By Smart-Programming

Point-to-point

About --------------------------------------------------------

Table of Contents

[Introduction to java 7](#_Toc132500131)

[Edition of Java 7](#_Toc132500132)

[Application 8](#_Toc132500133)

[Versions of java 8](#_Toc132500134)

[What is difference between C, C++ & Java 8](#_Toc132500135)

[Python vs Java 9](#_Toc132500136)

[Features of Java 10](#_Toc132500137)

[Naming Conventions in java 11](#_Toc132500138)

[Structure of java program 11](#_Toc132500139)

[1. Java Documentation Section 12](#_Toc132500140)

[2. Package Statement 12](#_Toc132500141)

[3. Import Statement 12](#_Toc132500142)

[4. Interface/Class Section **(Variables & Methods)** 13](#_Toc132500143)

[5. Main Class Section **(main method)** 13](#_Toc132500144)

[Preparing First java program 13](#_Toc132500145)

[Steps to prepare Java First Program 13](#_Toc132500146)

[Compile & Run Java Program :- 14](#_Toc132500147)

[Test cases of main method 14](#_Toc132500148)

[Deep explanation of javA JDK/ JRE/ JVM 15](#_Toc132500149)

[JVM (java virtual machine) 15](#_Toc132500150)

[JRE (java runtime environment) 15](#_Toc132500151)

[JDK (Java development kit) 16](#_Toc132500152)

[JVM (Java Virtual Machine) (detailed description) 17](#_Toc132500153)

[JVM Explanation diagram 18](#_Toc132500154)

[Working description of JVM Parts 18](#_Toc132500155)

[Class loader 19](#_Toc132500156)

[Class Loader Architecture 19](#_Toc132500157)

[Memory Areas 20](#_Toc132500158)

[Execution Engine 21](#_Toc132500159)

[Java Native Interface (JNI) 22](#_Toc132500160)

[Interview Questions 22](#_Toc132500161)

Type chapter title (level 3)3

Type chapter title (level 1)4

Type chapter title (level 2)5

Type chapter title (level 3)6

Chapter1

Java Introduction

# Introduction to java

J

ava is a class-based, object-oriented, high level, programming language that is designed to have as few implementation dependencies as possible. It is intended to let application developers **write once, and run anywhere (WORA),** meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java was first released in 1995 and is widely used for developing applications for desktop, web, and mobile devices. Java is known for its simplicity, robustness, and security features, making it a popular choice for enterprise-level applications.

**Some important points about java…**

1. **JAVA** was developed by **James Gosling** at **Sun Microsystems.**
2. It was developed in year (**1991 Development) (1995 Release)** and later acquired by **(Oracle Corporation)** symbol of language is **(Coffee Cup)**.
3. Before **Java,** its name was ***Oak***. Since Oak was already a registered company, so James Gosling and his team changed the name from **Oak to Java**.
4. Java makes writing, compiling, and debugging programming easy.
5. It helps to create reusable code and modular programs.
6. Java is a class-based, object-oriented, high level, programming language
7. general-purpose programming language made for developers to *write once run anywhere*
8. Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine.
9. The syntax of Java is mostly derived from c/c++ language.

# Edition of Java

**1. J2SE : -> Java 2 Standard Edition (Core Java)**

In this edition we learn java fundamentals

Projects : Standalone Applications

**2. J2EE : -> Java 2 Enterprise Edition (Advance Java)**

In this edition we learn server side programming

Projects : Enterprise Applications

**3. J2ME : -> Java 2 Micro Edition (it has been replaced by Android)**

In this edition we learn micro programming

Projects : Mobile Based Applications

# Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

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

# Versions of java

|  |  |  |
| --- | --- | --- |
| **Java Version Name**  **JDK 1.0** (Oak)  **JDK 1.1**  **J2SE 1.2** (Playground)  **J2SE 1.3** (Kestrel)  **J2SE 1.4** (Merlin)  **J2SE 5.0** (Tiger)  **Java SE 6** (Mustang)  **Java SE 7** (Dolphin)  **Java SE 8**  **Java SE 9**  **Java SE 10**  **Java SE 11**  **Java SE 12**  **Java SE 13**  **Java SE 14**  **Java SE 15**  **Java SE 16**  **Java SE 17**  **Java SE 18**  **Java SE 19** | ***Java Versions***  **1.0**  **1.1**  **1.2**  **1.3**  **1.4**  **1.5**  **1.6**  **1.7**  **1.8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19** | ***Release Date***  **January 1996**  **February 1997**  **December 1998**  **May 2000**  **February 2002**  **September 2004**  **December 2006**  **July 2011**  **March 2014**  **September, 21st 2017**  **March, 20th 2018**  **September, 25th 2018**  **March, 19th 2019**  **September, 17th 2019**  **March, 17th 2020**  **September, 15th 2020**  **March, 16th 2021**  **September, 14th 2021**  **March, 22nd 2022**  **September, 20th 2022** |

# What is difference between C, C++ & Java

|  |  |  |  |
| --- | --- | --- | --- |
| Language | C | C++ | Java |
| **Programming paradigm** | Procedural language | OOP (Object Oriented Programming) | Pure OOP (Object Oriented Programming) – 99% |
| **Origin** | Based on Assembly Language | Based on C Language | Based on C & C++ |
| **Designed by/year** | Dennis Ritchie / 1972 | Bjarne Stroustrup / 1979 | James Gosling / 1995 |
| **Translator** | Compiler | Compiler | Interpreter & Compiler |
| **File generated** | .exe file | .exe file | .class file |
| **Static/dynamic programming language** | Static Programming Language | Static Programming Language | Dynamic Programming Language |
| **Pre-processor directives** | Support header files (#include, #define) | Support header files (#include, #define) | Use packages (import) |
| **Platform dependent/independent** | Platform Dependent | Platform Dependent | Platform Independent |
| **Pointers** | Support pointers | Support pointers | Does not support pointers |
| **Inheritance** | No inheritance | Supports inheritance | Supports inheritance but not multiple inheritance |
| **Overloading** | Does not support overloading | Supports overloading | Supports overloading but not operator overloading |
| **Destructors** | Does not support constructor and destructor | Supports destructor | Does not support destructor |
| **Call by value / call by reference** | Supports both | Supports both | Support only Call By Value |

# Python vs Java

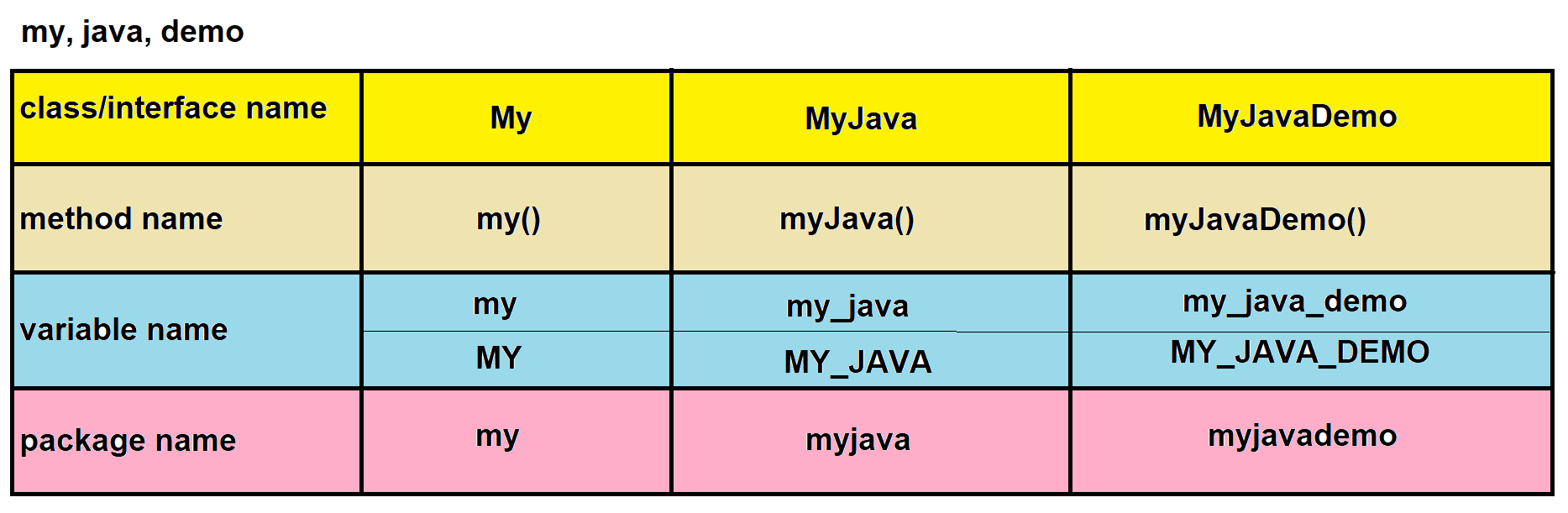
|  |  |  |
| --- | --- | --- |
| ****Python**** | | ****Java**** |
| 1. Python is Interpreted programming language, in which program can be run using a Python interpreter. | | Java is a **compiled language**. The compiled code is converted to **bytecode** and can be run on any platform that has **Java Virtual Machine (JVM).** |
| 1. Python supports imperative, object-oriented, functional and procedural paradigms. | | Java was developed for (Class-based and Object-oriented paradigms) (from **Java SE 8** version Java introduced some procedural oriented programming paradigm features like **Lamba Expression, functional interface, Functions, Predicates, Module System (java9**) etc). |
| 1. Python keeps the code concise, clean and readable. | | **More effort and time is nee**ded for the developers to keep the code readable in Java. |
| 1. Python is a dynamically typed language and doesn’t require the developer to declare variables. It allows the interpreter to detect and change the data type of the variable, need not to declare the variables.   Ex :- int variable\_1 = 100; (✕)  Variable\_2 = “Smart Programming”; (✓) | | Java is a **strongly/statically typed language**. it doesn’t allow the compiler to change the data type of the variable unless they are type-casted.  Ex :- int variable\_1 = 100; (✓)  Variable\_2 = “Smart Programming”; (✕) |
| 1. Python is slower, as it is interpretive language and the types are assumed during the runtime. | | Java is faster than Python as all the types are assigned during compilation. java support **(compilative and interpretive)**. |
| 1. Python only used interpreter for converting program code into machine code. | | Java support **compiler and interpreter**. it also support special type of inbuilt compiler called **(JIT Just in Time Compiler).** |
| 1. Python is less portable compared to java. | | As java supports more hardware’s like computers, mobiles, remotes, tv etc thus java is more portable. |
| 1. Python support multiple, multilevel, hierarchal, multilevel and Hybrid Inheritance. | | Java support **multiple, multilevel, Hierarchal** inheritance only. |
| 1. Python not needed (; Semicolon) to terminate the statement. | | Java need **(; Semicolon)** to terminate the statement. |
| 1. Python not needed to use { } rather then it uses Indentation. | | Java mandatory to use **{ }** |
| 1. Python uses easy syntax (easy to remember as it is close to humans). | | Java uses **hardcoded syntax** (if you miss any ; or { } then it will provide an error). |
| 1. Python has less number of line of code. | | Java has **more no of line of code.** |
| 1. Python It is less compatible with database and due to this reason python is rarely used in distributed (enterprise) applications. | Java has **more compatibility with database** as compared to python **(JDBC - Java Database Connectivity)** | |
| 1. Python provides less framework as compared to java for eg Django, Flask, etc. | Java provides **more frameworks** for eg Spring, Hibernate, JSF, Struts etc | |
| 1. Python use Tensorflow, Pythorch etc | Java use **Weka, Mallet, Deeplearning4j, MOA etc** | |
| 1. Python is best for Scientific & Numeric Computing, Machine Learning, AI etc | Java is best for **Desktop GUI Applications, Enterprise Applications, Mobile Applications, Embedded Systems etc** | |

# Features of Java

|  |
| --- |
| **1.Simple:-**   1. Java is using all the simplified syntax from C & C++. 2. Some C & C++ features were either removed (pointers, multiple inheritance) from java or was created automatic (memory management). 3. Java has less execution time period and less maintenance (less memory, less power). |
| **2.Platform Independent:-**   1. Java is a platform independent language which means java can be executed on any machine or operating system. |
| **3.Portable**   1. Java is portable language as it can be executed on multiple hardware’s. |
| **4.OOP (Object Oriented Programming) :-**   1. Java is OOP language thus java works around objects. 2. Java supports many OOP features like classes, objects and methods, inheritance, polymorphism, abstraction, encapsulation etc. |
| **5. Security:-**  Java is very secured language because   1. Java does not have pointers 2. Java has byte code verifier 3. Java has a security manager that defines the access for java classes |
| **6. Robust:-**  Java is Robust i.e. powerful language because   1. Java provides a lot of classes and interfaces to handle the exceptions 2. Automatic Memory Management. |
| **7. High Performance:-**  Java has high performance because  *Robust, Portable, Multithreaded Language*, etc. |
| 8. Distributed Applications:-   1. Java can be used for distributed application (enterprise applications). |
| 9. Java is compilative and interpretive language   1. Java use both compiler and interpreter for compiling source code to native code. |

# Naming Conventions in java

java is case sensitive language which means that declaring upper case and low3er case name are different **(int a=10; int A=10;)** both are different - different case.



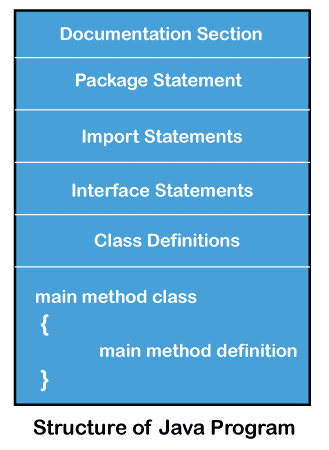
# Structure of java program

Let's see which elements are included in the structure of a [Java program](https://www.javatpoint.com/java-programs). A typical structure of a [Java](https://www.javatpoint.com/java-tutorial) program contains the following elements:

Suggested

Optional

Essential



* Documentation Section
* Package Declaration
* Import Statements
* Interface Section
* Class Definition
* Class Variables and Variables
* Main Method Class
* Methods and Behaviours

### 1. Java Documentation Section

1. It contains the basic information about java program for eg. author's name, date of creation, version, program name, company name, client name, description etc.
2. To create documentation section we use comments
3. There are 3 types of comments :-
   1. Single Line Comment (//)
   2. Multi-Line Comment (/\* ----- \*/)
   3. Documentation Comment (/\*\*-\*-\*-\*-\*-\*/)
4. To provide description or metadata of the program Java introduced new feature in JDK 1.5 version i.e. "Annotation" (@---)
5. If we use comments for documentation section then at compilation phase these comments will be removed by the compiler which cannot be read further, so to resolve this problem java introduced annotation which cannot be removed at compilation or runtime.
6. If you want to provide documentation section in html form, then we can use following command in CMD :-
7. javadoc FileName.java
   1. javap -version (to get JDK version)
   2. javap packagename.ClassName (to get the class details)

### 2. Package Statement

It is a group of similar type of classes or interfaces or packages

**Advantage of package :-**

1. It prevents naming conflicts of classes or interfaces

2. Easy to maintain

3. Accessibility to classes or interfaces can be controlled (Security)

4. Reusability

Types of packages :-

**1**. Predefined Package (which are already created by java for eg. lang, awt, swing, util, sql etc)

**2**. User Defined Package (which are created by the developer)

Syntax : package package\_name;

Example :-

package techdept.aa.bb;

class Employee

{ }

**Conditions for package :-**

1. Package statement should be only one in the single java file.

* + 1. Package statement should be the first statement in java file.
    2. Java provides one facility for naming convention for package that is we can create package on the same name of our domain name but in reverse order. For e.x com.smartprogramming.usajohn.techdept

com.smartprogramming.usajohn.managingdept

* + 1. This is OPTIONAL but you should always put all the classes or interfaces in any one package

### 3. Import Statement

Import statements is used to make available the classes or interfaces which we are going to use in present java file

**Syntax :**

1. import packagename.subpackagename.ClassName;

2. import packagename.\*;

1. if we use "import packagename.\*;" only classes or interfaces will be imported, not the subpackage.
2. If we don’t want to use import statement then we can directly write the packagename.ClassName to use that class.

3. This is **OPTIONAL** to use but mainly we use it in major level projects.

4. This line is written after the package line.

5. java.lang package is the only package which is imported by default

### 4. Interface/Class Section **(Variables & Methods)**

In this section we create classes or interfaces which contains variables, methods, constructors, blocks etc.

### 5. Main Class Section **(main method)**

* 1. In every project or java file there should be one main class which should contain main() method from which the execution of the program will start.

Syntax : // Basic windows hello Program

class Universe

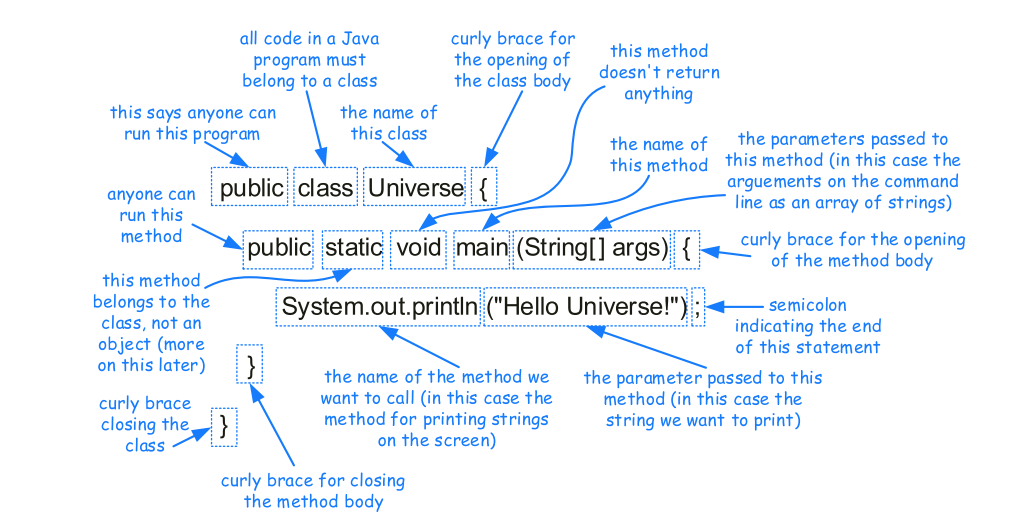
{

public static void main(String[] args){

System.out.println(“Welcome To Java”);

}}

Note - Full detailed description of java main method



# Preparing First java program

### Steps to prepare Java First Program

1. Download & Install Java Software’s

(JDK, Editor, IDE's)

2. Create Java First Program

* 1. Java Naming Conventions
  2. Structure of Java Program
  3. Java Program Explanation

3. Compile & Run Java Program

### Compile & Run Java Program

* 1. Open CMD.
  2. Type command **javac** and press enter (this step is to check whether java path is set or not) - if it says that "**'javac'** is not recognized as an internal or external command, operable program or batch file." that means java path is not set.
  3. If path is not set, then we have to set path and to set java path we have 2 options :-
     + - 1. **Temporary path set** (adding full path in Command Line)
         2. **Permanent path set** (Adding full path in **Windows Environment Variables**)
  4. Open the directory in CMD in which you have saved the java program.
  5. Compile the program (javac FileName.java).
  6. Run the program (java MainClassName).

### Test cases of main method

**1. If file name and class name is not same**

1. class file will be created on the name of class that we have created in java file.
2. Number of .class files that will be created depends on the number of classes that we have created in java file ex.(if we create 5 java class in a single java file then after compilation you will get 5 “**className**”.class file.
3. If the class is public then we have to save the file name as that class name only, but if there is no public class, then we can save the file name by any name.
4. We cannot create more than one public class in a single java file.

**2. main() method cases :-**

1. public static void main(String[] args)
2. public static void main(String[ ] args)
3. static public void main(String[] args)
4. public static void main(String []args)

Normal main Method with some sequence changes

1. public static void main(String [] args)
2. public static void main(String args[])
3. public static void main(String[] xyz)

Correct but 3 dots must have to write down

1. public static void main(String... args)
2. final public static void main(String[] args)
3. strictfp public static void main(String[] args)
4. synchronized public static void main(String[] args)

Synchronized main method

1. synchronized final public static void main(String[] args)
2. public static void main(Integer[] args)
3. public static void args(String[] args)

Correct syntax for method but

incorrect syntax for main() method

1. public static void xyz(String[] args)
2. public static void main(String args)
3. protected/private static void main(String[] args)

Error : Main method is not public in class Test

1. static void main(String[] args)

Error : Main method is not static in class Test

1. public void main(String[] args)

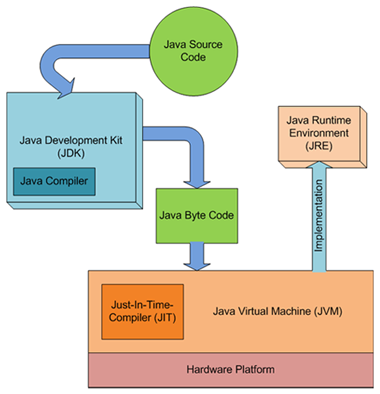
Incorrect & Compile time error because method syntax is incorrect as java needs always method name after return type

public void static main(String[] args)

public static void main(String args() {})

# Deep explanation of javA JDK/ JRE/ JVM

This is the simplest working Diagram of JDK (Java Development kit)



## JVM (java virtual machine)

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each [OS](https://www.javatpoint.com/os-tutorial) is different from each other. However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

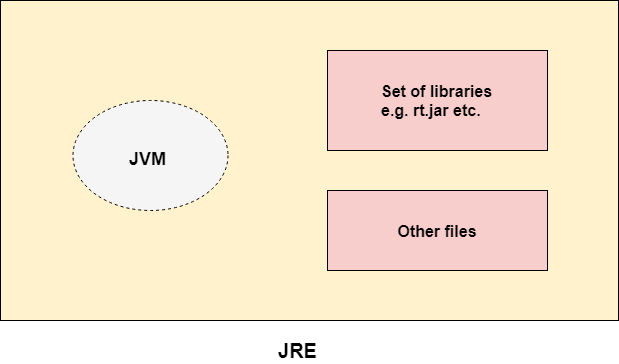
The JVM performs the following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

## JRE (java runtime environment)

JRE is an **acronym for Java Runtime Environment**. It is also written as Java **RTE**. The **Java Runtime Environment** is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



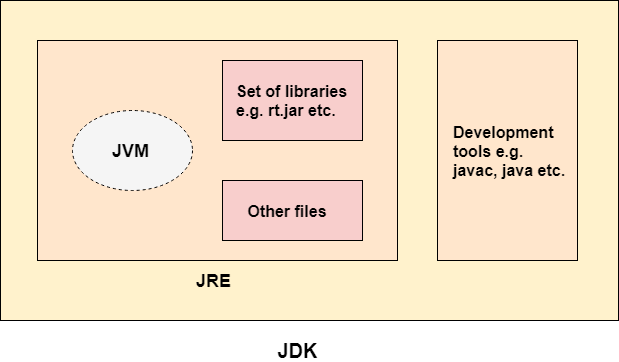
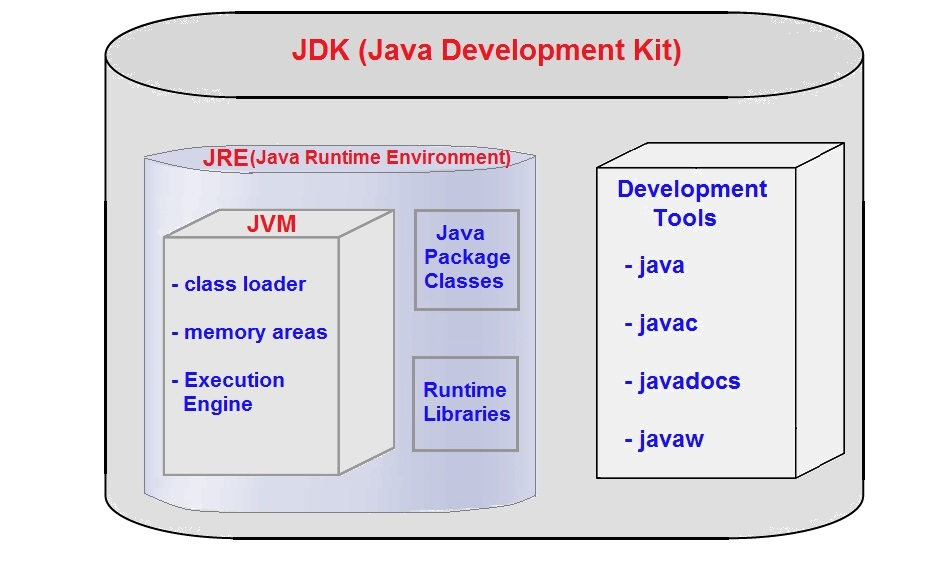
## JDK (Java development kit)

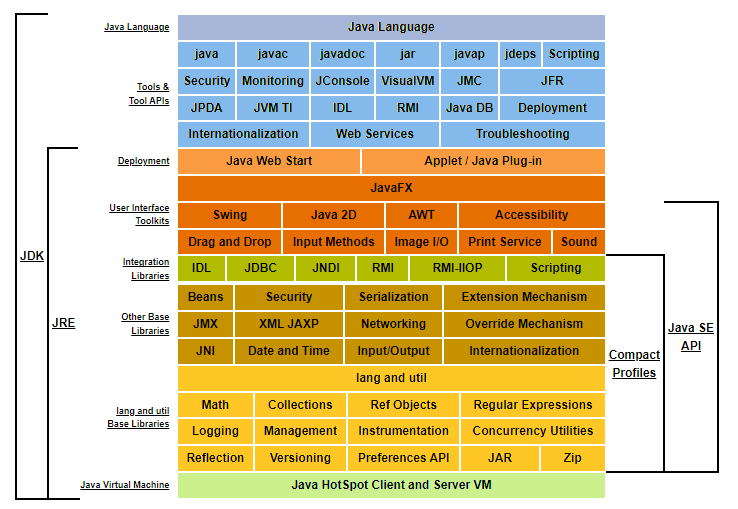
JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and [applets](https://www.javatpoint.com/java-applet). It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

Oracle has two products that implement Java Platform Standard Edition (Java SE) 8: Java SE Development Kit (JDK) 8 and Java SE Runtime Environment (JRE) 8. JDK 8 is a **superset of JRE 8**, and contains everything that is in JRE 8, plus tools such as the compilers and debuggers necessary for developing applets and applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language. Note that the JRE includes components not required by the Java SE specification, including both standard and on-standard\_Java\_components.

# JVM (Java Virtual Machine) (detailed description)

**What is Virtual Machine ?**

1. Before going for JVM, we should have knowledge of "Virtual Machine". A virtual machine is a software simulation of a machine which can perform operations similar to physical machine. Virtual machine is not physically present it’s a type of software which mimic like a full-fledged computer inside a computer.
2. A virtual machine usually known as a guest is created within another computing environment refereed as "**host**". Multiple virtual machine can exists within single host at one time.
3. For example "**Calculator**", we can buy physical calculator from market having some weight, size etc, this machine is physically exists. But the calculator present in our operating system, that does not physically present but it can perform all the tasks similar to that calculator we buy from market. SO calculator present in operating system is virtual machine which is not physically present but can perform all the tasks.

**Types of virtual machine**

1. Hardware Based or System Based Virtual Machine
2. Application Based or Process Based Virtual Machine
3. JVM(Java Virtual Machine) is Application based virtual machine.

### JVM (Java Virtual Machine)

1. JVM is an abstract machine.
2. It provides runtime environment (or say virtual machine) in which java bytecode can be executed.
3. It is called a virtual machine because it doesn't physically exist.
4. JVM is available for many hardware and software platforms.
5. JVM is platform dependent because configuration of each OS different and this makes java platform independent.
6. It can also run those programs which are written in other languages and compiled to Java bytecode.
7. It is a specification that provides a runtime environment in which Java bytecode can be executed. JVMs are available for many hardware and software platforms.
8. However, Java is platform independent.
9. There are three notions of the ***JVM:*specification*,*implementation*, and*instance***.*

**Specification -** A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.

**Implementation -** An implementation Its implementation is known as JRE (Java Runtime Environment).

**Instance -** Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

**JVM performs following major tasks**

1. Loads Java Class
2. Verifies java code
3. Allocates memory to variables (*local, instance and static variables*), current running methods etc
4. Executes the code
5. Provide runtime environment

**JVM provides definitions for the:**

1. Memory area
2. Class file format
3. Register set
4. Garbage-collected heap
5. Fatal error reporting etc.

Detailed JVM Diagram

Java Class File

Java Bytecode Stream

Javac text.java



Compiler

class test {

public static void main(String [] args) {

} }

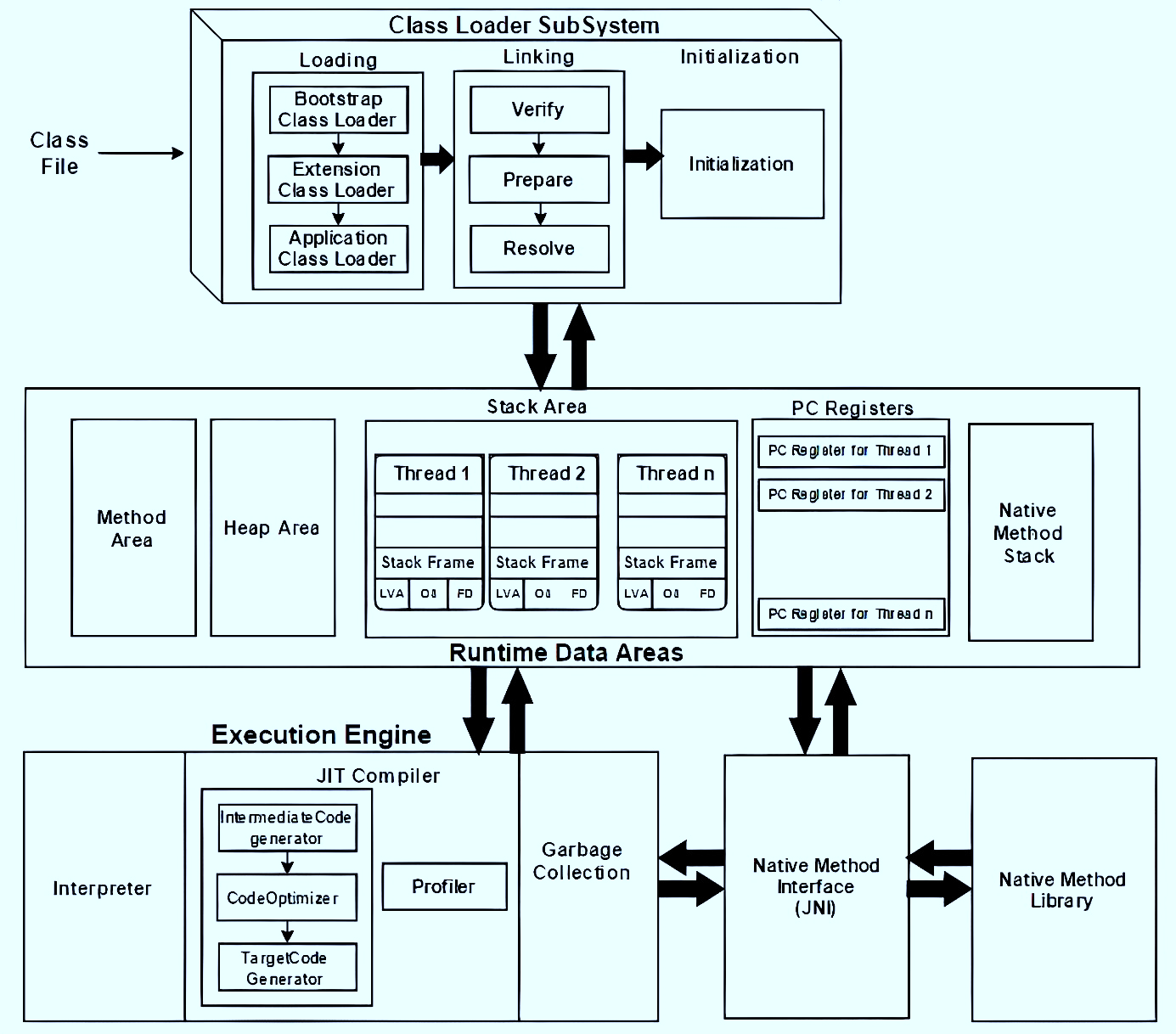
Basic working of converting JAVA code to JVM Byte Stream Code

Convert high level code into machine code

test.class

Class file to bytecode working

## JVM Explanation diagram



-Objects

-Instance variables

-Strings

-Arrays

- Next execution instruction address

-Current running method, local variables

- .class file details

- static variables

Security manager

Profiler (for hot spot)

Machine code / native code

**Memory area** where java elements are stored

all static variables are assigned with original values

static blocks will be executed from top to bottom

## Working description of JVM Parts

**JVM Deep Architecture**

1. It is a Runtime Engine responsible to run java-based applications.
2. It has two main tasks
   1. load.class file
   2. execute.class file

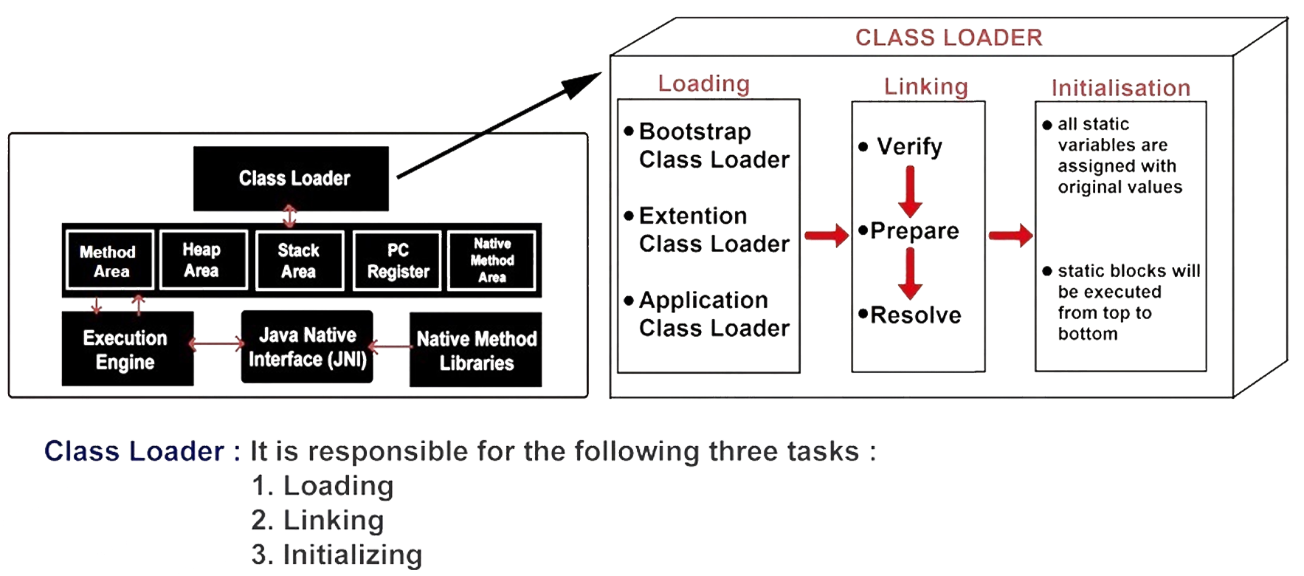
**Thus, architecture of JVM is broadly divided in three main categories**

1. Class Loader
2. Memory Areas
3. Execution Engine

## Class loader

**Classloader** is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap Classloader:** This is the first classloader which is the super class of Extension classloader. It responsible to loads the classes which present in **rt.jar** that **contains all class files** of Java Standard Edition like **java.lang** package classes, **java.net** package classes, **java.util** package classes, **java.io** package classes**, java.sql** package classes etc. (rt.jar file is present in bootstrap classpath i.e **jdk\jre\lib**).
2. **Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loads the jar files located inside $JAVA\_HOME/jre/lib/ext\*.jar directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It internally uses environment variable class path. It is also known as Application classloader.



### Class Loader Architecture

**A. Loading**

1. It will read .class file and store corresponding information in the method area.
2. For each class file, JVM will store following information in the method area :
   1. Fully qualified class name.
   2. Fully qualified parent class name.
3. Methods information.
4. Variables information.
5. Constructors’ information.
6. Modifiers information.
7. Constant pool information etc...

**B. Linking**

In linking three activities are performed:

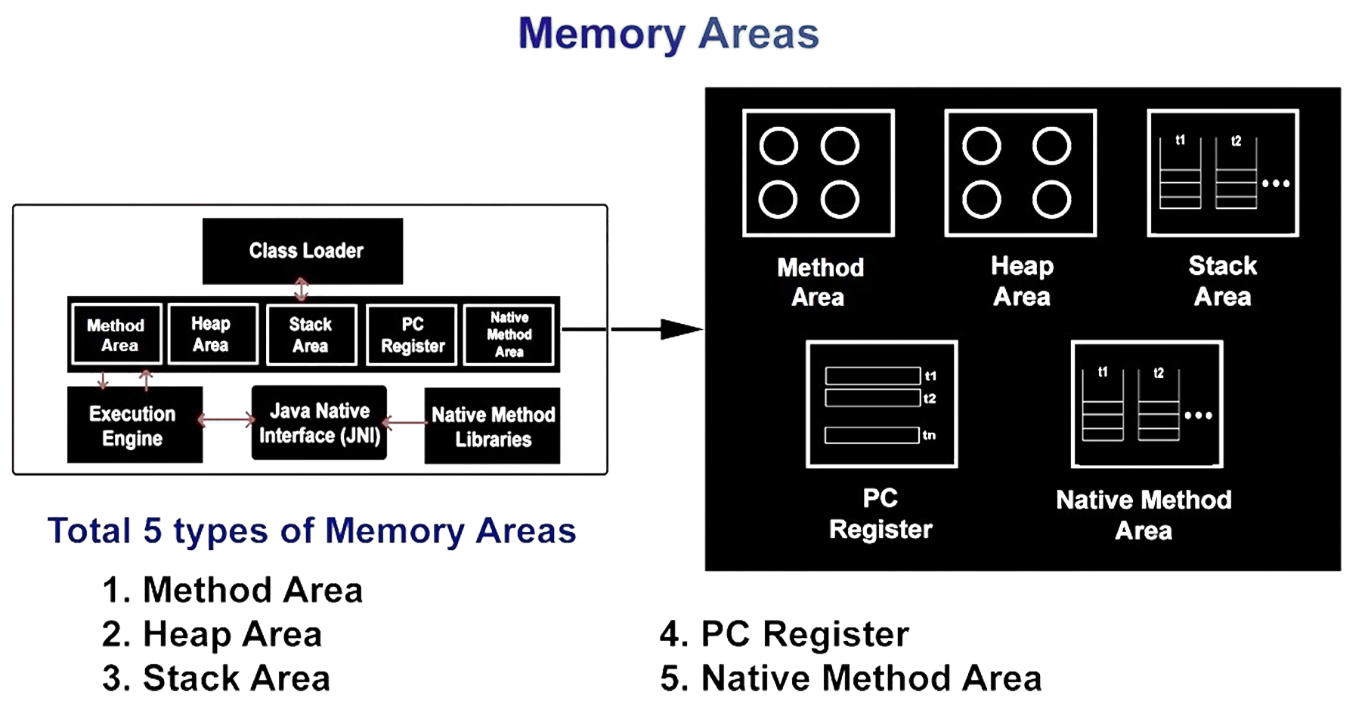
1. **Verification**
2. In this process Byte Code Verifier checks whether the .class file is generated by valid compiler or not and whether .class file is properly formatted or not.
3. If verification fails, then JVM will provide “**java.lang.VerifyError**” exception. Because of this process, java is secured.
4. **Preparation**
5. In this process JVM will allocate memory for class level static variables & assign default values (not original values).
6. **Resolution**
7. In this process symbolic names present in our program are replaced with original memory references from method area.

**C. Initializing**

In this process, two activities will be performed:

1. All static variables are assigned with original values.
2. static blocks will be executed from top to bottom.

## Memory Areas



**A. Method Area / Class Area**

1. Method area is created when JVM is started. It stores .class file information and static variables. Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
2. Per JVM one memory area, therefore multiple threads can access this area, so it is not thread safe.

**B. Heap Area**

1. Heap area is created when JVM is started.
2. It stores objects, instance variables and arrays (as every arrays is an object in java).
3. It can be accessed by multiple threads, so the data stored in heap area is not thread safe.

**C. Stack Area**

1. Whenever a new thread is created, a separate stack area will also be created. java stack stores frames. It holds/stores current running local variables and partial results, and plays a part in method invocation and return.
2. Each thread has a private JVM stack, created at the same time as thread.
3. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
4. After completing all method calls, the stack will become empty and that empty stack will be destroyed by thee **JVM** just before terminating the thread.
5. The data stored in the stack is available only for the corresponding thread and not available to the remaining thread, so this area is thread safe.

**D. PC Register Area**

1. It holds the address of next executing instruction.
2. For every thread, a separate pc register is created, so it is also thread safe.
3. Each JVM thread that carries out the task of a specific method has a **program counter register** associated with it. The non-native method has a PC that stores the address of the available JVM instruction whereas, in a native method, the value of the program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

**E. Native Method Stack Area**

1. Also called **C stacks**, native method stacks are not written in Java language. This memory is allocated for each thread when it’s created And it can be of a fixed or dynamic nature.
2. All native method calls invoked by the thread will be stored in the corresponding native method stack.
3. For every thread separate native method stack will be created. It is also thread safe.

## Execution Engine



**Execution Engine**

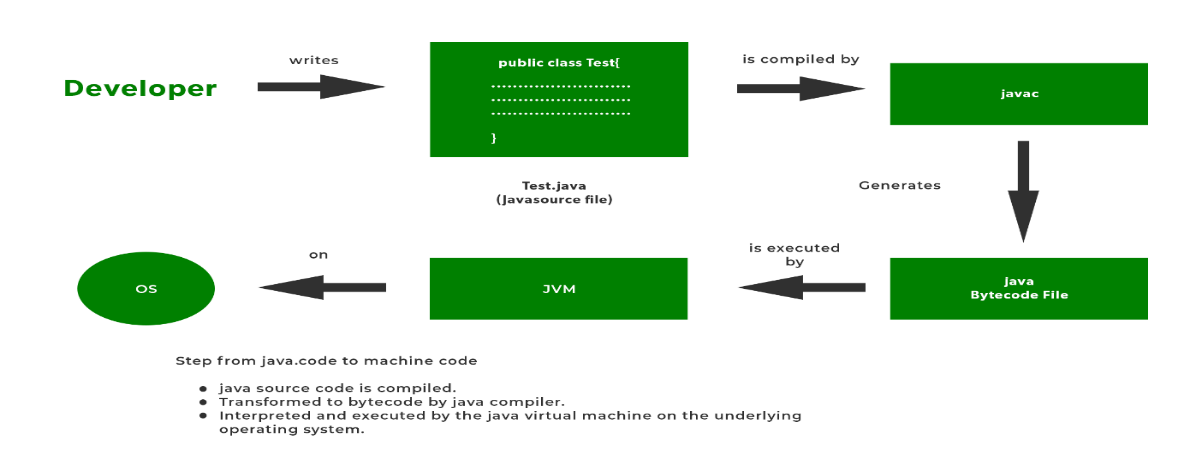
It is responsible to execute java class file. It contains mainly three components

1. A virtual processor
2. Interpreter
3. JIT Compiler
4. **Virtual Processor**

The JVPs are responsible for executing all server-based Java UDRs and applications. Although the JVPs are used for Java-related computation, they have the same capabilities as a CPU VP, and they can process all types of SQL queries.

1. **Interpreter**

A module that alternately decodes and executes every statement or line in some body of code. The Java interpreter decodes and executes bytecode for the Java virtual machine.



1. **JIT Compiler**
2. JIT stands for **Just-in-Time Compiler** which means that code gets compiled when it is needed, not before runtime.
3. The main purpose of JIT compiler is to improve performance.
4. JVM maintains a count as of how many times a function is executed. If this count exceeds a predefined limit or say threshold value, the JIT compiles the code into machine language which can directly be executed by the processor
5. Unlike the normal case in which javac compile the code into bytecode and then java.
6. The interpreter interprets this bytecode line by line converts it into machine code and executes.
7. Also next time this function is calculated same compiled code is executed again unlike normal interpretation in which the code is interpreted again line by line.
8. This makes execution faster. JIT compilation is applicable only for repeatedly required methods, not for every method.

## Java Native Interface (JNI)

1. Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.
2. It acts as mediator for java method calls & the corresponding native libraries i.e. JNI is responsible to provide information about native libraries to the JVM.
3. Native Method Library provides or holds native library information.
4. The java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

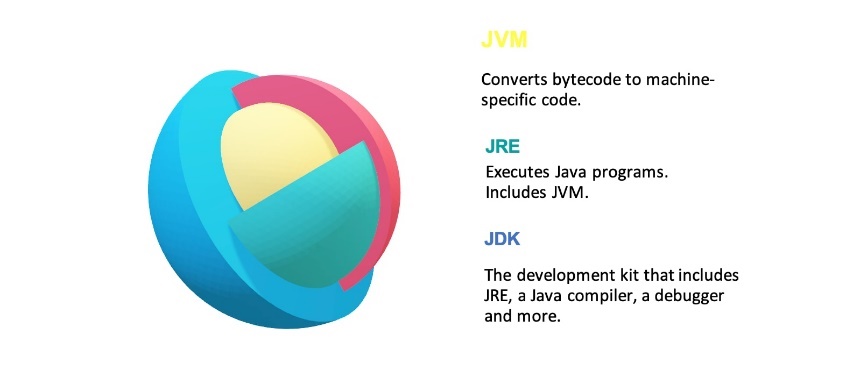
# **Interview Questions chapter - 1**

1. **What is the role of compiler? Or What happens in java compilation phase ?**

Compilers analyse and convert source code written in languages such as Java, C++, C# or Swift. They're commonly used to generate machine code or bytecode that can be executed by the target host system.

1. Compiler checks the syntax whether it is correct or not (whether the class, methods, interfaces, variable declaration etc is correct or not, Java is case sensitive language thus it checks upper case and lower-case keywords names).
2. Java is strongly typed language thus compiler checks whether the variables has stored correct value or not.
3. Compiler ignores all the comments.
4. Compiler generates the byte code.

1. **What happens in execution phase ?**
2. Byte code is loaded in JVM
3. Byte code verifier checks whether the byte code is correct or not
4. Memory allocation starts (variables, methods, class information etc)
5. Interpreter and JIT compiler executes our program and provides the output



1. **What is the role of JVM ?**

JVM is specifically responsible for converting bytecode to machine-specific

code and is necessary in both JDK and JRE. It is also platform-dependent

and performs many functions, including memory management and security.

1. **What is Virtual Machine ?**

A Virtual Machine (VM) is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual “guest” machines run on physical “host” machine. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host. This means that, for example, a virtual MacOS virtual machine can run on a physical PC.

It is a software simulation(copy) of a machine which performs operation similar to physical machine for

eg **calculator or JVM**

Types of Virtual Machine

1. Hardware Based VM or System Based VM
2. Application Based VM or Process Based VM
3. JVM is Application Based or Process Based VM

5. **What is JVM (Java Virtual Machine) ?**

1. JVM is Java Virtual Machine which is used to execute java byte code
2. JVM is Application Based or Process Based VM
3. Role of JVM
4. Architecture of JVM go to **pg-17**

6**. Explain deep architecture of Class Loader ? pg-18**

7**. How many memory areas are present in JVM?**

There are 5 memory areas are present in JVM are (Method area/Class area, Heap area, Stack Area, PC registers, Native method area)

8. **What data is stored in Method Area ?**

1. .class file information
2. static variables

9**. What data is stored in Heap Area ?**

1. Objects
2. Arrays
3. Instance Variables

10**. What data is stored in Stack Area ?**

1. Current Running Methods
2. Local Variables

11**. Explain the working of Execution Engine ?**

Execution Engine in Java is the core component of the JVM (java virtual machine) which communicates with different memory areas of the JVM. This component is used to execute the bytecode that is assigned to the runtime data areas via the classloader.

12**. What is JIT Compiler ?**

Bytecode is one of the most important features of java that aids in cross-platform execution. The way of converting bytecode to native machine language for execution has a huge impact on its speed of it. These bytecodes have to be interpreted or compiled to proper machine instructions depending on the instruction set architecture. Moreover, these can be directly executed if the instruction architecture is bytecode based. Interpreting the bytecode affects the speed of execution. In order to improve performance, JIT compilers interact with the [Java Virtual Machine (JVM)](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) at run time and compile suitable bytecode sequences into native machine code. While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring overhead for the translation process. This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently.

The JIT compiler is able to perform certain simple optimizations while compiling a series of bytecode to native machine language. Some of these optimizations performed by JIT compilers are data analysis, reduction of memory accesses by register allocation, translation from stack operations to register operations, elimination of common sub-expressions, etc. The greater the degree of optimization done, the more time a JIT compiler spends in the execution stage. Therefore, it cannot afford to do all the optimizations that a static compiler is capable of, because of the extra overhead added to the execution time and moreover its view of the program is also restricted.

13. **What is static or dynamic programming language?**

A dynamic language (Lisp, Perl, Python, Ruby) is designed to optimize programmer efficiency, so you can implement functionality with less code. A static language (C, C++, etc) is designed to optimize hardware efficiency, so that the code you write executes as quickly as possible.

14. **How java is dynamic programming language ?**

Java is considered dynamic because of Bytecode. The source code which is written in one platform that code can be executed in any platform. It loads the class file during runtime only. Hence, any thing that happens in runtime is dynamic.

15**. Why java does not require pre-processor ?**

Pre-processor directives are directives that a program, which would be compiled into machine code, needs to resolve before it can start executing on the processor. The compiler is responsible for resolving these directives even before it moves into the actual compilation phase. Java does not have any form of the **C #ifdef** or **#if** directives to perform conditional compilation. In theory, conditional compilation is not necessary in Java since it is a platform-independent language, and thus there are no platform dependencies that require the technique. In practice, however, conditional compilation is still often useful in Java--to provide slightly different user interfaces on different platforms, for example, or to support optional inclusion of debugging code in programs. docstore.mik.ua/orelly/java/javanut/ch02\_04.htm

16**. How java is platform independent ?**

Java is platform-independent because it uses a virtual machine. The Java programming language and all APIs are compiled into bytecodes. Bytecodes are effectively platform-independent. The virtual machine takes care of the differences between the bytecodes for the different platforms.

1. **Why java does not support pointers ?**

Java do not use pointers because using pointer the memory area can be directly accessed, which is a security issue. pointers need so memory spaces at the runtime. to reduce the usage of memory spaces java does not support pointers. and also pointers take more time at the run time. Most studies agree that pointers are one of the primary features that enable developers to inject bugs into their code. When Java was created, the intention was to create a language that is easy to learn and not prone to the bugs that C++ is prone to. It's not like c/c++ where we have to manage the memory management by destructors. In java automatic Garbage Collector works for memory management. Actually, Java references are pointers so everything in Java is accessed only through pointers.

1. **Why java has good memory management as compared to C & C++ ?**

The main difference between C++ and Java is the Garbage Collection. Java runs all the memory by the Garbage Collector. C++ developers should keep watch of their memory management. A complex C++ application can cause computer memory shortage.

1. **What do you mean by Platform Independent ?**

Software that can run on a variety of hardware platforms or software architectures. Platform-independent software can be used in many different environments, requiring less planning and translation across an enterprise.

1. **What is call by reference and call by value ?**

|  |  |
| --- | --- |
| Call By Value | Call By Reference |
| While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”. | While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References. |
| In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function. | In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function. |
| With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function. | With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them. |
| Thus actual values of a and b remain unchanged even after exchanging the values of x and y. | Thus actual values of a and b get changed after exchanging values of x and y. |
| In call-by-values, we cannot alter the values of actual variables through function calls. | In call by reference we can alter the values of variables through function calls. |
| Values of variables are passed by the Simple technique. | Pointer variables are necessary to define to store the address values of variables. |

1. **Is JVM Platform Dependent or Platform Independent ?**

Java is platform independent because of JVM

1. **How to set path in java ?**
2. In Search, search for and then select: System (Control Panel)
3. Click the **Advanced system settings** link.
4. Click **Environment Variables**. In the section **System Variables** find the PATH environment variable and select it. Click **Edit**. If the PATH environment variable does not exist, click New.
5. In the **Edit System Variable** (or **New System Variable**) window, specify the value of the PATH environment variable. Click **OK**. Close all remaining windows by clicking **OK**.
6. Reopen Command prompt window, and run your java code.
7. **What is difference between temporary path set and permanent path set ?**

In temporary path set access of java folder is limited to a particular **cmd** shell terminal or in permanent path set accessibility of javac and all of its tools in entire windows.

**Sting temporary path**

1. Open command prompt in Windows.
2. Copy the path of jdk/bin directory where java located (C:\Program Files\Java\jdk\_version\bin)
3. Write in the command prompt: SET PATH=C:\Program Files\Java\jdk\_version\bin and hit enter command.
4. **What happens in java compilation phase ?**

Compiling a Java program means taking the programmer-readable text in your program file (also called source code) and converting it to bytecodes, which are platform-independent instructions for the Java VM. Note: Part of the configuration process for setting up the Java platform is setting the class path

1. **Different ways to declare main() method ?**

# java data types

Java Fundamentals

Chapter 2

## Java Is a Strongly Typed Language

I

t is important to state at the outset that Java is a strongly typed language. Indeed, part of Java’s safety and robustness comes from this fact. Let’s see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

**What is Data types?**

A data type is a set of values and a set of operations defined on those values. The type of data that we are specifying to java is known as Data Type.

For example :-

10 - **int**

'a' - **char**

"deepak" - **String**

true - **boolean**

According to data type, languages are divided into 2 categories.

**1. Statically Type Languages** In this type of languages we have to specify the type of each data and thus compiler known which type of data we have provided. For examples C, C++, Java, FORTRAIN, Pascal etc.

**2. Dynamically Typed Languages** In this type of languages we don’t need to specify the type of data that we have provided. For examples Python, JavaScript, Objective C, Ruby etc.

## The Primitive Types

The data types which are already provided by java and whose size are fixed are known as **primitive data type**. 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 both terms will be used in this book. These can be put in four groups:

• **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.

• **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.

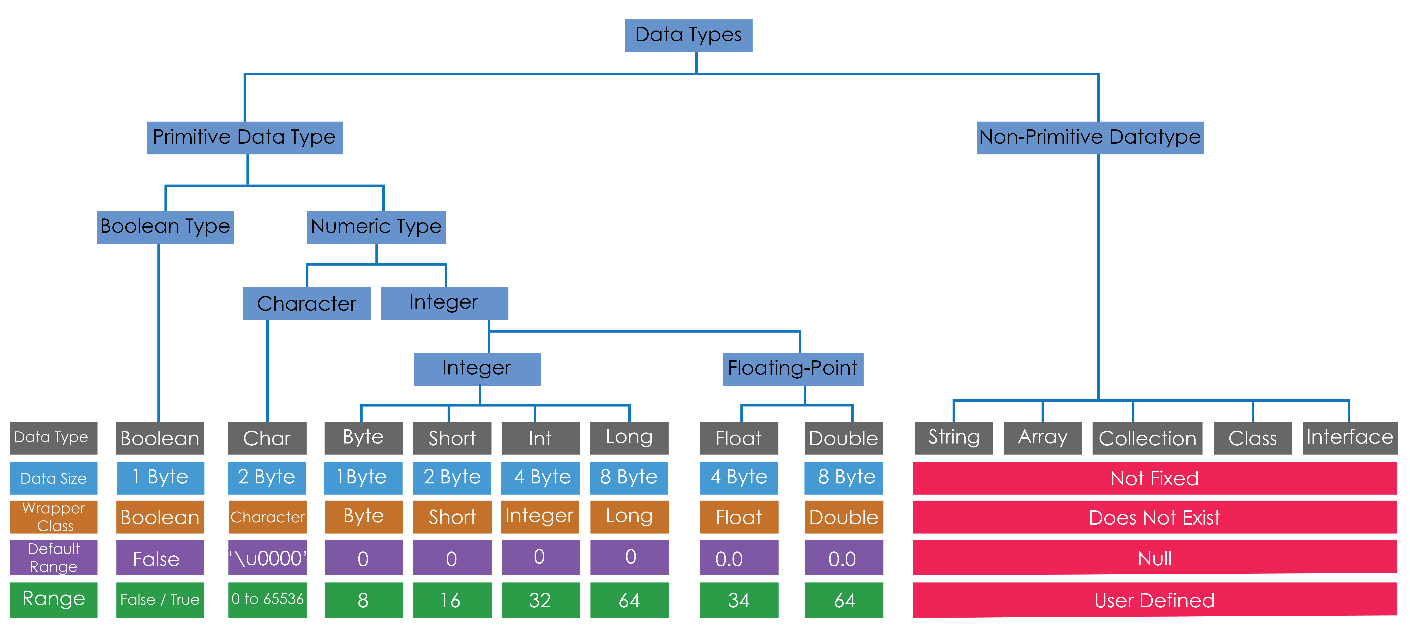
• **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.

• **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

1. To find the range of Integer primitive data type we can use the formula **2(n-1) to 2(n-1) - 1** (where n is no of bits).
2. To find the range (minimum and maximum value) of primitive data types (excluding Boolean) we can use **static final int** **MIN\_VALUE** & **MAX\_VALUE** variables.

## The User-defined types / Derived Datatype / Non-primitive type

A user-defined data type (**UDT**) is a data type that derived from an existing data type or primitive datatype. You can use UDTs to extend the built-in types already available and create your own customized data types. Sometimes these are known as **"reference variable"** or **"object reference".**

The size of non-primitive data type is not fixed. There are many users defined datatype which we used in java like as **Arrays, String, Class, Collection, Interface etc.** Hence, the data types that are defined by the user are known as user-defined data types.

## Concept of AutoBoxing and Unboxing

**Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their “*corresponding object wrapper classes”*. For example, converting an **int** to an **Integer**, a **double** to a **Double**, and so on. If the conversion goes the other way, this is called **unboxing**.

Here is the simplest example of autoboxing:

**Character ch = 'a';**

**int num1 = 10;** (int variable) to **Integer num2 = 10** (Integer Object)

**Unboxing** Converting an object of a wrapper type (**Integer**) to its corresponding primitive (**int**) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

* Passed as a parameter to a method that expects a value of the corresponding primitive type.
* Assigned to a variable of the corresponding primitive type.

List <**Double**> ld = new ArrayList<>(); // Making object of Double Class

ld.**add**(12.32); // Adding value in Double class Object

**double value = ld.get(0);** // Converting Double Object into double variable

# type casting

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the **conversion automatically**. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types. The process of converting the data from one data type to another data type is known as **type casting.** Let’s look at both automatic **type conversions** and **forcefully type casting** (also called loosely conversion).

There are 3 types of Type-Casting in Java

1. Primitive Data Type" Type Casting
   * 1. Widening Type Casting (Implicit Type Casting)
     2. Narrowing Type Casting (Explicit Type Casting)
     3. Automatic Type Promotion
2. "User Defined Data Type" Type Casting

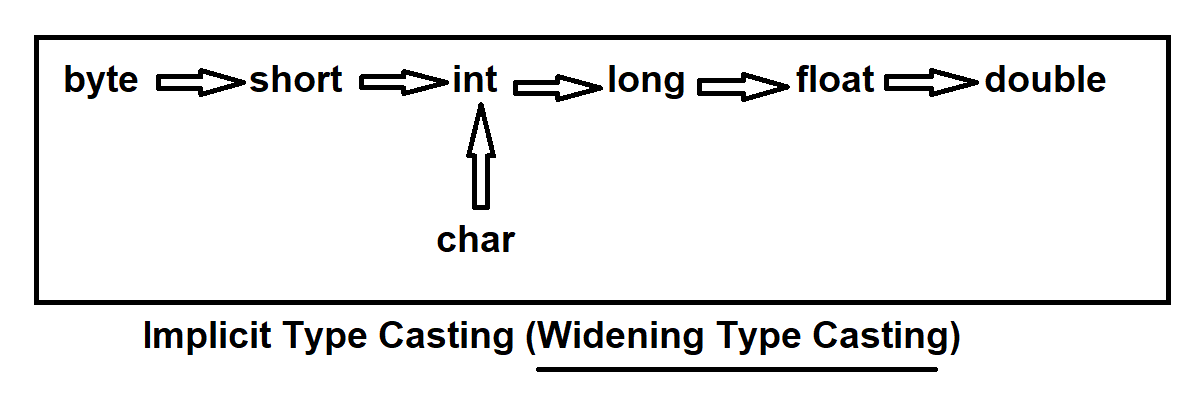
**1. "Primitive Data Type" Type Casting** When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met but only works on primitive datatypes.

• The two types are compatible.

• The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place.

1. Widening Type Casting (Implicit Type Casting)
   1. It is the process of converting data from lower data type to higher data type.

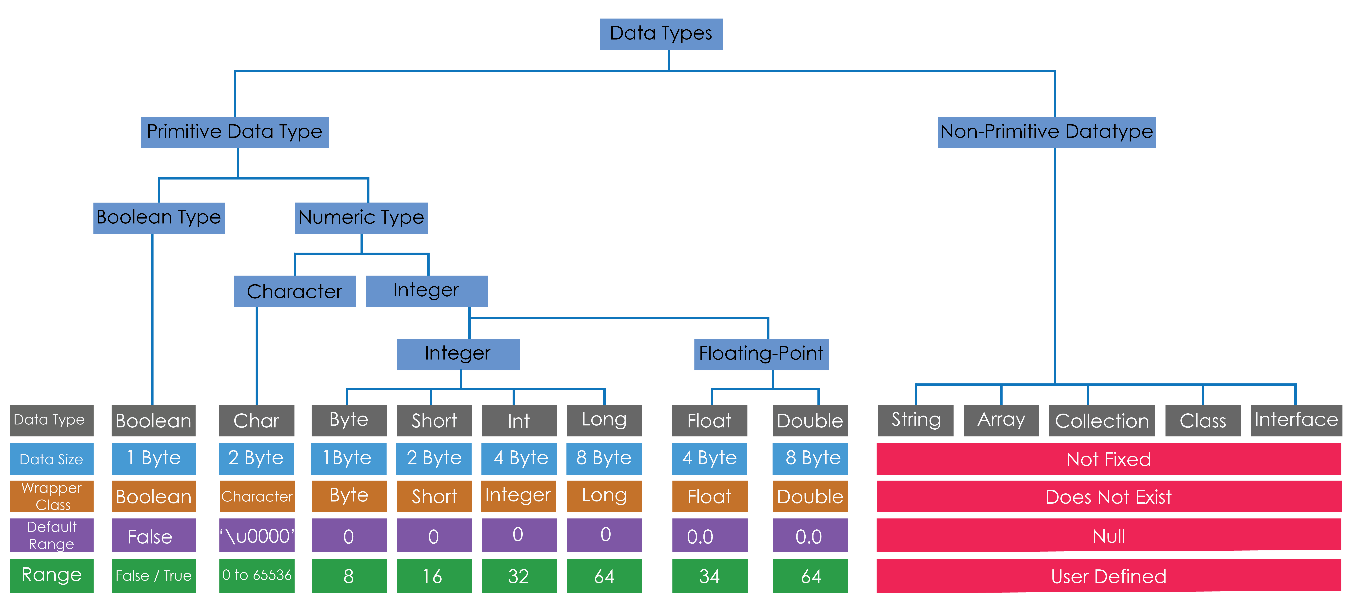


1. Narrowing Type Casting (Explicit Type Casting)
   1. In this case we convert higher data type into lower data type
   2. Narrowing Type Casting can be achieved by using "cast operator"

**Note: no1 & no2 , result**

no1 + no2 = result

1. If no1 & no2 is byte, short or int then result will be always in **int.**
2. If no1 & no2 is long, float, double etc then result will be in higher data type.



Widening Type Casting

Narrowing Type Casting

1. Automatic Type Promotion

In addition to assignments, there is another place where certain type In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following Java defines several *type promotion* rules that apply to expressions. They are as follows: First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands are **double**, the result is **double**.

byte b = 50;

b = (byte)(b \* 2);

which yields the correct value of 100(int).

**2. "User Defined Data Type" Type Casting** It is the process of converting data from one user defined data type to another user defined data type. For "user defined data type" type casting, both data types should have relation (either extends or implements).

*int* [ ] arrNumber = {10,30,20,45,26,30,22,11,33,15,44,55,66,}; // Integer Array  
String [ ] arrName = {"Dev","Ramesh","Suresh","Shyam","Ganesh","Kamlesh"}; // String Array

So here we converting Arrays Datatype to String Datatype using **Arrays.toString()** method.

String numbers = new String(Arrays.toString(arrNumber));

String names = new String(Arrays.toString(arrName));

### Type casting case study

Note : Checking variable data type in java use (((Object)s1).getClass().getSimpleName()) in System.out.println(); to get your primitive data type in console.

**1) Byte to short**

**byte** b1 = 100;

**short** s1 = b1;

System.***out***.println(s1+" --> is your Data\n"+((Object)s1).getClass().getSimpleName()+" --> this is your Datatype");

100 --> is your Data

Short --> this is your Datatype promoted byte to short

**2) Short to Int**

**short** s1 = 10;

**int** i1 = s1;

System.***out***.println(i1);

10 --> promoted short to int

**3) Int to Long**

**int** i1 = 10;

**long** l1 = i1;

System.***out***.println(l1);

10 --> promoted int to long

**4) Overflow type casting**

**int** i1 = 200;

**byte** b1 = i1;

System.***out***.println(b1); (throws an error)

Note 🡪 Because of int size is large then int and value of int cannot be fit in byte it will throw an exception are ->

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Type mismatch: cannot convert from int to byte at MyPracticeOfJava/typeCasting.java\_TypeCasting.main(java\_TypeCasting.java:14)

// to converting int to byte we will need to be cast into int to byte

**int** i1 = 200;

**byte** b1 = (byte)i1;

System.***out***.println(b1);

**o/p** : -56

**Note** - (because range of byte is less (-128 to 127) your result may be in negative from 1 to 127 value is going in increasing order and are range of byte will be crossed it will be start from -128 and increasing in ascending order that’s why result we get is -56.

For better understanding run below program

**for**(**short** i =0;i<200;i++) {

System.***out***.println(i +"-->"+ (**byte**)i);

}

**5) Adding Lower type**

**byte** b1 = 50;

**byte** b2 = 50;

**byte** b3 = b1+b2; (Type mismatch exception occur)

System.***out***.println(b1);

**Note** - Default result of two datatype ( int or less then int ) always be an int type. So the correct program is

**int** I = b1+b2;

System.***out***.println(I)

o/p -> 100

**6) Loosely type casting (Narrowing)**

**float** f1 = 55.5f;

**int** i2 = (**int**)f1;

System.***out***.println(f1+" float value");

System.***out***.println(i2+" int value");

o/p 🡪 55.5 (float value)

55 (int value) // 55.5 will remain 55 (0.5) is discarded and we lost our .5 value in higher to lower type conversion // format

**7) Automatic type promotion**

**byte** b1 = 55;

**int** i1 = 300;

**long** l1 = i1 + b1;

System.***out***.println(l1+" Result is in long ");

o/p -> 355 Result is in long

# number system

We i.e. Human beings communicate in words, characters and numbers but computers communicate only in numbers. The data or character that we provide to computer is converted into number (In java all characters have unique integer value known as ASCII). Computer (Programming Languages) communicates only in numbers thus there are 4 number systems:-

1. Binary Number System (Base 2)
2. Octal Number System (Base 8)
3. Decimal Number System (Base 10)
4. Hexa-Decimal Number System (Base 16)

**Java supports all number systems but default is "Decimal Number System"**

1. Binary Number System (Base 2)

Alphabet : 0,1

Prefix : 0b or 0B

2. Octal Number System (Base 8)

Alphabet : 0,1,2,3,4,5,6,7

Prefix : 0 (zero)

3. Decimal Number System (Base 10)

Alphabet : 0,1,2,3,4,5,6,7,8,9

Prefix : does not have any prefix

4. Hexa-Decimal Number System (Base 16) :-

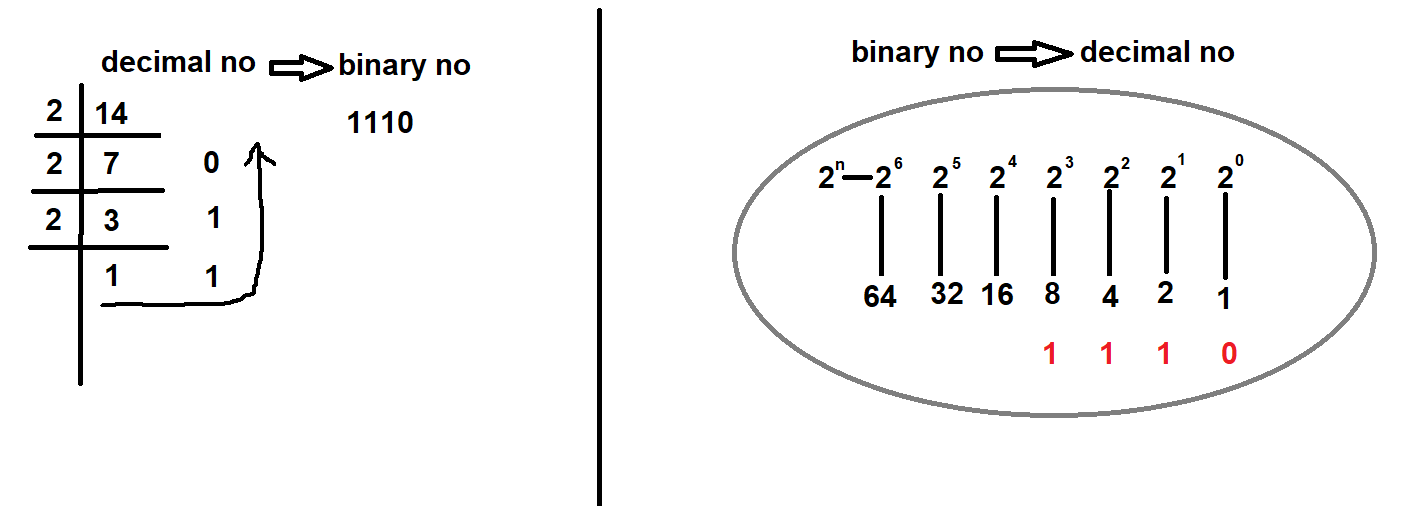
Alphabet : 0,1,2,3,4,5,6,7,8,9,A (10), B (11), C (12), D (13), E (14), F (15)

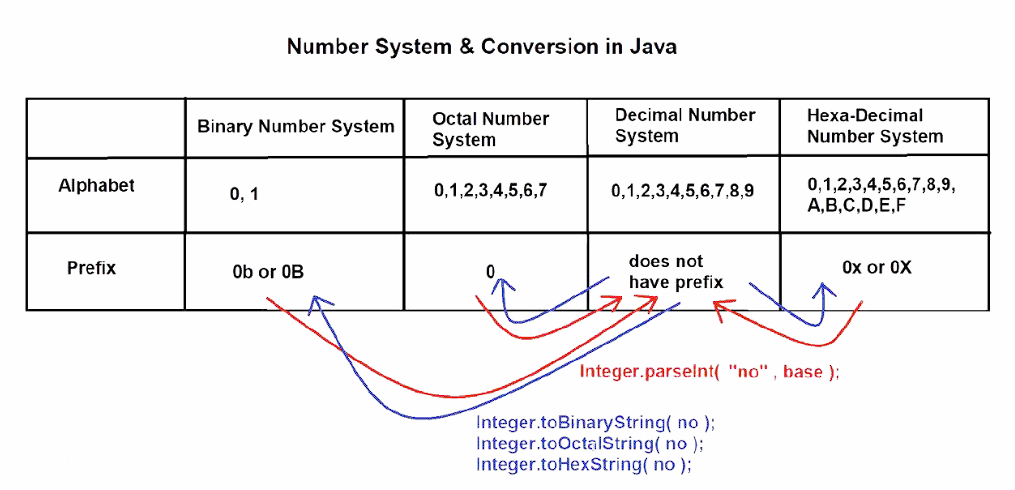
Prefix : 0x or 0X

## Conversion

1. If we want to convert Binary or Octal or Hexa-Decimal number system to Decimal number system -> Integer.parseInt( "no" , base );

2. If we want to convert Decimal number system to Binary or Octal or Hexa-Decimal number system -> Integer.toXxxString( no );





There are three method in java which can use to convert Decimal number system into other number system and these methods are present in **java.lang.Integer** class.

**int** number = 450;

System.***out***.println(Integer.*toBinaryString*(number)); o/p – 111000010 (Binary)

System.***out***.println(Integer.*toHexString*(number)); o/p - 1c2 (Hexadecimal)

System.***out***.println(Integer.*toOctalString*(number)); o/p – 702 (Octal)

Note that a local variable is not created until the method in which the variable is define is called. once the method is called then the all of the variable which is declared in that method is created and when the method execution is completed then all of variable is deleted and again free up the memory.

# variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next. In Java, all variables must be declared before they can be used. Variable is the name of memory location that contains the data the variable value can change according to programming logic. The basic form of a variable declaration is shown here:

*1) type identifier* = *value; // this is first way to declare variable in java (create single variable at a time)*

*2) type identifier value1, value2, value3, value4 …; // (all the variable is create same type but all of these is empty in second type of creation.)*

***Static variable initialization***

*Ex - >* **int**  *variable1 = 10; // this is integer type variable*

**char** c='a'; (c is the variable)

**String** *variable1, variable2, variable3, variable4, variable5; // this is string type variable*

**String** name= "deepak"; (name is the variable)

***Dynamic variable initialization***

// Demonstrate dynamic initialization.

class **Dynlnit** {

**public static** void main (String args [] )

**double** a = 3.0, b = 4.0;

// c is dynamically initialized

**double** c = Math.sqrt(a \* a + b \* b) ;

System.out.println("Hypotenuse is " + c);

}}

**Types of variables**

1. Local Variables

2. Instance Variables

3. Static Variables

**public** **class** A

{

**static** **int** m=100;//static variable

**void** method()

    {

**int** n=90;//local variable

    }

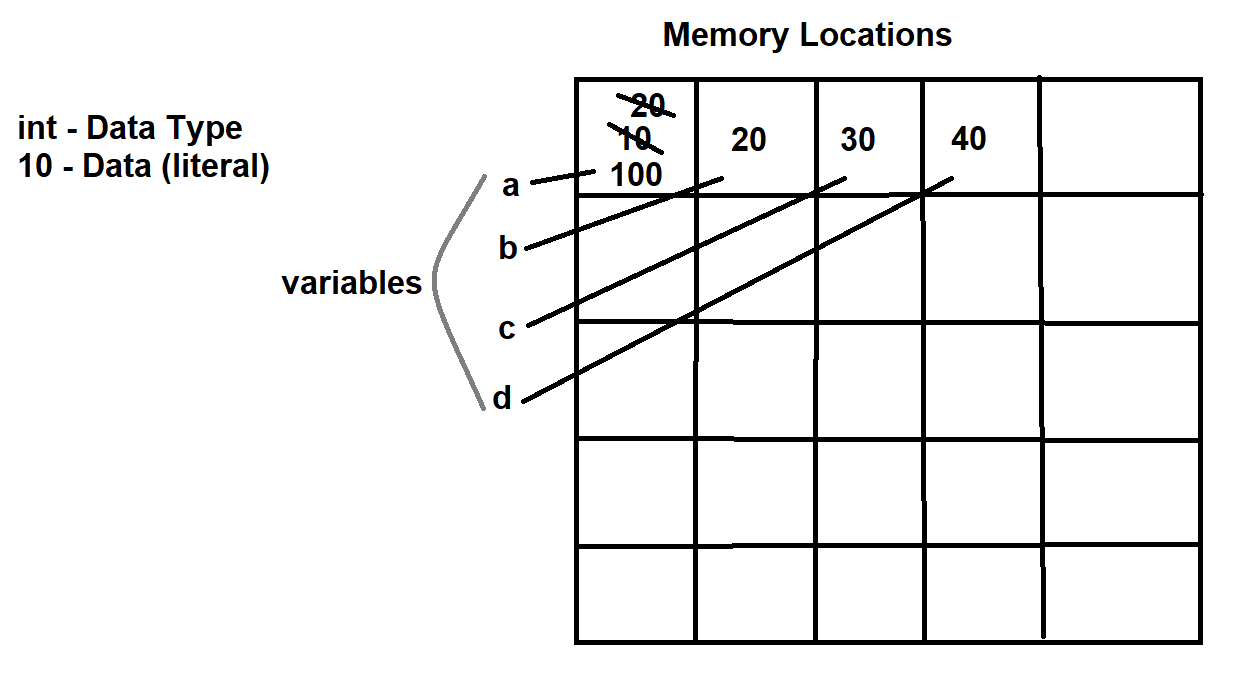
**public** **static** **void** main(String args[])

    {

**int** data=50;//instance variable

    }

}//end of class



## 1. LOCAL VARIABLES

|  |  |
| --- | --- |
|  |  |
| **Declaration** | Local variables are declared within the body of methods, constructors or blocks. |
| **Scope** | Local variables can be used within the methods or constructors or blocks but not outside them. |
| **Allocated** | Local variables get allocated when the methods or constructors or blocks are executed and get deleted from memory when that method or block or constructor execution completes. |
| **Stored Memory Area** | Local variables gets memory allocated in "**STACK AREA**". |
| **Default Values** | Local variables does not have any default value, if we don’t provide the value for local variables and use them, it will provide compile time error saying "variable name might not have been initialized". |
| **Access modifiers** | We cannot use access modifiers i.e. public, protected and private with local variables.  How to access Local Variables :  1. directly |

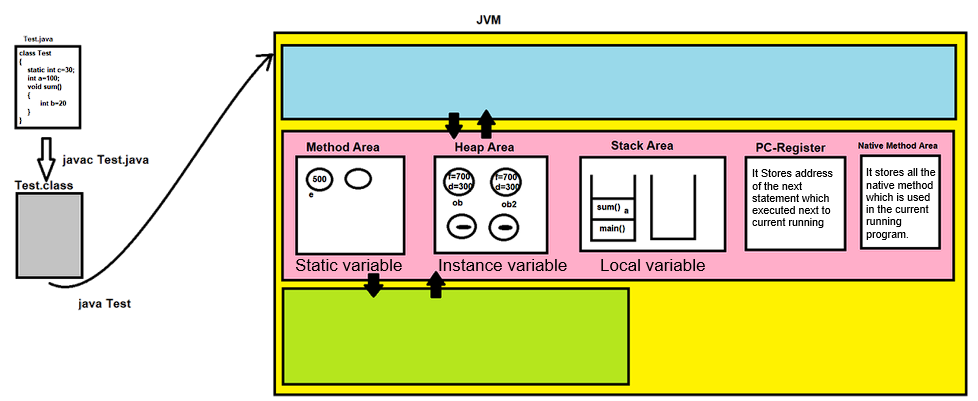
## 2. INSTANCE VARIABLES :-

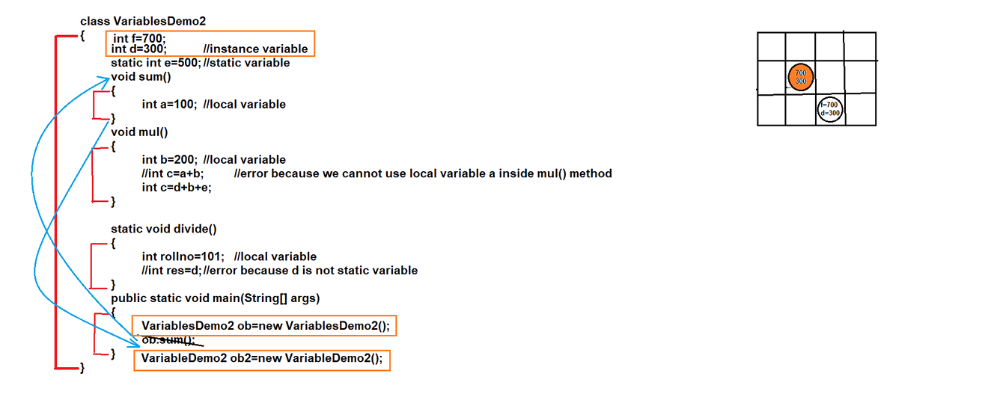
|  |  |
| --- | --- |
|  |  |
| **Declaration** | Instance variables are declared within the class but outside the methods or constructors or blocks. |
| **Scope** | Instance variables can be used within the class and every method or block or constructor but not inside the static methods or static blocks. |
| **Allocated** | Whenever new object is created, instance variables gets memory allocated and when that object is destroyed instance variables also gets deleted. |
| **Stored Memory Area** | Instance variables are stored in "HEAP AREA". |
| **Default Values** | Instance variables have default values for ex int - 0; boolean - false; float - 0.0; etc. |
| **Access Modifiers** | We can use access modifiers i.e. public, protected and private with instance variables.  How to access Instance Variables:  1. directly  2. by using object name |

## 3. STATIC VARIABLES

|  |  |
| --- | --- |
|  |  |
| **Declaration** | Static variables are also declared within the class but outside the methods or constructors or blocks and we also use "static" keyword with them. |
| **Scope** | Static variables can be used in every method or static methods or blocks or static blocks or constructors. |
| **Allocated** | When we run java program, .class file or byte code gets loaded in JVM and at that time only static variables also gets memory allocated. And when the .class file gets unloaded from JVM static variables gets destroyed from the memory. |
| **Stored Memory Area** | Static variables are stored in "METHOD AREA". |
| **Default Values** | Static variables have default values for ex int - 0; boolean - false; float - 0.0; etc. |
| **Access Modifiers** | We can use access modifiers i.e. public, protected and private with static variables.  How to access Static Variables:  1. directly  2. by object name  3. by class name |
|  |
|  |
|  |
|  |
|  |

Memory Allocation of Local, Instance & Static variables





# Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. This chapter describes all of Java’s operators except for the type comparison operator instanceof, in further chapters. Lets we starts from tokens -

## Tokens

Tokens are the smallest unit or say small building blocks of java program that are meaningful to the java compiler ex .. System.out.println("hello"); basically tokens are the specific part of the program which cannot be breakdown further in meaning full manner this is known as token in programming. tokens are **System**, **.**, **out**, **println**, **(**, **“**,**hello,”,)**,; . Our java program converts into tokens and then java compiler converts these tokens into java bytecode.

Different types of tokens are

|  |  |
| --- | --- |
|  |  |
| **Literals** | A constant value in Java is created by using a *literal* representation of it. For example, here are some literals: Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed. Any content value assigned to the variable is known as literal  🡪 For example :-   1. int a=10; //10 is literal 2. char c='x'; //x is literal 3. String name="deepak"; //deepak is literal 4. boolean b=true;   **Types of literals**   1. Integral Literals (byte, short, int, long) 2. Floating-Point Literals (float, double) 3. Character Literals (Single Quotes [''], Char literal in Integer literal, Unicode representation, escape character ) 4. String Literals 5. Boolean Literals |
| **Operators** | Depending on its type, an operator manipulates an arithmetic or logical value, or operand, in a specific way to generate a specific result. From handling simple arithmetic functions to facilitating the execution of complex algorithms, like security [encryption](https://www.techtarget.com/searchsecurity/definition/encryption), operators play an important role in the programming world. Operators are the special symbols which are used to perform any operation on one or more operands.  For example : c=a+b; (+ is operator and a,b are operands).  **Types of operators**  **1**. **Arithmetic Operators**  + Addition  - Subtraction  \* Multiplication  / Division  % Modulus  **2. Unary Operators**  posfix : no++, no--  prefix : ++no, --no, +no, -no, ~no, !no  **3. Assignment Operators**  = Assignment  += Addition assignment  -= Subtraction assignment  \*= Multiplication assignment  /= Divide assignment  &= And assignment  ^= Bitwise exclusive OR assignment  |= Bitwise OR assignment  <<= Shift left assignment  >>= Shift right assignment  >>>= Shift right zero fill assignment  **4**. **Relational Operators**  == Equals to operator  != Not equals to operator  < Less then operator  > Greater then operator  <= Less then equal to  >= Greater then equal to  **Instanceof** (Provides the output in true or false)  **5. Bitwise Operators**  **A**. Binary Bitwise Operators  **1**. Bitwise Logical Operators  & (bitwise AND)  | (bitwise OR)  ^ (bitwise exclusive OR)  **2**. Shift Operators  >> (right shift)  << (left shift)  >>> (zero fill right shift)        **B**. Unary Bitwise Operators  ~ One's compliment Operator  **6**. **Logical Operators**  && (logical AND)  || (logical OR)  ! (logical NOT)  **7**. **Ternary Operator**  ? (variable = condition ? expression 1 : expression2)  Ternary Operator by Smart Programming |
| **Separators** | In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is often used to terminate statements. The separators are shown in the following table:   |  |  |  | | --- | --- | --- | | Symbol | Name | Purpose | | () | Parentheses | Used to contain list of parameters in method definition and invocation. Also used for defining precedence in expression, containing expressions in control statements, and surrounding cast types. | | {} | Braces | Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes. | | [] | Braces | Used to declare array types. Also used when dereferencing array values. | | ; | Semicolon | Terminates statements. | | , | Comma | Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement. | | . | Period | Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable. | | … | Ellipses | Indicates a variable-arity parameter. | | @ | Ampersand | Begins an annotation. | |
| **Punctuators** | A punctuator is a token that has syntactic and semantic meaning to the compiler, but the exact significance depends on the context. A punctuator can also be a token that is used in the syntax of the pre-processor. There are three types of punctuators.  1) **?** - Question Mark  > used in ternary operator  2) **:** - Colon  > used after loop labels  3) **::** - Double Colon  > used to create method or constructor references |
| **Comments** | As mentioned, there are three types of comments defined in Java.  1) Single line comment - //  2) Multiline Comment - /\* \*/  3) Documentation Comment - /\* \*- \*- \*- \*- \*- \*- \*/  The third type is called a ***documentation comment***. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a  /\*\* and ends with a \*/. Documentation comments are explained in later. |
| **Keywords /**  **(Reserved Words)** | There are **53 keywords** currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. In general, these keywords cannot be used as identifiers, meaning that they cannot be used as names for a variable, class, or method.  **Used Keywords**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | abstract | continue | for | package | synchronized | | assert | default | if | private | this | | boolean | do | implements | protected | throw | | break | double | import | public | throws | | byte | else | instanceof | return | transient | | case | enum | int | short | try | | catch | extends | interface | static | void | | char | final | long | strictfp | volatile | | class | finally | native | super |  | | while | float | new | switch |  |   **Unused Keywords**  Note :   1. The keywords **const** and **goto** are reserved, even they are not currently in used they are no longer supported in java. 2. There are three words **true**, **false** and **null** look like keywords, but in actually they are **reserved literals**. However, they still can’t be used as identifiers in a program. 3. Keywords - 48 (Used keywords)   02 (Unused keywords)  03 (Reserved literals)  53 (Total Keywords) |
| **Identifiers** | Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are in point 8.  Invalid identifier names include these:    **Rules of Identifiers**   1. A variable name can consist of Capital letters **A-Z**, lowercase letters **a-z** digits **0-9**, and two special characters such as **\_** underscore and **$** dollar sign. 2. The first character must not be a digit. 3. Blank spaces cannot be used in variable names. 4. Java keywords cannot be used as variable names. 5. Variable names are case-sensitive. 6. There is no limit on the length of a variable name but by convention, but generally it should be in between 4 to 15 chars. 7. Variable names always should exist on the left-hand side of assignment operators. 8. Here are a few valid java variable name examples : *myvar, myVar, MYVAR, \_myVar, $myVar, myVar1, myVar\_1.* |

# **Interview Questions chapter - 2**

**1. What is ADT (Abstract Data Type) ?**

1. ADT is a type (or class) which holds the different types of objects with some specifications
2. The definition of ADT only mentions what operations are to be performed but not how these operations implemented

For examples : Stack ADT, List ADT, Queue ADT etc

**2. Why java is not purely OOP's Language ?**

Java is not purely OOP's Language because:-

1. Usage of Primitive Data Types

2. Usage of Static members

**3. How primitive variables passed to methods - by value or by reference :-**

Java supports only pass by value

**4. What is type casting ?**

Type casting is when you assign a value of one primitive data type to another type. If you convert lower range data type to higher then it would called *widening type casting* either higher type data can be converted or casted into lower range data type called *narrowing type casting*.

**5. What are different types of Type-Casting in java ?**

There are 2 types of Type-Casting in Java :-

1. "Primitive Data Type" Type Casting

1.1 Widening Type Casting (Implicit Type Casting)

1.2 Narrowing Type Casting (Explicit Type Casting)

2. "User Defined Data Type" Type Casting

**6. What is Type Checking & Type Casting ?**

**Type Checking:** Type Checking is the responsibility of compiler. It checks whether the syntax is correct or not and whether we are assigning lower data type to higher data type.

**Type Casting:** Type Casting is the responsibility of JVM. In this phase lower data type (for e.g. byte) will convert into higher data type (for e.g. int) and the value will be copied in higher data type.

**7. What are statically and dynamically typed languages?**

**Statically - In statically typed programming languages, type checking occurs at**[compile time](https://www.baeldung.com/cs/compile-load-execution-time#compile-time). At compile time, source code in a specific programming language is converted to a machine-readable format. This means that before source code is compiled, the type associated with each and every single variable must be known.

**Dynamic** - Conversely, **in dynamically typed languages, type checking takes place at**[runtime](https://www.baeldung.com/cs/runtime-vs-compile-time#runtime)**or execution time**. This means that variables are checked against types only when the program is executing. Some examples of programming languages that belong to this category are Python, JavaScript, Lisp, PHP, Ruby, Perl, Lua, and Tcl.

**8. What is difference between Primitive Data Type & Non-Primitive Data Type?**

Those datatype which are shipped with languages are called primitive data type and primitive datatype contain its own data value for ex byte = 1, short = 2, int = 4, long =8 but on the other side Non-primitive datatype are building with the help of primitive datatype and it has no limit to stores the data. These are as classes, array, String etc.

**9. What are Wrapper Classes?**

1. The classes which are used to convert primitive data into objects and objects into primitive data .
2. There are 8 wrapper classes: - Boolean, Character, Byte, Short, Integer, Long, Float & Double
3. Java introduced autoboxing and unboxing in J2SE 5.0 version which coverts primitive into object and object into primitive automatically.

**10. What is Autoboxing & Unboxing?**

1. Autoboxing is the automatic conversion of primitive data type into its corresponding wrapper classes by java compiler.
2. Unboxing is the automatic conversion of an object of wrapper type to its corresponding primitive value by java compiler

**11. What is instanceof operator ?**

As the name suggests, instanceof in Java is used to check if the specified object is an instance of a class, subclass, or interface. It is also referred to as the comparison operator because of its feature of comparing the type with the instance. This is used to verify if the specified object is the instance of specified class.

**12. What is difference between & and && operator?**

|  |  |
| --- | --- |
| **Difference between &** | **Difference between &&** |
| Bitwise AND | Logical AND |
| Operates on bit values | Operates on boolean values |
| In this case both side will get evaluated because there is no true or false case here | In this case If first expression is false, then it will not evaluate the second condition and it will return false directly |
| Used in bit manipulation | Used in conditional statements and loops |

**13. What is ternary operator?**

The Java ternary operator provides an abbreviated syntax to evaluate a true or false condition, and return a value based on the Boolean result. The Java ternary operator can be used in place of if..else statements to create highly condensed and arguably unintelligible code.

**boolean** b1 = (5>7) ? **true** : **false**;

System.***out***.println(b1);

o/p -> false

Control Statements

Chapter 3

# Control statements

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java’s program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java’s control statements are examined here.

## Types of control statements

Control Statements are those who can control the flow of the program

There are three types of Control Statements

### 1. Selection Statements

**if, if else, if-else if, nested if, switch**

#### if

The **if** statement is Java’s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

Syntax

If (*condition*) // in parentheses if always evaluate boolean condition (only check true/false)

*statement1;*

Example

**int** first = 10, second = 20;

**if**(first <second) {

System.***out***.println("first is smaller");

}

**o/p** first is smaller

Most often, the expression used to control the if will involve the relational operators. However, this is not technically necessary. It is possible to control the if using a **single boolean variable**, as shown in this code fragment:

**boolean** dataAvailable;

**if**(dataAvailable) { // here we use single bool variable to control if (if processDataFurther is true) if block is evaluated otherwise else block is evaluated

processDataFurther();

}

**else** {

waitForMoreData();

}

#### if else

To make a complete statement in our program then we will use **else with if** and without curly braces Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you’ll need to create a block, as in this fragment:

Syntax

*If(condition)*

*Statement1;*

*else*

*default condition;*

Examples

**int** validAge = 18, userAge=15;

**if**(userAge > validAge) {

System.***out***.println("You are valid candidate for voting");

}

**else** {

System.***out***.println("You are not valid candidate for voting");

}

**o/p –** you are not valid candidate for voting.

#### If else if / ladder if

A common programming construct that is based upon a sequence of nested **if’**s is the *if-else-if* ladder. It looks like this:

*if(condition)*

*statement;*

*else if(condition)*

*statement;*

*else if(condition)*

*statement;*

*...*

*else*

*statement;*

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Example

**public** **class** MultipleIfPrograms {

**public** **static** **void** main (String ...args) {

**float** hindi=58, english=89, maths=90, sanskrit=70, socialScience = 50;

**float** totalMarks = 500;

**float** obtainedNumber = hindi+english+maths+sanskrit+socialScience;

**float** percent = (obtainedNumber / totalMarks)\*100;

System.***out***.println("Your percentage is --> "+percent);

**if**(percent >= 90 && percent <= 100) {

System.***out***.println("You got grade A");

}

**else** **if**(percent >=80 && percent < 90 ) {

System.***out***.println("You got grade B");

}

**else** **if**(percent >=60 && percent <80) {

System.***err***.println("You got grade C");

}

**else** **if**(percent>=40 && percent<60) {

System.***out***.println("You got grade D");

}

**else** {

System.***out***.println("You failed");

}

}

}

**o/p** Your percentage is --> 71.4

You got grade C

#### Nested if

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that

is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

syntax

*if (condition)*

*if(condition)* // this is nested if inside if and

*else* // this else is connected with just upper if

*else*// this else is connected is connected with starting if

example

**public** **class** Nested\_if {

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

**int** i = 10,j=15,k=20;

**if** (i < j) { // this is first if block

**if**(i<k) { // this is second if block or nested if block

System.***out***.println("I is less then both");

}

**else** {

System.***out***.println("I is only less then j ");

}

}

**else** {

System.***out***.println("I is greater then all");

}

}

}

**o/p** I is less then both

**note** : if above **if** block is true then execution is goes to the **inner if** and check **inner if** condition. if **inner if** condition get false then execution will go onto the else part of **inner if** (not to the outer one else part ) and execution of if is terminated after completely execution. here we provide next program of if.

syntax example

if(condition) { // first if

if(condition) { // second if / if inside if

if(condition) { // third if / if inside if inside if

}

}

}

else{

if(condition){

}}

Here we will provide a simple if program for testing nested if part

**int** a=2,b=3,c=4,d=5;

**if**(a<b) {

**if**(b>c) {

**if**(c<d) {

System.***out***.println("Yes we execute third if 3");

}

System.***out***.println("Yes we execute second if 2");

}

System.***out***.println("yes we execute first if 1");

}

**else** {

**if**(5<10) {

System.***out***.println("I first if part of else block 1");

}

System.***out***.println("I am else part of first if ");

}

**note**: if first condition is false then if will go into else part otherwise first condition is true then it will check second if condition and second condition is false then it will execute completer first condition and terminate but in case all the if condition is get true then it will execute all parts of if / nested if and cannot else part. Else part only when executed when first if condition gets false because else part is connected with first if not all of its.

### switch

The **switch** statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement: For versions of Java prior to JDK 7, *expression* must resolve to type **byte**, **short**, **int**, **char**, or an enumeration. Beginning with JDK 7, *expression* can also be of type **String**. Each value specified in the **case** statements must be a unique constant expression (such as a literal value). Duplicate **case** values are not allowed. The type of each value must be compatible with the type of *expression*. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken. The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the **entire switch** statement. This has the effect of **“jumping out”** of the switch.

Syntax

switch (expression){

case *value*1:

// statement or function calling

Break;

case *value2*:

// statement sequence

break;

...

case *valueN* :

// statement sequence

break;

default:

// default statement sequence

}

}

Difference between **Switch** and **if**

1. The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
2. No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
3. A **switch** statement is usually more efficient than a set of nested **if**s.

Example : 1 // switch using user input (Integer)

**public** **static** **void** main(String [] args) { // taking input with user program of switch

Scanner scanner = **new** Scanner(System.***in***); // scanner class to take input from user

System.***out***.println("Enter no of week Day to get Full Day Name");

**byte** inp = scanner.nextByte();

**switch**(inp) { // here we can user string also

**case** 1:

System.***out***.println("Monday");

**break**;

**case** 2:

System.***out***.println("Tuesday");

**break**;

**case** 3:

System.***out***.println("Wednesday");

**break**;

**case** 4:

System.***out***.println("Thursday");

**break**;

**case** 5:

System.***out***.println("Friday");

**break**;

**case** 6:

System.***out***.println("Saturday");

**break**;

**case** 7:

System.***out***.println("Sunday");

**break**;

**default**:

System.***out***.println("You Input invalid number ");

**break**;

}

}

o/p Enter no of week Day to get Full Day Name

7 // seven is user input to get week day

Sunday

example : 2 // switch using for-loop

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

**for**(**int** inp =1;inp<=8;inp++) {

System.***out***.println();

**switch**(inp) {

**case** 1:

System.***out***.println("Monday");

**break**;

**case** 2:

System.***out***.println("Tuesday");

**break**;

**case** 3:

System.***out***.println("Wednesday");

**break**;

**case** 4:

System.***out***.println("Thursday");

**break**;

**case** 5:

System.***out***.println("Friday");

**break**;

**case** 6:

System.***out***.println("Saturday");

**break**;

**case** 7:

System.***out***.println("Sunday");

**break**;

**default**:

System.***out***.println("You Input invalid number ");

**break**;

}

}}

example : 3 // switch using string input and also uses multiple cases at one to get perfect result

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

System.***out***.println("Enter your season name To get Detail of them\n"

+ "1)Winter\n"

+ "2)Summer\n"

+ "3)Monsoon\n ");

Scanner sc = **new** Scanner(System.***in***);

String inpSeason = sc.next();

**switch** (inpSeason) {

**case** "winter":

**case** "Winter":

**case** "WINTER":

System.***out***.println("Winter temperature in between (5 - 15°C) ");

**break**;

**case** "summer":

**case** "Summer":

**case** "SUMMER":

System.***out***.println("Summer temperature in between (40 - 45°C)");

**break**;

**case** "monsoon":

**case** "Monsoon":

**case** "MONSOON":

System.***out***.println("Monsoon Temperature in between (20 - 25°C)");

**break**;

**default**:

System.***out***.println("Invalid Input");

**break**;

}

}}

**note** :

1. if you by mistake forgot to use break statement then all conditions will execute one by one after your selected condition.
2. In case of string switch statement want **constant string expression required** if you not provide it using variable then it will throw an exception.
3. If you provide input to switch statically then you must have to add **final keyword** before variable name to avoid an exception.

example : 4 // illustrating an example of switch without final in string input type

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

String name = "Deepak";

String name1 = "Amit";

String name2 = "Rohit";

String name3 = "Deepak";

String name4 = "Sumit";

**switch** (name) {

**case** name1:

System.***out***.println("Roll no 201");

**break**;

**case** name2:

System.***out***.println("Roll no 202");

**break**;

**case** name3:

System.***out***.println("Roll no 203");

**break**;

**case** name4:

System.***out***.println("Roll no 204");

**break**;

**default**:

System.***out***.println("Invalid Name");

**break**;

}

}

It will throw an exception

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

case expressions must be constant expressions

case expressions must be constant expressions

case expressions must be constant expressions

case expressions must be constant expressions

at MyPracticeOfJava/myPracticeProgramsAWT.Switch.main(Switch.java:14)

To avoid this exception, you must have to declare all the variable final like this and you will get your correct output

**final** String name1 = "Amit";

**final** String name2 = "Rohit";

**final** String name3 = "Deepak";

**final** String name4 = "Sumit";

example : 5 // this one is advanced switch

System.***out***.println("Enter no to identify");

**byte** input = sc.nextByte();

**switch**(input) {

**case** 1-> System.***out***.println("This is first statement");

**case** 2-> System.***out***.println("This is second statement");

**case** 3-> System.***out***.println("This is third statement");

**case** 4-> System.***out***.println("This is fourth statement");

**case** 5-> System.***out***.println("This is fifth statement");

**default** -> System.***out***.println("Invalid Input");

}

}

**note** : in this form of **switch** we do not have need to insert **break** statement. The case of in advance switch only execute once.

**Exercise**

1. write a program to find the greater number between 2 numbers
2. write a program to find the greater number between 3 numbers
3. write a program to check for leap year
4. write a program to create simple calculator using switch statement
5. write a program to find the greater number between 2 numbers using ternary operator
6. write a program to swap the 2 numbers
7. write a program to swap the 2 numbers without using third variable
8. write a program that takes number from the user and generate an integer between 1 and 7. it displays the weekday name using only if-else.
9. write a java program to find the number of days in a month using switch and also find the year is leap year or not Hint (logics include in case 2:)
10. write a java program that requires the user to enter a single character from the alphabet. Print Vowel or Consonant, depending on user input. if the user input is not a letter (between a and z or A and Z), or is a string of length > 1, print an error message.
11. write a java program to display the multiplication table of a give integer.

#### if test cases

1. if(condition) // not valid you must have to use (semicolon ;) after single if without braces or statement
2. if( var1 > var2) ; // valid if
3. if (condition ) // valid until more then one condition is present under if without curly braces

// statement

else

// statement

1. if (condition) // valid

// statement

else if (condition)

// statement

else

// statement

5. else // not valid throw exception else without if

// statement

6. if (condition) // valid

//statement

else{

if(condition){

// statement area of inner if of else part

}

}

7. if (condition){ //valid

if(condition){

// statement

}

}

else{

//statement

}

8. if (condition){ //valid

if(condition){

// statement

}

else{

//statement

if(condition)

//statement

}

else{

//statement

}

}

}

else{

if(

//statement

} .etc

9. // this one is special if test case in which we use method in if condition block

**public** **class** TestingIf {

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

// making object of testingIf class

TestingIf objif = **new** TestingIf();

// email1 & email2 is local variable of string type which can store string

String email1 ="abc123@gmail.com";

String email2 ="xyz456@yahoo.com";

// we can also use method in if condition but method must have to return value in boolean form.

**if**(objif.checkingMail(email2)) {

System.***out***.println("Google Verified Account ");

}

**else** {

System.***out***.println("Non - Google Account ");

}

}

// this method is checking the email which we get it in variable is google mail service or other mail service

**boolean** checkingMail(String variable) {

**if**(variable.endsWith("@gmail.com")) {

**return** **true**;

}

**else**

**return** **false**;

}

}

o/p Non – Google Account

### 2. Iteration / Looping Statements

**for, while, do-while, for-each loop**

#### for

You were introduced to a simple form of the **for** loop. As you will see, it is a powerful and versatile construct. There are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the newer **“foreach”** form, added by JDK 5. Both types of **for** loops are discussed here, beginning with the traditional form. Generally, we use loops for similar repetitive work. Generally, we used **for** loop when we know the number of iterations.

Syntax

**For(expression1; expression2; expression3){**

**// statement**

**}**

1. expression1 should be any java valid statement but preferred is declaration and initialization.
2. expression2 should always be conditional statement which should return boolean value.
3. expression3 should be any java valid statement but preferred is increment or decrement.

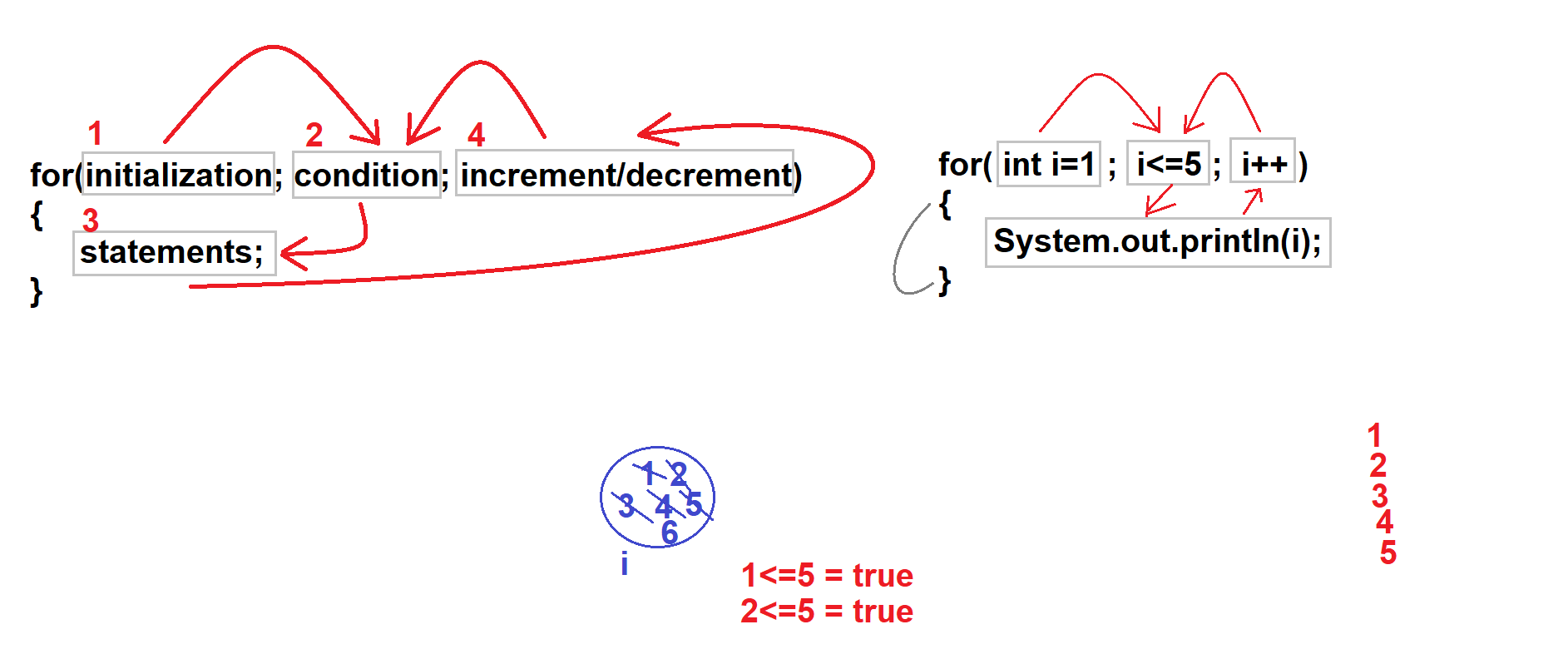
// forloop simple form

**for(initialization; condition; iteration){**

// this is the body party of the for loop

**}**

**Diagram**



Example // here we print the table of 2 using for loop

**for**(**int** i=0; i<=10; i++) { // here we simply initialize variable i inside for loop but you can done outside also.

System.***out***.println("2 \* "+i+" = "+2\*i);

}

##### correct cases

**for**(**int** i =0; i < 5; i++){

System.out.println(i);

}

**for**(**int** i =0; i < 5; i++){

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

}

System.out.println("HI");

op -> 1 2 3 4 5 HI

**int** i =0;

**for**( ; i < 5; i++){

System.out.println(i);

}

**int** i =0;

**for**(System.out.print("Hello Forloop"); i < 5; i++){

System.out.println(i+” ”);

}

o/p Hello Forloop 0 1 2 3 4

// using two and more condition in a single for loop

**for**(**int** i = 0, j = 10; i < 10 && j > 10 ; i++, j--){

System.out.println(i+" :: "+j);

}

/\* iterating more then two condition at a single time or we can say that multiple declaration at a time in single loop (remember more you declare more loop get complex) that’s why we not use these types of loops in general as much. \*/

**for**(**int** i =0,j = 100,k=200,l=300; i<=100 && j<=200 & k<=300 && l<=400; i++,j++,k++,l++){

System.out.println(i+" "+j+" "+k+" "+l);

}

##### Incorrect cases / infinite looping

**for**(**int** i = 0, j = 10; i < 10 , j > 10 ; i++, j--){

System.out.println(i+" :: "+j);

}

// in for loop only one declarative statement is allow and in this for loop more than two declarative statement is present. the correct form is when we use one declarative statement and provide multiple variables.

**for**(**int** i = 0, **int** j = 10; i < 10 && j > 10 ; i++, j--){

System.out.println(i+" :: "+j);

}

// this for loop goes into infinite stage because we didn't provide any condition and in this case java automatic provide true in condition part that’s why for-loop didn't know when to stop the loop. i will increment in each iteration and print the value of i till we manually terminate the program.

**for**(**int** i =0; ; i++){

System.out.println(i);

}

// In this for loop we didn't provide increment or decrement part of the loop and in this case loop condition cannot be upgraded or down-grade and loop check condition it will get true condition every time. in this case for loop always print 0 till we manually terminate the program.

**for**(**int** i =0; i< 5;){

System.out.println(i);

}

//same as above

**for**(**int** i =0 ; ;){

System.out.println(i);

}

//it will also goes into infinite condition

**for**(**int** i =0 ; ;System.out.println("Hello Java")){

System.out.println(i);

}

o/p Hello Java

0

Hello Java

0

// Infinite Times

// Stuck into infinite times and print always HI

**for**(;;){

// in increment we not provide any part

// in condition block it will automatically place true because we not provide any thing

// in increment and decrement part we left blank thats why it ignored

System.out.println("HI");

}

// This program will not compile because we will provide constant expression in condition part and compiler knows this for loop stuck into loop that’s why it will throw an error (Unreachable statement (it is compile time exception))

**for**(**int** i =0; **true**; i++){

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

}

System.out.println("HI");

// same as above compiler will automatically place true in condition part and also provide compile time exception (Unreachable statement)

**for**(**int** i =0; ; i++){

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

}

System.out.println("HI");

// it will compile and also run but do nothing

**for**(;;){

}

// it will also compile and run infinite times because only one statement is present without curly braces

**for**(;;)

System.out.println("hi");

// it will provide compile time error (illegal start of statement if we nothing write anything under for loop without curly braces

**for**(;;)

// in this case when you put semicolon end of the for loop then it will execute infinite times but nothing will print

**for**(;;) ;

// this is the not a write syntax of for-loop that’s why it throw an error (Illegal start of expression)

**for**(;;;)

// Stuck into infinite loop and print infinite times of hi

**for**(;;){

System.out.println("hi");

//In this example, the for loop continues to run until the boolean variable done is set to true. It does not test the value of i.

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

**boolean** terminate= **false**;

**for**(**int** i =0; !terminate; i++) {

System.***out***.println(i);

**if**(i == 100) {

terminate = **true**;

}

}

}

// here is the another example of iterating for loop using ! boolean condition

**int** i =0;

**boolean** check = **false**;

**for**(; !check ;) {

System.***out***.println(i);

**if**(i == 50) {

check = **true**;

}

i++;

}

#### for-each

A second form of **for** implements a “for-each” style loop. As you may know, contemporary language theory has embraced the for-each concept, and it has become a standard feature that programmers have come to expect. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. In Java, the for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is shown here:

**Syntax** for (*type itr-var* : *collection*) *statement-block*

***Example*** *for(int var : array\_Name){*

*Sytem.out.println(var);*

*}*

*Difference between for and enhanced for lop*

*Note : here Arrays Is just used only to explaining concept of for each loop. further we will surely discuss array in detail.*

*Simple for*

**int** arr [] = {1,4,7,9,2,6,12,87,23,10};

**int** sum =0;

**for**(**int** i =0; i< arr.length;i++) {

sum += arr[i];

}

System.***out***.println("Your sum is --> "+sum);

*Enhanced for*

**int** arr [] = {1,4,7,9,2,6,12,87,23,10};

**int** sum =0;

**for**(**int** val : arr) {

sum += val;

}

System.***out***.println("Your sum is --> "+sum);

As this output shows, the for-each style **for** automatically cycles through an array in sequence from the lowest index to the highest. Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement.

**Iterating in Multi-Dimensional**

The enhanced version of the **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. To understand the implications of this, consider the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row-order, from first to last.

*Using simple for loop*

*program sum of each row of 2-D array using simple for*

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

**int** [][]arr = {{10,20,42,30,15},

{41,51,61,62,63},

{71,81,91,42,99},

{55,31,21,41,77}};

**int** sumOfRow = 0,sumOfCompleteArray = 0;

**for**(**int** i =0;i< arr.length;i++) {

**for**(**int** j=0;j<arr.length;j++) {

System.***out***.print(arr[i][j]+" ");

sumOfRow += arr[i][j];

sumOfCompleteArray =sumOfCompleteArray+ arr[i][j];

**if**(j == arr.length-1) {

System.***out***.println(" = "+sumOfRow);

sumOfRow = 0;

}

}

System.***out***.println();

}

System.***out***.println("Complete 2-D Array Sum = "+sumOfCompleteArray);

}

**o/p** 10 20 42 30 = 102

41 51 61 62 = 215

71 81 91 42 = 285

55 31 21 41 = 148

Complete 2-D Array Sum = 750

*Making another same program using for each loop with taking random value using random class and get sum and average of complete array*

**import** java.util.Random;

**import** java.util.Scanner;

**public** **class** TwoD\_Array\_Program {

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

Random random = **new** Random();

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter rows of Array");

**int** rows = sc.nextInt();

System.***out***.println("Enter column of Array");

**int** column = sc.nextInt();

System.***out***.println("Enter lowest value of Array");

**int** lowest = sc.nextInt();

System.***out***.println("Enter highest value of Array");

**int** highest = sc.nextInt();

**int** [][]arr = **new** **int** [rows][column];

**int** sumOfCompleteArray = 0,averageOfArray = 0;

// Initializing an array

**for**(**int** i =0; i<arr.length; i++) {

**for**(**int** j =0; j<arr[i].length; j++) {

arr[i][j] = random.nextInt(lowest,highest);

}

}

**int** averageCount = 0;

// logic part using for each loop

**for**(**int** first[] : arr) {

**for**(**int** second : first) {

System.***out***.print(second + " ");

sumOfCompleteArray += second;

averageCount ++;

}

System.***out***.println();

}

System.***out***.println("Sum of Complete array is = "+sumOfCompleteArray );

System.***out***.println("Average of Completer array is = "+(sumOfCompleteArray/averageCount));

}

}

Enter rows of Array

10

Enter column of Array

10

Enter lowest value of Array

10

Enter highest value of Array

100

31 64 67 41 85 91 23 98 74 13

37 49 12 76 78 96 93 81 23 18

23 28 44 95 74 74 64 97 81 84

36 29 51 47 49 60 53 26 94 43

32 86 58 24 23 71 95 75 76 81

33 56 52 58 43 36 89 28 61 35

20 83 98 71 33 52 78 10 73 78

67 68 98 47 46 11 45 98 62 54

77 83 17 70 87 95 72 73 67 15

25 25 23 49 98 25 64 98 31 64

Sum of Complete array is = 5764

Average of Completer array is = 57

#### While

The **while** loop is Java’s most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

while(*condition*) {

// body of loop

}

The *condition* can be any **Boolean** expression. The body of the loop will be executed as long as the conditional expression is **true**. When *condition* becomes **false**, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

When we don’t know how many time we have to execute the loop then we have to use while loop.

**Cases :**

1. Its compulsory to provide the condition in while loop.

2. If while loop is infinite and we provide any expression after while loop then it will provide compile time error saying **"Unreachable statement**".

General program of while loop

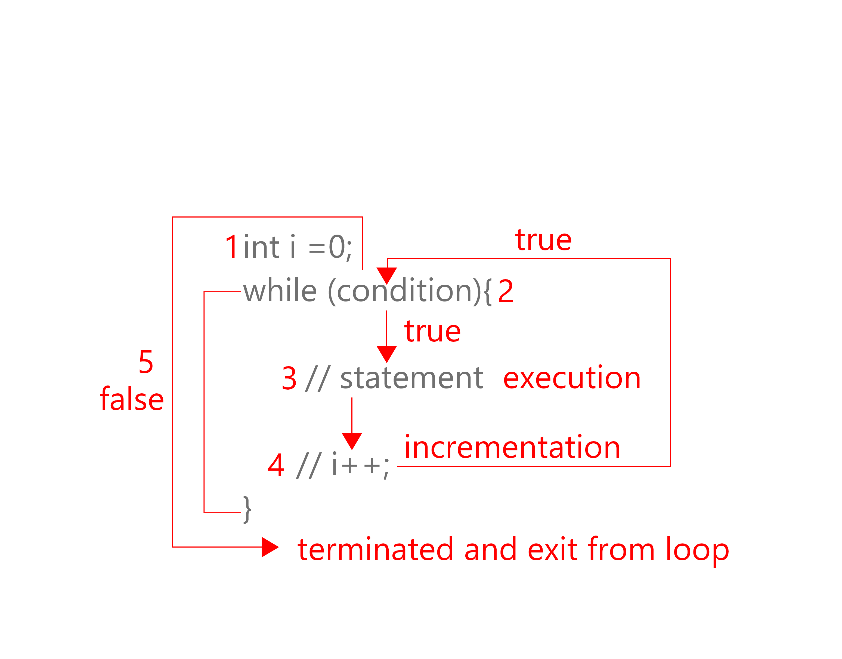
**int** i =0; // Initialization part

**while**(i<5) { // conditional checking part (also checking in boolean)

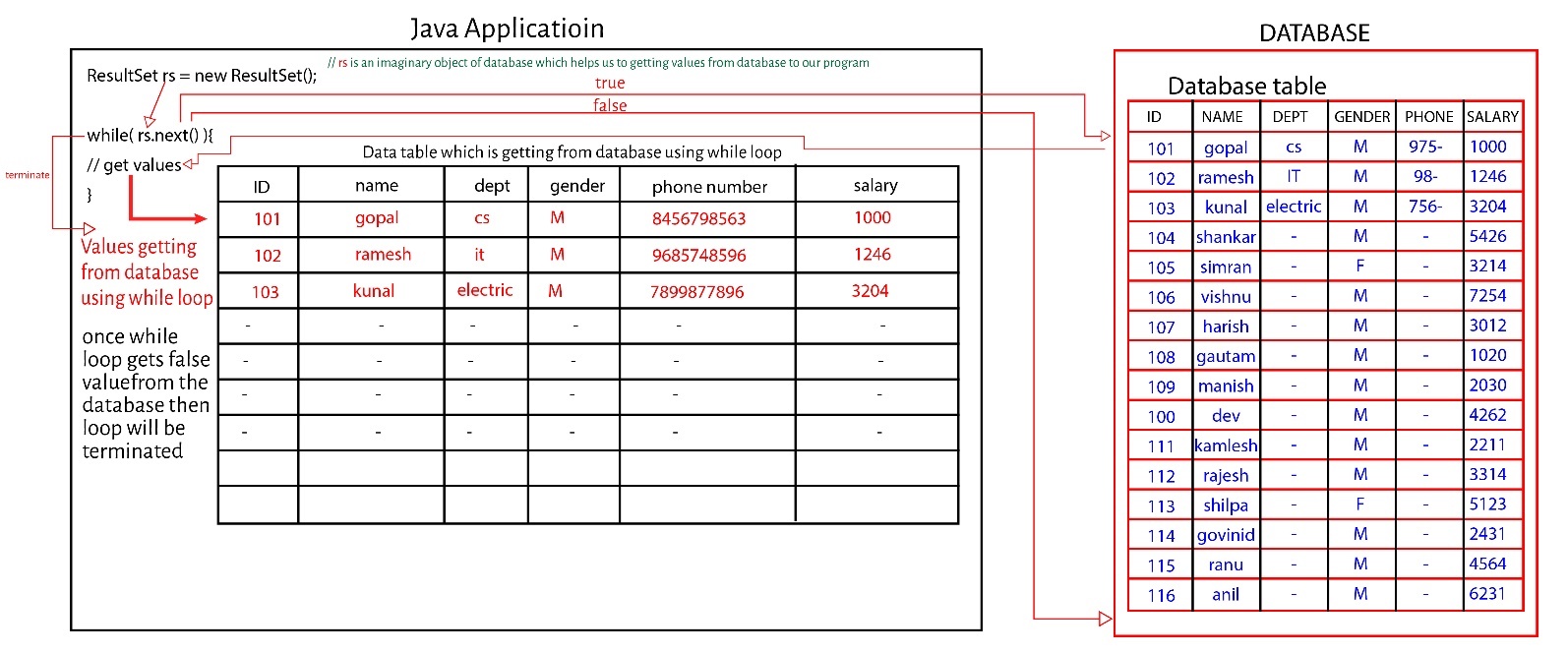
System.***out***.println(i); // statement part

i++; // increment or decrement part

}



// this diagram is an simplest and real life based example of while loop to getting data from database



##### Correct case

// this is the valid while loop case and it will also print 0 to 4 digits

**int** i =0;

**while**(i<=5){

System.out.println(i);

i++;

}

##### incorrect cases

//in this case condition of while loop is always true so it always print hi infinite times because in this while no termination or break statement id defined.

**while**(**true**){

System.out.println("hi");

}

// this can generate an error (Illegal start of expression)

**while**(){

System.out.println();

}

// this will throw an compile time error called (unreachable statement

**while**(**true**){

System.out.println("hi");

}

System.out.println("Hello");

// this loop also goes into infinite loop because we didn't include increment condition in while loop and value of i did not be upgraded in its life span. the value of i will always be lesser then 5 so it will print always 0.

**int** i =0;

**while**(i<5){

System.out.println(i);

}

// illegal start of expression

**int** i =1;

**while**(i<5)

// compile and run or goes into infinite loop

**while**(i<5);

// same as above and nothing display

**while**(i<5);

System.out.println("hi");

**Points to remember**

1. its compulsory to provide the condition in **while** loop.
2. **if** **while** loop is infinite and we provide any expression after **while** loop then it will provide compile time error saying "Unreachable statement".

while loop program to count how many digits in given integer number

**public** **class** IntegerCount {

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

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter value to get no of integer in you digit");

**long** val = sc.nextLong();

**int** count = 0;

**while**(val>0){

val = val/10;

System.***out***.println(val);

count ++;

}

System.***out***.println("Integer has -> "+count+" values");

}

}

Enter value to get no of integer in you digit

4567898456

456789845

45678984

4567898

456789

45678

4567

456

45

4

0

Integer has -> 10 values

Java program to get factorial value of integer number

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

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.println("Enter integer to get factorial value of it");

**int** num = scanner.nextInt();

**double** fact = 1;

**int** i =1;

**while**(i<=num) {

fact \*= i;

i++;

}

System.***out***.println("Factorial value of "+num+" is = "+fact);

}

Enter integer to get factorial value of it

5

Factorial value of 5 is = 120.0

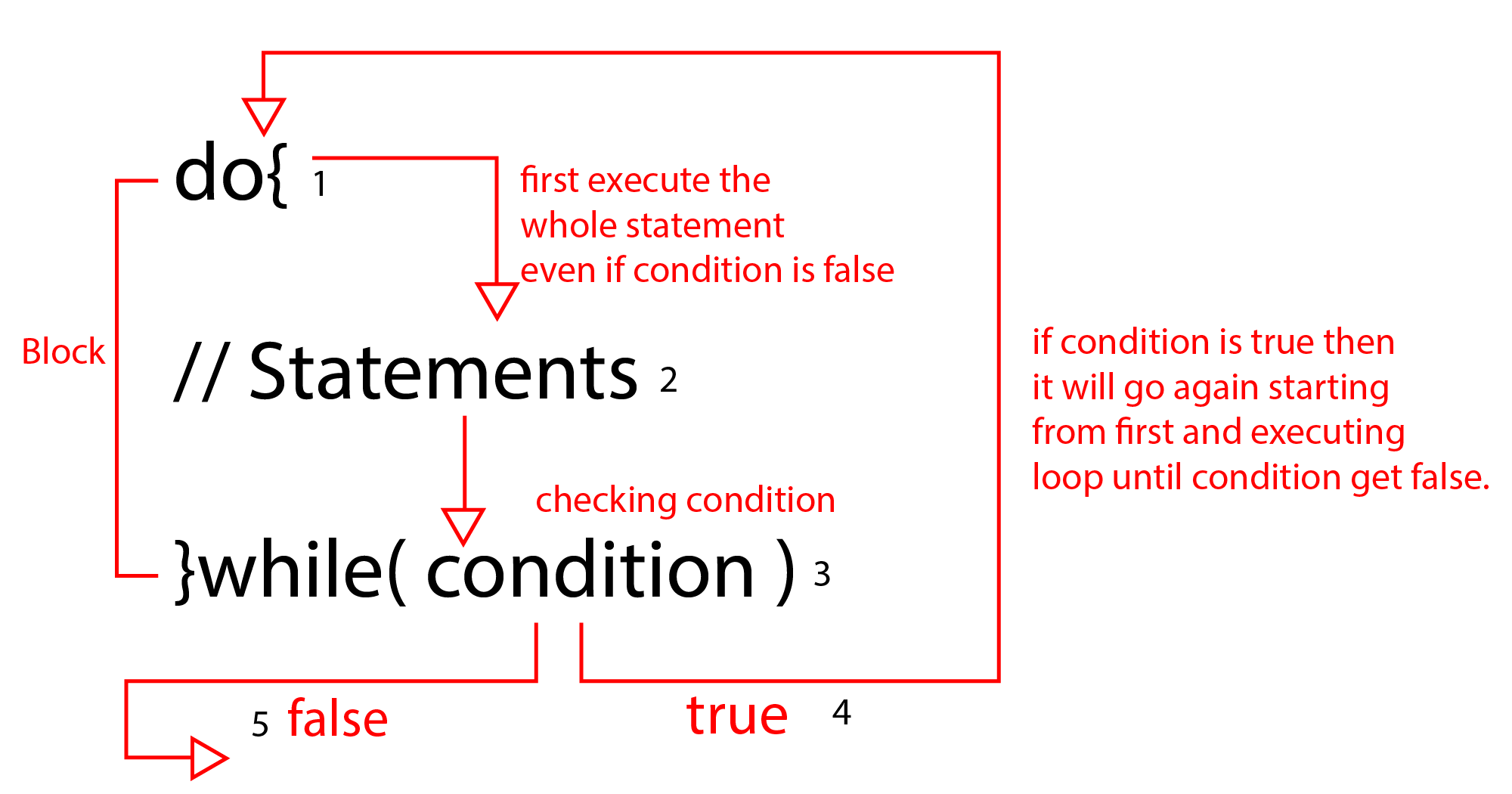
#### Do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, **because its conditional expression is at the bottom of the loop.** Its general form is

do {

// body of loop

} while (*condition*);



Program to reverse an integer number

**public** **class** ReverseAnIntegerNumber {

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

**int** num = 546738, reverse=0;;

**while**(num>0){

**int** temp =0;

temp = num %10;

reverse = reverse\*10+temp;

num = num/10;

System.***out***.println(reverse);

}

}

}

Actual given number is – 546738

Reversed number is – 837645

8

83

837

8376

83764

837645

Write a program to make a simple calculator using do-while loop

**import** java.util.Scanner;

**public** **class** SimpleCalculator {

// Main function of program

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

// Making object of scanner class to get an input from keyboard (object creation introduced in OOPs)

Scanner sc = **new** Scanner(System.***in***);

**int** check =0;

**do** { // all the code placed in do while loop to execute at least one time.

System.***out***.println("Enter your first number");

**int** first = sc.nextInt();

System.***out***.println("Enter your second number");

**int** second = sc.nextInt();

// displaying all the option which perform mathematically operation with our given number

System.***out***.println("What did you want to perform" +

"\n1) Addition" +

"\n2) Subtraction" +

"\n3) Multiplication" +

"\n4) Division " +

"\n5) Get Remainder");

**int** option =sc.nextInt(); // taking input what task is performed by out programs

// this is advance switch here we have no need to mention break statement (A simpler form of switch)

**switch**(option){

**case** 1 -> System.***out***.println(first+second);

**case** 2 -> System.***out***.println(first-second);

**case** 3 -> System.***out***.println(first\*second);

**case** 4 -> System.***out***.println(first/second);

**case** 5 -> System.***out***.println(first%second);

**default** -> System.***out***.println("Invalid input (choose between 1 to 5 only");

}

// this line is taking user input to confirm that to run program again or stop it

System.***out***.println("Are you want to calculate again" +

"\n0 yes" +

"\n1 no");

check = sc.nextInt();

System.***out***.println("-------------------------------------------------");

// this while is checking actual condition to process program further or terminate here

}**while**(check == 0);

}

}

Enter your first number

45

Enter your second number

23

What did you want to perform

1) Addition

2) Subtraction

3) Multiplication

4) Division

5) Get Remainder

1

68

Are you want to calculate again

0 yes

1 no

1 // pressing one is actually re run the program and asking the use input to perform further task again

### 3. Jump Statements

1. Jump statements are also known as **Transfer Statements.**
2. Jump statements are used to skip some statements inside the **loop** or used to **terminate** the loop immediately without checking the condition. Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here.

**break, continue, return**

#### break

In Java, the **break** statement has three uses.

1. First, as you have seen, it terminates a statement sequence in a **switch** statement.
2. Second, it can be used to exit a loop. Whenever we use break statement, then loop gets terminated without checking the condition and first statement after the loop will be executed
3. Third, it can be used as a **“civilized / replaced ”** form of goto.

##### Correct cases

// it will work because we use output statement after break block (not write immediately just after break)

**for**(**int** i =1; i<=5 ;i++){

**if**(i == 3){

System.out.println("Hi");

**break**;

//System.out.println("Hello");

}

System.out.println("Hello");

}

// the outer most for loop will run 3 times but the inner loop also run 3 time because we use break statement. once value of j is 3 break terminate the inner loop and passes control to the outer loop for further execution. here we also provide the output of the program.

**for**(**int** i =1; i<=3 ; i++){

System.out.println("Hi i = "+i);

**for**(**int** j =1; j<5; j++){

System.out.println("Hello j = "+j);

**if**(j == 3){

**break**;

}

}

}

output --

Hi i = 1

Hello j = 1

Hello j = 2

Hello j = 3

Hi i = 2

Hello j = 1

Hello j = 2

Hello j = 3

Hi i = 3

Hello j = 1

Hello j = 2

Hello j = 3

// if we use two loops inner and outer and we want to terminate both loop in once using one break statement then we have to use labels to jump the cursor from innermost loop to outermost loop.

outer: // this is called label (the outer one label)

**for**(**int** i =1; i<=3 ; i++){

System.out.println("Hi i = "+i);

inner: // the inner one label

**for**(**int** j =1; j<5; j++){

System.out.println("Hello j = "+j);

**if**(j == 3){

**break** outer; // we provide outer label name so out whole loop terminate in once if condition is match.

}

}

}

##### wrong cases

//there should not be statement after the break statement. it would throw and compile time error called **(Unreachable statement)**

**for**(**int** i =1; i<=5 ;i++){

**if**(i == 3){

System.out.println("Hi");

**break**;

System.out.println("Hello");

}

}

//Keep in mind that you cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile:

outer:

**for**(**int** i =0;i<10 ;i++) {

System.***out***.println(i);

}

**for**(**int** j =0;j< 10 ;j++) {

System.***out***.println(j);

**if**(j ==3) {

**break** outer;

}

}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The label outer is missing

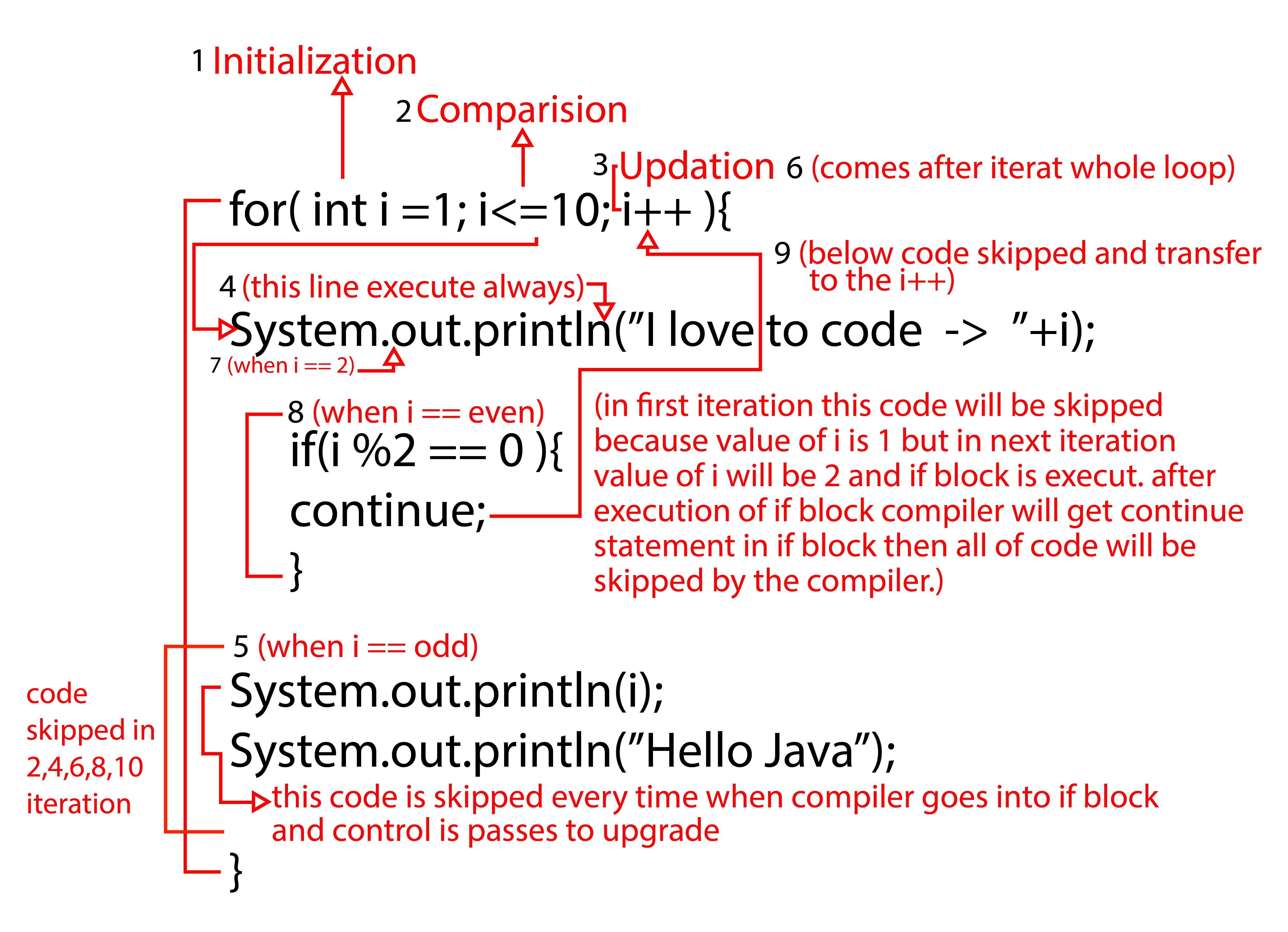
#### continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the

body of the loop, to the loop’s end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

1. there should not be any statement after continue statement.
2. if there are nested for loop and we want to skip the current iteration for outer for loop the n we can use labelled continue statement

**Here we present the very detailed diagram of continue block**

****

Program with continue statement

**case 1:**

// print statement is present under the if block

**for**(**int** i =1;i<=10;i++) {

**if**(i%2 ==0) {

**continue**;

}

System.***out***.print(i+” ”);

}

1 3 5 7 9

**case 2:**

// print statement about the if block

**for**(**int** i =1;i<=10;i++) {

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

**if**(i%2 ==0) {

**continue**;

}

}

1 2 3 4 5 6 7 8 9 10

**case 3:**

**for**(**int** i =1;i<=10;i++) {

**if**(i % 3 ==0 || i%2 ==0) {

**continue**;

}

System.***out***.print(i+” ”);

}

1 5 7

**case 4:**

**for**(**int** i =1;i<=10;i++) {

**if**(i % 3 ==0 || i%2 ==0) {

**continue**;

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

}

}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unreachable code

**case 5:**

**for**(**int** i =1;i<=5;i++) {

System.***out***.print("i -> "+i+", ");

**for**(**int** j =1;j<=3;j++) {

**if**(j==2) {

**continue**;

}

System.***out***.print("j -> "+j+", ");

}

System.***out***.println();

}

i -> 1, j -> 1, j -> 3,

i -> 2, j -> 1, j -> 3,

i -> 3, j -> 1, j -> 3,

i -> 4, j -> 1, j -> 3,

i -> 5, j -> 1, j -> 3

case 6:

// using labels with continue

outer: // this is outer label

**for**(**int** i =1;i<=5;i++) {

inner: // this is inner label

System.***out***.println("i -> "+i+", ");

**for**(**int** j =1;j<=3;j++) {

**if**(j==2) {

**continue** outer;

}

System.***out***.print("j -> "+j+", ");

}

System.***out***.println();

}

i -> 1,

j -> 1, i -> 2,

j -> 1, i -> 3,

j -> 1, i -> 4,

j -> 1, i -> 5,

j -> 1,

case 7:

// program in which using continue to demonstrate the working of continue statement

**for**(**int** i =0; i<=10;i++) {

**for**(**int** j =0; j<=10;j++) {

**if**(j>i) {

**continue**;

}

System.***out***.print(i\*j +" ");

}

System.***out***.println();

}

0

0 1

0 2 4

0 3 6 9

0 4 8 12 16

0 5 10 15 20 25

0 6 12 18 24 30 36

0 7 14 21 28 35 42 49

0 8 16 24 32 40 48 56 64

0 9 18 27 36 45 54 63 72 81

0 10 20 30 40 50 60 70 80 90 100

#### return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in OOPS, a brief look at **return** is presented here.

At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed.

1. In case of void method **return** type, we can use **empty return** statement.
2. **Return** statement must be the **last statement** in case of return method.

**case 1:**

// simply print easy java when calling m1 method and it return string

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

System.***out***.println(obj1.m1());

}

String m1() {

**return** "Easy Java";

}}

Easy Java

**case 2:**

// this program return 10 because method m2 return value 10 when it’s calling

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

System.***out***.println(obj1.m2());

}

**int** m2() {

**return** 10;

}}

10

**case 3:**

// if method return type is void and we can still return any value from It in print statement. then it will throw an exception.

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

System.***out***.println(obj1.m2());

}

**static** **void** m3() {

**return** 20;

}}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The method println(boolean) in the type PrintStream is not applicable for the arguments (void)

**case 4:**

// directly calling method without any print statement then program will throw an actual error but in just above program every thing is same except calling a method in print statement and the output we get is completely different in comparison of case 3: to case 4: case 4 shows actual error but case 3 shows printStream error both are quit different error.

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

obj1.*m3*();

}

**void** m3() {

**return** 20;

}}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Void methods cannot return a value

**case 5:**

// in void method we cannot use return statement but we can use print statement to display our result

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

obj1.m4();

}

**void** m4() {

System.***out***.println("I am not return anything");

}}

I am not return anything

**case 6:**

// here we can also use empty return statement in our program but we not get any benefit of this programming

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

obj1.m4();

}

**void** m4() {

System.***out***.println("Happy coder");

**return;**

}}

Happy coder

**case 7:**

// if our method return type is int and we return an empty value then it will throw an exception

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

ReturnProgramCases obj1 = **new** ReturnProgramCases();

obj1.*m1*();

}

**int** m1() {

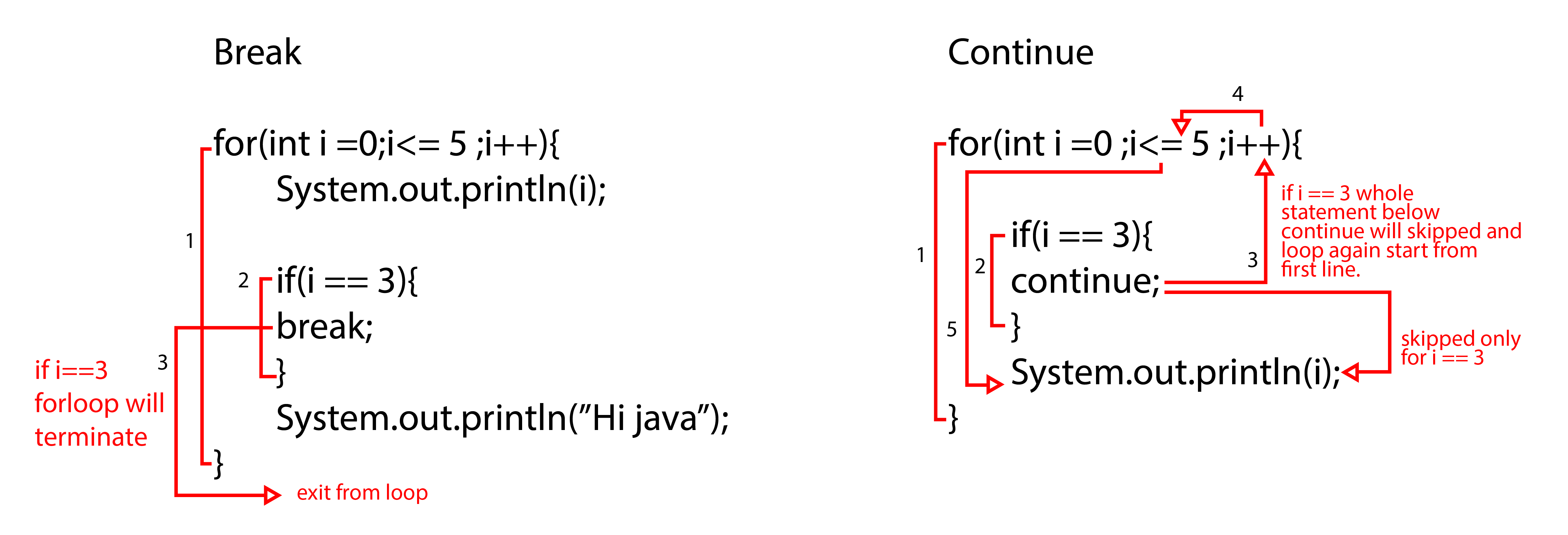
**return**;

}}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

This method must return a result of type int

**Difference between break and continue using diagram**



## types of conditional expressions

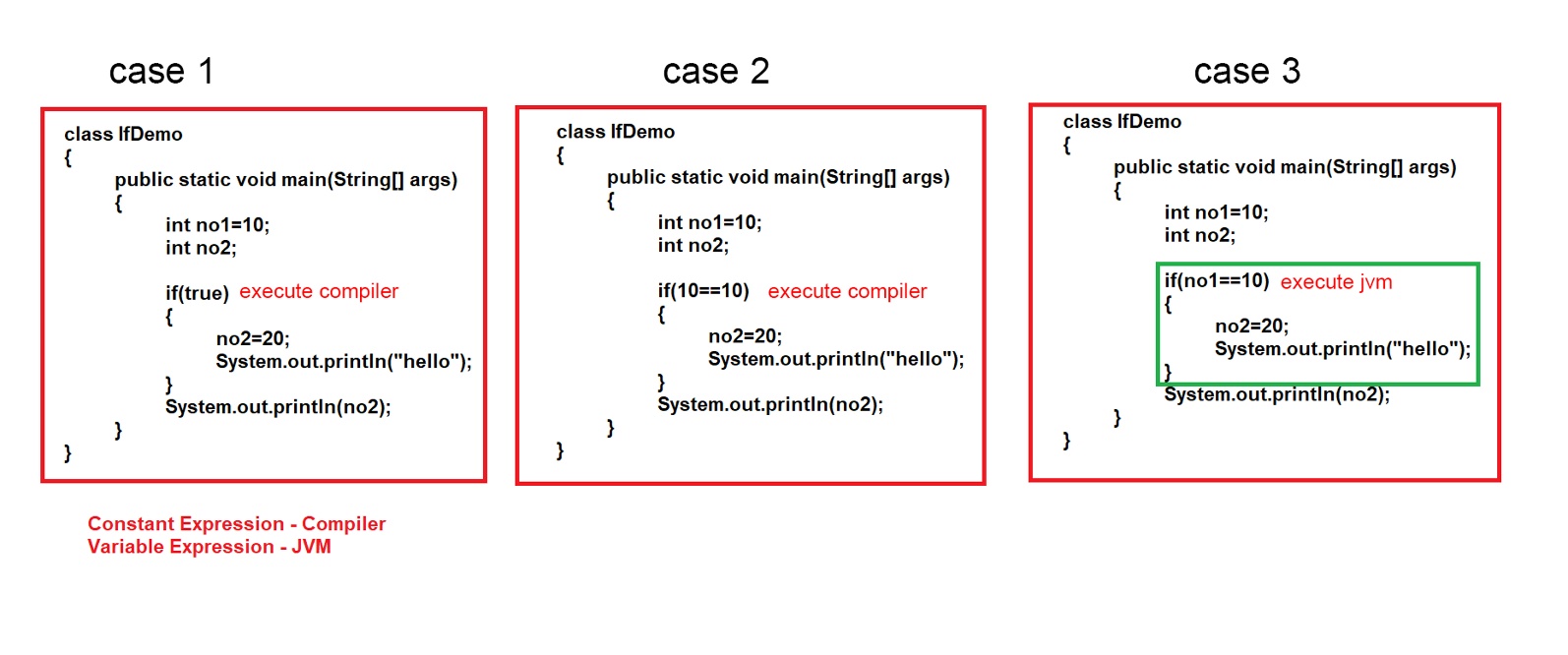
There are 2 types of conditional expressions

**1. Constant expression**

* + - * 1. These are those expressions in which we provide only constant values or final variables.
        2. Constant expressions are evaluated by **"Compiler".**

**2. Variable Expression**

* + - * 1. Variable expression are those in which we provide the variables.
        2. Variable expressions are evaluated by **"JVM".**



**case 1:** this program can run fine because we use constant expression in if block. constant expression is treated by the compiler In if part, boolean value is always true and compiler knows that the no 2 is present in if block is always initialized and used outside in print statement successfully.

**case 2:** this program will also work like previously one because if block contains 10 == 10 and it is constant expression that's why no 2 which present in if block is initialized with 20 and outside of if block no 2 will be printed. java compiler knows the condition present in if block is always true.

**case 3:** case 3 will generate an error because we use variable expression in if block and the value of no1 will different in different scenarios boolean result of if block is not always be true and it executed by jvm that’s why it throw an error.

**Case 1 program**

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

**int** a =10;

**int** b;

**if**(**true**) { // constant expression in if

b=20;

System.***out***.println("Value of A "+a);

}

System.***out***.println("Value of B "+b);

}

Value of A 10

Value of B 20

**Case 2 program**

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

**int** a =10;

**int** b;

**if**(10 == 10) { // constant expression in if

b=20;

System.***out***.println("Value of A "+a);

}

System.***out***.println("Value of B "+b);

}

Value of A 10

Value of B 20

**Case 3 program**

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

**int** a =10;

**int** b;

**if**(a == 10) { // variable expression in if ( value will be different (change manually) in each execution of program )

b=20;

System.***out***.println("Value of A "+a);

}

System.***out***.println("Value of B "+b);

}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The local variable b may not have been initialized

OOP’s (Object Oriented Programming Paradigm

Chapter 4

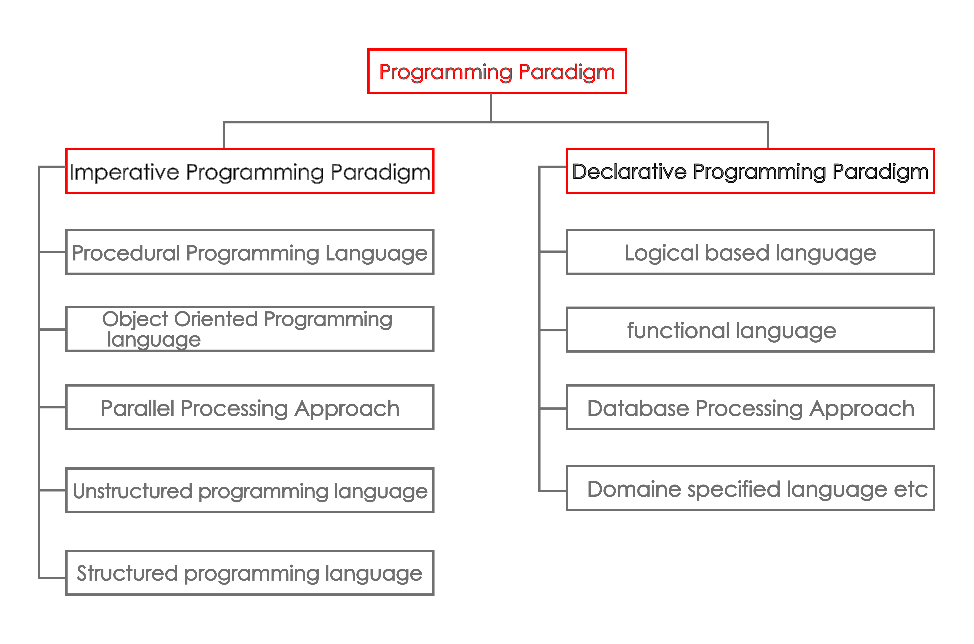
# what is Programming PARADIGM?

**Paradigm** can also be termed as method to solve some problem or do some tasks. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfil each and every demand.

1. Programming paradigm is a way or an approach to solve any problem or to achieve any task using any programming languages.
2. Programming paradigm are a way to classify programming languages based on their features.
3. There are 2 classifications of programming paradigm

1. Imperative Programming Paradigm

2. Declarative Programming Paradigm



## Difference b/w Imperative and Declarative

|  |  |  |
| --- | --- | --- |
| Features | Imperative | Declarative |
| **Programming style** | We have to specify step by step every task | We have to define he problem to achieve the task |
| **Task** | Users makes the decision and command to the compiler | Allows compiler to make decisions |
| **Smart** | User is more smart as compared to system | System is more smart as compared to user |
| **Programming focus** | “What” | “How” |
| **Primary flow control** | Loops, conditional, functions/methods etc | Functions calls (including recursion) |
| **Examples** | FORTRAN, Assembly languages, COBOL, C,C++,JAVA, Python etc… | SQL, Haskell, Prolog etc. |

## Structured vs unstructured programming

**Unstructured Programming Paradigm**

1. These are the first programming language categories that was introduced at starting point of computers.
2. For examples FORTRAN, COBOL, BASIC etc.
3. In these languages there was not fixed structure or way to achieve the task or to solve the problem.
4. In this type we use mnemonic codes.
5. In this part flow control was achieved by "goto" statement.
6. Hard to learn and difficult to achieve any task because number of lines of code were increased.

**Structured Programming Paradigm**

1. These were introduced after unstructured programming paradigm and used till now.
2. For examples PASCAL, C etc.
3. These have fixed structure to achieve any task.
4. These don’t use any mnemonic codes which makes this language easier (high level language syntax is used).
5. In this part a lot of flow control statements were introduced for example if, else, for, while etc.
6. Easier to learn and easy to achieve any task as there were improved code.
7. Structured programming paradigm have a lot of issues like less modularity, abstraction was not good, less security, less shareability, less code reusability and because of these reasons OOP's was introduced.

**Procedure Programming Paradigm**

1. In POP task can be divided into small parts known as functions/methods.
2. In POP top to down approach is used.
3. No Access Specifiers.
4. It deals with algorithms.
5. It uses less memory.
6. For example FORTRAN, PASCAL, C etc.
7. In POP there were a lot of issues less security and due to this reason OOPs were mostly adopted language by developers.

**Object Oriented Programming Paradigm**

1. In OOP program is divided into parts i.e., Objects.
2. In OOP bottom-up approach is used.
3. Have a lot of access modifiers or access specifiers.
4. OOP deals with data.
5. It needs more memory as compared to POP.
6. For examples Java, C++, C#, Python etc.

**NOTE : One language can use multiple programming paradigm**

## Object Oriented **v/s** Object Based Programming LANGUAGES?

|  |  |
| --- | --- |
| Object based programming language | Object oriented programming languages |
| Object based languages supports the usage of object and encapsulation. | Object oriented languages supports all the futures of oops including inheritance and polymorphism. |
| They does not support inheritance or, polymorphism or , both | They support built-in objects. |
| Object based languages does not support bult in objects | C#, java, VB.Net are the examples of object-oriented languages. |
| JavaScript, VB are the examples of object bases languages. |  |

# Classes, objects and methods

## Introduction to oops

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

**Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

**Smalltalk** is considered the first truly object-oriented programming language.

The popular object-oriented languages are [Java](https://www.javatpoint.com/java-tutorial), [C#](https://www.javatpoint.com/c-sharp-tutorial), [PHP](https://www.javatpoint.com/php-tutorial), [Python](https://www.javatpoint.com/python-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

**Points to remember**

1. Full form is **OOP’s** is Object Oriented Programming.
2. OOP is the programming paradigm based on the concept of Objects which contains the data (fields or variables) and methods.
3. It is the most popular programming paradigm used by the programmers
4. For examples: Java, Python, C++ etc
5. Features of OOP

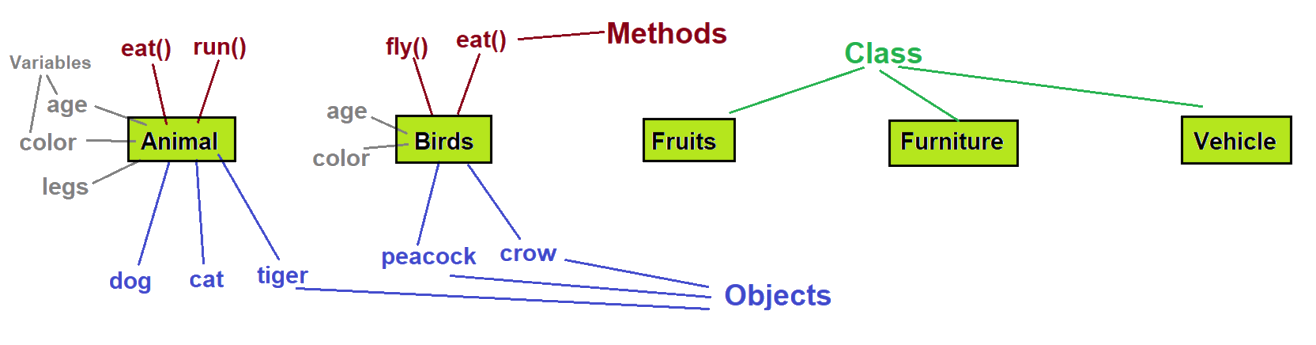


1. Class, Objects & Methods
2. message Passing
3. Inheritance & Composition
4. Polymorphism
5. Encapsulation
6. Abstraction

Apart from these concepts, there are some other terms which

are used in Object-Oriented design:

* 1. Coupling
  2. Cohesion
  3. Association
  4. Aggregation
  5. Composition
* **Real World Example of Class, Methods & Objects**



## Class

In Java, the **class** is a blueprint from which we can create an individual object. Java provides a [keyword](https://www.javatpoint.com/java-keywords) named class by which we can declare a class. Inside the class, we define **class** members and functions. It is not possible to create [Java programs](https://www.javatpoint.com/java-programs) without class. We can also refer a class as a **user-defined** [data type](https://www.javatpoint.com/java-data-types) because an object-oriented paradigm allows us to model real-world objects. In this section, we will focus on the **types of classes in Java.**  A class declaration constitutes of the following parts:

* Modifiers
* Class name
* Keywords
* The class body within curly brackets {}
  + 1. A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
    2. A class is a user defined blueprint or prototype which is used to create an object.
    3. Class is a logical entity or say it’s not a real-world entity or class is not physical.
    4. Real world example: - Animal, Birds, Vehicle, Fruits etc.
    5. Class represents the set of properties or methods that are common to all the objects of one type.
    6. Simply we can say that a class is a group of objects having common properties (attributes or variables), behaviour (methods), relationships & semantics.

Full Syntax

Keyword

**access-modifiers** **class** ClassName **extends** ParentClassName **implements** InterfaceName

{

//variables

//blocks

//constructors

//methods

//nested class, interfaces

}

Simple syntax

**access-modifiers** **class** ClassName

{

//variables

//methods

}

Simple class

**public class** Animal

{

**int** age=10;

**String** color=black;

}

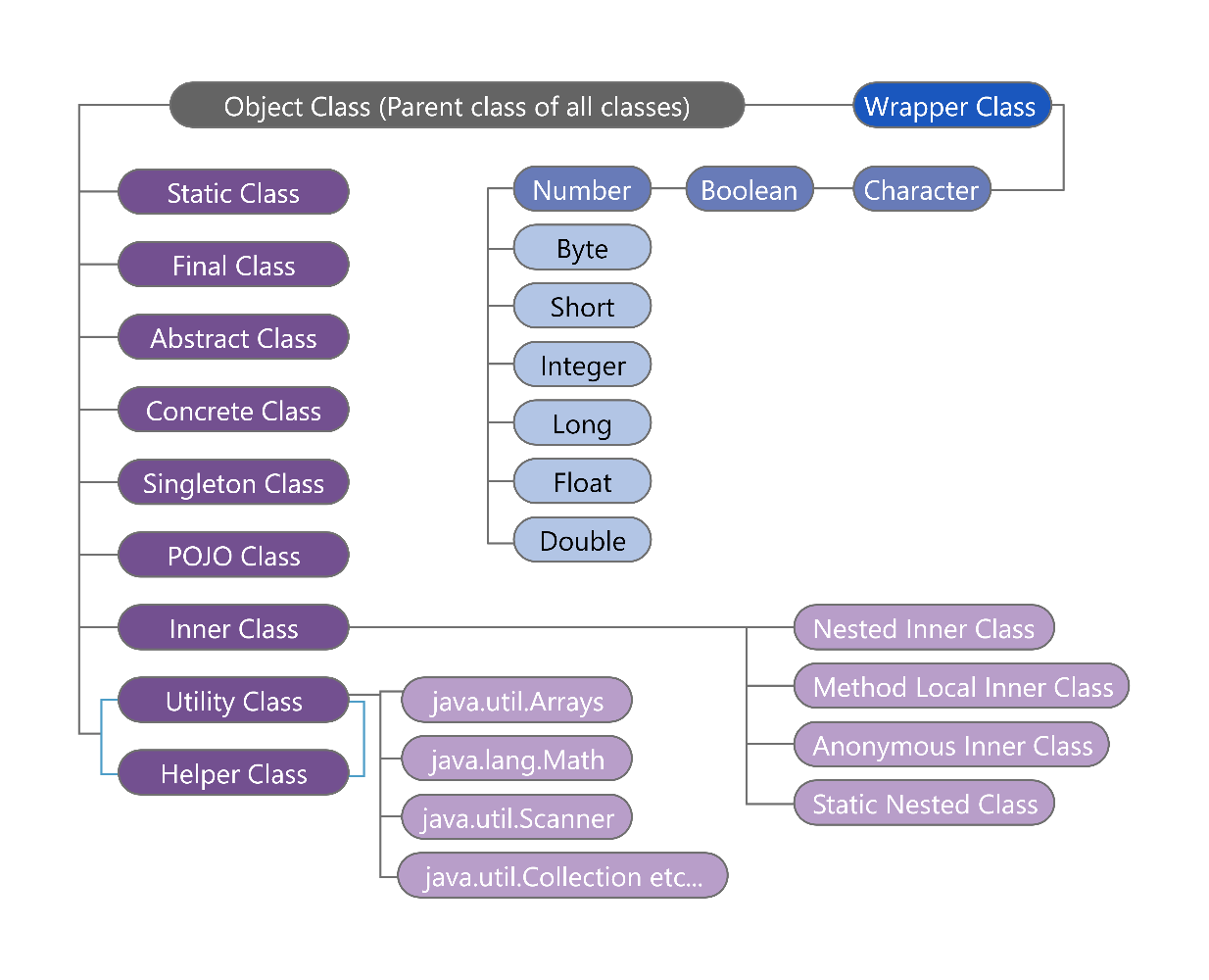
**Points to remember**

1. We can only use public or default access-modifiers but not **private or protected** with outer class.
2. For inner class we can use all access-modifiers i.e. **public, protected, default and private.**

note: here we covered all concepts of basic class but further we can talk some advance stuff regarding classes.

## types of classes

In java there are some more type of classes are present which we can explore now.



### 1 Static class

In [Java](https://www.javatpoint.com/java-tutorial), static is a keyword that manage objects in the memory. The static object belongs to the class instead of the instance of the class.

We can make a class [static](https://www.javatpoint.com/static-keyword-in-java) if and only if it is a nested class. We can also say that static classes are known as nested classes. It means that a class that is declared as static within another class is known as a static class. Nested static class does not require reference to the outer class. The purpose of a static class is to provide the outline of its inherited class.

The properties of the static class are:

* The class has only static members.
* It cannot access the member (non-static) of the outer class.
* We cannot create an object of the static class.

### 2 final class

The word final means that cannot be changed. The **final** class in Java can be declared using the [final keyword](https://www.javatpoint.com/final-keyword). Once we declare a class as final, the values remain the same throughout the program. The purpose of the final class is to make the class **immutable** like the String class. It is only a way to make the class immutable. Remember that the **final class cannot be extended**. It also **prevents the class from being sub-classed**.

### 3 abstract class

An [abstract class](https://www.javatpoint.com/abstract-class-in-java) is a that is declared with the keyword **abstract**. The class may or may not contain abstract methods. We cannot create an instance of an abstract class but it can be a subclass. These classes are incomplete, so to complete the abstract class we should extend the abstract classes to a concrete class. When we declare a subclass as abstract then it is necessary to provide the implementation of abstract methods. Therefore, the subclass must also be declared abstract. We can achieve data hiding by using the abstract class. An example of an abstract class is **AbstarctMap** class that is a part of the Collections framework.

### 4 concrete class

These are the regular Java classes. A derived class that provides the basic implementations for all of the methods that are not already implemented in the base class is known as a **concrete** class. In other words, it is regular Java classes in which all the methods of an abstract class are implemented. We can create an object of the concrete class directly. Remember that concrete class and abstract class are not the same. A concrete class may extend its parent class. It is used for specific requirements.

### 5 singleton class

A class that has only an object at a time is known as a **singleton class**. Still, if we are trying to create an instance a second time, that newly created instance points to the first instance. If we made any alteration inside the class through any instance, the modification affects the variable of the single instance, also. It is usually used to control access while dealing with the database connection and socket programming. If we want to create a singleton class, do the following:

* Create a private [constructor](https://www.javatpoint.com/java-constructor).
* Create a static method (by using the lazy initialization) that returns the object of the singleton class.

### 6 POJO class

In Java, POJO stands for **Plain Old Java Object.** A Java class that contains only private variables, setter and getter is known as **POJO** class. It is used to define Java objects that increase the reusability and readability of a Java program. The class provides encapsulation. It is widely used in Java because it is easy to understand these classes. POJO class has the following properties:

1. It does not extend the predefined classes such as Arrays, HttpServlet, etc.
2. It cannot contain pre-specified annotations.
3. It cannot implement pre-defined [interfaces](https://www.javatpoint.com/interface-in-java).
4. It is not required to add any constructor.
5. All instance variables must be private.
6. The getter/ setter [methods](https://www.javatpoint.com/method-in-java) must be public.

### 7 inner class

In java there are two type of inner classes are there

1. Static inner class
2. Non-static inner class.
3. **Static inner class or Static Nested class**

In Java, a static nested class is a nested class that is declared as **static** within another class. It is a class that is defined inside another class, but unlike regular nested classes, it is marked with the static keyword. A static nested class is primarily used for grouping related functionality or encapsulating functionality within the context of the outer class. It has the following characteristics:

1. **Accessibility:** A static nested class can be accessed directly using the outer class name, without the need to create an instance of the outer class.
2. **Namespace**: The static nested class has its own namespace and does not have access to the instance variables and methods of the outer class. However, it can access static members (variables and methods) of the outer class.
3. **Instantiation:** It can be instantiated independently of the outer class. To create an instance of a static nested class, you don't need an instance of the outer class.
4. **Non-static nested class / Inner class.**

A non-static nested class in Java is also known as an inner class. It is a class that is defined inside another class without the static keyword. Unlike static nested classes, non-static nested classes are associated with an instance of the outer class and have access to its instance variables and methods.

Here are some key characteristics of non-static nested classes:

1. **Association with Outer Class:** An instance of a non-static nested class is associated with an instance of the outer class. This means that you need to create an instance of the outer class before you can create an instance of the inner class.
2. **Access to Outer Class Members**: Non-static nested classes have access to the instance variables and methods of the outer class, including private members. They can refer to these members directly without any special syntax.
3. **Visibility**: The members (variables, methods, nested classes) of the inner class are scoped to the inner class itself. This means they can be accessed only within the context of the inner class. However, if the inner class is marked as public, it can be accessed by other classes outside the outer class using the syntax OuterClass.InnerClass.

**Types of Non-Static Inner Classes**

Java provides the two types of inner classes are as follows:

1. **Local Classes or Method Local Inner Class**
2. **Anonymous Classes or**[**Anonymous Inner Class**](https://www.javatpoint.com/anonymous-inner-class)
3. **Local Class or Method Local inner Class**

A local inner class, also known as a method-local inner class, is a class that is defined inside a method of another class. It has the following characteristics:

1. **Scope:** A local inner class is local to the method in which it is defined. It cannot be accessed or instantiated outside the method.
2. **Accessibility:** A local inner class can access the variables and parameters of the enclosing method, including local variables marked as final or effectively final. However, it cannot access non-final local variables or method parameters that are not marked as final.
3. **Visibility:** The local inner class is not visible outside the method, including other methods of the enclosing class. It is effectively encapsulated within the method where it is defined.
4. **Instantiation:** An instance of a local inner class can only be created within the method where it is defined. It cannot be instantiated or accessed from outside the method.

Local inner classes are often used when you need to define a class that has a close relationship with a specific method and is not required to be used outside that method. They can be useful for implementing callbacks or providing specialized behaviour within the context of the enclosing method.

1. **Anonymous class or anonymous inner class**

An anonymous inner class in Java is a special type of local inner class that does not have a specified name. It is declared and instantiated at the same time, typically as part of an expression or statement. Anonymous inner classes are often used to define and implement a single-use or ad-hoc class with overridden methods or additional functionality.

Here are some key characteristics of anonymous inner classes:

1. **Nameless:** Anonymous inner classes do not have a specified name, unlike regular inner classes.
2. **Inheritance or Implementation:** Anonymous inner classes can extend a class or implement an interface, just like regular classes.
3. **Syntax:** Anonymous inner classes are defined using a combination of class/interface definition and instantiation in a single expression.
4. **Overrides:** They can override methods from the superclass or interface and provide custom implementations.
5. **Scope:** Anonymous inner classes are typically defined and used within a limited scope, such as a method or constructor.

### 8 wrapper class

A wrapper class, also known as a wrapper object or simply a wrapper, is a class in Java that provides an object-oriented representation of a primitive data type. Wrapper classes are used to convert primitive types into objects so that they can be used in scenarios that require objects, such as collections, generics, and method overloading.

In Java, the following wrapper classes are provided for each primitive data type:

Boolean for boolean

Byte for byte

Short for short

Integer for int

Long for long

Float for float

Double for double

Character for char

Wrapper classes provide various utility methods to work with their respective primitive types. They also enable the conversion of primitive types to strings and vice versa through methods like **valueOf()** and **parseXxx(),** where Xxx represents the corresponding primitive type.

Here's an example that demonstrates the usage of wrapper classes:

**public** **class** WrapperExample {

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

Integer intObj = Integer.*valueOf*("123"); // Wrapper class object created from a string

**int** intVal = intObj.intValue(); // Unboxing: Wrapper object to primitive type

System.***out***.println("Integer value: " + intVal);

Double doubleObj = Double.*valueOf*("3.14"); // Wrapper class object created from a string

**double** doubleVal = doubleObj.doubleValue(); // Unboxing: Wrapper object to primitive type

System.***out***.println("Double value: " + doubleVal);

Character charObj = 'A'; // Autoboxing: Primitive type to wrapper object

**char** charVal = charObj.charValue(); // Unboxing: Wrapper object to primitive type

System.***out***.println("Character value: " + charVal);

}

}

In the example above, wrapper classes are used to convert primitive types to objects and vice versa. The **valueOf()** method is used to create a wrapper object from a string, while the **xxxValue()** methods are used to extract the primitive value from the wrapper object.

Wrapper classes are especially useful in scenarios where primitive types need to be used as objects or when working with generic classes that require objects. They provide a way to seamlessly switch between primitive types and their object counterparts in Java.

### 9 utility class

A utility class, as the name suggests, is a class that provides utility methods or functions that perform common tasks or provide common functionality. Utility classes are typically composed of static methods and do not require instantiation. They serve as a collection of related static methods that can be used by other classes throughout an application.

**Here are some key characteristics of utility classes:**

1. **Static Methods:** Utility classes consist primarily of static methods that encapsulate specific functionality or provide common operations. These methods can be accessed directly using the class name without the need to create an instance of the utility class.
2. **Stateless:** Utility classes are generally stateless, meaning they do not maintain any instance variables or state. They rely on the inputs passed as arguments to their methods and do not store any persistent data.
3. **Common Functionality:** Utility classes often provide functionality related to tasks such as string manipulation, mathematical calculations, date/time operations, file handling, data validation, conversions, and more. They encapsulate reusable code that can be shared across multiple classes or projects.
4. **Final and Non-instantiable:** Utility classes are often marked as final to prevent them from being subclassed. They typically have a private constructor, making them non-instantiable, as instantiation is not required for utility classes.

Here's an example of a utility class that provides string manipulation methods:

**public** **final** **class** StringUtil {

**private** StringUtil() {

// Private constructor to prevent instantiation

}

**public** **static** **boolean** isEmpty(String str) {

**return** str == **null** || str.isEmpty();

}

**public** **static** String reverse(String str) {

**return** **new** StringBuilder(str).reverse().toString();

}

**public** **static** String capitalize(String str) {

**if** (*isEmpty*(str)) {

**return** str;

}

**return** Character.*toUpperCase*(str.charAt(0)) + str.substring(1);

}

}

In the example above, StringUtil is a utility class that provides three static methods: isEmpty(), reverse(), and capitalize(). These methods perform common string manipulation tasks, such as checking if a string is empty, reversing a string, and capitalizing the first letter of a string.

Utility classes are widely used in Java applications to organize and share reusable code, improve code maintainability, and provide a centralized place for commonly used functionality. They help promote code reuse and reduce duplication across an application.

**Difference between Utility class vs Static class**

A utility class is typically used to group related static methods together. It contains only static methods, and its primary purpose is to provide common utility functions that can be used by other classes. Utility classes are often used for tasks such as mathematical calculations, string manipulation, date/time operations, file handling, etc. These utility classes are usually final and have a private constructor to prevent instantiation.

On the other hand, a static class is a nested class that can contain both static and non-static members (variables, methods, nested classes, etc.). It can be instantiated and used like any other class, but its members can be accessed without creating an instance of the enclosing class. Static classes are often used to encapsulate related functionality and provide a convenient way to group functionality together.

In summary, while both utility classes and static classes can contain static members, utility classes are typically used solely for providing utility methods, whereas static classes can have a wider range of functionality and can be instantiated and used as objects.

**Difference between Helper class vs Utility class**

In Java, the terms "helper class" and "utility class" are often used interchangeably to refer to a class that provides common functionality or utility methods. However, it's important to note that these terms are not standardized or defined by the Java language itself. They are more of a convention or naming pattern used by developers to indicate the purpose of a particular class.

Both a helper class and a utility class serve a similar purpose of providing reusable methods or functionality that can be used by other classes. They typically contain static methods that perform common operations or tasks, such as data manipulation, input/output operations, validation, conversions, and more.

The exact naming and structure of these classes may vary depending on the development team or individual preferences. Some developers may prefer to use "Helper" in the class name (e.g., "StringUtilHelper"), while others may use "Utility" (e.g., "FileUtility"). Ultimately, the choice of terminology and naming conventions is up to the developers and the specific project's style guide.

So, while the terms "helper class" and "utility class" may be used interchangeably, they both refer to classes that provide common functionality or utility methods in Java.

## where we can access classes, fields, member, method

Here’s some good reference information. In the declaration of a class or member the entity can be marked with zero or more modifiers. (But private, protected and public are supposed to be mutually exclusive.)

|  |  |
| --- | --- |
| Class Modifiers | |
| (no modifier) | accessible only within its own package |
| public | accessible wherever its package is |
| abstract | cannot be instantiated (may have abstract methods) |
| final | cannot be extended (subclassed) |
| sealed | has only the subclasses indicated |
| static | makes an inner-declared class really a top-level one |
|  |  |
| Member Modifiers | |
| private | accessible only within its class |
| (no modifier) | accessible only within its package |
| protected | accessible only within its package and to its subclasses |
| public | accessible wherever its class is |
|  |  |
| Field Modifiers | |
| final | value may not be changed, once assigned |
| static | only one instance of the field shared by all objects of the class |
| transient | field is not serialized |
| volatile | value may change asynchronously compiler must avoid certain optimizations |
|  |  |
| Method Modifiers | |
| final | may not be overridden |
| static | method does not apply to a particular instance |
| abstract | has no body; subclasses will implement |
| synchronized | requires locking the object before execution |
| native | implementation is not written in Java, but rather in some platform dependent way |

## object

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

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

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

An object has **three characteristics**:

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

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

**Object Definitions:**

1. An object is physical entity or *a real-world entity*.
2. An object is *a runtime entity*.
3. The object is *an entity which has state and behaviour*.
4. The object is *an instance of a class*.
5. Object is simple a memory block

Syntax :

1. Creation of an object

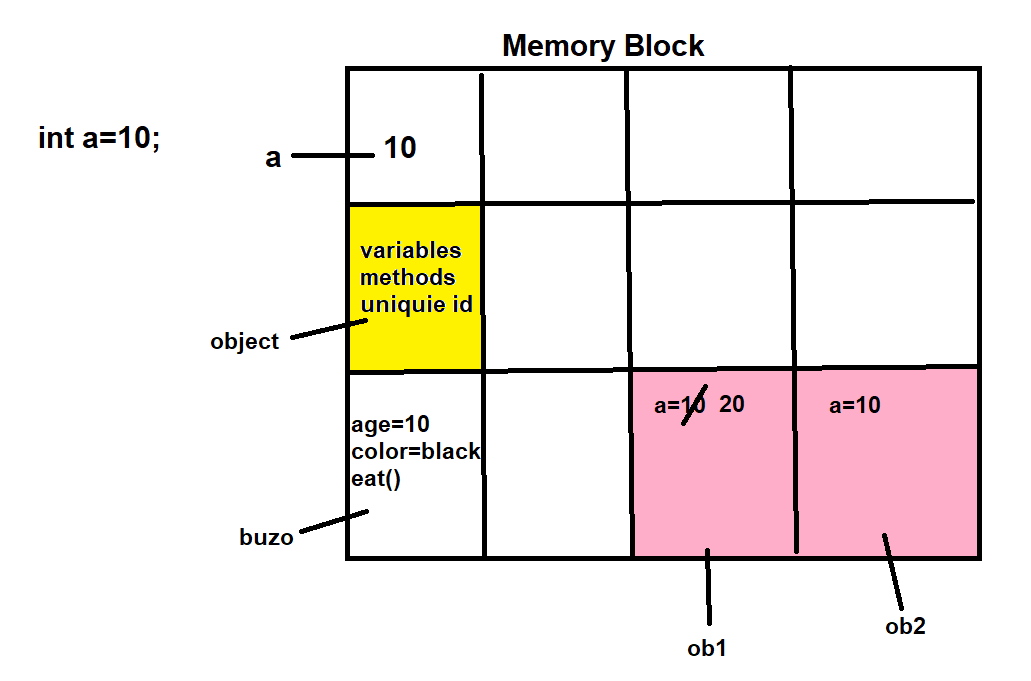
ClassName object\_name (ref\_variable\_name) = new ClassName();

Animal buzo = new Animal();

2. Calling variables or methods from object

object\_name.variable\_name; -> buzo.age;

object\_name.methodName(); -> buzo.eat();



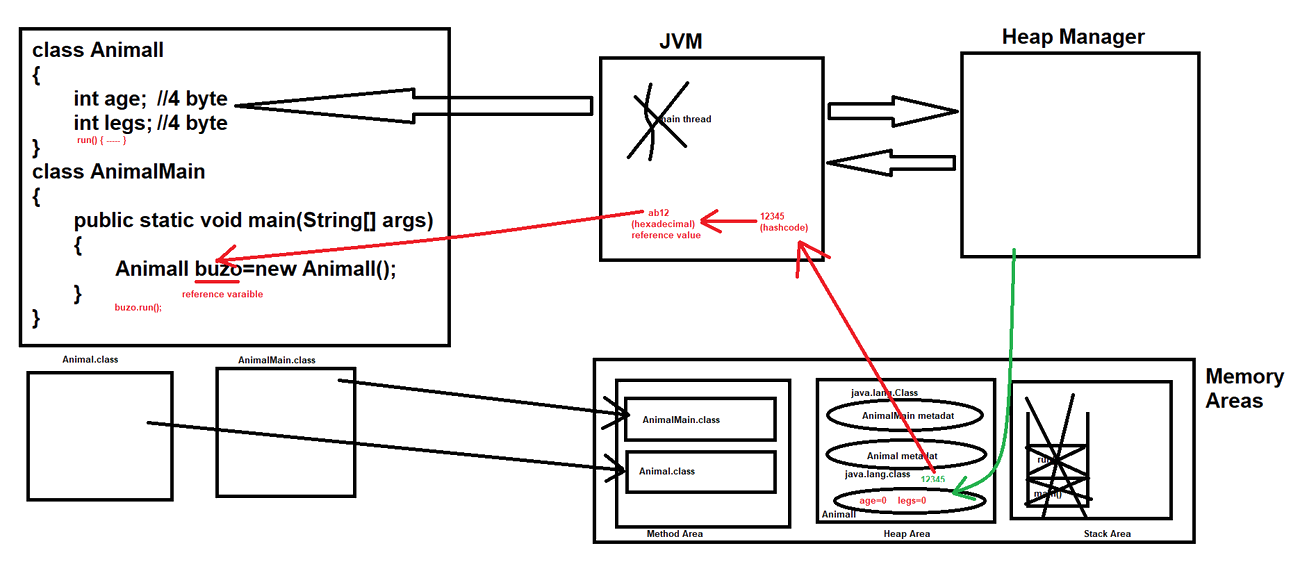
### How Objects are CREATED / Deep creation of Object

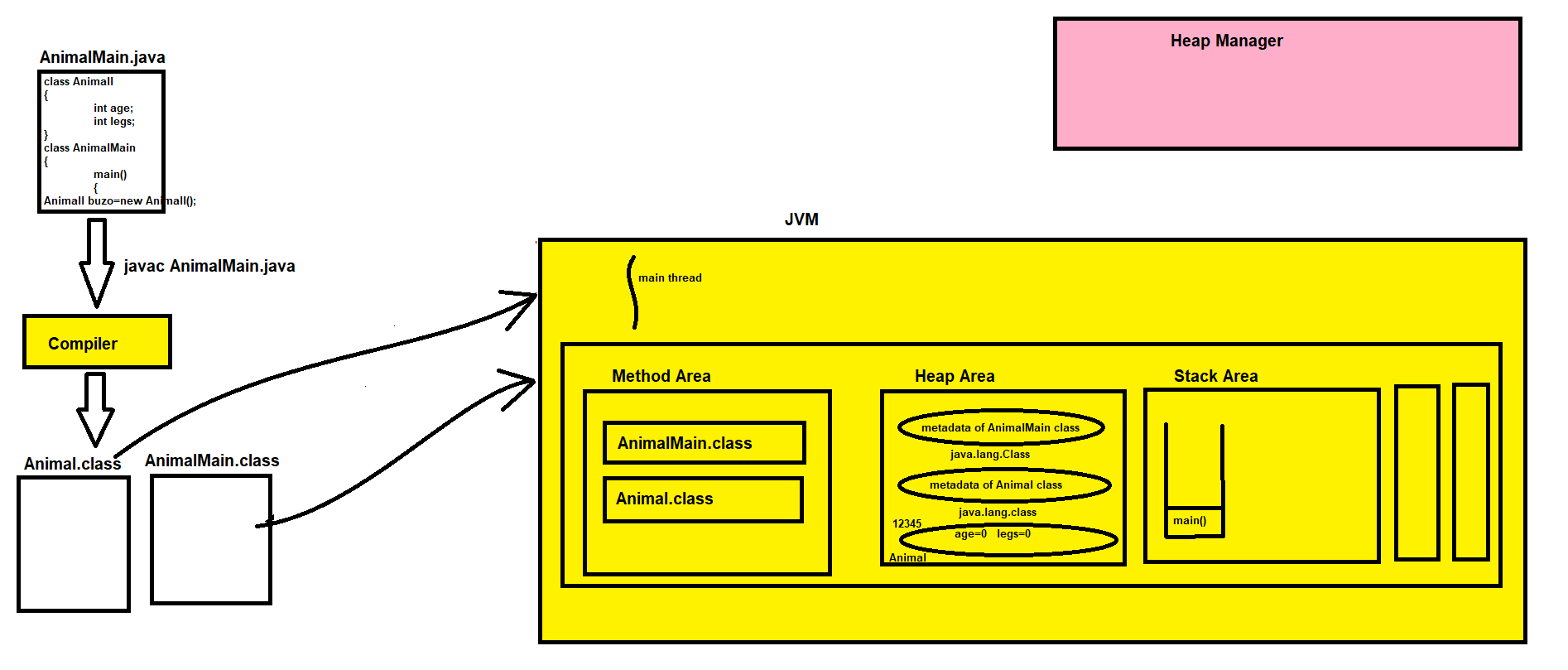
1. **When We compile the program i.e. javac AnimalMain.java**

Compiler will check the syntax and if syntax is correct then it will generate .class files (no of .class files generated depends on the no of classes we have created)

1. **When we run the program i.e. java AnimalMain**

1. AnimalMain.class file will be loaded in JVM memory area i.e. in Method Area.
2. An object of java.lang.Class class will be created in Heap Area in which AnimalMain class metadata will be stored.
3. Now Main Method will execute and for this JVM will create a new thread known as main thread.
4. As soon as main thread is created then JVM will create main stack in stack area.
5. Now first line in main method will execute i.e. Animal ob=new Animal(); now Animal.class file will load in method area and a new object of java.lang.Class class is created in heap area in which metadata of Animal class will be stored.
6. Now there is new keyword so new object creation process will start. JVM will instruct heap manager to create an object of Animal class but heap manager will ask for object size to JVM. Then JVM will calculate the size of object according to the number of instance variables that are declared in Animal class and this size will be taken by heap manager and heap manager will create an object in heap area.
7. As soon as heap manager creates an object, a unique integer value will be assigned to the object which is known as "hashcode".
8. This hascode value will be provided to the JVM and JVM will convert this hascode vallue into hexadecimal form and this hexadecimal value is known as reference value.
9. Now this hexadecimal value will be assigned to the variable which is known as reference variable.
10. Now object will be initialized that means all instance variables will be assigned by default values or by their original values.





1. java.lang.Class :- It represents the classes and interfaces which are used in running java application
2. java.lang.Object :- This is the parent class of all the classes in java.
3. When we create any class than that class will inherit Object class either directly or indirectly
4. java.lang.reflect.Method
5. java.lang.reflect.Constructor

## Methods

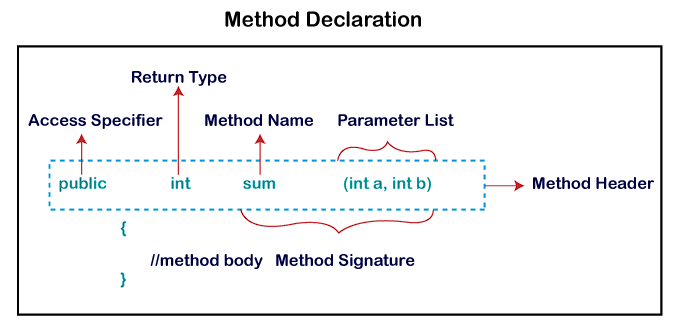
In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.

**method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main ()** method. Which executes the entire java program

### **Method Declaration**

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Full Syntax

**access-modifiers** **return-type** methodName(list of parameters) **throws** ExceptionClassName, -, -

{

//statements

}

Simple Syntax

**return-type** methodName(list of parameters)

{

//statements

}

Example

**void** eat() //method declaration

{ //method definition (body)

System.out.println("I’m eating");

}

**Method Signature**

Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list and used in method overloading**.

**Access Specifier**

Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

1. **Public:** The method is accessible by all classes when we use public specifier in our application.
2. **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
3. **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
4. **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

**Return Type**

Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword. **return** keyword is used to return any value from method.

**Method Name**

It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

**Single-word method name:** sum(), area()

**Multi-word method name:** areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

**Parameter List:**

It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

**Method Body:**

It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

**Some important point to remember**

1. when we pass the value in method **parameter** using **arguments** in case copy of actual value will be send not original value is sent.
2. Java supports only **call by value** not **call by reference**.
3. In case of passing **array** in method parameter will send the reference of the array not send the entire copy of the array.
4. We can also overload **main method** in java but **JVM** call the default main method by checking its syntax.

ex

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

**public static void** main (**int** value){}

**public static void** main (**boolean** check){}

// here we overload main method two times but JVM call always main method which have (String args[])

1. There are four types of method in programming languages bases on return type.
   1. Method with argument and return type.
   2. Method with argument but no return type.
   3. Method with no argument but return type.
   4. Method with no argument and no return type.

### Types of Method

There are mainly two types of methods in Java (1,2) others method are includes in 1,2 :

1. **Predefined Method**

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),random()** etc.

1. **User-defined Method / Concrete method**

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement. (Concrete methods are those method which have full body implementation)

**int** a = 100;

**static** **int** b = 200;

**public static void** findEvenOdd(**int** num) { //method body

**if**(num%2==0)

System.out.println(num+" is even");

**else**

System.out.println(num+" is odd");

// we can access both static and non-static data member in concrete method and also modify the value of if.

a = 10;

b = 20;

System.out.println(a+b); o/p = 20

}

note: all of these methods are using described in details further

1. **Static method**
   1. A method that has **static keyword** is known as static method.
   2. In other words, a method that belongs to a class rather than an instance of a class is known as a static method.
   3. We can also create a static method by using the keyword **static** before the method name.
   4. The main advantage of a static method is that we can call it **without creating an object**.
   5. It can access **static data members** and also change the value of it.
   6. It is used to create an instance method. It is invoked by using the **class name directly**.
   7. The best example of a static method is the **main()** method.

example

**int** first = 10; // this instance variable belongs to object (access only with class object)

**static int** second = 10; // this static variable belongs to class (no need of object to access)

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

int third = 30 ; // this is local variable and only accessible inside the block.

first = 20; // this will throw an **compile time error** Non-static field 'first' cannot be referenced from a static context.

// we only able to access **static data member** and **local data member** in static data method.

}

1. **Instance method**

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. There are two types of instance method:

* 1. **Accessor Method (Getters method)**

**Accessor Method:** The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. It is also known as **getters**. It returns the value of the private field. It is used to get the value of the private field.

**private** Id =10;

**public** **int** getId() {

**return** Id;

}

* 1. **Mutator Method (Setters method)**

**Mutator Method:** The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word **set**. It is also known as **setters** or **modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

**Private** roll = 101;

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

1. **Abstract method**
2. The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method.
3. It always declares in the **abstract class**.
4. It means the class itself must be abstract if it has abstract method. “*We cannot create object of abstract class because it types of method that’s have no method body. In any case we able to create body of abstract method then we also able to call abstract method and we call it then what will this method is perform because it contains nothing.”*
5. To create an abstract method, we use the keyword **abstract**.

Ex : **abstract** **void** detail { **abstract** **void** method\_Name(); } // and we cannot create object of it

1. **Factory method**

It is a method that returns an object to the class to which it belongs. All static methods are factory methods. For example, **NumberFormat obj = NumberFormat.getNumberInstance();**

note : Only providing hint briefly described in further Chapters (Anonymous method and class in (Java 8 Features in Lambda Expressions) or (Synchronized method and Processes in Multi-threading)

1. **Synchronized method**

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. (An important exception: **final** fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with [liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html), as we'll see later in this lesson.

The synchronization is mainly used

1. To prevent thread interference.
2. To prevent consistency problem.
3. **Anonymous method or function / lambda functions**
4. Anonymous function is a function define as not bound to an **identifier**. Because, these are a form of nested function in allowing access to variables in the scope of the containing function (non-local functions). So, this means anonymous functions need to be implemented using closures. Simply, lambda is an anonymous function which can be passed around in a concise way.  **Or,**
5. An anonymous function (**function literal, lambda abstraction, lambda function, lambda expression or block**) is [function](https://en.wikipedia.org/wiki/Function_(computer_science)) definition that is not [bound](https://en.wikipedia.org/wiki/Name_binding) to an identifier.
6. Anonymous functions are often arguments being passed to [higher-order functions](https://en.wikipedia.org/wiki/Higher-order_function) or used for constructing the result of a higher-order function that needs to return a function.

Also, a lambda expression represents an **anonymous function** and it comprises of a set of parameters, a lambda operator (->) and a function body.

**Return Type**

1. When there is a single statement, the return type of the anonymous function is the same as that of the body expression.
2. When there is more than one statement enclosed in curly brackets then the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Examples: // only works with lambda functions / methods (normal method use **return** keyword for return type.)

1. **Zero parameter**

() -> System.out.println(” No Parameters “);

1. **One Parameter**

(parameter) -> System.out.println(“One parameter is : ” + parameter);

or without parenthesis; parameter -> parameter (Identity function example)

1. **Multiple parameters**

(parameter1, parameter2) -> parameter1 + parameter2;

1. **Parameter Types**

(int a, String name) -> System.out.println(“id:” + a + ” name:” + name);

1. **Code block**

(parameter1, parameter2) -> { return parameter1 + parameter2; }

1. **Nested lambda expression**

**Two** nested expressions with the first one as closure

1. **As objects**

The lambda expression is recommended to variable, lastly it is summoned by the interface strategy it carried out

1. **Varargs: Variable-Length Arguments**

Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called ***varargs***and it is short for ***variable-length arguments***. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

1. When we want single type of result with our method but we have different parameters with same data type then we use varargs.
2. Varargs takes value in form of array so whenever you want to use varargs then you have to use loops to fetching all the values.
3. Varargs can be overloads but the condition is varargs is always present in last of the parameter list.
4. Always remember only **three period …(dots)** are allowed to create varargs methods.
5. Syntax modifier return\_type methodName(datatype parameterName){}

public void addition (int …a){};

public int adding (int … args){}

public int multiplication(int a, short … args){}

public static int print(int …a){}

public int doit(int a, int b, int c ,double d, int … var){}

public int adding1(int … args, float b ){} // wrong syntax

public int doit(int a, int b, int c ,double d, int … var, boolean stop){}

public int printing(int … first, int … second){} // wrong syntax

remember some points to avoid ambiguity when you overload varargs

example 1:

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

print();

// in this case the print will throw an error because method print() is empty and compiler doesn’t know to call which print (int one or boolean one ) and in this case this will throw an compile time error

Ambiguous method call. Both

print (boolean...)

print (long...)

}

**static** **int** print(**int** … a){}

**static** **int** print(**boolean** … b){}

example 2:

public static void main(String[] args)

testing(10,20,30,); // throws an error Ambiguous method call. Both

}

|  |
| --- |
| **static** **void** testing(**int** ... a){  **for**(**int** print : a){  System.***out***.println(print);  }  }  **static** **void** testing(**int** a , **int** ...c){    **for**(**int** check : c){  System.***out***.println(check);  }  } |

Does this translate into a call to **testing(int …a)**, with one varargs argument, or into a call to **testing(int a, int …c)** with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous. Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.

# Constructors

In Java, a constructor is a special method that is used to initialize objects of a class. It is called automatically when an object of the class is created using the new keyword. The purpose of a constructor is to set initial values to the instance variables of the object and perform any necessary setup or initialization tasks.

Here are some key points about constructors in Java:

1. **Method Name**: Constructors have the same name as the class they belong to. They do not have a return type, not even void.
2. **Automatic Invocation**: When an object is created using the new keyword, the constructor associated with the class is automatically invoked to initialize the object.
3. **Overloading**: Just like regular methods, constructors can be overloaded. This means that a class can have multiple constructors with different parameter lists. This allows creating objects with different initialization options.
4. **Default Constructor**: If a class does not explicitly define any constructors, Java provides a default constructor that takes no arguments. The default constructor initializes the object with default values or performs no initialization if no default values are specified.
5. **Parameterized Constructor**: A parameterized constructor is a constructor that takes one or more parameters. It allows you to pass values to initialize the object with specific values.
6. **“this” Keyword:** The this keyword is used to refer to the current object within a constructor. It can be used to differentiate between instance variables and parameters with the same name.
7. **when constructors are executed** : Constructors are executed exactly at the time of object creation, not before or after object creation.
8. **how constructors are executed** : Constructors are executed automatically when we create an object

**syntax :-**

**access-modifiers** ClassName(list of parameters) **throws** Exception1, Exception2, --

{

//initialization code

}

1. We can use any access-modifier for the constructor i.e. **public, private, protected or default**. This is done to control the object creation
2. We cannot use abstract, final, static, synchronized etc keywords with constructors

### TYPES OF CONSTRUCTORS

There are 3 types of constructors

1. Default Constructors (compiler)
2. 0-Argument Constructors (programmer)
3. Parametrized Constructors (programmer)

**1. Default Constructors**

1. Whenever we don’t create any constructor in class, then compiler will always create a constructor which is known as default constructor.
2. Default constructors are used to provide the default values to the objects like 0, null etc depending on the type.
3. **Note:** If programmer creates any one constructor, then compiler will not generate default constructor
4. Prototype of default constructor:
   1. Access-modifier of default constructor will be same at that of class access-modifier.
   2. Access-modifier of default constructor cannot be private or protected because outer class cannot be private or protected.
   3. Default constructor has only one line of code i.e. super();

**2. 0-Argument Constructors / Non parameterised constructor**

1. These constructors are created by the programmer

class Test

{

Test()

{

}

}

**3. Parametrized Constructors**

1. These constructors are created by the programmer

class Test

{

Test(int a, int b)

{

}

}

**What is difference between Methods & Constructors**

|  |  |  |
| --- | --- | --- |
| 1 | Methods always have return type. | Constructors does not have any return type even void. |
| 2 | Methods can have any valid name. | Constructors always have same name as that of class name. |
| 3 | Methods are used to perform any particular task. | Constructors are always used to initialize an object. |
| 4 | We have to call the methods explicitly by using object name or class name. | Constructors are called automatically when we create an object. |
| 5 | If we don’t create any method then compiler will not generate any method. | If we don’t create any constructor then compiler will generate default constructor. |

**Topics related to constructor**

1. Constructors with inheritance.

2. Constructors overloading and overriding.

3. Constructors chaining (using this keyword).

4. Use of super keyword with constructor.

5. Constructors with abstract class & interface.

6. Constructors with exception handling.

7. Copy constructor.

**NOTE**

1. Constructor is predefined class present in **java.lang.reflect** package.
2. This Constructor class is used to get constructor related information.

**Program Flow**

