The toLowerCase() method of string in Java is used to convert all the characters of the string to the lowercase(or small) characters.

Syntax:

```
string.toLowerCase()
```

Example:

```
String str = "KEyBOArD";  
String strLower = str.toLowerCase();  
System.out.println(strLower);
```

Output:

keyboard

Example:

```
String str = "KEYBOARD";  
str = str.toLowerCase();  
System.out.println(str);
```

Output:

keyboard

11. String trim() Method

Details: The trim() method of string in Java is used to trim (or remove) the extra white spaces from the specified string from both the ends.

Syntax:

```
string.trim()
```

Example:

```
String str = "    Coding is    ";  
System.out.println(str + " awesome");
```

```
str = str.trim();  
System.out.println(str + " awesome");
```

Output:

```
    Coding is    awesome  
Coding is awesome
```

Explanation: In the above example, we have used the trim() method to trim the extra white spaces of string str at the ends only.

Please note that it does not remove extra whitespaces if they are present between the text and not at the ends. Use this [Online Compiler](#) to compile your Java code.

12. String replace(char oldChar, char newChar)

Details: The replace() method of string in Java as the name suggests is used to replace all the specified character of the string with another character.

Syntax:

```
string.replace(oldChar, newChar)
```

Example:

```
String str = "Hi, i will be replaced with a";  
str = str.replace('i','a');  
System.out.println(str);
```

Output:

```
Ha, a wall be replaced with a
```

Concatenating Strings

String concatenation in Java involves combining two strings using the + operator or the concat() method. It is important to note that the original string remains unchanged unless explicitly modified.

Syntax:

```
String resultantStr = str1 + str2 + ...
```

Example-1:

```
String str1 = "Scaler ";
```

```
String str2 = " Topics";
```

```
String concatenated = str1 + str2;
```

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

Output:

Scaler Topics

Creating Format Strings

Details: The format() method is used to format the string as per the passed argument. This method accepts two arguments: format string and argument(s), then returns the formatted string.

Syntax:

```
string.format(stringWithFormatSpecifier, anotherString)
```

Example:

```
String str = "India";
```

```
String str1 = String.format("Country: %s", str);
```

```
System.out.println(str1);
```

Output:

Country: India

Explanation: In the above example, we have used the %s format specifier which is reserved for strings, Hence India got printed in place of %s. Similarly, there are format specifiers for int(%d), float(%f), etc. which can be used for formatting of these data type values.

Escape Character () in Java Strings

Escape character is the character with a backslash \ before it.

Example:

```
public class Main {
```

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

```
        String escStr = "How to learn \"coding\" easily?";
```

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

```
}  
}
```

Output:

Main.java:4: error: ';' expected

```
String escStr = "How to learn "coding" easily?";  
                ^
```

Main.java:4: error: ';' expected

```
String escStr = "How to learn "coding" easily?";  
                ^
```

2 errors

Explanation:

To include double quotes within a Java string, we use the escape character \. Placing a backslash before the double quotes indicates that they should be treated as part of the string, avoiding errors.

Example-1

```
String escStr = "How to learn \"coding\" easily?";  
System.out.println(escStr);
```

Output:

How to learn "coding" easily?

Some more examples of escape character for tab, backslash, and a new line. **Example-2**

```
String tabStr = "This space\t is of tab's size.";  
System.out.println(tabStr);
```

```
String backSlashStr = "This is a backslash \\ here.";  
System.out.println(backSlashStr);
```

```
String newLineStr = "The text\n after this will be in new line";  
System.out.println(newLineStr);
```

Output:

This space is of tab's size.

This is a backslash \ here.

The text

after this will be in new line

Java Strings: Mutable or Immutable

In Java, strings are immutable, meaning their values cannot be changed once initialized. This is because a single string object in the String constant pool can have multiple references. Modifying the value of one reference could affect other strings or reference variables, leading to conflicts. To prevent these conflicts, string objects are made immutable in Java.

The concept of the String Constant Pool is closely tied to string immutability in Java.

Although it may seem like string values can be changed in previous sections of this article, the actual string value remains unchanged.

Let us understand with the help of an example:

```
// Unmodified string str
```

```
String str = "Happy Learning";
```

```
System.out.println(str); // Output: Happy Learning
```

```
// Modified string str
```

```
str = str + " to all";
```

```
System.out.println(str); // Output: Happy Learning to all
```

In Java, strings are immutable. When we concatenate a string like " to all" with str, a new string value is created. str then points to this newly created value, while the original value remains unchanged. This behavior demonstrates the immutability of strings in Java.

INTRODUCTION TO PACKAGES IN JAVA

Let's find out why we even need packages in the first place. Say you have a laptop and a bunch of data you want to store. Data includes your favorite **movies**, **songs**, and **images**.

So, do you store them all in a **single folder** or make a **separate category** for each one and store them in their corresponding folder?

It is obvious that anyone would like to create separate folders for images, videos, songs, movies, etc. And the reason is the ease of **accessibility** and **manageability**.

How does Packages in Java improve accessibility and manageability?

If you had clustered everything inside a single folder, you would probably spend a lot of time finding the correct file. But, separating your files and storing them in the correct folder saves you a lot of time.

This way, we're improving accessibility. **Accessibility and manageability go kind of hand in hand.** Managing a file is related to how easy it's to modify a specified file to incorporate future needs.

Though management includes other factors as well, like how well the code is written, having those files in a structured format is an essential requirement.

You can think of it like that – “What if we write this article in a not-so-structured format and combine everything inside a single paragraph?” You obviously will have a hard time reading it. Isn't it?

So using the same analogy in application development, but here, packages in Java provide little more than that. In simple words, packages in Java are nothing but a folder that contains related files.

These files could be of type Java classes or interfaces or even sub-packages (don't forget the storing data inside the laptop analogy).

Types of Packages in Java

Packages in Java can be categorised into 2 categories.

1. Built-in / predefined packages
2. User-defined packages.

Built-in packages

When we install **Java** on a personal computer or laptop, many packages are automatically installed. Each of these packages is unique and capable of handling various tasks. This eliminates the need to build everything from scratch. Here are some examples of built-in packages:

- **java.lang**
- **java.io**
- **java.util**
- **java.applet**
- **java.awt**
- **java.net**

Let's see how you can use an inbuilt package in your Java file.

Importing java.lang

```
import java.lang.*;
```

```
public class Example {
```

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

```
        double radius = 5.0;
```

```
double area = Math.PI * Math.pow(radius, 2);  
System.out.println("Area: " + area);  
  
String message = "Hello, World!";  
int length = message.length();  
System.out.println("Length: " + length);  
  
int number = 42;  
String binary = Integer.toBinaryString(number);  
System.out.println("Binary: " + binary);  
}  
}
```

Output:

Area: 78.53981633974483

Length: 13

Binary: 101010

Explanation:

- In this example, we import `java.lang.*` to have access to some of the fundamental classes provided by Java, such as `Math`, `String`, and `Integer`.
- The code demonstrates the usage of these classes by calculating the area of a circle, determining the length of a string, and converting an integer to its binary representation.

User-defined packages

User-defined packages are those that developers create to incorporate different needs of applications. In simple terms, User-defined packages are those that the users define. Inside a package, you can have Java files like classes, interfaces, and a package as well (called a sub-package).

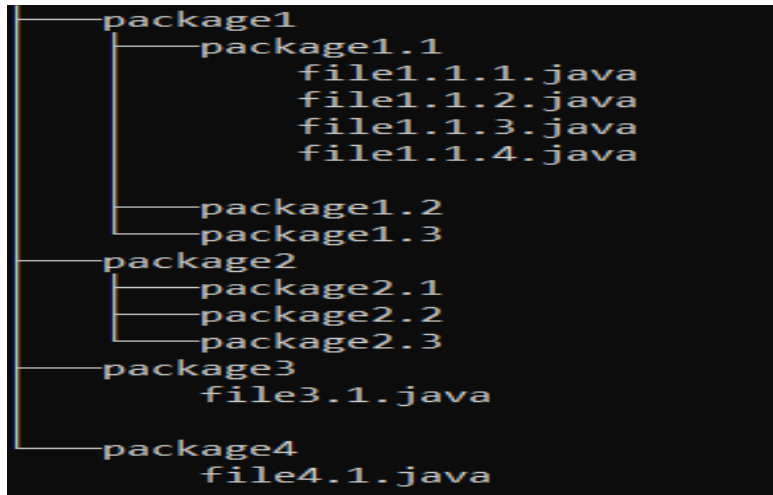
Sub-package

A package defined inside a package is called a sub-package. It's used to make the structure of the package more generic. It lets users arrange their Java files into their corresponding packages. For example, say, you have a package named `cars`. You've defined supporting Java files inside it.

But now you want to define another package called `superCars`. Should we define it inside the `car's` package or outside as another package? At this point, you must say that it should go inside `cars` because `superCars` are also related to `cars`.

But to be more generic, we're defining it as a separate package, but having it inside cars makes them part of cars.

Let's look over the package's structure using an example.



We've defined four packages, and some of them even have sub-packages defined inside them. We distributed Java files to each package. By this example, you can calculate the importance of giving a good name to files and placing them inside a designated package. **The Sooner the application grows, the more the importance of packages and this structured format becomes prominent.**

Create, Compile and Run a Java Package Program

Use package keyword to create a package in Java

```
package myFirstPackage;

class Main {

    public static void main(String args[]) {

        System.out.println("Woohoooo!! I created my first package");

    }

}
```

Let's understand the code snippet. We've defined a package named myFirstPackage. After that, we've made a class called Main, and since this is a public class, we must save this file with the same name as the class i.e. Main.java.

After compilation, when we execute this program, it'll print us the output. (Hop over to the next point to know how to compile and execute your program)

A constraint that we must follow – Sequence of the program must start from **declaring the package**, then **single or multiple import lines** and at last **class definitions**.

Compile a Java Package

To compile the Java package, open the command prompt and run the following command

```
javac -d destinationDirectoryPathWithName javaFileName
```

-d switch is used to define the destination folder to keep the generated Java class file. To use the current directory as a destination folder, use (dot) in place of the destination directory path with the name.

```
javac -d . Main.java
```

Run a Java File using Package Name

To run a Java package program:

After compiling (or performing step number (b)), use the below command to run your first Java package program.

```
java myFirstPackage.Main
```

Accessibility of Packages in Java

As you may know, in Java, we do everything inside classes. So, limiting access to variables and methods by different files must be handled for security purposes. We mainly focus on package-level accessibility here, but you can check out the table below to know more.

Note: Java has four modifiers: public, protected, private, and default.

Except for private members of a class, all other members can be accessed by all other classes defined inside the same package. Sub-classes defined inside the different packages only access to public and protected members.

Here's the table to get more clarity.

Access Modifier	Same class	Same package subclass	Same package non-subclass	Difference Package subclass	Different package non-subclass
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No
Protected	Yes	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes	Yes

From the above table, you can infer whether data members of a class with certain modifiers can be accessed by:

1. The same class in which these data members are defined
2. The classes available inside the same package
3. The sub-class of this Java file that is stored outside this package
4. All other classes are available anywhere inside the system.

Let's understand them in more detail.

Same class

If we make a class named A and define data members inside it with any modifiers (public, protected, default, and private), then we can access them inside class A.

Same Package

We have single or multiple classes inside a package. Let's say we've two classes inside a package named A and B. If we try to import class A inside class B, then except for private data members of class A all others will be accessible to class B.

Same Package Sub-class

Let's create two classes Class A and B where Class B inherits from Class A. In such a scenario, Class B can access all members of Class A except private members. It can access the default and protected members as it belongs to the same package.

Different Package Sub-class

Say you have two packages, packageA, and packageB. The packageA contains class A and packageB contains class B. Class B is a subclass of class A but since they both are in different packages, that's why class B can access only public and protected members of class A, default and private members will be hidden.

Global

Let's say a class B inside a packageB wants to access class A of packageA. Now, class B is **NOT** a subclass of class A. In this case, only public members of class A will be accessible to class B.

How to Access A Package?

Accessing a package inside another Java program file is required in almost every real-life project (or as soon as you move ahead of the basics of Java).

We use the import keyword to add desired package files into the current Java program. There are mainly three ways to access a package and its files.

Using packageName (dot) *

```
package packageB;
```

```
import packageA.*;
```

```
class B {
```

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

```
        // A class is present inside packageA
```

```
        A obj = new A();
```

```
}  
}
```

Explanation:

- Using asterisk (*) made all the classes and interfaces available in the package accessible. One thing to note here is that if a package contains a sub-package, then it'll **NOT** be accessed by doing this. **We need to import sub-package separately.**
- In the above code snippet, we've made a class named B with a main() method.
- We're able to make an object of class A just because we imported packageA and since we're using * (asterisk), it imports all the files that are defined inside packageA, and class A is one of them.

Using packageName.className

```
package packageB;
```

```
import packageA.A;
```

```
class B {
```

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

```
        // A class is present inside packageA
```

```
        A obj = new A();
```

```
    }
```

```
}
```

Explanation:

- By specifying className or another file, only that file will be imported into your current Java file. This is prominent behavior because you usually don't want to import everything inside a package, just a single file.
- Otherwise, it costs you a lot of memory which may cause other issues. This way of accessing the file is most efficient and widely used.
- In the above code snippet, we've made a class named B with a main() method. We can make an object of class A just because we imported class A from packageA.
- Though we're allowed to make an object of class A and use it, if we tried to make an object of any other class that is defined inside packageA then the Java compiler will definitely not like it and instantly show an error.

- This is because we're not using * (that imports everything from a package). If we import a single file, then all other files are inaccessible until we import them manually using the same procedure.

Using the Fully Qualified Name

```
package packageB;
```

```
class B {
```

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

```
        // No import statement specified as before
```

```
        packageA.A obj = new packageA.A();
```

```
    }
```

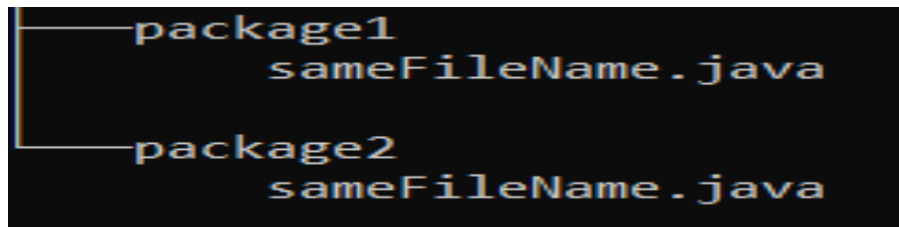
```
}
```

Explanation:

- Imagine a situation where you've imported two packages, and both of them have a file of the same name. Don't you think the advantage of the java packages (prevent name collision) is found to be void here?
- How can a [online Java compiler](#) resolve two files of the same name? Now you may get why this option is used to access a particular file.
- So, in short, it's likely to be used in cases where two or more packages have the same file name, and you want to access each of them.

Handling Naming collision error

Having two classes with the same name in a single file produces a name collision error, but having them on different packages can prevent it. So, packages in Java also help in preventing naming collision errors. See example.



```
package1
    sameFileName.java
package2
    sameFileName.java
```

We've made two packages, package1 and package2. Both packages contain a file of the same name that is **sameFileName.java**. Having them inside a single package causes a name collision error, and we use two packages to thwart it.

Advantages of Packages in Java

1. Increases ease of managing your application. How? See, as we discussed at the beginning, if we separate files and arrange them into appropriate **packages** (folder), it makes files easily accessible, letting us manage them easily.
2. Provides an extra layer of protection to your files (discussed later).
3. Prevents **Naming Collision** error.

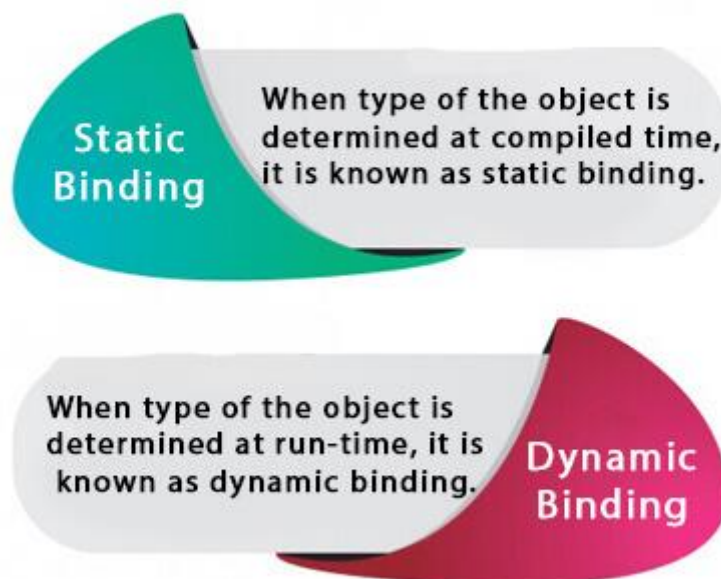
STATIC BINDING AND DYNAMIC BINDING

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

Static vs Dynamic Binding



Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

Backward Skip 10sPlay VideoForward Skip 10s

1. **int** data=30;

Here data variable is a type of int.

2) References have a type

```
class Dog{  
  
    public static void main(String args[]){  
  
        Dog d1;//Here d1 is a type of Dog  
  
    }  
}
```

3) Objects have a type

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{ }  
  
  
class Dog extends Animal{  
  
    public static void main(String args[]){  
  
        Dog d1=new Dog();  
  
    }  
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog{  
  
    private void eat(){System.out.println("dog is eating...");}  
  
  
    public static void main(String args[]){  
  
        Dog d1=new Dog();  
  
        d1.eat();  
  
    }  
}
```

```
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{  
    void eat(){System.out.println("animal is eating...");}  
}  
  
class Dog extends Animal{  
    void eat(){System.out.println("dog is eating...");}  
  
    public static void main(String args[]){  
        Animal a=new Dog();  
        a.eat();  
    }  
}
```

Output:

dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

FINAL KEYWORD IN JAVA

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.

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

javatpoint.com

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.

Backward Skip 10sPlay VideoForward Skip 10s

```
class Bike9{  
    final int speedlimit=90;//final variable  
    void run(){  
        speedlimit=400;  
    }  
    public static void main(String args[]){  
        Bike9 obj=new Bike9();  
        obj.run();  
    }  
} //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

```
class Bike{  
    final void run(){System.out.println("running");}  
}  
  
class Honda extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda honda= new Honda();  
        honda.run();  
    }  
}
```

Output:

Compile Time Error

3) Java final class

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

Example of final class

```
final class Bike{ }  
  
class Honda1 extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda1 honda= new Honda1();  
        honda.run();  
    }  
}
```

Output:

Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{  
    final void run(){System.out.println("running...");}  
}  
  
class Honda2 extends Bike{  
    public static void main(String args[]){  
        new Honda2().run();  
    }  
}
```

Output:

running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{  
    int id;  
    String name;  
    final String PAN_CARD_NUMBER;  
    ...  
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```

class Bike10{

    final int speedlimit;//blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}

```

Output: 70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```

class A{

    static final int data;//static blank final variable

    static{ data=50;}

    public static void main(String args[]){
        System.out.println(A.data);
    }
}

```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```

class Bike11{

```

```
int cube(final int n){  
    n=n+2;//can't be changed as n is final  
    n*n*n;  
}  
public static void main(String args[]){  
    Bike11 b=new Bike11();  
    b.cube(5);  
}  
}
```

Output:

Compile Time Error