INTRODUCTION TO OOP AND JAVA FUNDAMENTALS

1.1 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

List of object-oriented programming languages

Ada 95	Fortran 2003	PHP since v4, greatly enhanced in v5
BETA	Graphtalk	Python
C++	IDLscript	Ruby
C#	J#	Scala
COBOL	Java	Simula
Cobra	LISP	Smalltalk
ColdFusion	Objective-C	Tcl
Common Lisp	Perl since v5	

Abstraction

Abstraction is one of the key concepts of object-oriented programming (OOP) languages. Its main goal is to handle complexity by hiding unnecessary details from the user. This enables the user to implement more complex logic on top of the provided abstraction without understanding about all the hidden complexity.

For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the desired location without worrying about the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows us to layer the semantics of complex systems, breaking them into more manageable pieces.

- Hierarchical abstractions of complex systems can also be applied to computer programs.
- The data from a traditional process-oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects.
- Thus, each of these objects describes its own unique behavior.
- We can treat these objects as concrete entities that respond to messages telling them to do something.

Objects And Classes

Object

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

Class

A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Objects in Java

If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If we compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

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

Following is an example of a class.

```
public class Dog {
    String breed;
    int age;
    String color;
    void barking()
    {
    }
}
```

A class can contain any of the following variable types.

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

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

Encapsulation

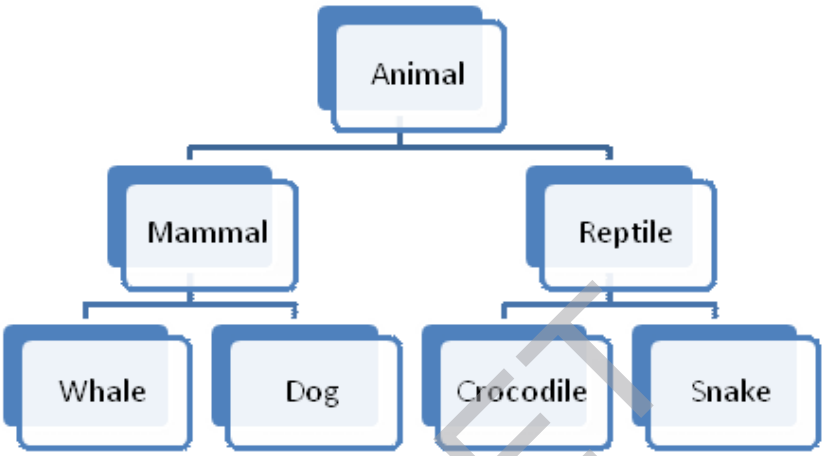
Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

- In Java, the basis of encapsulation is the class. There are mechanisms for hiding the complexity of the implementation inside the class.
- Each method or variable in a class may be marked private or public.
- The public interface of a class represents everything that external users of the class need to know, or may know.
- The private methods and data can only be accessed by code that is a member of the class.
- Therefore, any other code that is not a member of the class cannot access a private method or variable.

- Since the private members of a class may only be accessed by other parts of program through the class' public methods, we can ensure that no improper actions take place.

Inheritance

Inheritance is the process by which one object acquires the properties of another object.



For example, a Dog is part of the classification Mammal, which in turn is part of the Animal class. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, inheritance makes it possible for one object to be a specific instance of a more general case.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

For eg, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose.

Consider a stack (which is a last-in, first-out LIFO list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.

1.2 OOP CONCEPTS IN JAVA

OOP concepts in Java are the main ideas behind Java's Object Oriented Programming. They are:

Object

Any entity that has state and behavior is known as an object. It can be either physical or logical.

For example: chair, pen, table, keyboard, bike etc.

Class & Instance

Collection of objects of the same kind is called class. It is a logical entity.

A Class is a 3-Compartment box encapsulating data and operations as shown in figure.

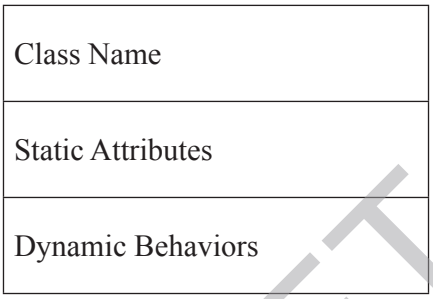


Figure: Class Structure

The followings figure shows two classes 'Student' and 'Circle'.

Name (Identifier)	Student	Circle
Variables (Static Attributes)	name, gender, dept, marks	radius, color
Methods (Dynamic Behaviors)	getDetails() calculateGrade()	getRadius() printArea()

Figure: Examples of classes

A class can be visualized as a three-compartment box, as illustrated:

1. Name (or identity): identifies the class.
2. Variables (or attribute, state, field): contain the static attributes of the class.
3. Methods (or behaviors, function, operation): contain the dynamic behaviors of the class.

An instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition. The term "object" usually refers to instance.

For example, we can define a class called "Student" and create three instances of the class "Student" for "John", "Priya" and "Anil".

The following figure shows three instances of the class Student, identified as "John", "Priya" and "Anil".

John : Student	Priya : Student	Anil : Student
name = "John" gender = "male"	name = "Priya" gender = "female"	name = "Anil" gender = "male"
dept = "CSE" mark = 88	gender = "female" dept = "IT"	gender = "male" dept = "IT"
getDetails() calculateGrade()	getDetails() calculateGrade()	getDetails() calculateGrade()

Figure: Instances of a class 'Student'

Abstraction

Abstraction refers to the quality of dealing with ideas rather than events. It basically deals with hiding the details and showing the essential things to the user.

We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it.

Abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. It avoids repeating the same work multiple times. In java, we use abstract class and interface to achieve abstraction.

Abstract class:

Abstract class in Java contains the 'abstract' keyword. If a class is declared abstract, it cannot be instantiated. So we cannot create an object of an abstract class. Also, an abstract class can contain abstract as well as concrete methods.

To use an abstract class, we have to inherit it from another class where we have to provide implementations for the abstract methods there itself, else it will also become an abstract class.

Interface:

Interface in Java is a collection of abstract methods and static constants. In an interface, each method is public and abstract but it does not contain any constructor. Along with abstraction, interface also helps to achieve multiple inheritance in Java.

So an interface is a group of related methods with empty bodies.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. It means to hide our data in order to make it safe from any modification.

The best way to understand encapsulation is to look at the example of a medical capsule, where the drug is always safe inside the capsule. Similarly, through encapsulation the methods and variables of a class are well hidden and safe.

A java class is the example of encapsulation.

Encapsulation can be achieved in Java by:

- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the variables values.

Inheritance

This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes.

For eg, a child inherits the properties from his father.

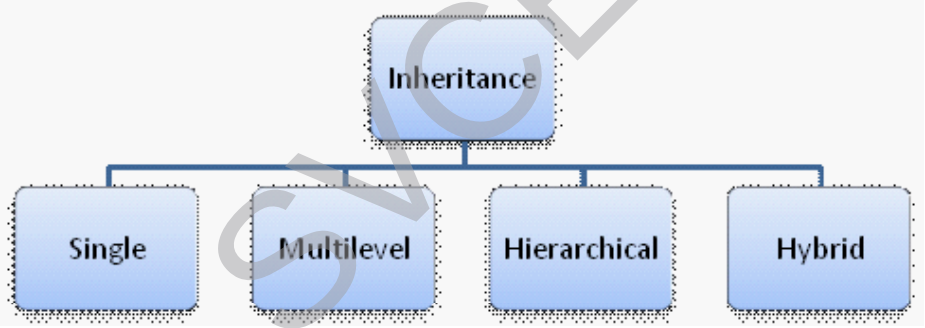
Similarly, in Java, there are two classes:

1. Parent class (Super or Base class)

2. Child class (Subclass or Derived class)

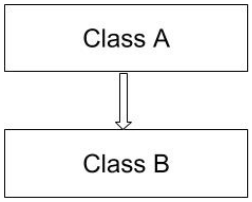
A class which inherits the properties is known as 'Child class' whereas a class whose properties are inherited is known as 'Parent class'.

Inheritance is classified into 4 types:



Single Inheritance

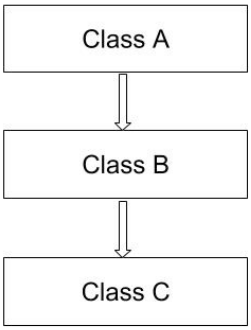
It enables a derived class to inherit the properties and behavior from a single parent class.



Here, Class A is a parent class and Class B is a child class which inherits the properties and behavior of the parent class.

Multilevel Inheritance

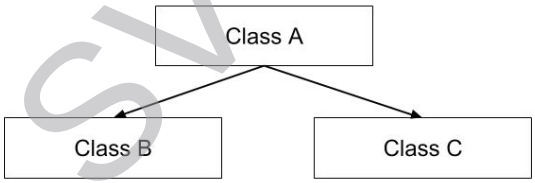
When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.



Here, class B inherits the properties and behavior of class A and class C inherits the properties of class B. Class A is the parent class for B and class B is the parent class for C. So, class C implicitly inherits the properties and methods of class A along with Class B.

Hierarchical Inheritance

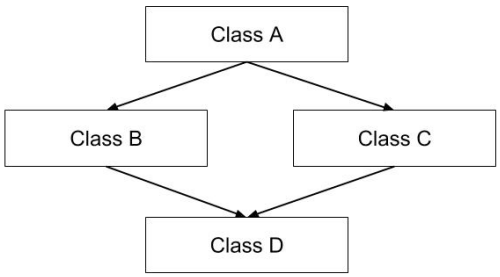
When a class has more than one child class (sub class), then such kind of inheritance is known as hierarchical inheritance.



Here, classes B and C are the child classes which are inheriting from the parent class A.

Hybrid Inheritance

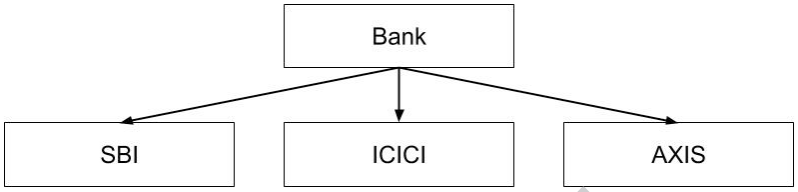
Hybrid inheritance is a combination of *multiple* inheritance and *multilevel* inheritance. Since multiple inheritance is not supported in Java as it leads to ambiguity, this type of inheritance can only be achieved through the use of the interfaces.



Here, class A is a parent class for classes B and C, whereas classes B and C are the parent classes of D which is the only child class of B and C.

Polymorphism

Polymorphism means taking many forms, where 'poly' means many and 'morph' means forms. It is the ability of a variable, function or object to take on multiple forms. In other words, polymorphism allows us to define one interface or method and have multiple implementations.



For eg, Bank is a base class that provides a method rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS are the child classes that provide different rates of interest.

Polymorphism in Java is of two types:

- Run time polymorphism
- Compile time polymorphism

Run time polymorphism:

In Java, runtime polymorphism refers to a process in which a call to an overridden method is resolved at runtime rather than at compile-time. Method overriding is an example of run time polymorphism.

Compile time polymorphism:

In Java, compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time. Method overloading is an example of compile time polymorphism.

1.3 CHARACTERISTICS OF JAVA

Simple :

- Java is Easy to write and more readable.
- Java has a concise, cohesive set of features that makes it easy to learn and use.
- Most of the concepts are drawn from C++, thus making Java learning simpler.

Secure :

- Java program cannot harm other system thus making it secure.
- Java provides a secure means of creating Internet applications.
- Java provides secure way to access web applications.

Portable :

- Java programs can execute in any environment for which there is a Java run-time system.
- Java programs can run on any platform (Linux, Window, Mac)
- Java programs can be transferred over world wide web (e.g applets)

Object-oriented :

- Java programming is object-oriented programming language.
- Like C++, java provides most of the object oriented features.
- Java is pure OOP Language. (while C++ is semi object oriented)

Robust :

- Java encourages error-free programming by being strictly typed and performing run-time checks.

Multithreaded :

- Java provides integrated support for multithreaded programming.

Architecture-neutral :

- Java is not tied to a specific machine or operating system architecture.
- Java is machine independent.

Interpreted :

- Java supports cross-platform code through the use of Java bytecode.
- Bytecode can be interpreted on any platform by JVM (Java Virtual Machine).

High performance :

- Bytecodes are highly optimized.
- JVM can execute bytecodes much faster .

Distributed :

- Java is designed with the distributed environment.
- Java can be transmitted over internet.

Dynamic :

- Java programs carry substantial amounts of run-time type information with them that is used to verify and resolve accesses to objects at run time.

1.4 JAVA RUNTIME ENVIRONMENT (JRE)

The Java Runtime Environment (JRE) is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.

JRE is part of the Java Development Kit (JDK), but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation.

JRE consists of the following components:

Name of the component	Elements of the component
Deployment technologies	Deployment Java Web Start Java Plug-in
User interface toolkits	Abstract Window Toolkit (AWT) Swing Java 2D Accessibility Image I/O Print Service Sound Drag and Drop (DnD) Input methods.
Integration libraries	Interface Definition Language (IDL) Java Database Connectivity (JDBC) Java Naming and Directory Interface (JNDI) Remote Method Invocation (RMI) Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) Scripting.

base libraries	International support Input/Output (I/O) Eextension mechanism Beans Java Management Extensions (JMX) Java Native Interface (JNI) Math Networking Override Mechanism Security Serialization and Java for XML Processing (XML JAXP).
Lang and util base libraries	lang and util Management Versioning Zip Instrument Reflection Collections Concurrency Java Archive (JAR) Logging Preferences API Ref Objects Regular Expressions.
Java Virtual Machine (JVM)	Java HotSpot Client Server Virtual Machines

1.5 JAVA VIRTUAL MACHINE (JVM)

The JVM is a program that provides the runtime environment necessary for Java programs to execute. Java programs cannot run without JVM for the appropriate hardware and OS platform.

Java programs are started by a command line, such as:

java <arguments> <program name>

This brings up the JVM as an operating system process that provides the Java runtime environment. Then the program is executed in the context of an empty virtual machine.

When the JVM takes in a Java program for execution, the program is not provided as Java language source code. Instead, the Java language source must have been converted (or compiled) into a form known as Java bytecode. Java bytecode must be supplied to the JVM in a format called class files. These class files always have a .class extension.

The JVM is an interpreter for the bytecode form of the program. It steps through one bytecode instruction at a time. It is an abstract computing machine that enables a computer to run a Java program.

1.6 SETTING UP AN ENVIRONMENT FOR JAVA

Local Environment Setup

Download Java and run the .exe to install Java on the machine.

Setting Up the Path for Windows

Assuming Java is installed in c:\Program Files\java\jdk directory −

- Right-click on 'My Computer' and select 'Properties'.
- Click the 'Environment variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

1.7 POPULAR JAVA EDITORS

To write Java programs, we need any of the following:

- **Notepad** − Text editor
- **Netbeans** − A Java IDE that is open-source and free
- **Eclipse** − A Java IDE developed by the eclipse open-source community

1.8 JAVA SOURCE FILE STRUCTURE

When we write a Java source program, it needs to follow a certain structure or template as shown in the following figure:

```
// Filename: NewApp.java

// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilePackage;

// PART 2: (ZERO OR MORE) import declarations
import java.io.*;
import java.util.*;

// PART 3: (ZERO OR MORE) top-level class and interface declarations
public class NewApp { }

class AClass { }

interface IOne { }

class BClass { }

interface ITwo { }
// ...
// end of file
```

Figure: Java Source File Structure

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A Java source file can have the following elements that must be specified in the following order:

1. An optional package declaration to specify a package name.
2. Zero or more import declarations.
3. Any number of top-level type declarations. Class, enum, and interface declarations are collectively known as type declarations.

Part 1: ***Optional Package Declaration***

A package is a pack (group) of classes, interfaces and other packages. Packages are used in Java in order to prevent naming conflicts, to control access, to make searching / locating and usage of classes, interfaces, enumerations and annotations easier, etc.

Rules:

- The package statement should be the first line in the source file.
- There can be only one package statement in each source file.
- If a package statement is not used, the class, interfaces, enumerations, and annotation types will be placed in the current default package.
- It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named animals:

```
/* File name : Animal.java */
package animals;
interface Animal
{
  public void eat();
  public void travel();
}
```

Part 2: ***Zero or More import Declarations***

The import statement makes the declarations of external classes available to the current Java source program at the time of compilation. The import statement specifies the path for the compiler to find the specified class.

Syntax of the import statement:

```
import packagename;

or

import packagename.* ;
```

We may import a single class or all the classes belonging to a package.

- To import a single class, we specify the name of the class
- To import all classes, we specify *.

Examples of the import statement:

Statement in Java	Purpose
import mypackage.MyClass;	imports the definition of the MyClass class that is defined in the mypackage package.
import mypackage.reports.accounts.salary.EmpClass;	imports the definition of EmpClass belonging to the mypackage.reports.accounts.salary package.
import java.awt.*;	imports all the classes belonging to the java.awt package.

Part 3: ***Zero or More top-level Declarations***

The Java source file should have one and only one public class. The class name which is defined as public should be the name of Java source file along with .java extension.

Source File Declaration Rules

- There can be only one public class per source file.
- A source file can have multiple non-public classes.
- The public class name should be the name of the source file which should have .java extension at the end.
- For eg, if the class name is public class Employee{}, then the source file should be as Employee.java.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

1.9 COMPILATION

In Java, programs are not compiled into executable files. Java source code is compiled into bytecode using javac compiler. The bytecodes are platform-independent instructions for the Java VM. They are saved on the disk with the file extension `.class`. When the program is to be run, the bytecode is converted into the machine code using the just-in-time (JIT) compiler. It is then fed to the memory and executed.

Java code needs to be compiled twice in order to be executed:

1. Java programs need to be compiled to bytecode.
2. When the bytecode is run, it needs to be converted to machine code.

The Java classes / bytecode are compiled to machine code and loaded into memory by the JVM when needed for the first time.

Compiling the Program

The Java compiler is invoked at the command line with the following syntax:

```
javac ExampleProgram.java
```

Interpreting and Running the Program

Once the java program successfully compiles into Java bytecodes, we can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so your computer can understand and run the program.

The Java interpreter is invoked at the command line with the following syntax:

```
java ExampleProgram
```

Quick compilation procedure

To execute the first Java program, follow the steps:

1. Open text editor. For example, Notepad or Notepad++ on Windows; Gedit, Kate or SciTE on Linux; or, XCode on Mac OS, etc.
2. Type the java program in a new text document.
3. Save the file as HelloWorld.java.
4. Next, open any command-line application. For example, Command Prompt on Windows; and, Terminal on Linux and Mac OS.
5. Compile the Java source file using the command: javac HelloWorld.java
6. Once the compiler returns to the prompt, run the application using the following command:

```
java HelloWorld
```

1.10 FUNDAMENTAL PROGRAMMING STRUCTURES IN JAVA

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line. A single-line comment begins with a // and ends at the end of the line.

Syntax	Example
//Comment	//This is single line comment

2) Java Multi Line Comment

This type of comment must begin with /* and end with */. Anything between these two comment symbols is ignored by the compiler. A multiline comment may be several lines long.

Syntax	Example
/*Comment starts continues continues . . . Commnent ends*/	/* This is a multi line comment */

3) Java Documentation Comment

This type of comment is used to produce an HTML file that documents our program. The documentation comment begins with a /** and ends with a */.

Syntax	Example
/**Comment start * *tags are used in order to specify a parameter *or method or heading *HTML tags can also be used *such as <h1> * *comment ends*/	/** This is documentation comment */

1.11 DATA TYPES

Java is a **statically typed and also a strongly typed language. I**n Java, each type of data (such as integer, character, hexadecimal, etc.) is predefined as part of the programming language and all constants or variables defined within a given program must be described with one of the data types.

Data types represent the different values to be stored in the variable. In java, there are two categories of data types:

- Primitive data types
- Non-primitive data types

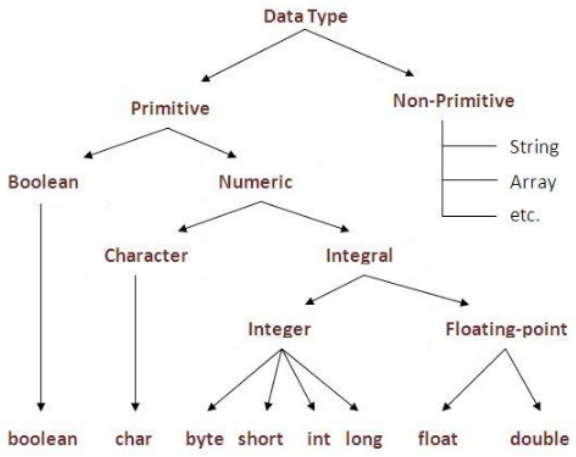


Figure: Data types in java

The Primitive Types

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types and they are grouped into the following four groups:

i) ***Integers*** - This group includes byte, short, int, and long. All of these are signed, positive and negative values. The width and ranges of these integer types vary widely, as shown in the following table:

Name	Width in bits	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Table: Integer Data Types

ii) ***Floating-point numbers*** – They are also known as real numbers. This group includes float and double, which represent single- and double-precision numbers, respectively. The width and ranges of them are shown in the following table:

Table: Floating-point Data Types

Name	Width in bits	Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

iii) ***Characters*** - This group includes char, which represents symbols in a character set, like letters and numbers. char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars.

iv) ***Boolean*** - This group includes boolean. It can have only one of two possible values, true or false.

1.12 VARIABLES

A variable is the holder that can hold the value while the java program is executed. A variable is assigned with a datatype. It is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. There are three types of variables in java: local, instance and static.

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Before using any variable, it must be declared. The following statement expresses the basic form of a variable declaration –

datatype variable [= value][, variable [= value] ...] ;

Here data type is one of Java's data types and variable is the name of the variable. To declare more than one variable of the specified type, use a comma-separated list.

Example

```
int a, b, c;         // Declaration of variables a, b, and c.
int a = 20, b = 30;  // initialization
byte B = 22;         // Declaratrion initializes a byte type variable B.
```

Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

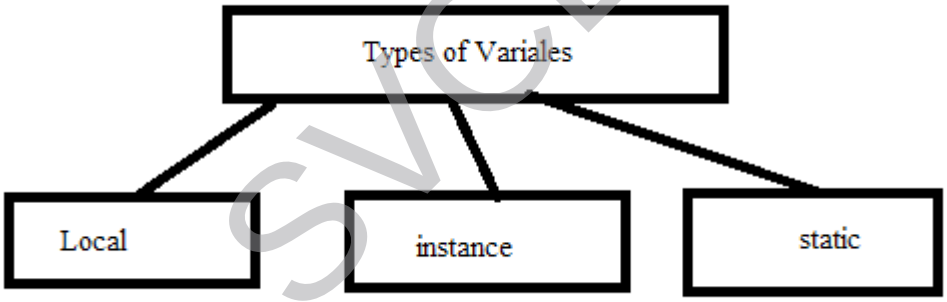


Fig. Types of variables

Local Variable

- Local variables are declared inside the methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered
- Local variable will be destroyed once it exits the method, constructor, or block.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.
- Access specifiers cannot be used for local variables.

Instance Variable

- A variable declared inside the class but outside the method, is called instance variable. Instance variables are declared in a class, but outside a method, constructor or any block.
- A slot for each instance variable value is created when a space is allocated for an object in the heap.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. It is recommended to make these variables as private. However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values.
    - numbers, the default value is 0,
    - Booleans it is false,
    - Object references it is null.
- Values can be assigned during the declaration or within the constructor.
- Instance variables cannot be declared as static.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. ObjectReference.VariableName.

Static variable

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- Only one copy of each class variable per class is created, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.

- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is same as instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables.
    - numbers, the default value is 0;
    - Booleans, it is false;
    - Object references, it is null.
- Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables cannot be local.
- Static variables can be accessed by calling with the class name ClassName. VariableName.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

1.13 OPERATORS

Operator in java is a symbol that is used to perform operations. Java provides a rich set of operators to manipulate variables.For example: +, -, *, / etc.

All the Java operators can be divided into the following groups −

- Arithmetic Operators :

  Multiplicative : `* / %`

  Additive : `+ -`
- Relational Operators

  Comparison : `< > <= >= instanceof`

  Equality : `== !=`
- Bitwise Operators

  bitwise AND : `&`

  bitwise exclusive OR : `^`

  bitwise inclusive OR : `|`

  Shift operator: << >> >>>

- Logical Operators

  logical AND : &&

  logical OR : ||

  logical NOT : ~ !

- Assignment Operators: =
- Ternary operator: ? :
- Unary operator

  Postfix : *expr*++ *expr*–

  Prefix : ++*expr* --*expr* +*expr* -*expr*

The Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations in the same way as they are used in algebra. The following table lists the arithmetic operators −

Example:

int A=10,B=20;

Operator	Description	Example	Output
+ (Addition)	Adds values A & B.	A + B	30
- (Subtraction)	Subtracts B from A	A - B	-10
* (Multiplication)	Multiplies values A & B	A * B	200
/ (Division)	Divides B by A	B / A	2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A	0

// Java program to illustrate arithmetic operators

```
public class Aoperators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
      String x = "Thank", y = "You";
       System.out.println("a + b = "+(a + b));
      System.out.println("a - b = "+(a - b));
```

```
        System.out.println("x + y = "+x + y);
        System.out.println("a * b = "+(a * b));
        System.out.println("a / b = "+(a / b));
            System.out.println("a % b = "+(a % b));
    }
}
```

The Relational Operators

The following relational operators are supported by Java language.

Example:

int A=10,B=20;

Operator	Description	Example	Output
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B)	true
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B)	true
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B)	true
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B)	true
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B)	true
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B)	true
instance of Operator	checks whether the object is of a particular type (class type or interface type) (Object reference variable) instanceof (class/interface type)	boolean result = name instanceof String;	True

```
// Java program to illustrate relational operators
public class operators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10;
      boolean condition = true;
       //various conditional operators
      System.out.println("a == b :" + (a == b));
      System.out.println("a < b :" + (a < b));
      System.out.println("a <= b :" + (a <= b));
      System.out.println("a > b :" + (a > b));
      System.out.println("a >= b :" + (a >= b));
      System.out.println("a != b :" + (a != b));
      System.out.println("condition==true :" + (condition == true));
   }
}
```

Bitwise Operators

Java supports several bitwise operators, that can be applied to the integer types, long, int, short, char, and byte. Bitwise operator works on bits and performs bit-by-bit operation.

Example:

int a = 60,b = 13;

binary format of a & b will be as follows −

a = 0011 1100

b = 0000 1101

Bitwise operators follow the truth table:

a	b	a&b	a|b	a^b	~a
0	0	0	0	1	1
0	1	0	1	0	1
1	0	0	1	0	0
1	1	1	1	1	0

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators −

int A=60,B=13;

Operator	Description	Example	Output
& (bit-wise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is	12 (in binary form : 0 0 0 0 1100)
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B)	61 (in binary form: 0011 1101)
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001	49 (in binary form: 0011 0001)
~ (bitwise compli-ment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.	-61 (in binary form: 1100 0011)
<< (left shift)	The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000	240 (in binary form: 1111 0000)

>> (right shift)	The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111	15 (in binary form: 1111)
>>> (zero fill right shift)	The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111	15 (in binary form: 0000 1111)

// Java program to illustrate bitwise operators

```
public class operators
{
   public static void main(String[] args)
   {
      int a = 10;
      int b = 20;
     System.out.println("a&b = " + (a & b));
     System.out.println("a|b = " + (a | b));
     System.out.println("a^b = " + (a ^ b));
     System.out.println("~a = " + ~a);
   }
}
```

Logical Operators

The following are the logical operators supported by java.

Example:

A=true;

B=false;

Operator	Description	Example	Ouptput
&& (logical and)	If both the operands are non-zero, then the condition becomes true.	(A && B)	false
|| (logical or)	If any of the two operands are non-zero, then the condition becomes true.	(A || B)	true
! (logical not)	Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B)	true

Assignment Operators

The following are the assignment operators supported by Java.

Operator	Description	Example
= (Simple assignment operator)	Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+= (Add AND assignment operator)	It adds right operand to the left operand and assigns the result to left operand.	C += A is equivalent to C = C + A
-= (Subtract AND assignment operator)	It subtracts right operand from the left operand and assigns the result to left operand.	C -= A is equivalent to C = C – A
*= (Multiply AND assignment operator)	It multiplies right operand with the left operand and assigns the result to left operand.	C *= A is equivalent to C = C * A

/= (Divide AND assignment operator)	It divides left operand with the right operand and assigns the result to left operand.	C /= A is equivalent to C = C / A
%= (Modulus AND assignment operator)	It takes modulus using two operands and assigns the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

// Java program to illustrate assignment operators

```
public class operators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c, d, e = 10, f = 4, g = 9;
       c = b;
      System.out.println("Value of c = " + c);
      a += 1;
```

```
        b -= 1;
        e *= 2;
        f /= 2;
        System.out.println("a, b, e, f = " +
                        a + "," + b + "," + e + "," + f);
    }
}
```

Ternary Operator

Conditional Operator (? :)

Since the conditional operator has three operands, it is referred as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

Following is an example −

Example:

```
public class example
{
public static void main(String args[])
{
int a, b;
a = 10;
b = (a == 0) ? 20: 30;
System.out.println( "b : " +  b );
}
}
```

Unary Operators

Unary operators use only one operand. They are used to increment, decrement or negate a value.

Operator	Description
- Unary minus	negating the values
+ Unary plus	converting a negative value to positive
++ :Increment operator	incrementing the value by 1
— : Decrement operator	decrementing the value by 1
! : Logical not operator	inverting a boolean value

// Java program to illustrate unary operators

```
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
boolean condition = true;
c = ++a;
System.out.println("Value of c (++a) = " + c);
c = b++;
System.out.println("Value of c (b++) = " + c);
c = --d;
System.out.println("Value of c (--d) = " + c);
c = --e;
System.out.println("Value of c (--e) = " + c);
System.out.println("Value of !condition =" + !condition);
}
}
```

Precedence of Java Operators

Operator precedence determines the grouping of operands in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, the following expression,

x = 10 + 5 * 2;

is evaluated. So, the output is 20, not 30. Because operator * has higher precedence than +.

The following table shows the operators with the highest precedence at the top of the table and those with the lowest at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	>() [] . (dot operator)	Left to right
Unary	>++ - - ! ~	Right to left
Multiplicative	>* /	Left to right
Additive	>+ -	Left to right
Shift	>>> >>> <<	Left to right
Relational	>> >= < <=	Left to right
Equality	>== !=	Left to right
Bitwise AND	>&	Left to right
Bitwise XOR	>^	Left to right
Bitwise OR	>|	Left to right
Logical AND	>&&	Left to right
Logical OR	>||	Left to right
Conditional	?:	Right to left
Assignment	>= += -= *= /= %= >>= <<= &= ^= |=	Right to left

1.14 CONTROL FLOW

Java Control statements control the flow of execution in a java program, based on data values and conditional logic used. There are three main categories of control flow statements;

Selection statements: if, if-else and switch.

Loop statements: while, do-while and for.

Transfer statements: break, continue, return, try-catch-finally and assert.

Selection statements

The selection statements checks the condition only once for the program execution.

If Statement:

The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program.

The simple if statement has the following syntax:

```
if (<conditional expression>)
<statement action>
```

The following program explains the if statement.

```
public class programIF{
public static void main(String[] args)
{
int a = 10, b = 20;
if (a > b)
System.out.println("a > b");
if (a < b)
System.out.println("b < a");
}
}
```

The If-else Statement

The if/else statement is an extension of the if statement. If the condition in the if statement fails, the statements in the else block are executed. The if-else statement has the following syntax:

```
if (<conditional expression>)
        <statement action>
else
        <statement action>
```

The following program explains the if-else statement.

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

```
int a = 10, b = 20;
if (a > b)
{
System.out.println("a > b");
}
else
{
System.out.println("b < a");
}
}
}
```

Switch Case Statement

The switch case statement is also called as multi-way branching statement with several choices. A switch statement is easier to implement than a series of if/else statements. The switch statement begins with a keyword, followed by an expression that equates to a no long integral value.

After the controlling expression, there is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label.

The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block. If none of the case label values match, then none of the codes within the switch statement code block will be executed.

Java includes a default label to use in cases where there are no matches. A nested switch within a case block of an outer switch is also allowed. When executing a switch statement, the flow of the program falls through to the next case. So, after every case, you must insert a break statement.

The syntax of switch case is given as follows:

```
switch (<non-long integral expression>) {
        case label1: <statement1>
        case label2: <statement2>
        ...
        case labeln: <statementn>
        default: <statement>
        } // end switch
```

The following program explains the switch statement.

```
public class ProgramSwitch
{
public static void main(String[] args)
{
int a = 10, b = 20, c = 30;
int status = -1;
if (a > b && a > c)
{
status = 1;
}
else if (b > c)
{
status = 2;
}
else
{
status = 3;
}
switch (status)
{
case 1:System.out.println("a is the greatest");
```

```
break;
case 2:System.out.println("b is the greatest");
break;
case 3:System.out.println("c is the greatest");
break;
default:System.out.println("Cannot be determined");
}
}
}
```

Iteration statements

Iteration statements execute a block of code for several numbers of times until the condition is true.

While Statement

The while statement is one of the looping constructs control statement that executes a block of code while a condition is true. The loop will stop the execution if the testing expression evaluates to false. The loop condition must be a boolean expression. The syntax of the while loop is

```
while (<loop condition>)
<statements>
```

The following program explains the while statement.

```
public class ProgramWhile
{
public static void main(String[] args)
{
int count = 1;
System.out.println("Printing Numbers from 1 to 10");
while (count <= 10)
{
System.out.println(count++);}
}
}
}
```

Do-while Loop Statement

The do-while loop is similar to the while loop, except that the test condition is performed at the end of the loop instead of at the beginning. The do—while loop executes atleast once without checking the condition.

It begins with the keyword do, followed by the statements that making up the body of the loop. Finally, the keyword while and the test expression completes the do-while loop. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

The syntax of the do-while loop is

do

<loop body>

while (<loop condition>);

The following program explains the do--while statement.

```
public class DoWhileLoopDemo {
public static void main(String[] args) {
int count = 1;
System.out.println("Printing Numbers from 1 to 10");
do {
System.out.println(count++);
} while (count <= 10);
}
}
```


For Loop

The for loop is a looping construct which can execute a set of instructions for a specified number of times. It's a counter controlled loop.

The syntax of the loop is as follows:

for (<initialization>; <loop condition>; <increment expression>)

<loop body>

- initialization statement executes once before the loop begins. The <initialization> section can also be a comma-separated list of expression statements.
- test expression. As long as the expression is true, the loop will continue. If this expression is evaluated as false the first time, the loop will never be executed.

- Increment(Update) expression that automatically executes after each repetition of the loop body.
- All the sections in the for-header are optional. Any one of them can be left empty, but the two semicolons are mandatory.

The following program explains the for statement.

```
public class ProgramFor {
public static void main(String[] args) {
System.out.println("Printing Numbers from 1 to 10");
for (int count = 1; count <= 10; count++) {
System.out.println(count);
}
}
}
```

Transfer statements

Transfer statements are used to transfer the flow of execution from one statement to another.

Continue Statement

A continue statement stops the current iteration of a loop (while, do or for) and causes execution to resume at the top of the nearest enclosing loop. The continue statement can be used when you do not want to execute the remaining statements in the loop, but you do not want to exit the loop itself.

The syntax of the continue statement is

```
continue; // the unlabeled form
continue <label>; // the labeled form
```

It is possible to use a loop with a label and then use the label in the continue statement. The label name is optional, and is usually only used when you wish to return to the outermost loop in a series of nested loops.

The following program explains the continue statement.

```
public class ProgramContinue
{
public static void main(String[] args) {
System.out.println("Odd Numbers");
```

```
for (int i = 1; i <= 10; ++i) {
if (i % 2 == 0)
continue;
System.out.println(i + "\t");
}
}
}
```

Break Statement

The break statement terminates the enclosing loop (for, while, do or switch statement). Break statement can be used when we want to jump immediately to the statement following the enclosing control structure. As continue statement, can also provide a loop with a label, and then use the label in break statement. The label name is optional, and is usually only used when you wish to terminate the outermost loop in a series of nested loops.

The Syntax for break statement is as shown below;

```
break; // the unlabeled form
break <label>; // the labeled form
```

The following program explains the break statement.

```
public class ProgramBreak {
public static void main(String[] args) {
System.out.println("Numbers 1 - 10");
for (int i = 1;; ++i) {
if (i == 11)
break;
// Rest of loop body skipped when i is even
System.out.println(i + "\t");
}
}
}
```

The transferred statements such as try-catch-finally, throw will be explained in the later chapters.

1.15 DEFINING CLASSES IN JAVA

A class is an entity that determines how an object will behave and what the object will contain. A class *is* the basic building block of an object-oriented language such as Java. It is acting as a template that describes the data and behavior associated with instances of that class.

When you instantiate a class means creating an object. The class contains set of variables and methods.

The data associated with a class or object is stored in variables; the behavior associated with a class or object is implemented with methods. A class is a blueprint from which individual objects are created.

```
class MyClass {
   // field,
   //constructor, and
   // method declarations
}
```

Example:

```
class Myclass{
   public static void main(String[] args)
   {
      System.out.println("Hello World!"); //Display the string.
   }
}
```

The keyword class begins the class definition for a class named name. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The "Hello World" application has no variables and has a single method named main.

In Java, the simplest form of a class definition is

```
class name {
   . . .
}
```

In general, class declarations can include these components, in order:

1. ***Modifiers :*** A class can be public or has default access.
2. ***Class name:*** The name should begin with a initial letter.

3. ***Superclass(if any):*** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. ***Interfaces(if any):*** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. ***Body:*** The class body surrounded by braces, { }.

1.16 CONSTRUCTORS

Every class has a constructor. If the constructor is not defined in the class, the Java compiler builds a default constructor for that class. While a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

Rules for writing Constructor

- Constructor(s) of a class must have same name as the class name in which it resides.
- A constructor in Java cannot be abstract, final, static and synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Following is an example of a constructor −

Example

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

Types of Constructors

There are two type of constructor in Java:

1. No-argument constructor:

A constructor that has no parameter is known as default constructor.

If the constructor is not defined in a class, then compiler creates default constructor (with no arguments) for the class. If we write a constructor with arguments or no-argument then compiler does not create default constructor. Default constructor provides the default values to the object like 0, null etc. depending on the type.

// Java Program to illustrate calling a no-argument constructor

```
import java.io.*;
class myclass
{
   int num;
String name;
    // this would be invoked while object of that class created.
   myclass()
   {
      System.out.println("Constructor called");
   }
}
 class myclassmain
{
   public static void main (String[] args)
   {
      // this would invoke default constructor.
      myclass m1 = new myclass();
 // Default constructor provides the default values to the object like 0, null
      System.out.println(m1.num);
      System.out.println(m1.name);
   }
}
```

2. Parameterized Constructor

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use parameterized constructor.

// Java Program to illustrate calling of parameterized constructor.

```
import java.io.*;
 class myclass
{
   // data members of the class.
   String name;
   int num;
    // contructor with arguments.
   myclass(String name, int n)
   {
      this.name = name;
      this.num = n;
  }
}
 class myclassmain{
   public static void main (String[] args)
   {
      // this would invoke parameterized constructor.
      myclass m1 = new myclass("Java", 2017);
      System.out.println("Name :" + m1.name + " num :" + m1.num);
   }
   }
```

There are no "return value" statements in constructor, but constructor returns current class instance. We can write 'return' inside a constructor.

1.17 CONSTRUCTOR OVERLOADING

Like methods, we can overload constructors for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

// Java Program to illustrate constructor overloading

```
import java.io.*;
class myclass
{
   // constructor with one argument
   myclass (String name)
   {
      System.out.println("Constructor with one " + "argument - String : " + name);
   }
  // constructor with two arguments
   myclass (String name, int id)
   {
  System.out.print("Constructor with two arguments : " +" String and Integer : " + name
+ " "+ id);
   }
    // Constructor with one argument but with different type than previous.
   myclass (long num)
   {
      System.out.println("Constructor with one argument : " +"Long : " + num);
   }
}
 class  myclassmain
{
   public static void main(String[] args)
   {
     myclass m1 = new myclass ("JAVA");
     myclass m2 = new myclass ("Python", 2017);
     myclass m3 = new myclass(3261567);
   }
}
```

Constructors are different from methods in Java

- Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
- Constructor(s) do not any return type while method(s) have the return type or **void** if does not return any value.
- Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

Creating an Object

The class provides the blueprints for objects. The objects are the instances of the class. In Java, the new keyword is used to create new objects.

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

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

1.18 METHODS IN JAVA

A method is a collection of statement that performs specific task. In Java, each method is a part of a class and they define the behavior of that class. In Java, method is a jargon used for method.

Advantages of methods

- Program development and debugging are easier
- Increases code sharing and code reusability
- Increases program readability
- It makes program modular and easy to understanding
- It shortens the program length by reducing code redundancy

Types of methods

There are two types of methods in Java programming:

- Standard library methods (built-in methods or predefined methods)
- User defined methods

Standard library methods

The standard library methods are built-in methods in Java programming to handle tasks such as mathematical computations, I/O processing, graphics, string handling etc. These

methods are already defined and come along with Java class libraries, organized in packages. In order to use built-in methods, we must import the corresponding packages. Some of library methods are listed below.

Packages	Library Methods	Descriptions
java.lang.Math All maths related methods are defined in this class	acos()	Computes arc cosine of the argument
	exp()	Computes the e raised to given power
	abs()	Computes absolute value of argument
	log()	Computes natural logarithm
	sqrt()	Computes square root of the argument
	pow()	Computes the number raised to given power
java.lang.String All string related methods are defined in this class	charAt()	Returns the char value at the specified index.
	concat()	Concatenates two string
	compareTo()	Compares two string
	indexOf()	Returns the index of the first occurrence of the given character
	toUpperCase()	converts all of the characters in the String to upper case
java.awt contains classes for graphics	add()	inserts a component
	setSize()	set the size of the component
	setLayout()	defines the layout manager
	setVisible()	changes the visibility of the component

Example:

Program to compute square root of a given number using built-in method.

```
public class MathEx {
   public static void main(String[] args) {
      System.out.print("Square root of 14 is: " + Math.sqrt(14));
   }
}
```

Sample Output:

```
Square root of 14 is: 3.7416573867739413
```

User-defined methods

The methods created by user are called user defined methods.

Every method has the following.

- Method declaration (also called as method signature or method prototype)
- Method definition (body of the method)
- Method call (invoke/activate the method)

Method Declaration

The syntax of method declaration is:

Syntax:

return_type method_name(parameter_list);

Here, the return_type specifies the data type of the value returned by method. It will be void if the method returns nothing. method_name indicates the unique name assigned to the method. parameter_list specifies the list of values accepted by the method.

Method Definition

Method definition provides the actual body of the method. The instructions to complete a specific task are written in method definition. The syntax of method is as follows:

Syntax:

```
modifier return_type method_name(parameter_list){
// body of the method
}
```

Here,

Modifier	–	Defines the access type of the method i.e accessibility region of method in the application
return_type	–	Data type of the value returned by the method or void if method returns nothing
method_name	–	Unique name to identify the method. The name must follow the rules of identifier
parameter_list	–	List of input parameters separated by comma. It must be like datatype parameter1,datatype parameter2,…… List will be empty () in case of no input parameters.
method body	–	block of code enclosed within { and } braces to perform specific task

The first line of the method definition must match exactly with the method prototype. A method cannot be defined inside another method.

Method Call

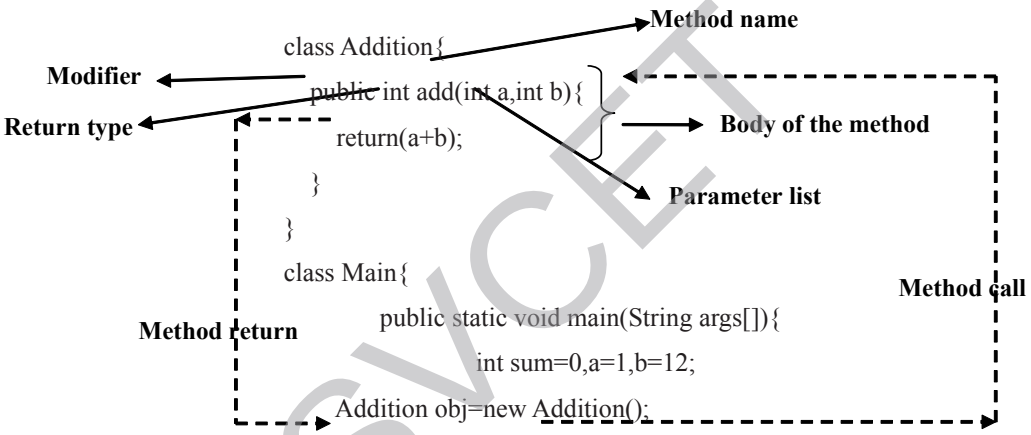
A method gets executed only when it is called. The syntax for method call is.

Syntax:

method_name(parameters);

When a method is called, the program control transfers to the method definition where the actual code gets executed and returns back to the calling point. The number and type of parameters passed in method call should match exactly with the parameter list mentioned in method prototype.

Example:




```
class Addition{
    public int add(int a,int b){
        return(a+b);
    }
}
class Main{
    public static void main(String args[]){
        int sum=0,a=1,b=12;
        Addition obj=new Addition();
        sum=obj.add(a,b);
        System.out.println("Sum:"+sum);
    }
}
```

Sample Output:

Sum:13

Memory allocation for methods calls

Method calls are implemented using stack. When a method is called, the parameters passed in the call, local variables defined inside method, and return value of the method are stored in stack frame. The allocated stack frame gets deleted automatically at the end of method execution.

Types of User-defined methods

The methods in C are classified based on data flow between calling method and called method. They are:

- Method with no arguments and no return value
- Method with no arguments and a return value
- Method with arguments and no return value
- Method with arguments and a return value.

Method with no arguments and no return value

In this type of method, no value is passed in between calling method and called method. Here, when the method is called program control transfers to the called method, executes the method, and return back to the calling method.

Example:

Program to compute addition of two numbers (no argument and no return value)

```
public class Main{
   public void add(){          //  method definition with no arguments and no return value
      int a=10,b=20;
      System.out.println("Sum:"+(a+b));
   }
   public static void main(String[] args) {
      Main obj=new Main();
      obj.add();                //  method call with no arguments
   }
}
```

Sample Output:

Sum:30

Method with no arguments and a return value

In this type of method, no value is passed from calling method to called method but a value is returned from called method to calling method.

Example:

Program to compute addition of two numbers (no argument and with return value)

```
public class Main {
   public int add(){   // method definition with no arguments and with return value
      int a=10,b=20;
      return(a+b);
   }
   public static void main(String[] args) {
      int sum=0;
      Main obj=new Main();
      sum=obj.add();   /* method call with no arguments. The value returned from the method is assigned to variable sum */
      System.out.println("Sum:"+sum);
   }
}
```

Sample Output:

```
Sum:30
```

Method with arguments and no return value

In this type of method, parameters are passed from calling method to called method but no value is returned from called method to calling method.

Example:

Program to compute addition of two numbers (with argument and without return value)

```
public class Main {
   public void add(int x,int y){   // method definition with arguments and no return value
      System.out.println("Sum:"+(x+y));
   }
   public static void main(String[] args) {
      int a=10,b=20;
      Main obj=new Main();
      obj.add(a,b);   // method call with arguments
   }
}
```

Sample Output:

```
Sum:30
```

Method with arguments and a return value.

In this type of method, there is data transfer in between calling method and called method. Here, when the method is called program control transfers to the called method with arguments, executes the method, and return the value back to the calling method.

Example:

Program to compute addition of two numbers (with argument and return value)

```
public class Main {
   public int add(int x,int y){       // function definition with arguments and return value
      return(x+y);    //return value
   }
   public static void main(String[] args) {
      int a=10,b=20;
      Main obj=new Main();                    /* method call with arguments. The value returned from
                                                 the method is displayed within main() */
      System.out.println("Sum:"+obj.add(a,b));
   }
}
```

Sample Output:

```
Sum:30
```

1.19 PARAMETER PASSING IN JAVA

The commonly available parameter passing methods are:

- Pass by value
- Pass by reference

Pass by Value

In pass by value, the value passed to the method is copied into the local parameter and any change made inside the method only affects the local copy has no effect on the original copy. In Java, parameters are always passed by value. All the scalar variables (of type int, long, short, float, double, byte, char, Boolean) are always passed to the methods by value. Only the non-scalar variables like Object, Array, String are passed by reference.

Note:

Scalar variables are singular data with one value; Non scalar variables are data with multiple values.

Example:

```
Pass by value
class Swapper{
int a;
int b;
Swapper(int x, int y)  // constructor to initialize variables
{
a = x;
b = y;
}
void swap(int x, int y) // method to interchange values
{
        int temp;
        temp = x;
        x=y;
        y=temp;
    }
}
class Main{
    public static void main(String[] args){
        Swapper obj = new Swapper(10, 20);  // create object
        System.out.println("Before swapping: a="+obj.a+" b="+obj.b);
        obj.swap(obj.a,obj.b); // call the method by passing class object as parameter
        System.out.println("Before swapping: a="+obj.a+" b="+obj.b);
    }
}
```

/* only the local copy *x*, *y* gets swapped. The original object value *a*, *b* remains unchanged*/

Sample Output:

```
Before swapping: a=10 b=20
After swapping: a=10 b=20
```

Here, to call method swap() first create an object for class Swapper. Then the method is called by passing object values *a* and *b* as input parameters. As these values are scalar, the parameters are passed using pass by value technique. So the changes carried out inside the method are not reflected in original value of *a* and *b*.

Pass by Reference

In pass-by-reference, reference (address) of the actual parameters is passed to the local parameters in the method definition. So, the changes performed on local parameters are reflected on the actual parameters.

Example:

```
class Swapper{
    int a;
    int b;
    Swapper(int x, int y)  // constructor to initialize variables
    {
            a = x;
            b = y;
    }
    void swap(Swapper ref) // method to interchange values
    {
            int temp;
            temp = ref.a;
            ref.a = ref.b;
            ref.b = temp;
    }
}
class PassByRef{
    public static void main(String[] args){
            Swapper obj = new Swapper(10, 20);  // create object
            System.out.println("Before swapping: a="+obj.a+" b="+obj.b);
```

/* Object is passed by reference. So the original object value *a*, *b* gets changed*/

```
            obj.swap(obj); // call the method by passing class object as parameter
            System.out.println("After swapping: a="+obj.a+" b="+obj.b);
        }
    }
```

Sample Output:

Before swapping: a=10 b=20

After swapping: a=20 b=10

In this example, the class object is passed as parameter using pass by reference technique. So the method refers the original value of *a* and *b*.

Method using object as parameter and returning objects

A method can have object as input parameter (*see* pass by reference) and can return a class type object.

Example:

```
class Addition{
    int no;
    Addition(){}
    Addition(int x){
        no=x;
    }
    public Addition display(Addition oa){
        Addition tmp=new Addition();
        tmp.no=no+oa.no;
        return(tmp);
    }
}
class Main{
    public static void main(String args[]){
        Addition a1=new Addition(10);
        Addition a2=new Addition(10);
        Addition a3;
      a3=a1.display(a2);  // method is invoked using the object a1 with input parameter a2
```

/*method with same class object as input parameter & return value*/

```
        System.out.println("a1.no="+a1.no+" a2.no="+a2.no+" a3.no="+a3.no);
    }
}
```

Sample Output:

a1.no=10 a2.no=10 a3.no=20

Here, display() accepts class Addition object a2 as input parameter. It also return same class object as output. This method adds the value of invoking object a1 and input parameter a2. The summation result is stored in temporary object tmp inside the method. The value returned by the method is received using object a3 inside main().

1.20 METHOD OVERLOADING

Method overloading is the process of having multiple methods with same name that differs in parameter list. The number and the data type of parameters must differ in overloaded methods. It is one of the ways to implement polymorphism in Java. When a method is called, the overloaded method whose parameters match with the arguments in the call gets invoked.

Note: Overloaded methods are differentiable only based on parameter list and not on their return type.

Example:

Program for addition using Method Overloading

```
class MethodOverload{
    void add(){
        System.out.println("No parameters");
    }
    void add(int a,int b){              // overloaded add() for two integer parameter
        System.out.println("Sum:"+(a+b));
    }
    void add(int a,int b,int c){        // overloaded add() for three integer parameter
        System.out.println("Sum:"+(a+b+c));
    }
    void add(double a,double b){        // overloaded add() for two double parameter
        System.out.println("Sum:"+(a+b));
```

```
    }
}
public class Main {
    public static void main(String[] args) {
        MethodOverload obj=new MethodOverload();
        obj.add();              // call all versions of add()
        obj.add(1,2);
        obj.add(1,2,3);
        obj.add(12.3,23.4);
    }
}
```

Sample Output:

```
No parameters
Sum:3
Sum:6
Sum:35.7
```

Here, *add()* is overloaded four times. The first version takes no parameters, second takes two integers, third takes three integers and fourth accepts two double parameter.

1.21 ACCESS SPECIFIERS

Access specifiers or access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class. It determines whether a data or method in a class can be used or invoked by other class or subclass.

Types of Access Specifiers

There are 4 types of java access specifiers:

1. Private
2. Default (no speciifer)
3. Protected
4. Public

The details about accessibility level for access specifiers are shown in following table.

Access Modifiers	Default	Private	Protected	Public
Accessible inside the class	Yes	Yes	Yes	Yes
Accessible within the subclass inside the same package	Yes	No	Yes	Yes
Accessible outside the package	No	No	No	Yes
Accessible within the subclass outside the package	No	No	Yes	Yes

Private access modifier

Private data fields and methods are accessible only inside the class where it is declared i.e accessible only by same class members. It provides low level of accessibility. Encapsulation and data hiding can be achieved using private specifier.

Example:

```
Role of private specifier
class PrivateEx{
   private int x;                    // private data
   public int y;                     // public data
   private PrivateEx(){}             // private constructor
   public PrivateEx(int a,int b){    // public constructor
      x=a;
      y=b;
   }
}
public class Main {
   public static void main(String[] args) {
      PrivateEx obj1=new PrivateEx();      // Error: private constructor cannot be applied
      PrivateEx obj2=new PrivateEx(10,20);  // public constructor can be applied to obj2
      System.out.println(obj2.y);           // public data y is accessible by a non-member
      System.out.println(obj2.x);           //Error: x has private access in PrivateEx
   }
}
```

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

Default access modifier

If the specifier is mentioned, then it is treated as default. There is no default specifier keyword. Using default specifier we can access class, method, or field which belongs to same package, but not from outside this package.

Example:

Role of default specifier

```
class DefaultEx{
   int y=10; // default data
}
public class Main {
   public static void main(String[] args) {
      DefaultEx obj=new DefaultEx();
      System.out.println(obj.y);       // default data y is accessible outside the class
   }
}
```


Sample Output:

10

In the above example, the scope of class DefaultEx and its data *y* is default. So it can be accessible within the same package and cannot be accessed from outside the package.

Protected access modifier

Protected methods and fields are accessible within same class, subclass inside same package and subclass in other package (through inheritance). It cannot be applicable to class and interfaces.

Example:

```
Role of protected specifier
class Base{
   protected void show(){
      System.out.println("In Base");
   }
}
public class Main extends Base{
   public static void main(String[] args) {
      Main obj=new Main();
      obj.show();
   }
}
```

Sample Output:

```
In Base
```

In this example, *show()* of class Base is declared as protected, so it can be accessed from outside the class only through inheritance. Chapter 2 explains the concept of inheritance in detail.

Public access modifier

The public access specifier has highest level of accessibility. Methods, class, and fields declared as public are accessible by any class in the same package or in other package.

Example:

```
Role of public specifier
class PublicEx{
   public int no=10;
}
```

```
public class Main{
   public static void main(String[] args) {
      PublicEx obj=new PublicEx();
      System.out.println(obj.no);
   }
}
```

Sample Output:

10

In this example, public data *no* is accessible both by member and non-member of the class.

1.22 STATIC KEYWORD

The static keyword indicates that the member belongs to the class instead of a specific instance. It is used to create class variable and mainly used for memory management. The static keyword can be used with:

- Variable (static variable or class variable)
- Method (static method or class method)
- Block (static block)
- Nested class (static class)
- import (static import)

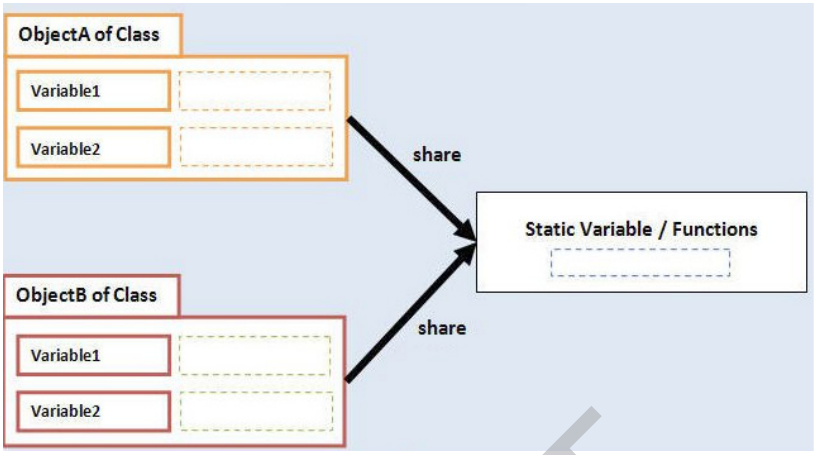
Static variable

Variable declared with keyword static is a static variable. It is a class level variable commonly shared by all objects of the class.

- Memory allocation for such variables only happens once when the class is loaded in the memory.
- scope of the static variable is class scope (accessible only inside the class)
- lifetime is global (memory is assigned till the class is removed by JVM).
- Automatically initialized to 0.
- It is accessible using ClassName.variablename
- Static variables can be accessed directly in static and non-static methods.

Example :

Without static	With static
class StaticEx{ **int no=10;** **StaticEx(){** **System.out.println(no);** **no++;** **}** **}** **public class Main{** **public static void main(String[] args)** **{** **StaticEx obj1=new StaticEx();** **StaticEx obj2=new StaticEx();** **StaticEx obj3=new StaticEx();** **}** **}** *Sample Output:* **10** **10** **10**	**class StaticEx{** **static int no=10;** **StaticEx(){** **System.out.println(no);** **no++;** **}** **}** **public class Main{** **public static void main(String[] args)** **{** **StaticEx obj1=new StaticEx();** **StaticEx obj2=new StaticEx();** **StaticEx obj3=new StaticEx();** **}** **}** *Sample Output:* **10** **11** **12**



Static Method

The method declared with static keyword is known as static method. *main()* is most common static method.

- It belongs to the class and not to object of a class.
- A static method can directly access only static variables of class and directly invoke only static methods of the class.
- Static methods cannot access non-static members(instance variables or instance methods) of the class
- Static method cant access this and super references
- It can be called through the name of class without creating any instance of that class. For example, ClassName.methodName()

Example:

```
class StaticEx{
   static int x;
   int y=10;
   static void display(){
      System.out.println("Static Method "+x);   // static method accessing static variable
   }
   public void show(){
      System.out.println("Non static method "+y);
      System.out.println("Non static method "+x); // non-static method can access static variable
```

```
    }
}
public class Main
{
    public static void main(String[] args) {
            StaticEx obj=new StaticEx();
            StaticEx.display();     // static method invoked without using object
            obj.show();
    }
}
```

Sample Output:

```
Static Method 0
Non static method 10
Non static method 0
```

In this example, class StaticEx consists of a static variable x and static method display(). The static method cannot access a non-static variable. If you try to access y inside static method display(), it will result in compilation error.

```
static void display(){          /*non-static variable y cannot be referred from a
                                static context*/
  System.out.println("Static Method "+x+y);
}
```

Static Block

A static block is a block of code enclosed in braces, preceded by the keyword *static*.

- The statements within the static block are first executed automatically before main when the class is loaded into JVM.
- A class can have any number of static blocks.
- JVM combines all the static blocks in a class as single block and executes them.
- Static methods can be invoked from the static block and they will be executed as and when the static block gets executed.

Syntax:

```
static{
```

…….

}

Example:

```
class StaticBlockEx{
   StaticBlockEx (){
      System.out.println("Constructor");
   }
   static {
      System.out.println("First static block");
   }
   static void show(){
      System.out.println("Inside method");
   }
   static{
      System.out.println("Second static block");
      show();
   }

   public static void main(String[] args) {
            StaticBlockEx obj=new StaticBlockEx ();
    }
    static{
       System.out.println("Static in main");
    }
}
```

Sample Output:

```
First static block
Second static block
Inside method
```

Static in main

Constructor

Nested class (static class)

Nested class is a class declared inside another class. The inner class must be a static class declared using keyword static. The static nested class can refer directly to static members of the enclosing classes, even if those members are private.

Syntax:

```
class OuterClass{
        ……..
        static class InnerClass{
                ……..
        }
}
```

We can create object for static nested class directly without creating object for outer class. For example:

OuterClassName.InnerClassName=new OuterClassName.InnerClassName();

Example:

```
class Outer{
   static int x=10;
   static class Inner{
      int y=20;
      public void show(){
         System.out.println(x+y);          // nested class accessing its own data & outer class static data
      }
   }
}
class Main{
   public static void main(String args[]){
      Outer.Inner obj=new Outer.Inner();  // Creating object for static nested class
      obj.show();
   }
}
```

Sample Output:

30

Static Import

The static import allows the programmer to access any static members of imported class directly. There is no need to qualify it by its name.

Syntax:

Import static package_name;

Advantage:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage:

- Overuse of static import makes program unreadable and unmaintable.

Example:

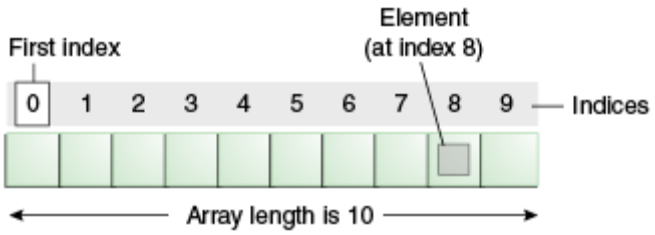
```
import static java.lang.System.*;
class StaticImportEx{
  public static void main(String args[]){
   out.println("Static Import Example");          //Now no need of System.out
}
}
```

Sample Output:

Static Import Example

1.23 ARRAYS

Array is a collection of elements of similar data type stored in contiguous memory location. The array size is fixed i.e we can't increase/decrease its size at runtime. It is index based and the first element is stored at 0^{th} index.



Advantages of Array

- Code Optimization: Multiple values can be stored under common name. Date retrieval or sorting is an easy process.
- Random access: Data at any location can be retrieved randomly using the index.

Disadvantages of Array

- Inefficient memory usage: Array is static. It is not resizable at runtime based on number of user's input. To overcome this limitation, Java introduce collection concept.

Types of Array

There are two types of array.

- One Dimensional Array
- Multidimensional Array

One Dimensional Array

Declaring Array Variables

The syntax for declaring an array variable is

Syntax:

dataType[] arrayName; //preferred way

Or

dataType arrayName [];

Here datatype can be a primitive data type like: int, char, Double, byte etc. arrayName is an identifier.

Example:

int[] a;

Instantiation of an Array

Array can be created using the *new* keyword. To allocate memory for array elements we must mention the array size. The size of an array must be specified by an *int* value and not *long* or *short*. The default initial value of elements of an array is 0 for numeric types and *false* for boolean.

Syntax:

arrayName=new datatype[size];

Or

dataType[] arrayName=new datatype[size]; //declaration and instantiation

Example:

```
int[] a=new int[5];       //defining an integer array for 5 elements
```

Alternatively, we can create and initialize array using following syntax.

Syntax:

```
dataType[] arrayName=new datatype[]{list of values separated by comma};
                    Or
dataType[] arrayName={ list of values separated by comma};
```

Example:

```
int[] a={12,13,14};
int[] a=new int[]{12,13,14};
```

The built-in *length* property is used to determine length of the array i.e. number of elements present in an array.

Accessing array elements

The array elements can be accessed by using indices. The index starts from 0 and ends at (array size-1). Each element in an array can be accessed using *for* loop.

Example:

Program to access array elements.

```
class Main{
   public static void main(String args[]){
      int a[]=new int[]{10,20,30,40};//declaration and initialization
      //printing array
      for(int i=0;i<a.length;i++)//length is the property of array
         System.out.println(a[i]);
   }
}
```

Sample Output:

```
10
20
30
40
```

The for-each loop

The for-each loop is used to traverse the complete array sequentially without using an index variable. It's commonly used to iterate over an array or a Collections class (eg, Array-List).

Syntax:

```
for(type var:arrayName){
    Statements using var;
}
```

Example:

Program to calculate sum of array elements.

```
class Main{
  public static void main(String args[]){
    int a[]=new int[]{10,20,30,40};//declaration and initialization
    int sum=0;
    for(int i:a)      // calculate sum of array elements
      sum+=i;
    System.out.println("Sum:"+sum);
  }
}
```

Sample Output:

```
Sum:100
```

Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays.

Syntax:

```
dataType[][] arrayName=new datatype[rowsize][columnnsize];     // 2 dimensional array
dataType[][][] arrayName=new datatype[][][];                   // 3 dimensional array
```

Example:

```
int[][] a=new int[3][4];
```

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Example:

Program to access 2D array elements

```
class TwoDimEx
{
   public static void main(String args[])
   {
      // declaring and initializing 2D array
      int arr[][] = { {1,1,12},{2,16,1},{12,42,2} };
       // printing 2D array
      for (int i=0; i< arr.length; i++)
      {
         for (int j=0; j < arr[i].length ; j++)
            System.out.print(arr[i][j] + " ");
          System.out.println();
      }
   }
}
```

Sample Output:

```
1 1 12
2 16 1
12 42 2
```

Jagged Array

Jagged array is an array of arrays with different row size i.e. with different dimensions.

Example:

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

    int[][] a = {
        {11, 3, 43},
        {3, 5, 8, 1},
        {9},
    };

    System.out.println("Length of row 1: " + a[0].length);
    System.out.println("Length of row 2: " + a[1].length);
    System.out.println("Length of row 3: " + a[2].length);
  }
}
```

Sample Output:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

Passing an array to a method

An array can be passed as parameter to method.

Example:

Program to find minimum element in an array

```
class Main{
   static void min(int a[]){
      int min=a[0];
      for(int i=1;i<a.length;i++)
         if(min>a[i])
            min=a[i];
```

```
        System.out.println("Minimum:"+min);
    }
    public static void main(String args[]){
        int a[]={12,13,14,5};
        min(a);//passing array to method
    }
}
```

Sample Output:

Minimum:5

Returning an array from a method

A method may also return an array.

Example:

Program to sort array elements in ascending order.

```
class Main{
    static int[] sortArray(int a[]){
        int tmp;
        for(int i=0;i<a.length-1;i++)  {          //code for sorting
            for(int j=i+1;j<=a.length-1;j++)  {
                if(a[i]>a[j]){
                    tmp=a[i];
                    a[i]=a[j];
                    a[j]=tmp;
                }
            }
        }
        return(a);        // returning array
    }
    public static void main(String args[]){
        int a[]={33,43,24,5};
        a=sortArray(a);//passing array to method
```