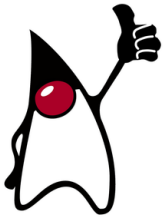
LavaTech Technology

Core Java



Call: 096073 31234
Email: info@lavatechtechnology.com
Website: https://lavatechtechnology.com
Address: Pune, Maharashtra

PUBLISHED BY LAVATECH TECHNOLOGY

lavatechtechnology.com

January 2022

YOU CAN

TOTALLY

DO THIS!

Contents

1 Introduction to Java 9

1.1 Getting started with Java 9

1.1.1 What is Java? 10
1.1.2 Uses of Java 11
1.1.3 Problem with C and C++ 12
1.1.4 Advantages of Java over C++ 12
1.1.5 How does program execution works? 13
1.1.6 The way Java works 14
1.1.7 Compiler V/S JVM 15
1.1.8 JVM components 17
1.1.9 History of Java 18

1.2 Installing Java and IDE 22

1.2.1 Java installation 23
1.2.2 IDE installation 24
1.2.3 Our first Java program 25
1.2.4 Executing our first Java program 28
1.2.5 Using JShell 32

2 Java Language Fundamentals 35

2.1 Identifiers, variables and more 35

2.1.1 Java identifiers 36
2.1.2 Java grammar 37
2.1.3 Java variable 38
2.1.4 Literals 39
2.1.5 Java reserved words or keywords 40
2.1.6 Java Comments 41
2.1.7 How Objects Can Change Your Life? 42

2.2 Introductions to OOPs 45

2.2.1 Java source file 47
2.2.2 main() method 51
2.2.3 Command-line argument 55
2.2.4 Pillars of OOPs 57

3 Data types in Java 59

3.1 Getting started with data type 59

3.1.1 Java is strongly typed 60
3.1.2 Types of data types 60

3.2 Integer 61

3.2.1 byte 62
3.2.2 short 63
3.2.3 int 63
3.2.4 long 64
3.2.5 Integer literals 65

3.3 Floating-point 68

3.3.1 Float 68
3.3.2 Double 68
3.3.3 Floating-point literals 69

3.4 Character 72

3.4.1 What is ascii & unicode? 72
3.4.2 char datatype 73
3.4.3 Character literals 74

3.4.4 Escape character 76
3.5 Boolean 77
3.6 Type conversion 78

4 Arrays 81
4.1 Arrays in detail 81
4.1.1 Array introduction 82
4.1.2 Array declaration 83
4.1.3 Array creation 85
4.1.4 Array initialisation 90
4.1.5 Array declaration, creation and initialisation in one line 93
4.1.6 length variable 94
4.1.7 Anonymous Arrays 97
4.1.8 Array element assignments 98
4.1.9 Array variable assignments 99

5 Operators 101
5.1 Operators in Java 101
5.1.1 Arithematic Operator 103
5.1.2 String Concatenation Operator 106
5.1.3 Increment/Decrement Operator 107
5.1.4 Relational Operator 110
5.1.5 Equality Operator 111
5.1.6 Bitwise Operator 113
5.1.7 Boolean complement operator 115
5.1.8 Short circuit Operator 116
5.1.9 Assignment Operator 118
5.1.10 Conditional Operator 120
5.1.11 new Operator 120
5.1.12 [] Operator 120
5.1.13 Operator Precedence 121

6 Flow Control 123
6.1 Flow control statement 123

6.2 Selection Statements 124
6.2.1 if...else 124
6.2.2 switch case 128
6.3 Iteration 135
6.3.1 while() 135
6.3.2 do-while() 137
6.3.3 for() 140
6.3.4 for-each loop 143
6.4 Transfer Statements 145
6.4.1 break 145
6.4.2 continue 147
6.4.3 Labeled break & continue 148
6.4.4 return 150
6.4.5 try..catch..finally 150
6.4.6 assert 150

7 Functions 151
7.1 var_arg methods 151
7.2 Selection Statements 152

8 Packages in Java 153
8.1 Packages in detail 153
8.1.1 import keyword 154
8.1.2 Packages 158

9 Polymorphism & Inheritance 163
9.1 Class 163
9.1.1 Attributes in detail 164
9.1.2 Methods in detail 173
9.1.3 Class level access modifier 186
9.2 Constructors and garbage collection 191
9.2.1 new operator 192

9.2.2 Constructor 193
9.2.3 this keyword 195
9.2.4 Constructor Chaining. 196
9.2.5 super() 197
9.2.6 super keyword 199
9.2.7 Constructor overloading 200
9.2.8 this() 202
9.2.9 super(),this() V/S super,this 204
9.2.10 Access modifier and constructor 205
9.2.11 Instance block 209
9.2.12 Static initializer 210
9.2.13 Constructor V/S Instance block V/S Static block 212
9.2.14 Destroying object 212
9.2.15 The Stack and the Heap: where things live 213

9.3 Inheritance 214

9.3.1 Types of inheritance 217
9.3.2 Final class 221
9.3.3 Has-A relationship 222
9.3.4 Has-A V/S Is-A relationship 225

9.4 Polymorphism 226

9.4.1 Method overloading 228
9.4.2 Method overriding 230
9.4.3 Overloading V/S Overriding 234
9.4.4 Method hiding 235

9.5 Data Encapsulation 237

9.5.1 Data Hiding 238
9.5.2 Data Abstraction 239
9.5.3 Tightly encapsulated class 240
9.5.4 Coupling 240
9.5.5 Cohesion 242

10 Abstract Class & Interface 243

10.1 Abstract class 243

10.1.1 What is abstract class? 244

10.1.2 What is concrete class? 246
10.1.3 What is abstract method? 247
10.1.4 Illegal modifier: final abstract 249

10.2 The ultimate superclass: Object 250

10.2.1 Object type-casting 254

10.3 Interface 255

10.3.1 Interface methods 256
10.3.2 extends V/S implements 258
10.3.3 Interface variables 259
10.3.4 Interface method naming conflicts 260
10.3.5 Interface variable naming conflicts 262
10.3.6 Marker interface 263
10.3.7 Adapter Classes 264
10.3.8 Interface V/S Abstract class 266

11 Java Built-in packages and API 267

11.1 Overview of java API 267

11.1.1 String, String Buffer and String Builder 268
11.1.2 Exception Handling 269
11.1.3 Threads and multithreading 270
11.1.4 Wrapper Classes 271
11.1.5 Data Structures 272
11.1.6 JAVA COLLECTION FRAMEWORKS 273
11.1.7 File Handling 274
11.1.8 Serialization 275

LINUX PADHO!

JOB KI CHINTA MAAT KARO

1. Introduction to Java

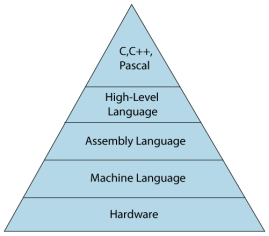
1.1 Getting started with Java

In this section, you are going to learn:

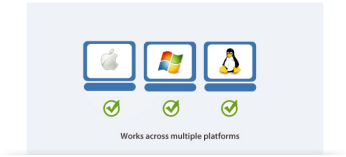
1. **What is Java?**
2. **Uses of Java**
3. **Problem with C and C++**
4. **Advantages of Java over C++**
5. **How does program execution works?**
6. **The way Java works**
7. **Compiler V/S JVM**
8. **JVM components**
9. **History of Java**

1.1.1 What is Java?

- Java is a popular **high-level programming language** that is **platform-independent**, **object-oriented** and **open source**.
- Let's understand each bold words in detail:
  - **High level programming langauge**: Language that are in english and can be understood by humans.



  - **Platform-independent**: Java code can run on any platform (i.e OS) using Java Virtual Machine (JVM).



  - **Object-Oriented**: Java is built around objects.

  - **Open source**: Java source code is available for anyone to view and modify.

1.1.2 Uses of Java

- **Enterprise-level applications** - Eg: SAP, IBM websphere, Salesforce, Oracle E-Business suite

- **Web applications** - Eg: LinkedIn, Netflix, Twitter, Amazon, Airbnb

- **Mobile applications** - Eg: Instagram, WhatsApp, Google Maps, Uber

- **Games** - Eg: Minecraft, RuneScape, Puzzle Pirates

- **Financial Applications** - Eg: Quicken, Bloomberg Terminal

- **Scientific Applications** - Eg: BioJava, ImageJ

1.1.3 Problem with C and C++

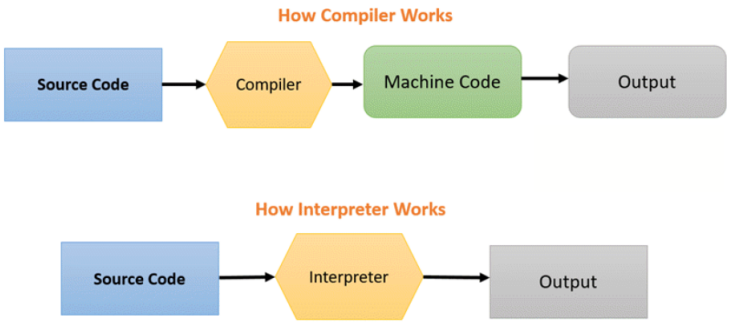
- **Original idea for Java was not the Internet!**
- Java was created to be **platform-independent language** that can be embedded in various consumer electronic devices, eg: **microwave ovens and remote controls**.
- The trouble with C and C++ is that they are **compiled for a specific target**.
- A full C++ compiler targeted for a specific CPU is needed to compile a C++ program for different CPU.
- The problem is that **compilers are expensive and time-consuming to create**.
- James Gosling (founder of Java) began work on a portable, platform-independent language that could be used to produce code that would run on any CPUs.
- This led to the creation of Java.

1.1.4 Advantages of Java over C++

- **Security**: No danger of reading bogus data when accidentally going over the size of an array.
- **Automatic memory management**:
  - Garbage collector allocate and deallocate memory for objects.
  - Pointers are not necessary.
- **Simplicity**:
  - No pointers, unions, templates, structures, multiple inheritance.
  - Anything in Java can be declared as a class.
- **Support for multithreaded execution**: Support development of multithreaded software.
- **Portability**: Support WORA (Write it once, run it anywhere), using Java virtual machine (JVM).

1.1.5 How does program execution works?

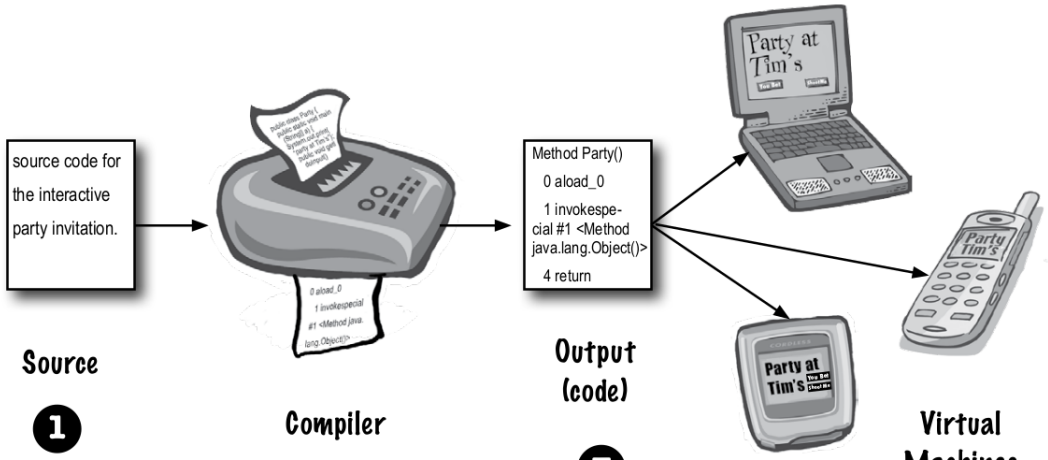
- **Compiler:**
  - A compiler is a **software program that converts source code written in a high-level programming language into machine code**, which can be executed by a computer.
  - The compiler performs:
    - Syntax analysis
    - Semantic analysis
    - Bytecode generation
- **Interpreter:**
  - An interpreter is a program that reads and executes code **line-by-line**, without the need to compile the entire program beforehand.
  - Interpreters run code in a virtual environment, where each line of code is executed as soon as it is read.



- **Java Virtual Machine (JVM):**
  - The JVM is a virtual machine that is responsible for executing Java bytecode.
  - The JVM is an example of an interpreter, as it reads Java bytecode and executes it line-by-line.
  - The JVM performs:
    - Memory management
    - Security
    - Garbage collection

1.1.6 The way Java works

So, what exactly happens to developer-written Java code until the actual execution?



Source

1

Create a source document. Use an established protocol (in this case, the Java language).

Compiler

2

Run your document through a source code compiler. The compiler checks for errors and won't let you compile until it's satisfied that everything will run correctly.

Output (code)

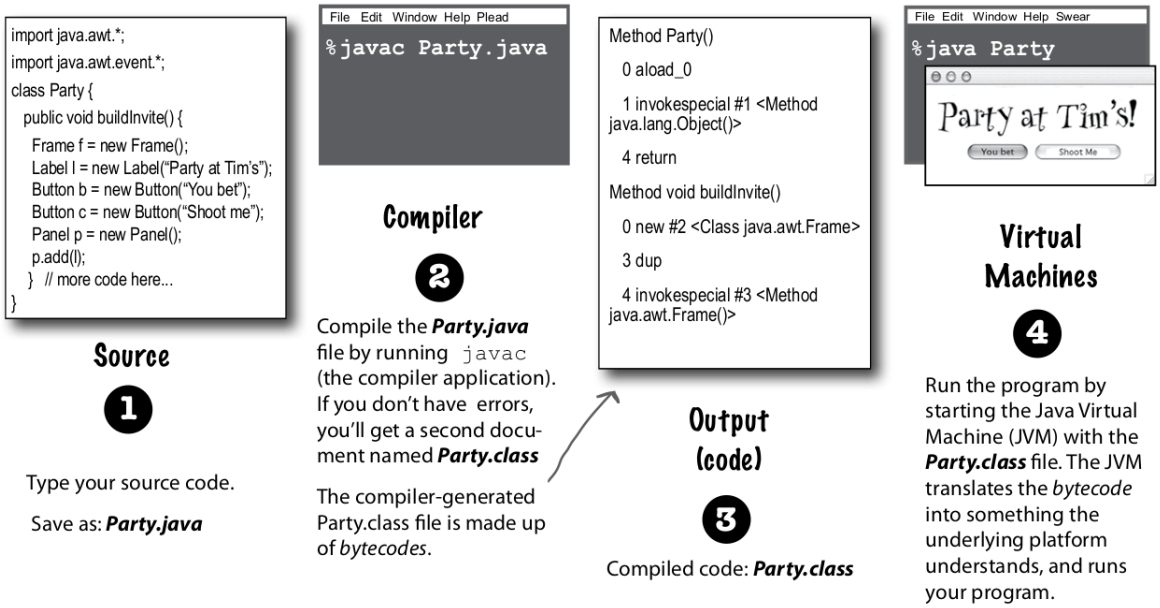
3

The compiler creates a new document, coded into Java ***bytecode***. Any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform-independent.

Virtual Machines

4

Your friends don't have a physical Java Machine, but they all have a ***virtual*** Java machine (implemented in software) running inside their electronic gadgets. The virtual machine reads and *runs* the bytecode.



Source

1

Type your source code.

Save as: ***Party.java***

Compiler

2

Compile the ***Party.java*** file by running `javac` (the compiler application). If you don't have errors, you'll get a second document named ***Party.class***

The compiler-generated Party.class file is made up of *bytecodes*.

Output (code)

3

Compiled code: ***Party.class***

Virtual Machines

4

Run the program by starting the Java Virtual Machine (JVM) with the ***Party.class*** file. The JVM translates the *bytecode* into something the underlying platform understands, and runs your program.

1.1.7 Compiler V/S JVM

Compiler and JVM battle over the question, "Who's more important?"

Ans: Go through below discussion to find the answer:

JVM: I am Java. I'm the guy who actually makes a program run. The compiler just gives you a file in bytecode after checking it's syntax. I'm the one who run it.

Compiler: Excuse me? Without me, you would have to translate everything from source code and be very very slow!

JVM: Your work is not important. A programmer could just write bytecode by hand. You might be out of a job soon, buddy.

Compiler: That's arrogant. A programmer writing bytecode by hand is next to possible, some scholars might write, not everyone!

JVM: But you still didn't answer my question, what you actually do?

Compiler: Remember that Java is a strongly-typed language, I can't allow variables to hold data of the wrong type. This is a security feature, implement by ME!

JVM: Your type checking is not very strict! Sometimes people put the wrong type of data in an array of different type.

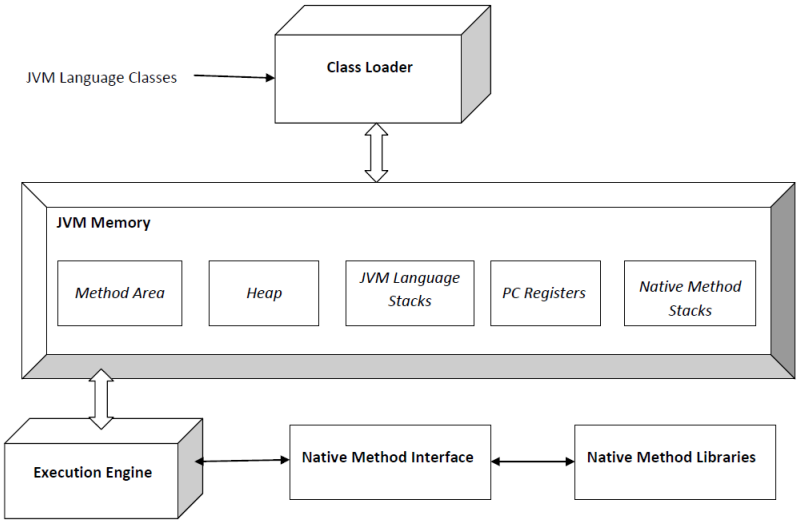
Compiler: Yes, that can emerge at runtime that only you can catch to allow dynamic binding. But my job is to stop anything that would never succeed at runtime.

JVM: OK. Sure. But what about security? Look at all the security stuff I do! You just perform silly syntax checking.

Compiler: Listen, I'm the first line of defense. I also prevents access violations, such as code trying to invoke a private method. I stop people from touching code they're not meant to see.

JVM: Whatever. I have to do that same stuff too!

1.1.8 JVM components



- **Class Loader:** Responsible for loading the class files into the memory of the JVM.
- **Execution Engine:** Responsible for executing the bytecode that is loaded into the memory. It includes:
  - **Interpreter:** Reads and executes the bytecode one instruction at a time.
  - **JIT compiler:** Compiles the bytecode into machine code for fast execution.
- **Garbage Collector:** It periodically frees up the memory that is not used by the Java application.
- **Runtime Data Area:** It is memory space allocated by the JVM for the execution of the Java application. It includes:
  - Method area
  - Heap
  - Stack
  - PC registers
- **Native Method Interface (JNI):** Allows Java code to call code written in other programming languages like C and C++. The JNI allows Java applications to interact with OS and hardware.

1.1.9 History of Java

What year Java was invented?

Ans: 1995

What company invented Java?

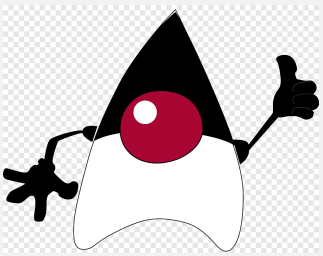
Ans: Sun Microsystems

Who is founder of Java?

Ans: James Gosling

What is Java mascot?

Ans: A cartoon character named Duke



What is the original name of Java ?

Ans: "Oak" after the oak tree that was outside Gosling's office.

What was the reason for changing original name?

Ans: "Oak" was already trademarked

What is the inspiration behind Java's name?

Ans: Java language is named after coffee grown on the Indonesian island

What is original Java logo?

Ans: Original logo:

Who has the current ownership of Java?

Ans: Oracle acquired Java in 2009

What are the Java versions?

Ans: Below are the Java version details:

Version	Year
JDK Alpha and Beta	1995
JDK 1.0	Jan, 1996
JDK 1.1	Feb, 1997
J2SE 1.2 or **Java2** (codename: Playground)	Dec, 1998
J2SE 1.3 or **Java2** (codename: Kestrel)	May, 2000
J2SE 1.4 or **Java2** (codename: Merlin)	Feb, 2002
J2SE 1. 5 or **Java5** (codename: Tiger)	Sep, 2004
Java SE 1.6 or **Java6** (codename: Mustang)	Dec, 2006
Java SE 1.7 or **Java7** (codename: Dolphin)	July, 2011
Java SE 1.8 or **Java8** (codename: Spider)	(18th March, 2014)
Java SE 1.9 or **Java9**	September, 2017
Java 10	March, 2018
Java SE 11	September 2018
Java SE 12	March 2019
Java SE 13	September 2019
Java SE 14	March 2020
Java SE 15	September 2020
Java SE 16	March 2021
Java SE 17	September 2021

Why is Java 2 consider very significant in history of Java?

Ans: Starting **Java 2**, it is composed of three parts:

- **J2SE (Java 2 Platform, Standard Edition) or JSE**, a computing platform for the development and deployment of portable code for **desktop and server environments**.
- **J2EE (Java 2 Platform, Enterprise Edition) or JEE**, extending Java SE with enterprise features such as **distributed computing and web services**.
- **J2ME (Java 2 Platform, Micro Edition) or JME**, a computing platform for **embedded and mobile devices**.

Other major highlights of this release:

- **JIT compiler** became part of JVM (means turning code into executable code became a faster operation).
- **Swing graphical API** was introduced as alternative to AWT.
- Java collections framework (for working with sets of data) was introduced.

I see Java 2 and Java 5.0, but was there a Java 3 and 4? And why is it Java 5.0 but not Java 2.0?

Ans: The joys of marketing...

- When the version of Java shifted from 1.1 to 1.2, the changes to Java were so many that the marketers decided a whole new "name", so they started calling it Java 2, even though the actual version of Java was 1.2.
- But versions 1.3 and 1.4 were still considered Java 2.
- There never was a Java 3 or 4.
- Beginning with Java version 1.5, the marketers decided a new name was needed.
- The next number in the name sequence would be "3", but calling Java 1.5 Java 3 seemed more confusing, so they decided to name it Java 5.0 to match the "5" in version "1.5".

1.2 Installing Java and IDE

In this section, you are going to learn:

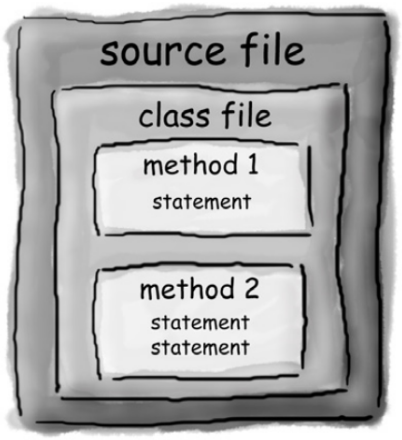
1. **Java installation**
2. **IDE installation**
3. **Our first Java program**
4. **Using JShell**

1.2.1 Java installation

1.2.2 IDE installation

1.2.3 Our first Java program

- Code structure in Java:



Put a class in a source file.

Put methods in a class.

Put statements in a method.

What goes in a source file?

Ans:

- A source code file with the **".java"** extension holds one class definition.
- The source file name should be **"classname.java"**.
- The class represents a piece of your program.
- The class must go within a pair of curly braces.
- Eg: Below code name will be Dog.java and class name will be Dog -

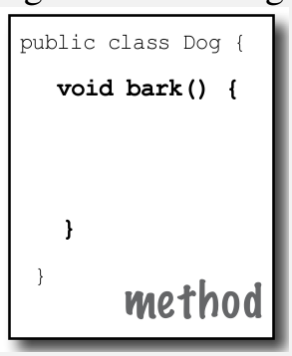
```
public class Dog {

}
```

class

What goes in a class?

Ans:

- A class has one or more methods.
- Your methods must be declared inside a class.
- Eg: The class Dog contains a method called bark()



What goes in a method?

Ans:

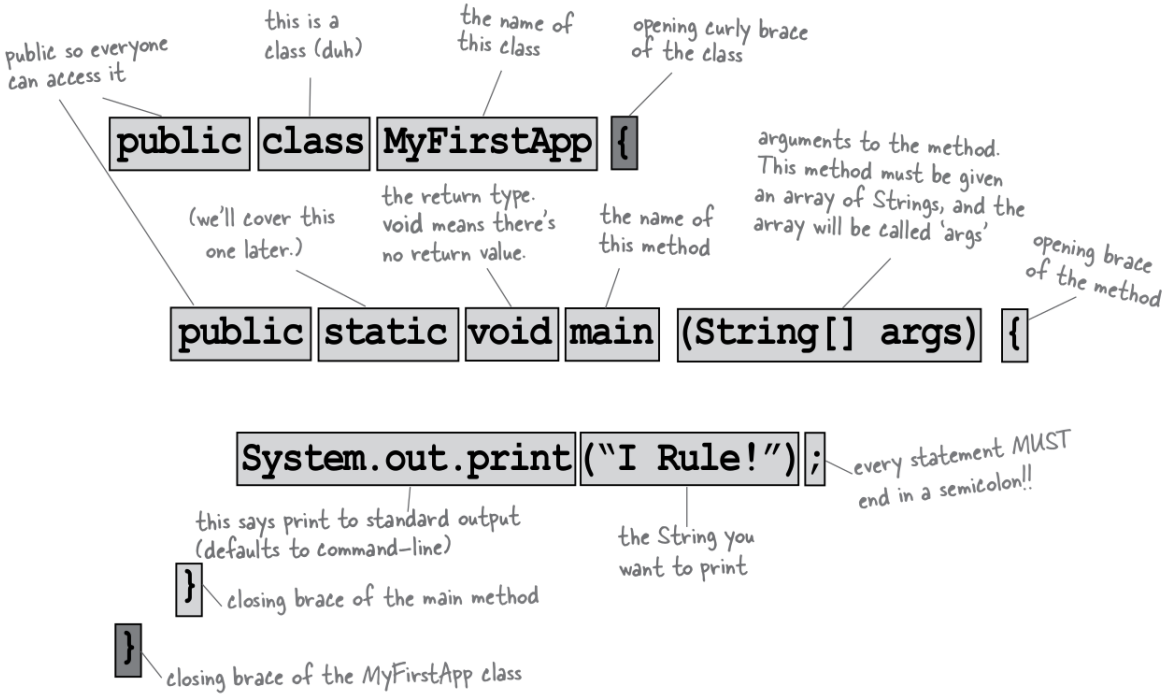
- Method code is basically a set of statements.
- Method kind of like a function or procedure.
- When the JVM starts running, it looks for the method inside the class that looks exactly like:

Code:

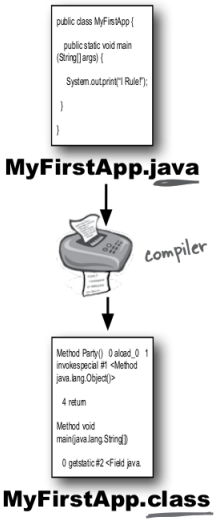
```
public static void main (String[] args) {
    // your code goes here
}
```

- JVM runs everything inside { } of your main method.
- Every Java application has to have at least one class, and at least one main method (not one main per class; just one main per application).

Overall, a basic Java program would look something like below:



Running your Java Code:




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

1 Save

`MyFirstApp.java`

2 Compile

```
javac MyFirstApp.java
```

3 Run

```
File Edit Window Help Scream
%java MyFirstApp
I Rule!
The World
```

1.2.4 Executing our first Java program

- Using simple text editor:

MyFirstApp.java

```
public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("I Rule!");
        System.out.println("The World");
    }
}
```

Command:

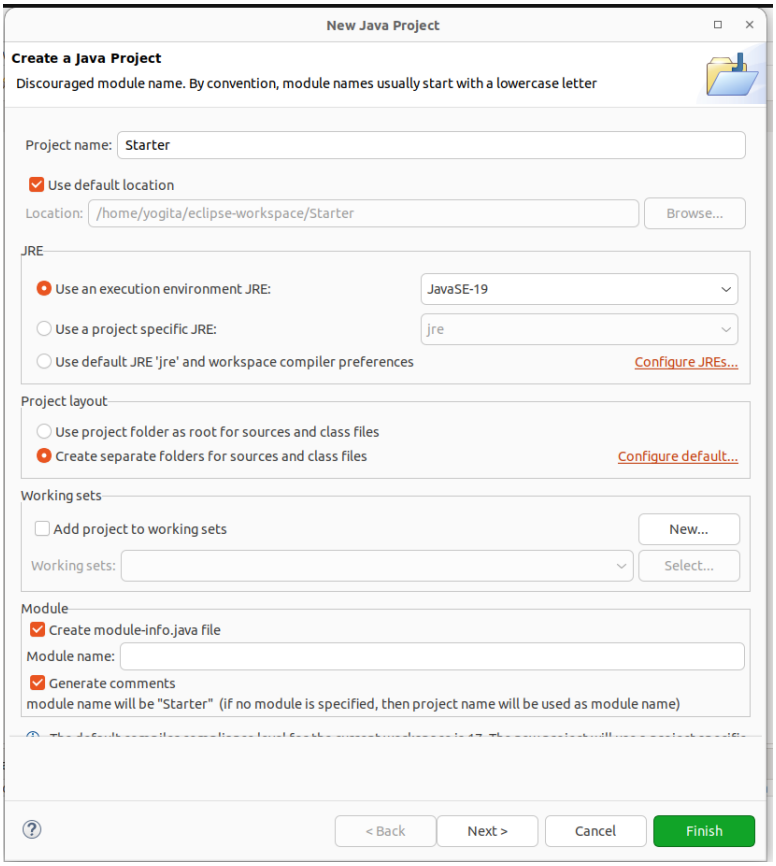
```
javac MyFirstApp.java
java MyFirstApp
```

Output:

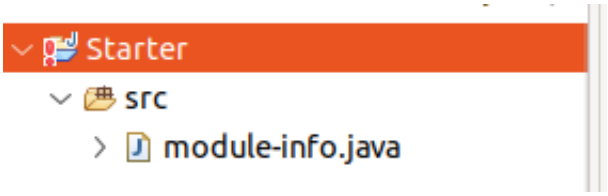
```
I Rule!
The World
```

- Using Eclipse:
  1. Create New Project:

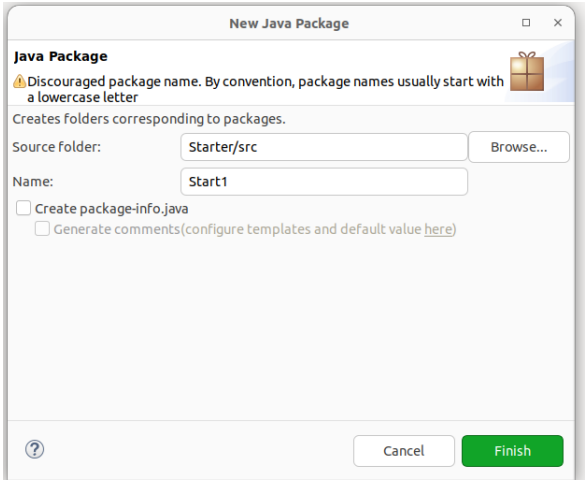
     File -> New -> Java Project -> Add "Starter" as project name



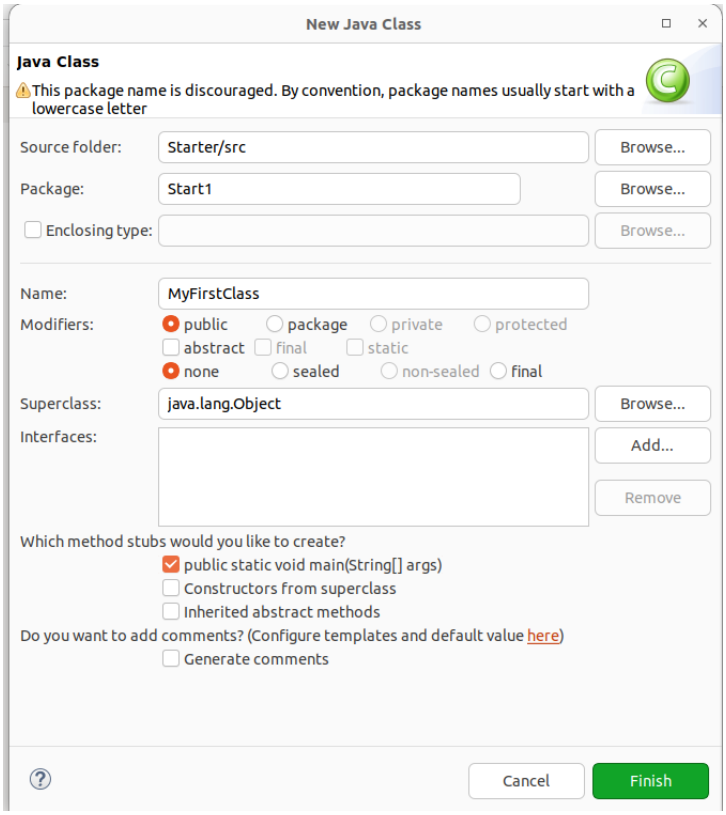
  2. This will create directory structure as shown below:



3. Right click the src -> Select “Package” -> Add “Start1” as package name as shown below:



4. Right click the “Start1” -> Select New -> Class -> Add “MyFirstClass” as classname as shown below:

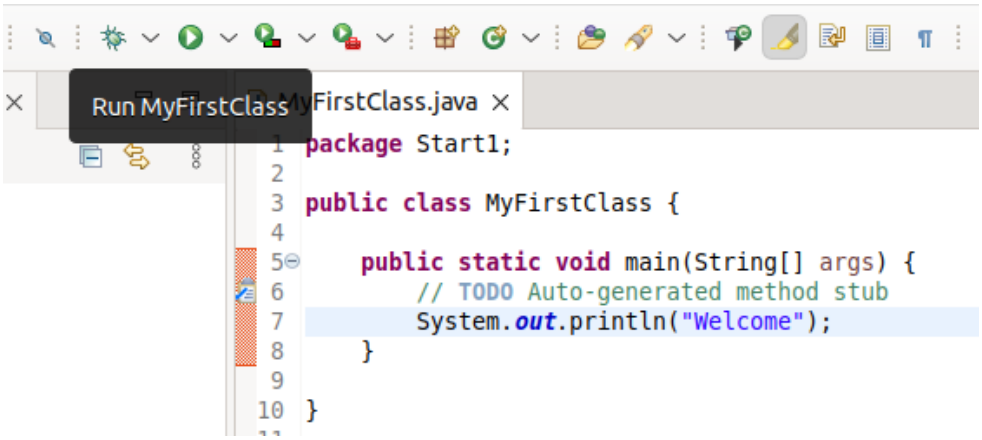


5. This will create a class with below structure:

Code:

```
package Start1;
public class MyFirstClass {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Welcome Back!");
    }
}
```

6. Execute the code by pressing the run button:



1.2.5 Using JShell

- JShell is a Read-Eval-Print Loop (REPL)
- It evaluates declarations, statements, and expressions as they are entered, and then it immediately shows the results.
- It was introduced in Java 9.
- To use jshell, type "jshell" command in command prompt:

Command:

```
$ jshell
| Welcome to JShell – Version 11.0.18
| For an introduction type: /help intro

jshell> int i=42;
i ==> 42

jshell> float j=3.4f;
j ==> 3.4

jshell> i+j
$3 ==> 45.4

jshell> String text = "Welcome To World of Java";
text ==> "Welcome To World of Java"

jshell> text.toUpperCase()
$5 ==> "WELCOME TO WORLD OF JAVA"
```

- To display all variables declared:

Command:

```
jshell> /vars
| int i = 42
| float j = 3.4
| float $3 = 45.4
| String text = "Welcome To World of Java"
| String $5 = "WELCOME TO WORLD OF JAVA"
```

- To save all valid statements of Jshell to a file:

Command:

```
jshell> /save filename.java


jshell> /exit
| Goodbye
```

- Content of filename.java:

filename.java

```
int i=42;
float j=3.4f;
i+j
String text = "Welcome To World of Java";
text.toUpperCase()
```

- You can open the file back in the jshell using open command:

Command:

```
jshell> /open filename.java


jshell> i
i ==> 42
```

Small steps every day.....

2. Java Language Fundamentals

2.1 Identifiers, variables and more

In this section, you are going to learn:

1. **Java identifiers**
2. **Java grammar**
3. **Java variable**
4. **Literals**
5. **Java reserved words or keywords**
6. **Java comments**
7. **How objects can change your life?**

2.1.1 Java identifiers

A Java identifier is a name of a variable, function, class, module or other object.
Eg:

```
package Starter;

public class Test {
    public static void main(String[] args) {
        int x=999;
        System.out.println(x);
    }
}
```

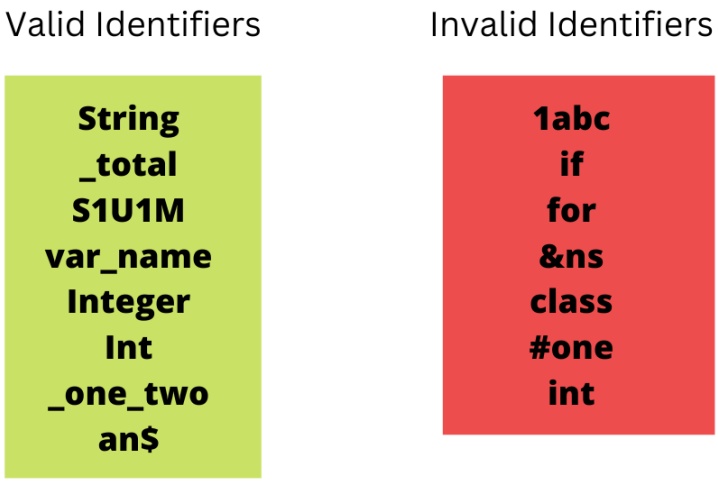
In above code, there are total 6 identifiers:

- Starter - name of package
- Test - name of class
- main - name of function
- String - name of class
- args - name of object
- x - name of integer variable

Rules of identifiers:

- Names can contain A-Z, a-z, 0-9, _, and $ signs.
- Names cannot begin with number.
- Names are case sensitive ("myVar" and "myvar" are different variables).
- Reserved words (like Java keywords, such as int or boolean) cannot be used as names.
- Names can be of any length, but it's not recommended to have big names.

- Developers should declare identifiers using the **Camel case** writing style (e.g., StringBuilder, isAdult)



2.1.2 Java grammar

- Java is **case sensitive**.
- **Block delimiters**: Except for import, package, interface (or @interface), enum and class declarations, everything else in a Java source file must be declared between curly brackets ().
- Code lines are ended in Java by the **semicolon (;) symbol**

2.1.3 Java variable

- The variable is the basic unit of storage in a Java program.

Syntax:

```
type identifier = value;
or
type identifier = value, identifier = value, identifier = value ... ;
```

Here,

- type = datatype or name of class or interface
- identifier = name of variable

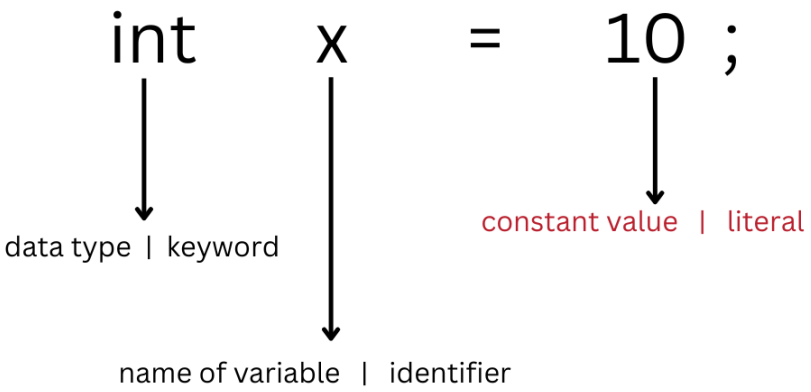
To declare more than one variable of the specified type, use a comma-separated list.

New.java

```
class New {
    public static void main(String[] args) {
        int a, b, c;
        int d = 3, e, f = 5;
        byte z = 22;
        double pi = 3.14159;
        char x = 'x';
    }
}
```

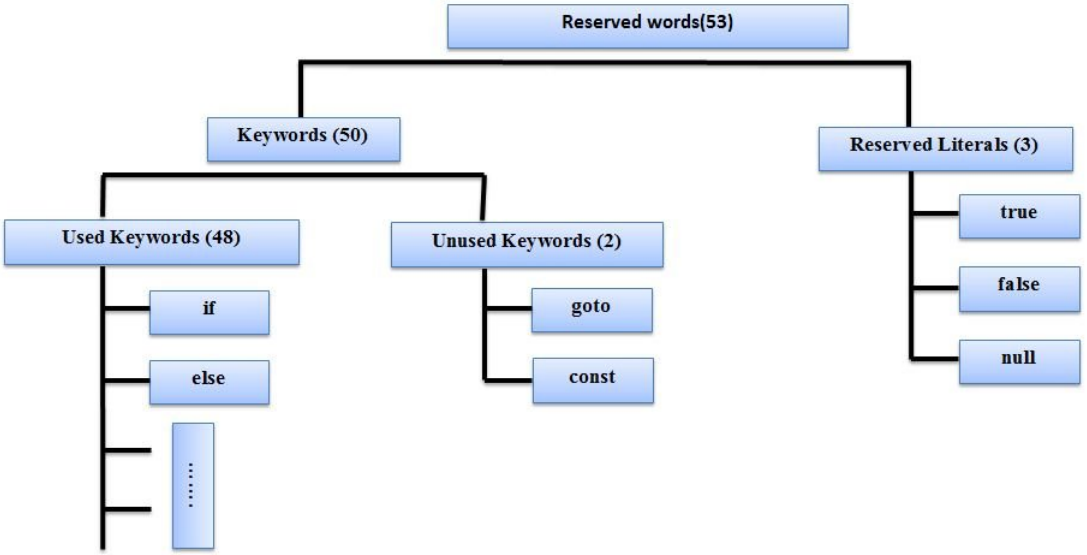
2.1.4 Literals

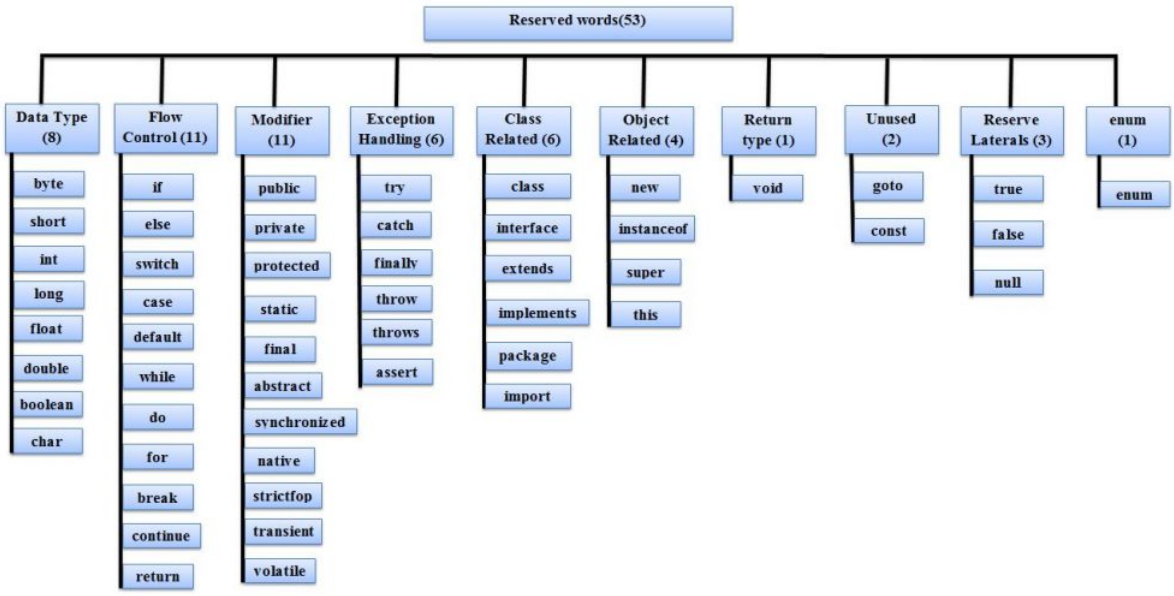
- A constant value which can be assigned to the variable is called **literal**
- Eg:



2.1.5 Java reserved words or keywords

- Words having **predefined meaning**
- Java 17 has a total of 60 reserved words.





Note:

const and **goto** are not used anymore!

2.1.6 Java Comments

- Java comments refer to text that are not considered part of the code execution and ignored by the compiler.
- 3 ways to add comments:
  - **//** : Used for single line comments:

Code:

```
// testing
```

  - **/** ... ***/**: Javadoc comments, special comments that are exported using special tools into the documentation of a project called Javadoc API

Code:

```
/**
* Returns the sum of two integers.
* @param a the first integer to add
* @param b the second integer to add
* @return the sum of a and b
*/
public int add(int a, int b) {
    return a + b;
}
```

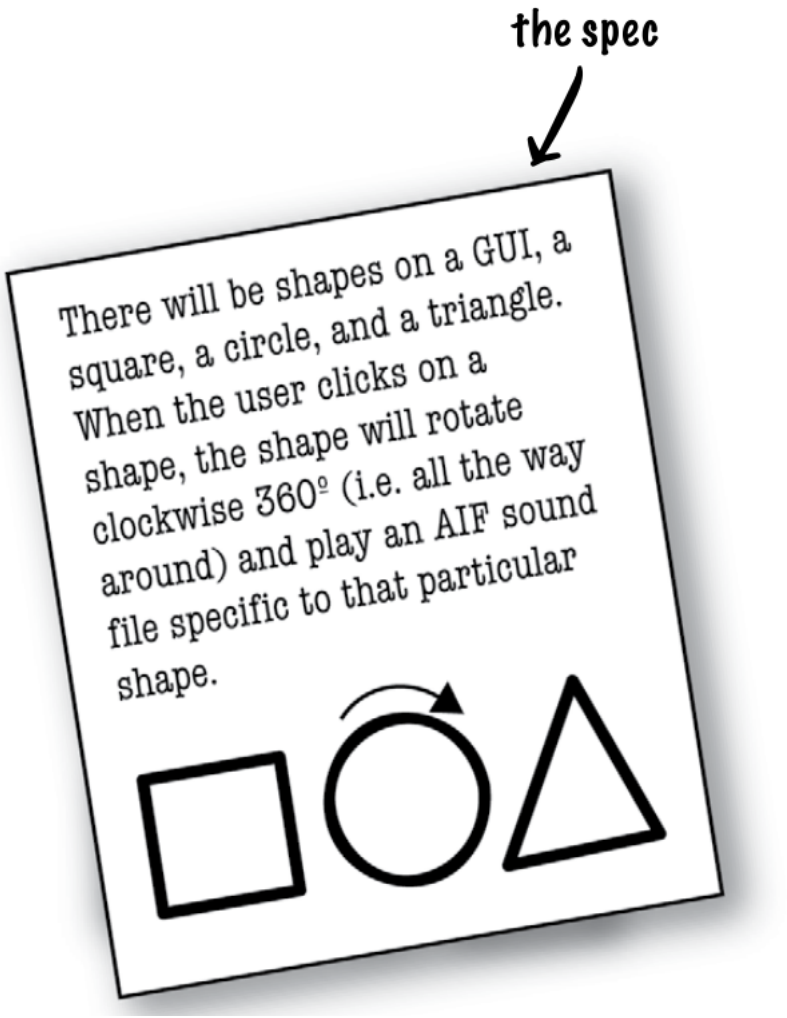
  - **/* ... */** : used for multiline comments

Code:

```
/* This is
a multi line
statement */
```

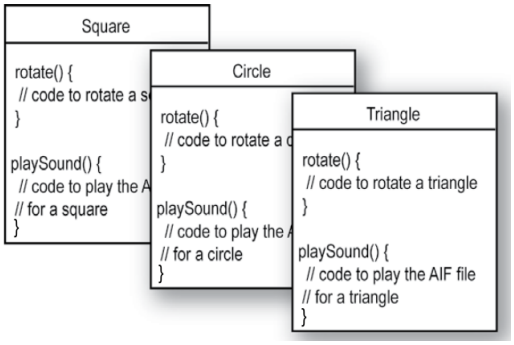
2.1.7 How Objects Can Change Your Life?

- So far, we put all of our code in the **main() method**.
- **That's not exactly object-oriented**.
- Leave the procedural world behind, get the out of main(), and start making some objects of our own.
- We'll look at what makes object-oriented (OO) development in Java so much fun.
- Let's understand this with a use-case:
  Creat a Shape Application with below requirement:

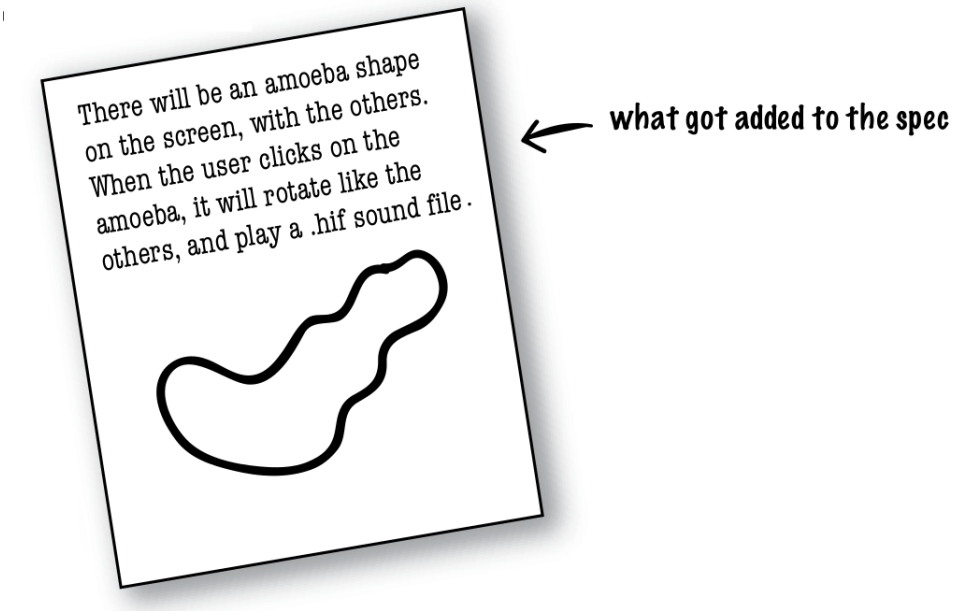


- For this, Raj followed Procedure-Oriented approach while Ram followed Object-Oriented approach to develop the code.

Raj's Procedure-Oriented approach	Ram's Object-Oriented approach

Code:
rotate(shapeNum) {
//code here
}
playSound(shapeNum) {
//code here
}
	

- But wait! There's been a spec change.



- In order to reflect the new requirements, here's what Raj and Ram decided to do:

Back in Raj's cube	At Ram's laptop in a cafe

Raj will need to make changes in existing code and perform the testing again:
 	
Code:
rotate(shapeNum) {
//new changes to add amoeba
}
playsound(shapeNum) {
//add change for amoeba
}
	Ram just need to create one more class called "Amoeba":
	
	Amoeba
	rotate() {
	// code to rotate an amoeba
	}
	playSound() {
	// code to play the new
	// .hif file for an amoeba
	}

So what do you like about OO?

Ans:

- Design software as per real world usage.
- New changes can be incorported easily.
- Not messing around with code already tested, just to add a new feature.
- Data & methods that operate on that data are together in one class.
- Code can be re-used in other applications.

Let's dig a bit deeper into OOP's

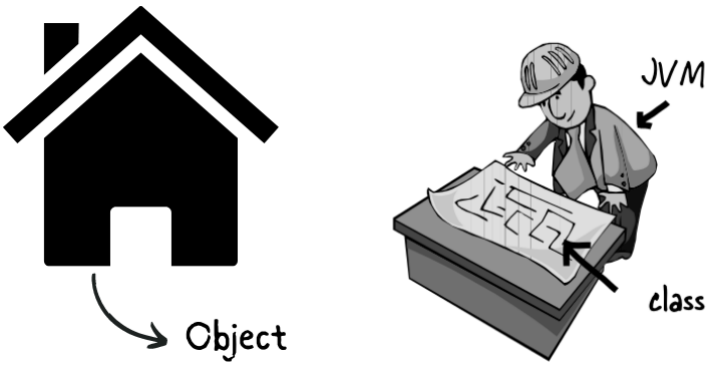
2.2 Introductions to OOPs

Object-**O**riented **P**rogramming (OOP) is a programming paradigm that revolves around the concept of "objects".

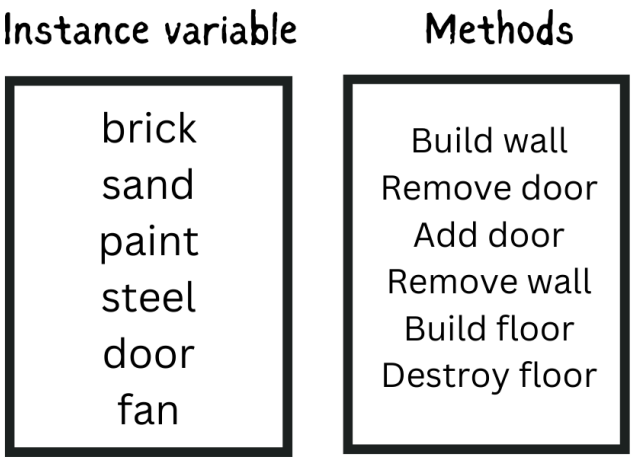
In Java, OOP is implemented through the use of classes and objects.

Class & Object

- A class is a blueprint for an **object**.
- It tells the JVM how to make an object of that particular type.
- An Object is real world entity. It is an instance of class.



- A class consists of instance **variable and methods**:
  - **Instance variable:** Represents the object data.
  - **Methods:** Things an object can do are called methods.

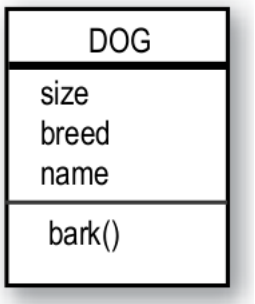


- Java code for class and object would look something like below:

```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

instance variables

a method




```
class DogTestDrive {
   public static void main (String[] args) {
     Dog d = new Dog();
     d.size = 40;
     d.bark();
   }
}
```

make a Dog object

use the dot operator (.) to set the size of the Dog and to call its bark() method

dot operator

2.2.1 Java source file

- A Java program can contain any number of classes in a single program.
- However, only 1 class can be declared as **public**.
- If there is a public class, then **name of the program and name of public class must be matched**, otherwise the program will result in compile time error.
- Below are some use-cases on this:
  - **Case I**: If there is no public class and the program contains multiple class, then the program name can be anything.

Lava.java

```
class A {}
class B {}
class C {}
```

Command:

```
$ javac Lava.java
// This will create A.class, B.class, C.class
```

  - **Case II**: If **class B is public**, then name of the program should be **B.java**, otherwise we will get compile-time error saying: **class B is public should be declared in file named B.java**:

Lava.java

```
class A {}
public class B {} ✗// Lava.java is incorrect name
class C {}
```

B.java

```
class A {}
public class B {} ✓
class C {}
```

– **Case III**: If class B and C are declared as public and name of program is "B.java", then we will get compile time error saying: class C is public, should be declared in a file named C.java

B.java

```
class A {}
public class B {}
public class C {} ✗        // Two class cannot be
public
```

– **Class IV:** If a program contains main() for multiple class, then executing those specific class will execute their respective main()

Lava.java

```
class A {
    public static void main(String[] args) {
        System.out.println("A class main");
    }
class B {
    public static void main(String[] args) {
        System.out.println("B class main");
    }
class C {
    public static void main(String[] args) {
```

```
            System.out.println("C  class  main");
    }
class  D  {}
}
```

Command:

```
$  javac  Lava.java
A.class  B.class  C.class  D.class

$  java  A
A  class  main

$  java  B
B  class  main

$  java  C
C  class  main

$  java  D
RuntimeError:  NoSuchMethodError:  main()

$  java  Lava
RuntimeError:  NoClassDefFoundError:  Lava
```

Conclusion:

- While executing a java program, for every class present in that program, a separate **“.class”** will be generated.
- You can compile a java program (Java source file), but you can run a java **".class"** file.
- On executing a java class, the corresponding class main() will be

executed.

- If the class doesn't contain main(), then you will get runtime exception.
- If the corresponding .class file is not available, then you will get runtime exception.
- It is not recommended to declare multiple classes in a single source file.
- It is recommended to declare only one class per source file and name of the program to be same as class name.

2.2.2 main() method

- **main()** serves as the entry point for a Java program.
- When a Java program is executed, the JVM starts by looking for the main() method in the class specified in the command line arguments, and then executes the code inside it.

Syntax:

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

- At runtime, JVM always searches for main method with the above prototype:
  - **public:** To call main() from anywhere
  - **static:** without existing object also, JVM has to call this method
  - **void:** main() method wont return anything to JVM
  - **main:** This is the name which is configured inside JVM
  - **String[] args:** command line argument

Note:

The main() syntax is very strict and if we perform any change then we will get runtime error from JVM saying "NoSuchMethodError: main".

- Changes allowed in main():
  - Order of modifier can be changed:
    Eg: static public void main(String[] args)
  - The command line argument's string array can have different syntax:
    Eg: public static void main(String args[])
  - Identifier of the string array can change:
    Eg: public static void main(String name[])
  - String array can be taken as var_arg parameter
    Eg: public static void main(String... args)

- main() method can be declared with following modifiers:
  - final
  - synchroised
  - strictfop

Eg:

New.java

```
class New {
        static final synchronized strictfp public void
main(String... name){
            System.out.println("Valid main method");
    }
}
```

- There can be multiple main() methods (i.e main() method over-loading is possible!). However JVm will always call String[] argument main method only.

New.java

```
class New {
    public static void main(String[] args){
        System.out.println("Starting");
    }     public static void main(int[] args){
        System.out.println("Sample 2");
    } }
```

Output:

```
Starting
```

- **Inheritance:** While executing child class, if child does not contain main(), then parent class main() will be executed.

New.java

```
class New {
    public static void main(String[] args){
        System.out.println("Starting");
    }
}
class C extends New {}
```

Command:

```
$ javac New.java <- Creates C.class New.clas New.java
$ java New
Starting
$ java C
Starting
```

- **Method hiding**: Child class can override parent class's main(). This is not method overriding but it is method hiding.

  Eg:

New.java

```
class New {
    public static void main(String[] args){
        System.out.println("Starting New");
    }
}
class C extends New {
    public static void main(String[] args){
        System.out.println("Starting C");
    }
}
```

Command:

```
$ javac New.java <- Creates C.class New.clas New.java
$ java New
Starting New
$ java C
Starting C
```

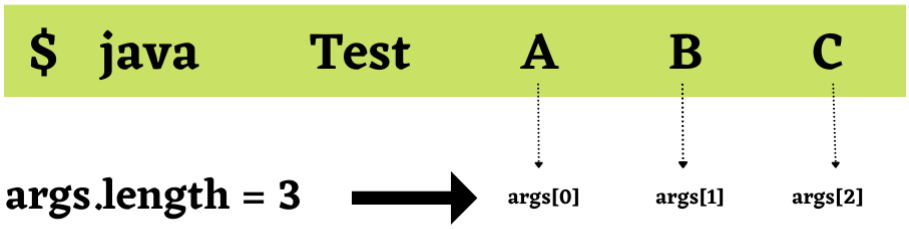
Note:

- Whether class contains main() method or not, and whether main() method is declared according to requirement or not, **these things are won't be checked by compiler**.
- At runtime, **JVM is responsible to check these things**.
- If JVM unable to find main() method, then will throw runtime exception!

2.2.3 Command-line argument

- Command line arguments are values passed to Java program when it is run from the command line.
- With these command line arguments, JVM creates an array and pass it to main().
- Command line arguments can be accessed using the args parameter of the main().
- Args parameter is an array of String objects.
- You can customise behaviour of main() using command-line argument:

```
public static void main(String[] args) // ← Here, String[] args contains command line args
```



- Command line argument are always String[]

New.java

```
class New {
    public static void main(String... args) {
        for(int i = 0; i < args.length; i++) {
        System.out.println(args[i]);
        }
    }
}
```

Command:

```
$ javac New.java
$ java New 1 23 3
1
23
3
```

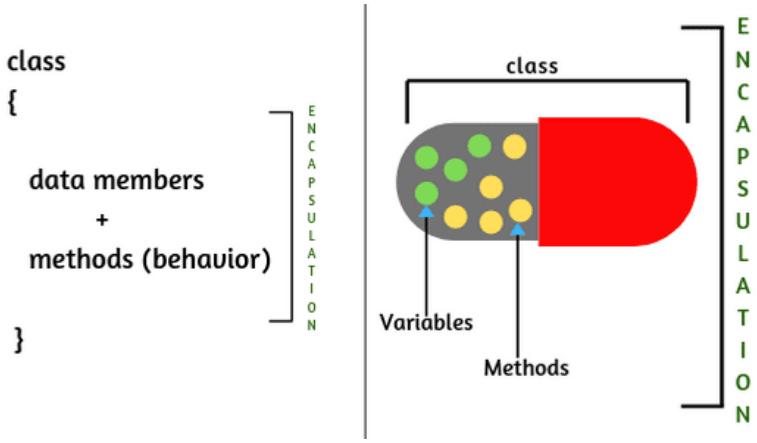
- Command line arguments are separated by space. To give one argument with space character, using "" :

Command:

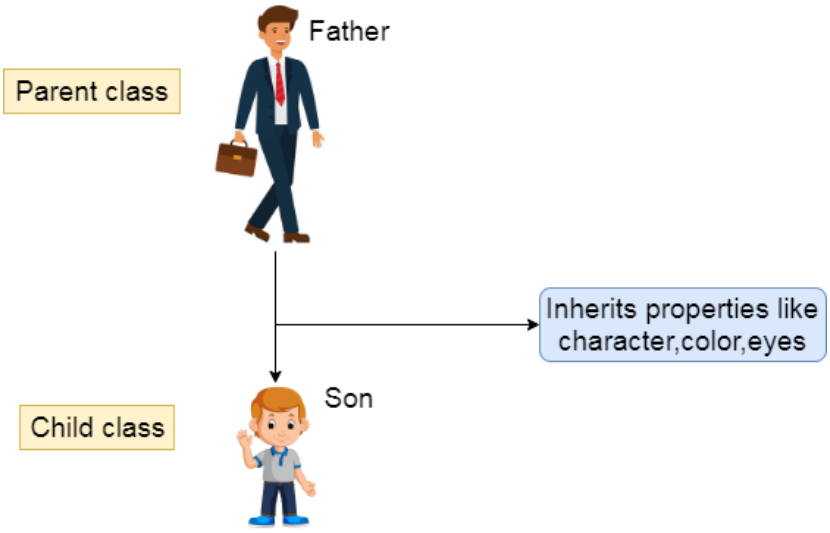
```
$ java New "Note Book"
```

2.2.4 Pillars of OOPs

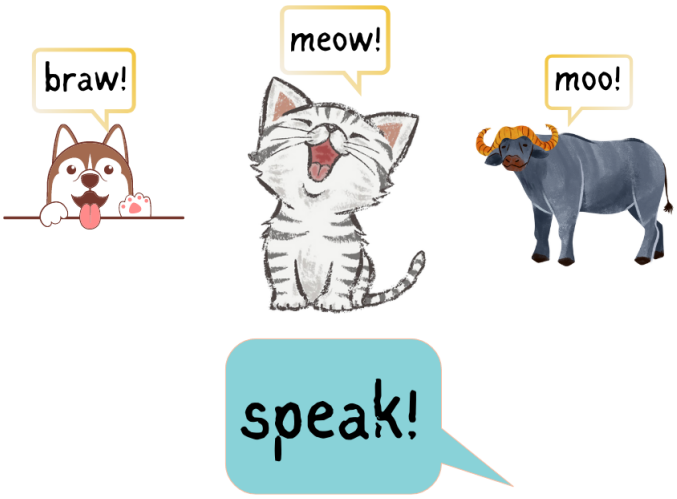
- **Encapsulation:**
  - Technique of bundling data and methods in a class.
  - With encapsulation, data cannot be accessed from outside the class.
  - This protects the data from accidental modification.



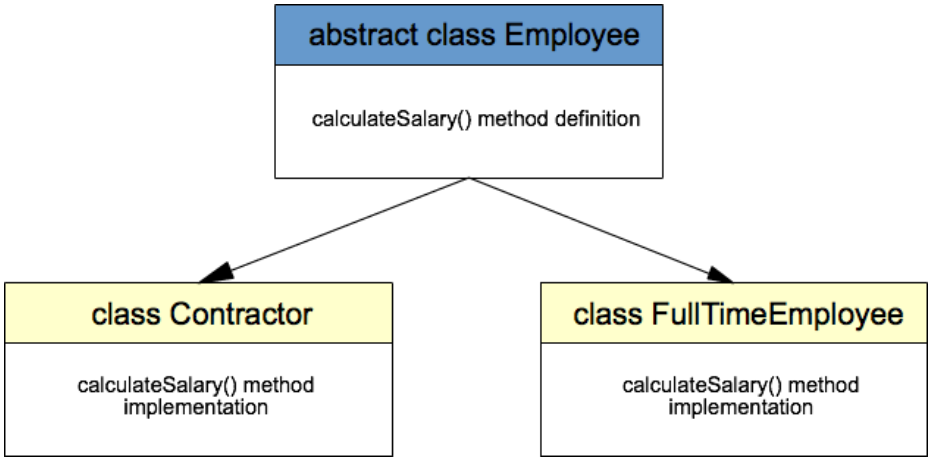
- **Inheritance:**
  - Inheritance is one class acquires the properties (methods and fields) of another class.
  - It allows code reusability and saves time and effort.



- **Polymorphism:**
  - Polymorphism is ability of objects of different classes to be treated as if they belong to a common superclass.
  - Polymorphism allows methods to be written that can work with objects of many different classes, as long as they share a common interface.
  - This enables code to be more flexible and adaptable to changing requirements.



- **Abstraction:**
  - Abstraction hides implementation details.
  - It shows only the essential features of an object.



Small steps every day.....

3. Data types in Java

3.1 Getting started with data type

In this section, you are going to learn:

1. **Java is strongly typed**
2. **Types of data types**
3. **Integer data type in detail**
4. **Floating-point data type in detail**
5. **Character data type in detail**
6. **Boolean data type in detail**

3.1.1 Java is strongly typed

This means:

- Every variable has a type and every type is strictly defined
- All assignments are checked for type compatibility
- There are no automatic conversions of conflicting types
- Compiler checks all variables to ensure that the types are compatible.
- Any type mismatches errors must be corrected before the compiler will finish compiling the class.

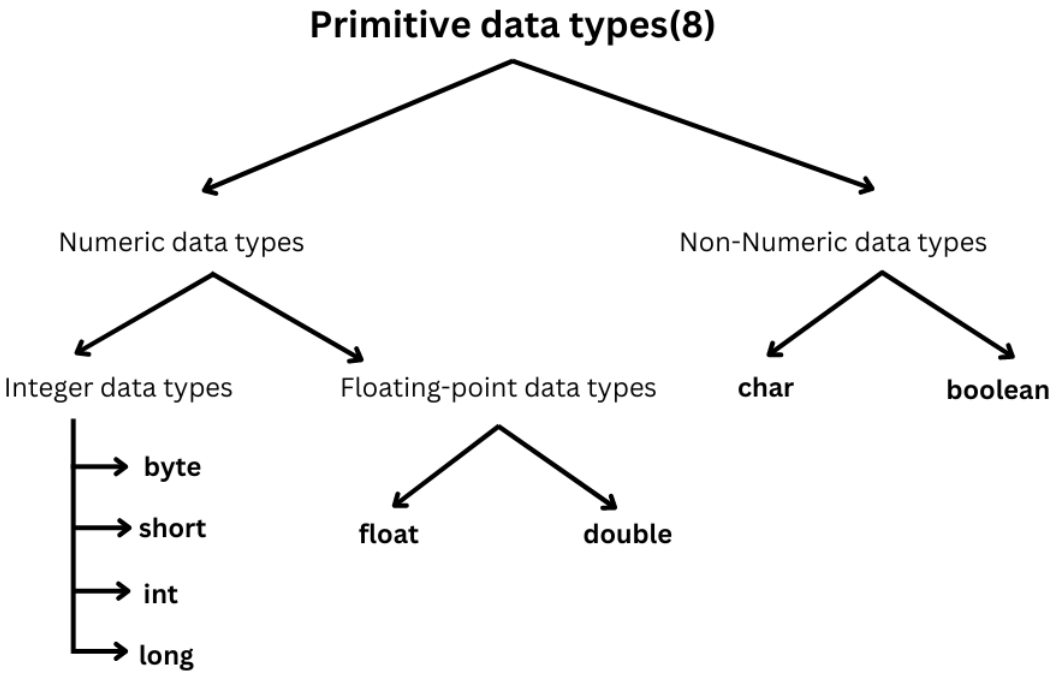
3.1.2 Types of data types

There are two types of data types in Java:

- **Primitive data types**: Includes boolean, char, byte, short, int, long, float and double.
- **Reference data types**:
  - These are not predefined by the language.
  - They are instead created by the programmer using class definitions.
  - Examples of reference data types include:
    * Classes
    * Interfaces
    * Arrays
    * Strings
    * Enumerations

We shall see primitive data type in detail in this chapter.

3.2 Integer



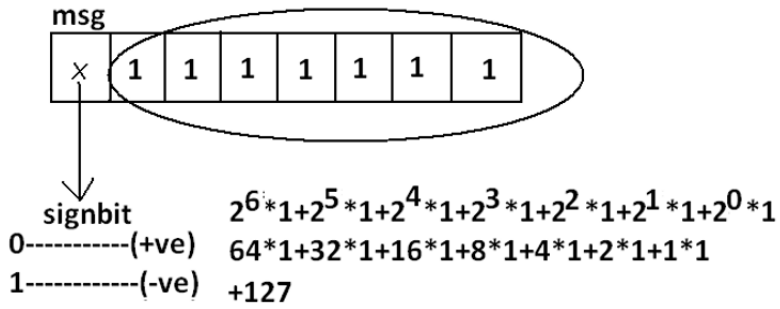
There are 4 types of integer in Java:

- byte
- short
- int
- long

3.2.1 byte

- byte keyword is an 8-bit signed integer.

Size	**1 byte (8 bits)**
MAX_VALUE	**+127**
MIN_VALUE	**-128**
Range	**-128 to 127**



- Left most bit (also called **m**ost **s**ignificant **b**it) is sign bit , where
  - 0 is positive number
  - 1 is negative number

Valid bytes values

```
byte b=10;
byte b=127;
byte b=-120;
```

Invalid bytes values

```
byte b=10.5;
byte b=true;
byte b="lava";
```

- **Where is byte used?**
  - **Reading and writing binary data**
  - **Image processing**
  - **Audio processing**
  - **Network programming**

3.2.2 short

- Least frequently used data type.
- short keyword is 16-bit signed integer.

Size	**2 byte (16 bits)**
MAX_VALUE	32767
MIN_VALUE	-32768
Range	**-2^{15} to $2^{15}-1$**

- **Where is short data type used?**
  - Short data type used for 16 bit processor like 8085.

Valid short values

```
short s = 32767;
short s = -32767;
```

Invalid short values

```
short s=10.5;
short s=true;
```

3.2.3 int

- Mostly commonly used data type is int.
- int keyword is 32-bit signed integer.

Size	**4 byte (32 bits)**
MAX_VALUE	2147483647
MIN_VALUE	-2147483648
Range	**-2^{31} to $2^{31}-1$**

Valid int values

```
int x = 2147483647;
int x = -90;
```

Invalid int values

```
int b=10.5;
int b=true;
int b="lava";
```

3.2.4 long

- long keyword is 64-bit signed integer.

Size	**8 byte (64 bits)**
MAX_VALUE	2^{63}-1
MIN_VALUE	-2^{63}
Range	**-2^{63}** to **2^{63}-1**

- **Where is long data type used?**
  - Eg 1: Amount of distance travelled by light in 1,000 days. To hold this value integer is not enough. Hence, long is used. long l = 1,26,000 X 60 X 60 X 24 X 1000 miles.
  - Eg 2: The number of characters present in a big file may exceed int range. Hence, the return type of length() is long but not integer.

Code:

```
long l = f.length()
```

Long literals

- **long** data type can be suffixed with "l" or "L".
- Below are valid **long** data type:

Code:

```
long l = 10L; ✓
long b = 10; ✓
```

- However, below declaration will result in error:

Code:

```
int x = 10L; ✗
```

3.2.5 Integer literals

- For integral datatypes like byte, short, int & long, we can specify literal value in the following base:
  - **Decimal literal (base-10)**:
    - Allowed digits are 0-9

Code:
```
int x = 10;
```

  - **Binary literal (base-2)**:
    - From Java 1.7 version, integral literal can be represented as binary value.
    - Allowed digits are 0 and 1
    - Literal value should be pre-fixed with "0B" or "0b"

Code:
```
int x = 0B10;
int y = 0b10101;
```

  - **Octal literal (base-8)**:
    - Allowed digits are 0-7
    - Literal value should be pre-fixed with 0

Code:
```
int x = 017;
```

  - **Hexadecimal literal (base-16)**:
    - Allowed digits are 0-9, a-f or A-F
    - Literal value should be pre-fixed with "0X" or "0x"

Code:
```
int x = 0X13aA;
int x = 0x45Fe;
```

  - **Usage of _ in numeric literal**:
    - From Java 1.7 version, we can use "_" in middle of big numbers to increase integer's readability.

* At the time of compilation, these "_" symbols will be removed automatically.

Code:

```
int x = 78_32_34_23;
int y = 0Xaa_ff_11;
int z = 034_12_10;
int a = 0B11__00_10_11;
```

* "_" symbol cannot be used in the starting or end of integer. Below are **invalid** declarations:

Code:

```
int x = _78_32_34_23;
int y = 0Xaa_ff_11_;
```

Program to convert octal and hexadecimal form of integer to decimal form:

test.java

```
package Starter;
class test
{
    public static void main (String[] args)
   {
      int x = 10;
      int y = 061;
      int z = 0x9a;
      System.out.println(x+","+y+","+z);
   }
}
```

Output:

```
10,49,154
```

Output Explaination:

In Java, integeral literals are always represents in decimal literals forms:

- Octal to decimal form:

$$(61)_8 = (?)_{10}$$

$$6 \times 8^1 + 1 \times 8^0 = 49$$

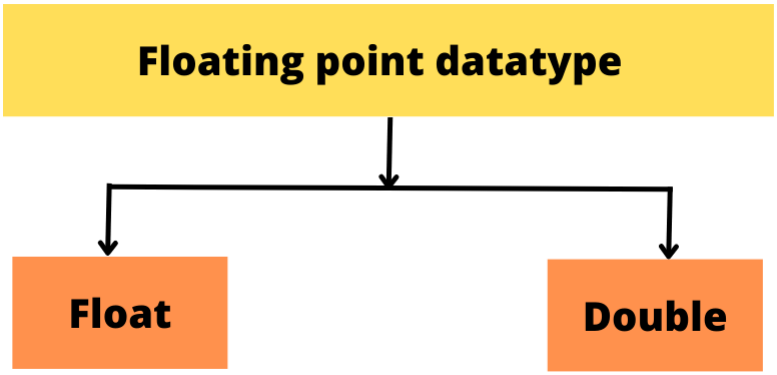
- Hexadecimal to decimal form:

$$(9a)_{16} = (?)_{10}$$

$$9 \times 16^1 + 10 \times 16^0 = 154$$

3.3 Floating-point

Floating-point numbers, also known as real numbers. There are two kinds of floating-point type:



3.3.1 Float

- Represent **single-precision** numbers (upto 7 decimal digits)

Size	**4 byte (32 bits)**
MAX_VALUE	3.4e38
MIN_VALUE	-3.4e38
Range	-3.4e38 to 3.4e38

3.3.2 Double

- Represent **double-precision** numbers (upto 16 decimal digits)

Size	**8 byte (64 bits)**
MAX_VALUE	1.7e+308
MIN_VALUE	1.7e-308
Range	1.7e-308 to 1.7e+308

3.3.3 Floating-point literals

- By default, floating-point numbers are represented in double form.
- So below declaration will result in error:

Code:

```
float f = 1.0 ✗
```

- Correct way to represent float data type is by suffixing **"F"** or **"f"** to the floating-point number as shown:

Code:

```
float f = 1.6F; ✓
float f = 7.8f; ✓
```

- Double data type can be represented using suffix **"D"** or **"d"** or no suffix as below:

Code:

```
double a = 12.67; ✓
double b = 13.7d; ✓
double c = 123.456D; ✓
double d = 0123.456; ✓
double e = 0789.9; ✓
```

- Floating-point literal are only in decimal form, not in octal and hexa decimal forms. Below are **invalid** declarations:

Code:

```
float a = 045.8; ✗
float b = 0X56.9; ✗
double c = 0X56.9; ✗
```

- We can assign integral literal directly to floating-point variables like double and float. Below are valid declarations:

Code:

```
double a = 0456; ✓
double b = 0XFace; ✓
double c = 10; ✓
float a = 0456; ✓
float b = 0XFace; ✓
float c = 10; ✓
```

- **Expontential format:** This is scientific notation to represent very large or small floating-point values. Use the letter “e” or “E” to indicate the exponent:

Code:

```
double a = 1.2e3; ✓
float b = 1.3e4F; ✓
```

- **Hexadecimal floating-point literals:** You can represent double and float in hexadecimal form using the letter “p” or “P”:

Code:

```
double d = 0x12.2P2; ✓
float e = 0x12.2P2f; ✓
```

- **Usage of _ in floating literal**:
  - From Java 1.7 version, we can use "_" in middle of big numbers to increase floating-point’s readability.
  - At the time of compilation, these "_" symbols will be removed automatically.
  - Eg:

Code:

```
float x = 78_3.2_34_23f; ✓
double y = 12_45_23__23_2323.90; ✓
```

- "_" symbol cannot be used in the starting or end of integer or decimal point. Below are **invalid** declarations:
- Eg:

Code:

```
float x = 78_3.2_34_23f_; ✓
double y = _12_45_23__23_2323.90; ✓
double z = 12_45_2_.3__23_2323.90; ✓
```

3.4 Character

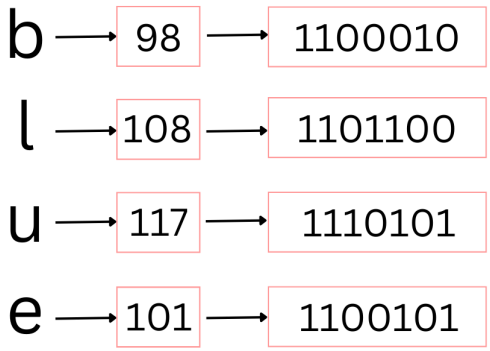
- In order to understand char data type, we need to understand what is "ASCII" and "UNICODE"

3.4.1 What is ascii & unicode?

- **ASCII (American Standard Code for Information Interchange)**:
  - Is a character encoding system that **represents text in computers**.
  - It uses a 7-bit code to represent 128 characters, including the letters of the English alphabet, digits, punctuation marks, and some control codes.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]



- **Unicode**:
  - It is a character encoding standard designed to support the representation of **all the world's languages**.

3.4.2 char datatype

- Java **uses unicode** to represent **char datatype**.
- There are **no negative chars**.
- Character is represented in **single quotes**.

Size	**2 byte (16 bits)**
MAX_VALUE	65,535
MIN_VALUE	0
Range	0 to 65,535

Eg:

Code:

```
class New {
    public static void main(String[] args) {
        char a = 88;
        char b = 'x';
        System.out.println(a);
        System.out.println(b);
    }
}
```

3.4.3 Character literals

- Char literal can be represented as **single character within single quotes**.

Code:

```
char ch='a'; ✓
```

- Below char literal declaractions will **result into compile time errors**:

Code:

```
char ch = "a"; ✗
char ch = a; ✗
char ch = 'ab'; ✗
```

- Char literal can also be **represented as integral literal** which represents the unicode value of character.
  The unicode value can be specified in decimal, octal and hexa decimal form.

Code:

```
char ch1 = 97; ✓
char ch2 = 0xFace; ✓
char ch3 = 0777; ✓
char ch = 65535; ✓
```

  Note that allowed range is **0 to 65535**.

- Char literal can also be represented in **unicode representation** by using "\uXXXX" syntax where "XXXX" is 4 digit hexa decimal number.

Code:

```
char ch1 = '\u0052'; ✓
char ch2 = '\u0932'; ✓
char ch3 = \uface; ✓
```

- Char literal can also **represent escape sequence characters**.

Code:

```
char ch1 = '\n'; ✓
char ch2 = '\t'; ✓
```

- Below are some more example of **invalid** char declarations:

Code:

```
char ch1 = 65536; ✗
char ch2 = 0XBear; ✗
char ch3 = '\m'; ✗
char ch4 = '\iface'; ✗
```

3.4.4 Escape character

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.
The following table shows the Java escape sequences:

Escape Character	Decimal Point
\n	New line
\t	Horizontal Tab
\r	Carriage return (Move to first character in next line)
\b	Back Space
\f	form feed
\’	single quote
\"	double quote
\\	Back Slash

Eg:

Code:

```
System.out.println("This is line one \n And this is line two");
System.out.println("This is \t tab space");
System.out.println("Sunflower \r Forest");
System.out.println("Sunflower \f Forest \f ground");
System.out.println("C:\\lavatech_technology");
System.out.println("Sunflower\’s Forest");
```

3.5 Boolean

- Boolean is datatype for logical values.
- Boolean is return by all relational operators.

Size	**1 byte (8 bits)**. The actual memory usage may depend on JVM implementation.
Value	true, false

Valid boolean values

```
boolean b = true;
boolean flag = false;
```

Invalid boolean values

```
boolean b = True;
boolean b = "True";
boolean b = 0;
```

Note:

- The values of true and false do not convert into any numerical representation.
- The true literal in Java does not equal 1, nor does the false literal equal 0.

Like C and C++, below code will not result in interpreting 0 and 1 as boolean and result in error

```
int x = 0;
if (x)
{
   System.out.println("Hello");
}
else
{
   System.out.println("Hi");
}
```

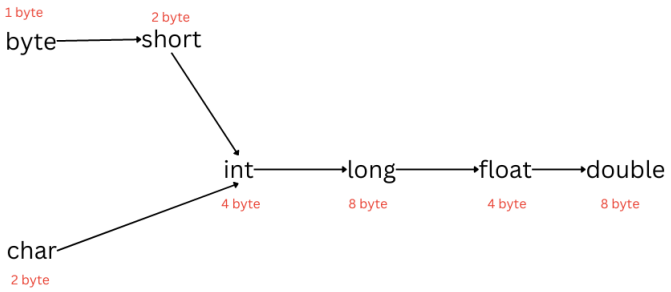
```
while(1)
{
   System.out.println("Hello") ;
}
```

Compile-time error: incompatible types
found: int
required: boolean

3.6 Type conversion

Converting one primitive data type into another is known as type conversion. There are two types of type conversions:

- **Implicit type casting or widening:**
  - **Converting a lower datatype to a higher datatype** is known as widening or up-casting.
  - Compiler is responsible to perform implicity type casting.
  - There is no loss of information in this type casting.
  - The following are various possible conversions:



Eg:

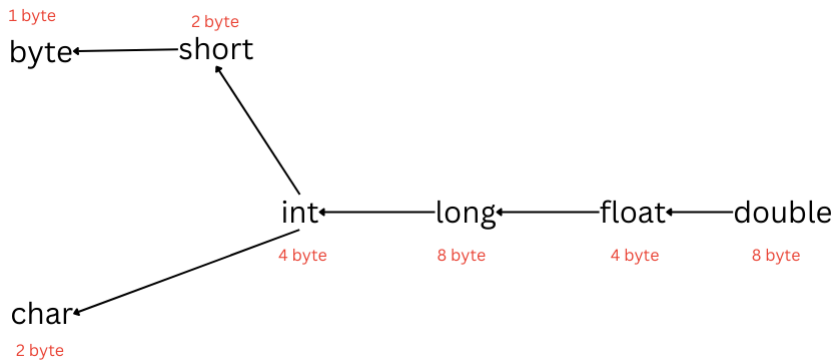
Code:

```
int x = 'a';
System.out.println(x); // output: 97

double d = 10;
System.out.println(d); // output: 10.0
```

Note:

- **Long value can be assigned to float variable** because both are following different memory representations internally.

- **Explicit type casting or narrowing:**
  - Converting a higher datatype to a lower datatype is known as narrowing or down casting.
  - Programmer is responsible to perform explicit type casting.
  - Loss of information is possible.
  - The following are various possible conversions:



  - Except for boolean, all datatypes can be type-cast to other primitive data-type.
  - Type-cast operator for each primitive datatype:

Data type	Type-case operator	Example
byte	(byte)	double d = 130.4; byte x = (byte) d;
short	(short)	double d = 130.4; short x = (short) d;
int	(int)	double d = 130.4; int x = (int) d;
long	(long)	double d = 130.4; long x = (long) d;
float	(float)	double d = 130.4; float x = (float) d;
double	(double)	float d = 130.4; double x = (double) d;
char	(char)	double d = 130.4; char x = (char) d;

Eg 1:

Code:

```
int x = 130;
//byte b = x; ✗
byte b = (byte) x; ✓
System.out.println(b); // output: -126
```

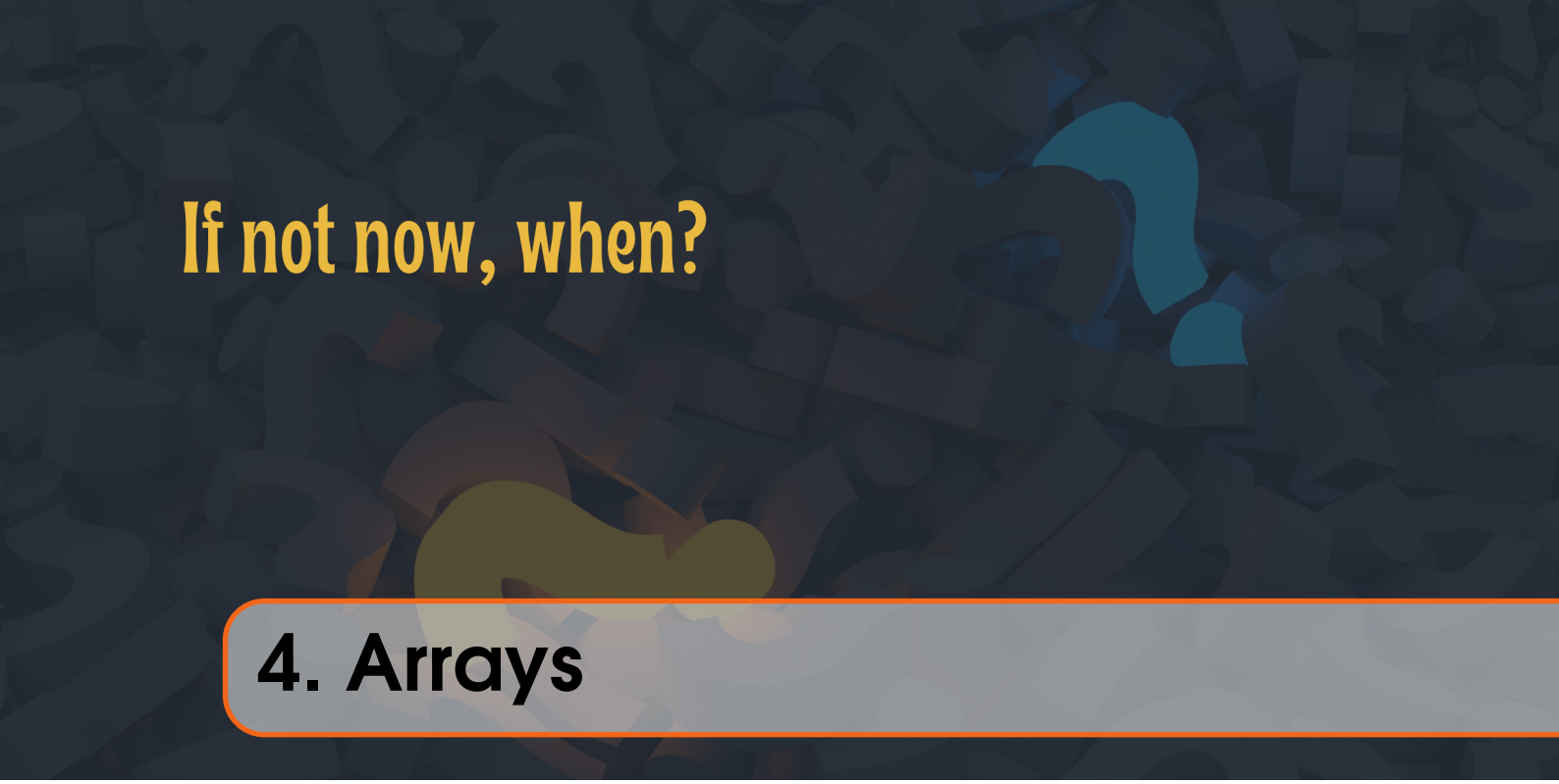
Output explaination:

- Integer is 32-bit in size & byte is 8-bit in size.
- 32-bit binary representation of 130 is 0000000...10000010.
- To convert integer to byte datatype, mean downsize decimal representation of 130 to fit in 8 bit.
- Which means the binary number **0000000...10000010** is down-sized to **10000010**
- Note that left-most bit is "1" and hence number is now represented as 2's complement.
- 2's complement of **10000010** is **11111101+1 => 11111110 => -126**

Eg 2: If we assign floating point values to integral types, by explicit type casting, the digits after decimal point will be lost.

Code:

```
double d = 130.456;
int x = (int) d;
System.out.println(x); // output: 130

byte b = (byte) d;
System.out.println(b); // output: -126
```



4. Arrays

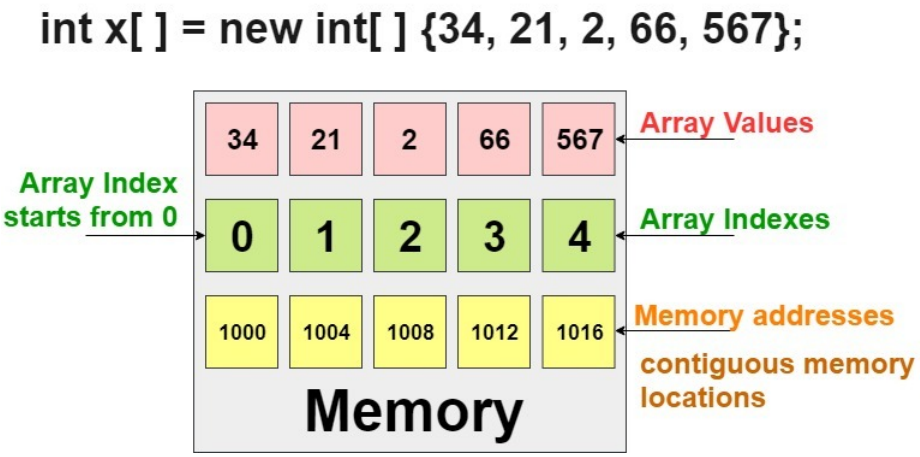
4.1 Arrays in detail

In this section, you are going to learn:

Selection Statements	Iterative Statements	Transfer Statements
• if..else • switch()	• while() • do-while() • for() • for-each loop (Java 1.5)	• break • continue • return • try..catch.finally • assert

4.1.1 Array introduction

An array is **indexed collection of fixed number of homogeneous data elements**.



Advantage:

- Array can represent huge number of values using single variable that will improve readability of code.

Disadvantage:

- Array are fixed in size.
- Once an array is created, they cannot be increased or decreased.
- Array size need to be mentioned in advance, which is not always possible.

4.1.2 Array declaration

- **One dimensional array declaration**:

Syntax:

int[] x; (Recommended as name of variable is clearly separated from type)
int []x;
int x[];

Note:

Array declaration **cannot define size** of array.

Code:

```
int[6] x; ✗
```

- **Two-dimensional array declaration**:

Syntax:

int[][] x; (Recommended)
int [][]x;
int x[][];
int[] []x;

- **3-dimensional array declaration:**

Syntax:

int[][][] x; (Recommended)
int [][][]x;
int x[][][];
int[] [][]x;
int[] x[][];
int[] []x[];

- **More combinations**:
  - Declaring variable **"a"** and **"b"** with **1** dimension:

Code:

```
int[]  a,b;
```

  - Declaring variable **"a"** with 2 dimension and variable **"b"** with 1 dimension

Code:

```
int[]  a[],b;
```

  - Declaring variable **"a"** and **"b"** with 2 dimensions.

Code:

```
int[]  a[],b[];
int[]        a[],b;
```

  - Declaring variable **"a"** with 2 dimension and variable **"b"** with 3 dimension:

Code:

```
int[]        []a,b[];
```

Note:

"[]" is allowed only in front of first variable.

Code:

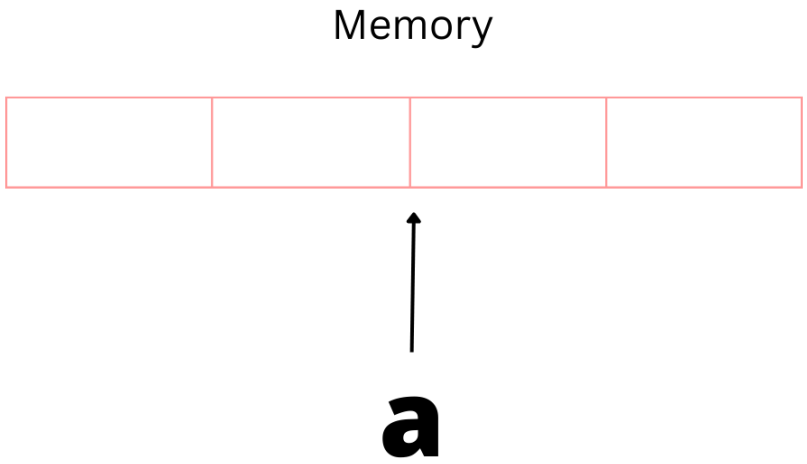
```
int[]      []a,[]b;  ✗
int[]      []a,[]b,[]c;  ✗
```

4.1.3 Array creation

Things to note about array:

- In Java, every array is an Object.
- **"new"** operator is used to create an object.
- Hence, we can create array by using **new** operator.
- **One dimensional array creation**:

```
Syntax:
int[] a = new int[4];
```



Important:

- At the time of array creation, **size should be mentioned compulsorily.**
- An array can be of zero size.

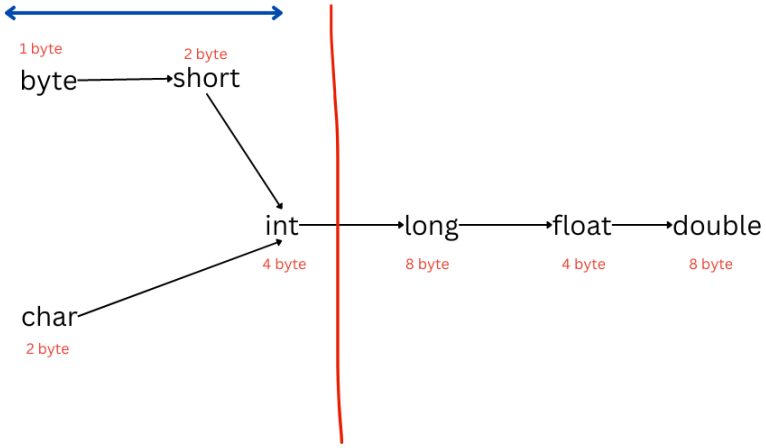
```
int[] x = new int[]; ✗
int[] x = new int[6]; ✓
int[] x = new int[0]; ✓
```

- Java compiler will never throw error for negative size of array. However, Java Virtual Machine will throw runtime error:

NegativeArraySizeException.

```
int[] x = new int[-3]; ✗
```

– Allowed data types for mentioning array size are:
  * integer
  * byte
  * short
  * char




```
int[] x = new int[10]; ✓
int[] x = new int['a']; ✓
byte b = 20;
int[] x = new int[b]; ✓
short s = 30;
int[] x = new int[s]; ✓
```

Below array creation will result in error:

```
int[] x = new int[10l]; ✗
int[] x = new int[3.5]; ✗
```

– Maximum size of array can be 2147483647:

```
int[] x = new int[2147483647]; ✓
int[] x = new int[2147483648]; ✗
```

– For every array type, corresponding classes are available and these classes are part of Java language and not available to the programmer level.

Array type	**Corresponding class name**
int[]	[I
int[][]	[[I
double[]	[D
short[]	[S
byte[]	[B
boolean[]	[Z

Eg: You can find name of class for different array type:

```
int[] a = new int[3];
System.out.println(a.getClass().getName());
```

Output:

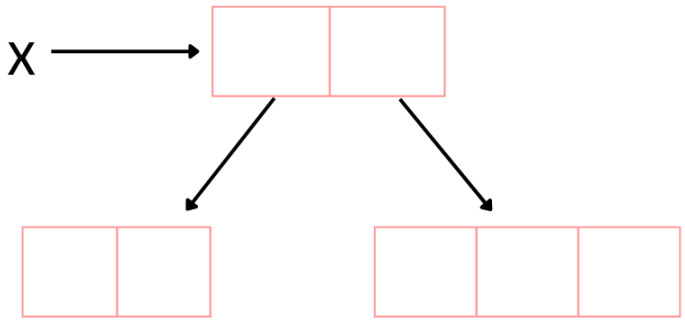
```
[I
```

- **Two-dimensional array creation**:
  - In Java, two dimensional array is not implmeneted using matrix approach.
  - Array of arrays approach is followed for multi-dimensional array creation.
  - Advantage of array of arrays approach is improved memory utilisation.

There are different ways of creating two-dimensional array.

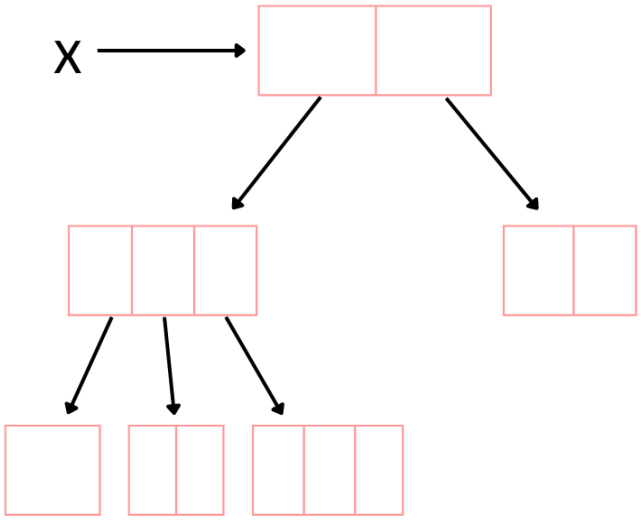
- **Base size**: In this we specifiy the size of first dimension at the time of array creation.

```
int[][] x = new int[2][];
x[0] = new int[2];
x[1] = new int[3];
```



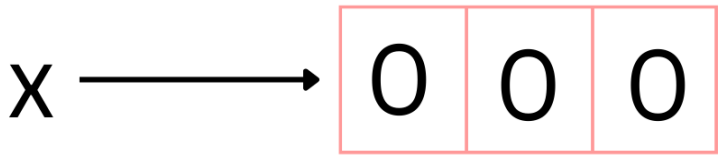
- Three-dimensional array creation:

```
int[][][] x = new int[2][][];
x[0] = new int[3][];
x[0][0] = new int[1];
x[0][1] = new int[2];
x[0][2] = new int[3];
x[1] = new int[2][2];
```



4.1.4 Array initialisation

- **One dimensional array**:
  Once we create an array, every array element is by default initialized with default values.



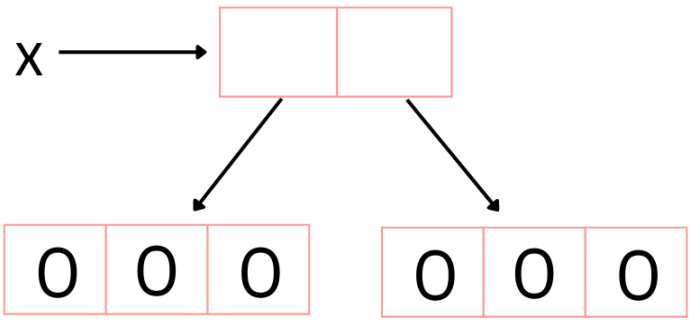

```
int[] a = new int[3];
System.out.println(a);
System.out.println(a[0]);
```

Output:

```
[I@422a8473
0
```

Whenever we are trying to print any reference variable, internally two string method will be called, which is implemented by default to return the string in the following form:
class_name@hexadecimal_form

- **Two-dimensional array**:
  Example 1:

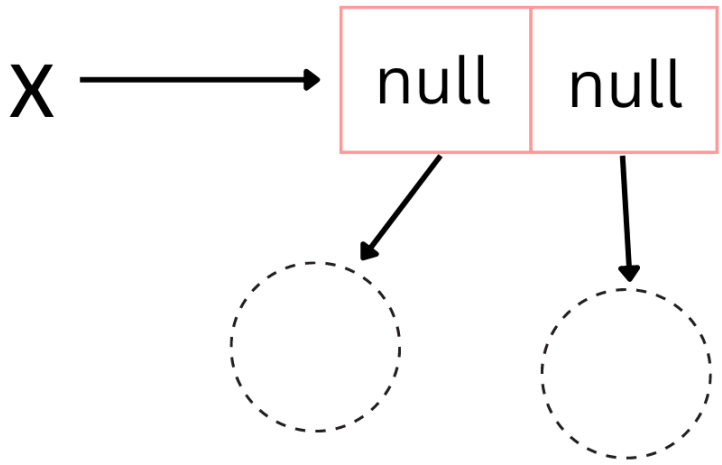


```
int[][] a = new int[2][3];
System.out.println(a);
System.out.println(a[0]);
System.out.println(a[0][0]);
```

Output:

```
[[I@5a39699c
[I@129a8472
0
```

Example 2:

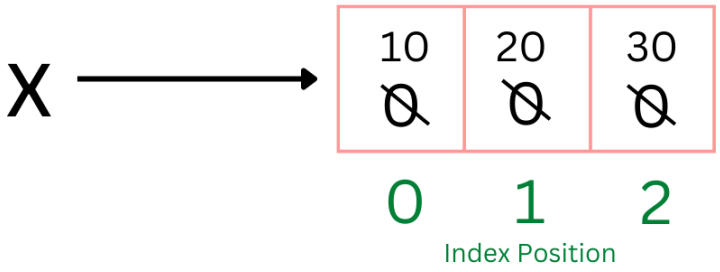



```
int[][] a = new int[2][];
System.out.println(a);
System.out.println(a[0]);
System.out.println(a[0][0]);
```

Output:

```
[[I@5a39699c
null
Exception in thread "main" java.lang.NullPointerException:
```

- **Over-riding array value:**
  Once we create an array, every array element by default initialised with default values.
  We can over-ride default values with custom values.




```
int[] a = new int[3];
a[0]=10;
a[1]=20;
a[2]=30;
System.out.println(a[0]);
System.out.println(a[1]);
System.out.println(a[2]);
```

Output:

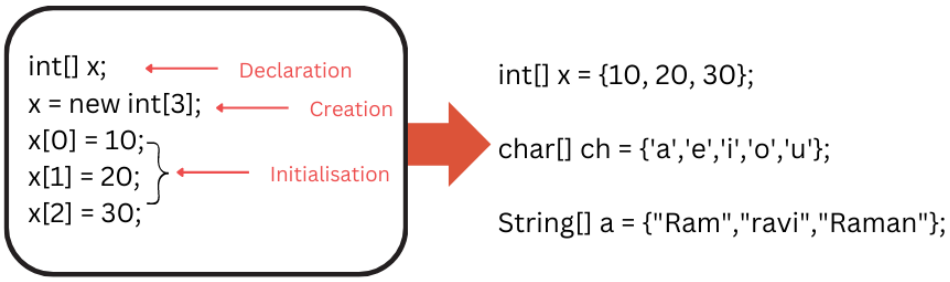
```
10
20
30
```

Note: Trying to access array element with out of range index (either positive or negative integer value) will result in runtime exception: **"ArrayINdexOutOfBoundException"**

4.1.5 Array declaration, creation and initialisation in one line

- **One dimensional array**:

  We can declare, create and initialise an array in a single line (shorcut representation):



Code:

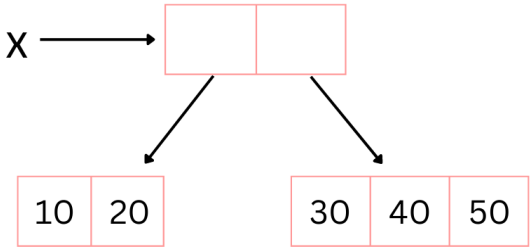
```
int[] x = {10,20,30};
char[] ch = {'a','e','i','o','u'};
String[] a = {"Ram" , "Ravi"};
```

You can declare the array & provide it's value as shown below:

Code:

```
int[] x;
x = {10,20,30};
```

- **Multi-dimensional array**:



Code:

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

4.1.6 length variable

length variable:

- length is final variable applicable for arrays.
- length variable is used to display size of an array.
- Value returned by length is fixed as array once created cannot change it's size.

Code:

```
int[] x = new int[6];
System.out.println(x.length); ✓
```

Output:

```
6
```

- length variable is **not** applicable on string objects.

Code:

```
String s="lavatech";
System.out.println(s.length); ✗
```

- In multi-dimensional arrays, length variable represents only base size, but not total size.

Code:

```
int[][] x = new int[6][3];
System.out.println(x.length);
```

Output:

```
6
```

length():

- length() is present in String class.
- length() method is final variable applicable for string objects.

- It returns number of characters present in the string.

Code:

```
String  s="lavatech";
System.out.println(s.length()); ✓
```

Output:

```
8
```

- length variable is applicable for arrays, but not for string objects.
- length() is applicable for string objects, but not for arrays.

  Example:

Code:

```
String[]  s=  {"A","AA","AAA"};
System.out.println(s.length); ✓
System.out.println(s.length()); ✗
System.out.println(s[0].length); ✗
System.out.println(s[0].length()); ✓
```

Output:

```
3
error
error
1
```

- There is no direct way to find total length of multi-dimensional array.

  Total length of multi-dimensional array can be found as follows:

Code:

```
int[][]  x  =  new  int[3][3];
System.out.println(x.length);
System.out.println(x[0].length+x[1].length+x[2].length);
```

Output:

```
3
9
```

4.1.7 Anonymous Arrays

- Anonymous arrays are nameless arrays.
- These arrays are used for instant one-time purpose.

Syntax:

Single dimension array: **new datatype[]{}**
Multi-dimension array: **new datatype[][]{{},{}}**

- While creating anonymous arrays, you cannot mention it's size:

Code:

```
new int[3]{10,20,30} ✗
new int[]{10,20,30} ✓
new int[][]{{10,20,30},{40,50,60}} ✓
```

- In below example, main() is calling sum() using an anonymous arrays:

Test.java

```
{
    public static void main (String[] args)
    {
        sum(new int[]{10,20,30,40,50});
    }
    public static void sum(int[] x)
    {
        int total = 0;
        for(int x1 :x)
        {
            total = total+x1;
        }
        System.out.println("The sum is : "+total);
```

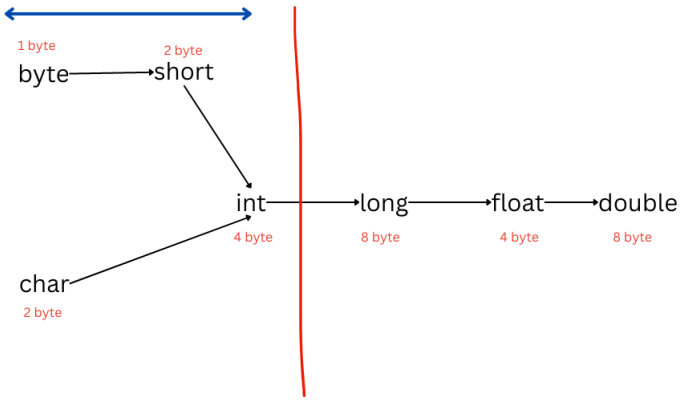
```
    }
}
```

Output:

```
The sum is : 150
```

4.1.8 Array element assignments

In case of **primitive type arrays**, as array elements, you can provide any type which can be **implicitly promoted to declared type**. Example:



Code:

```
int[] x = new int[5];
x[0] = 10;
x[1] = 'a';
byte b = 20;
x[2] = b;
short s = 30;
x[3] = s;
x[4] = 10L; ✗
```

Similary, in case of float type arrays, the allowed datatypes are:

- byte
- short
- char
- int
- long
- float

4.1.9 Array variable assignments

- Data type level promotion are not applicable at array level.
- Eg: Char datatype can be promoted to int type. Whereas, char array cannot be promoted to int array.

Code:

```
int[] x = {10,20,30,40};
char[] ch = {'a','b','c','d'};
int[] b = x; ✓
int[] c = ch; ✗
```

Which of the following promotions will be performed automatically:

Conversion	Answer
char → int	✓
char[] → int[]	✗
int → double	✓
int[] → double[]	✗
float → int	✗
float[] → int[]	✗

Don't fear failure.

Not failure, but low aim is the crime.

5. Operators

5.1 Operators in Java

In this section, you are going to learn:

1. **Arithematic Operator**
2. **String Concatenation Operator**
3. **Increment/Decrement Operator**
4. **Relational Operator**
5. **Equality Operator**
6. **Bitwise Operator**
7. **Boolean complement Operator**
8. **Short circuit Operator**
9. **Type cast Operator**
10. **Assignment Operator**
11. **Conditional Operator**
12. **new Operator**
13. **[] Operator**

14.

5.1.1 Arithematic Operator

Operator	Example

Addition(+)	`int a = 5;`
	`int b = 10;`
	`int c = a + b; // c will be 15`

Subtraction(-)	`int a = 10;`
	`int b = 5;`
	`int c = a - b; // c will be 5`

Multiplication(*)	`int a = 2;`
	`int b = 3;`
	`int c = a * b; // c will be 6`

Modulus (%)	`int a = 10; int b = 3; int c = a % b; // c will be 1`

Division(/)	`int a = 10;`
	`int b = 3;`
	`int c = a / b; // c will be 3 (the remainder is discarded)`
	
	`double d = 10.0;`
	`double e = 3.0;`
	`double f = d / e; // f will be 3.33333`

Important Points:

- **Implicit type casting:** If we apply any arithmetic operator between 2 variables "a" and "b", the result type is always:

  ```
  maximum(int, type of a, type of b)
  ```

  Using above formula,

  - Byte + byte = int
  - Byte + short = int
  - Short + short = int
  - Byte + long = long
  - Long + double = double
  - Float + long = float
  - Char + char = int
  - Char + double = double

Eg:

Code:

```
System.out.println('a'+'b'); // output: 195
System.out.println('a'+3.29); // output: 100.29
```

- **Infinity**:
  - In integral arithmetic **(byte short int long), infinity cannot be represented** and JVM will return runtime error.

    Code:

    ```
    System.out.println(10/0); ✗
    ```

  - But in **floating point arithmetic (float, double), infinity can be represented**. For this, Float and Double classes contains below 2 constants:
    - POSITIVE_INFINITY;
    - NEGATIVE_INFINITY;

    Code:

    ```
    System.out.println(10/0.0); // output: Infinity
    System.out.println(-10/0.0); // output: -Infinity
    ```

- **NaN (not a number):**
  - In **integer arithmetic (byte, short, int, long) , undefined results cannot be represented** and JVM will return runtime error.

    Code:

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

  - But, in floating point arithmetic **(float,double), undefined results can be represented as NaN constant**.

Code:

```
System.out.println(0.0/0); // output: NaN
System.out.println(-0/0.0); // output: NaN
```

5.1.2 String Concatenation Operator

- The only overloaded operator in Java is **"+"** operator.
- It can act as arithmetic addition operator as well as string concatecation operator.

Code:

```
System.out.println(10+20); // output: 30
System.out.println("ab"+"cd"); // output: abcd
```

Note:

Apart from "+", Java does not support **operator overloading**!

- Working of **"+"** with string:
  - If atleast one argument is string type, + operator acts as concatenation operator.
  - If both arguments are number type, + operator acts as arithmetic addition operator.

Eg:

Code:

```
String a = "lavatech";
int b=10, c=20, d=30;
System.out.println(a+b+c+d); // output: lavatech102030
System.out.println(b+c+d+a); // output: 60lavatech
System.out.println(b+c+a+d); // output: 30lavatech30
System.out.println(b+a+c+d); // output: 10lavatech2030
```

5.1.3 Increment/Decrement Operator

- The increment(++) and decrement(–) operators are unary operators.
- They are used to increment or decrement the value of a variable by 1.
- **Increment Operator (++):** Used in two ways:
  - **Prefix (++var)**: Variable is incremented first and then used in the expression.

Code:

```
int a = 5;
int b = ++a; // b will be 6, a will be 6
```

  - **Postfix (var++)**: Variable is used in the expression and then incremented.

Code:

```
int a = 5;
int b = a++; // b will be 5, a will be 6
```

- **Decrement Operator (–):** Works in a similar way and can be used in prefix and postfix forms:

Code:

```
int a = 5;
int b = –a; // b will be 4, a will be 4
int c = a–; // c will be 4, a will be 3
```

Summary:

Expression	Initial value of x	Value of y	Final value of x
y=++x;	10	11	11
y=++x;	10	10	11
y=++x;	10	9	9
y=++x;	10	10	9

Important Points:

- Increment/decrement is **applicable only on variable and not on constant**.

Code:

```
System.out.println(++10); ✗
```

- **Listing** of increment/decrement operators **not allowed**.

Code:

```
int x=10;
int y = ++(++x); ✗
```

- **For final variables**, increment/decrement operators **cannot** be used:

Code:

```
final int x=10;
System.out.println(x++); ✗
```

- Increment/decrement is applicable on all primitive type, **except boolean datatype**:
  - Integer example:

Code:

```
int x=10;
x++;
System.out.println(x); // output: 11
```

  - Character example:

Code:

```
char ch = 'a';
ch++;
System.out.println(ch); // output: 'b'
```

- Boolean example:

Code:

```
boolean b=true;
b++; ✗
```

Difference between “x++” and “x=x+1”

- We know that: If we apply any arithmetic operator between 2 variables “a” and “b”, the result type is always:

```
maximum(int, type of a, type of b)
```

- This is the reason why using "x=x+1" can result in compile-time error:

Code:

```
byte b=10;
b = b+1; ✗
```

- But in case of increment/decrement operators, internal type casting will be performed automatically:

Code:

```
byte b=10;
b++; ✓
```

5.1.4 Relational Operator

- Below are available relational operator in Java:

```
<  ----> less than
<= ----> less than equal to
>  ----> greater than
>= ----> greater than equal to
== ----> equal to
!= ----> not equal to
```

- We can apply relational operator for every primitive datatype, except boolean datatype.
- Eg:

Code:

```
System.out.println(10>20); // output: false
System.out.println('a'>20); // output: false
System.out.println('b'>2.0); // output: false
//System.out.println(true > false); ✗
```

- We can't apply relational operators for object types

Code:

```
System.out.println('lava' > 'lavatech'); ✗
```

- Chaining of relational operators is not allowed.

Code:

```
System.out.println(10>20>30); ✗
```

5.1.5 Equality Operator

- Equality operators can be used for every primitive type including boolean.

Code:

```
System.out.println(10==20); // output: fasle
System.out.println('a' == 'b'); // output: false
System.out.println('a' == 97.0); // output: true
System.out.println(false == false); // output: true
```

- Equality operators can be applied for object types also. For object references, "r1" & "r2", "r1==r2" returns true, if both reference pointing to the same object (reference comparison or address comparison)

Code:

```
Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
System.out.println(t1 == t2); // output: false
System.out.println(t1 == t3); // output: true
```

There should be some relation between argument types(either child to parent or parent to child or same type). Otherwise, it will result in compile-time error.

Code:

```
Thread t1 = new Thread();
Object o = new Object();
String s = new String("lava");
System.out.println(t1==o) ; // output: false
System.out.println(o==s); // output: false
//System.out.println(s==t1); ✗
```

For any object reference "r": "r==null" ← is always False

Code:

```
String s1 = new String("lava");
System.out.println(s1 == null) ; // output: false

String s2 = new String();
System.out.println(s2==null); // output: false

String s3 = null;
System.out.println(s3==null); // output: true
```

5.1.6 Bitwise Operator

& ---> Bitwise And
| ---> Bitwise Or
^ ---> Bitwise Xor
~ ---> Bitwise Complement
>> ---> Bitwise left shift
<< ---> Bitwise right shift

Bitwise & - If both bits are 1, then only 1 otherwise 0

Sample Code	Output	Explaination

int a=4;
int b=5;
System.out.println(a & b)
	4	100
		101
		——
		100

Bitwise | - If atleast one bit is 1, then only 1 otherwise 0

Sample Code	Output	Explaination

int a=4;
int b=5;
System.out.println(a | b)
	5	100
		101
		——
		101

Bitwise ^ - Also called x-or. If both bits are different, then 1, otherwise 0

Sample Code	Output	Explaination

int a=4;
int b=5;
System.out.println(a ^ b)
	1	100
		101
		—
		001

Bitwise ~ - Bitwise complement operator, 1 becomes 0 and 0 becomes 1

Sample Code	Output	Explaination
int a=4; System.out.println(~ a)	-5	32-bit os represents number with total 32 bits as 000...000100 ~ 000...000100 = 111...111011 Left most bit is sign bit where, 1 is negative and 0 is positive Since left most bit is now 1, number is negative Negative number is represented as 1's complement + 1 ie. 100..000100 + 1 = 100...000101

Bitwise » - Bitwise right shift.

Remove "x" bit from right side and add "x" 0 to left side.

Sample Code	Output	Explaination

int a=10;
int x=2;
System.out.println(a » x)
	2	000...1010 » 2
		000...0010

Bitwise « - Bitwise left shift.

Remove "x" bit from left side and add "x" 0 to right side.

Sample Code	Output	Explaination

int a=10;
int x=2;
System.out.println(a « x)
	40	000...1010 « 2
		000...101000

Note:

- &, | , ^ are applicable for both boolean and integral type.
- » , « are applicable for integral type only.
- ~ applicable for only integral type but not for boolean type.

5.1.7 Boolean complement operator

- The Boolean complement operator (!) is a unary operator that negates the value of a Boolean expression.

Code:

```
boolean a = true;
boolean b = !a; // output: b is false
```

Note:

! operator can be applied only on boolean data-type.

5.1.8 Short circuit Operator

In Java, the short-circuit operators are:

- **&& (logical AND)**: Same as bitwise &
- || **(logical OR)**: Same as bitwise |

& , |	&& , ||
Both arguments should be evaluated always	Second argument evaluation is optional
Relatively performance is low	Relatively performance is high
Applicable for both boolean and integral types	Applicable only for boolean but not for integral types
Eg: For x & y, both x and y will be evaluated. Similarly, For x | y, both x and y will be evaluated.	Eg: For x && y, y will be evaluated if and only if x is true i.e if x is false then y wont be evaluated. Similarly, For x || y, y will be evaluated if and only if x is false i.e if x is true then y wont be evaluated.

Consider below code showing different behaviour of &, &&, |, ||:

Code:

```
int x = 10, y = 15;
if( ++x < 10 & ++y > 15) {
    x++;
}
else {
    y++;
}
System.out.println(x + "..." + y); // output: 11...17
```

Code:

```
int x = 10, y = 15;
if( ++x < 10 | ++y > 15) {
    x++;
}
else {
    y++;
}
System.out.println(x + "..." + y); // output: 12...16
```

Code:

```
int x = 10, y = 15;
if( ++x < 10 && ++y > 15) {
    x++;
}
else {
    y++;
}
System.out.println(x + "..." + y); // output: 11...16
```

Code:

```
int x = 10, y = 15;
if( ++x < 10 || ++y > 15) {
    x++;
}
else {
    y++;
}
System.out.println(x + "..." + y); // output: 12...16
```

5.1.9 Assignment Operator

There are 3 types of assignment operators:

- **Simple:** The assignment operator (=) is used to assign a value to a variable.

Code:

```
int x = 10;
```

- **Chained:**

Code:

```
int a,b,c,d;
a=b=c=d=20;
```

We can't perform chained assignment directly at the time of declaration:

Code:

```
int a=b=c=d=20; ✗

int b,c,d;
int a=b=c=d=20; ✓
```

- **Compound:**
  - Assignment operator can be mixed with other operators.
  - Such type of assignment operators are called compound assignment operators.

Syntax:

```
variable operator= expression;
```

Code:

```
int a = 10;
a += 20;
System.out.println(a) ; // output: 30
```

– In case of compound assignment operator, internal type casting will be performed automatically:

Code:

```
byte b=10;
b = b+1; ✗

byte b = 10;
b++; // output: 11

byte b=10;
b+=1; // output: 11
```

– Below are sample examples of compound assignment operator:

Code:

```
int x = 5; x += 3; // x is 8
x -= 2; // x is 6
x *= 4; // x is 24
x /= 3; // x is 8
x %= 5; // x is 3
x &= 1; // x is 1
x |= 2; // x is 3
x ^= 3; // x is 0
x «= 2; // x is 0
x »= 1; // x is 0
```

5.1.10 Conditional Operator

- The conditional operator (also known as the ternary operator)
- It is **if-else statement in a single line**.

Syntax:

```
condition ? expression1 : expression2
```

- If condition is true, then the expression1 is evaluated else expression2 is evaluated and its value is returned.

Code:

```
int a = 23, b = 30;
System.out.println( (a>b)? "a is greater" : "b is greater");
```

5.1.11 new Operator

- **new** operator is used to create an object.

Syntax:

```
Class obj = new class();
```

Eg:

Code:

```
String name = new String("Ram");
```

5.1.12 [] Operator

- [] operator is used to declare and create arrays.

Code:

```
int[] x = new int[10];
```

5.1.13 Operator Precedence

Unary operators	[] , x++ , x– ++x , –x , ~ , !
Arithematic operators	* , / , % + , -
Shift operators	» , »> , «
Comparison operators	< , <= , > , >= , instanceof
Equality operators	== , !=
Bitwise operators	& , ^ , |
Short circuit operators	&& , ||
Conditional operator	?:
Assignment operators	= , += , -= , *=

Evaluation order of operands:

- In java, we have only operator precedence and no operands precedence.
- Before applying any operator, all operands will be evaluated from left to right.
- Eg:

Code:

```
System.out.println(1+2*3/4+5*6);  //  output:  32
```

Output explaination:

  - 1+2*3/4+5*6
  - 1+6/4+5*6
  - 1+1+5*6
  - 1+1+30
  - 32

Don't fear failure.
Not failure, but low aim is the crime.

6. Flow Control

6.1 Flow control statement

In this section, you are going to learn:

Selection Statements	Iterative Statements	Transfer Statements
• if..else • switch()	• while() • do-while() • for() • for-each loop (Java 1.5)	• break • continue • return • try..catch.finally • assert

6.2 Selection Statements

Below are 2 selection statements:

- **if..else**
- **switch case**

Let's see each of these in detail.

6.2.1 if...else

- The **if** statement allows you to execute a block of code if a certain condition is true.

Syntax:

```
if (condition)
    Action if is true
```

Syntax:

```
if (condition) {
    Action if is true
}
else {
    Action if is false
}
```

- The argument to if statement should be boolean type only, else there will be compile-time error.
- **Else part and curly braces are optional.**
- Without curly braces only one statement is allowed under if statement, which should **not be declarative statement**.

- Eg 1:

Code:

```
if (true)
    System.out.println("Hello");
```

- Eg 2:

Code:

```
if (true); ✓// Note: Semicolon is also valid statement.
```

- Eg 3:

Code:

```
if (true)
    int x = 10; ✗// Compile-time error for declarative
statement
```

- Eg 4:

Code:

```
if (true) {
    int x = 10;
}
```

- Eg 5:

Code:

```
int x = 0;
if (x) { ✗// Compile-time error as not a boolean
    System.out.println("Hello");
}
else {
    System.out.println("Hi");
}
```

- Eg 6:

Code:

```
int x = 0;
if (x=20) { ✗// Compile-time error as not a boolean
    System.out.println("Hello");
}
else {
    System.out.println("Hi");
}
```

- Eg 7:

Code:

```
int x = 0;
if (x==20) {
    System.out.println("Hello");
}
else {
    System.out.println("Hi"); // output: Hi
}
```

- Eg 8:

Code:

```
boolean b = false;
if (b == false) {
    System.out.println("Hello"); // output: Hello
}
else {
    System.out.println("Hi");
}
```

Dangling else

- There is no dangling else problem in Java.
- Every else is mapped to the nearest if statement.

Code:

```
if (true)
if (true)
    System.out.println("Hello"); // output: Hello
else
    System.out.println("Hi");
```

6.2.2 switch case

- If several options ar available, then it is not recommended to use nested if..else statement, as it reduces readability.
- Solution: switch statement

Syntax:

```
switch(argument) {
    case arg-1:
        action1;
        break;
    case arg-2:
        action2;
        break;
    case n:
        action-n;
        break
    default:
        Default action
}
```

- Curly braces are mandatory.
- **Case and default are optional**, i.e an empty switch statement is a valid Java syntax.

Code:

```
int x = 10;
switch(x) {} ✓
```

- Allowed argument types in switch statement:
  - Upto Java 1.4 version -> **byte, short, char, int**
  - From Java 1.5 version -> byte, short, char, int, **wrapper classes (Byte, Short, Character, Integer), enum**

- From Java 1.7 versionbyte, short, char, int, wrapper classes (Byte, Short, Character, Integer), enum, **string**

- Inside switch, every statement should be under some case or default.

Code:

```
int x = 10;
switch(x){
    System.out.println(); ✗      // Compile-time error!
}
```

- Case argument should be compile-time constant (i.e constant expression).

Code:

```
int x=10;
int y=20;
switch(x) {
    case 10:
        System.out.println(10);
        break;
    case y; ✗      // Compile-time error!
        System.out.println();
        break;
}
```

Note:

If y is declared as "final", then there will be no compile-time error.

- Switch arguments and case label can be expressions. But, case label should be constant expression.

Code:

```
int x = 10;
switch(x+1) {
    case 10:
        System.out.println(10);
        break;
    case 10+20+30:
        System.out.println(60);
        break;
}
```

- **Case label should be in range of switch arg type**, else it will result in compile-time error.

Eg 1:

Code:

```
byte b = 10;
switch(b) {
    case 10:
        System.out.println(10);
        break;
    case 100:
        System.out.println(100);
        break;
    case 1000: ✗      // Compille-time error!
        System.out.println(1000);
        break;
}
```

Eg 2:

Code:

```
byte b = 10;
switch(b+1) { // Implicit type-caste to integer
    case 10:
        System.out.println(10);
        break;
    case 100:
        System.out.println(100);
        break;
    case 1000:
        System.out.println(1000);
        break;
}
```

- Duplicate case labels are not allowed.

Code:

```
int x = 10;
switch(x) {
    case 12:
        System.out.println(12);
        break;
    case 97:
        System.out.println(97);
        break;
    case 'a': ✗     // Duplicate labels error
        System.out.println(1000);
        break;
}
```

Summary for case-label argument

- It should be constant expression.
- The value should be in the range of switch argument type.
- Duplicate case label are not allowed

Fall-through inside switch

- Within the switch, if any case is matched, from that case onwards all statements will be executed until break or end of the switch.
- This is called fall-through inside switch.
- The main advantage of fall inside the switch is, we can define common action for multiple cases (code-reuseablilty).

Syntax:

```
switch(argument)  {
      case  arg-1:
      case  arg-2:
      case  arg-3:
            action;
            break;
      case  arg-4:
      case  arg-5:
      case  arg-6:
            action;
            break;
}
```

Eg:

Code:

```
int x = 0; switch(x) {
    case 0:
        System.out.println(0);
    case 1:
        System.out.println(1);
        break;
    case 2:
        System.out.println(2);
    default:
        System.out.print("default");
}
```

Output:

```
0
1
```

- **Default case:**
  - Within the switch, **you can take default case at most once**.
  - Default case will be executed if and only if, there is no case matched.
  - Within the switch, you can write default case anywhere but it is recommended to write as last case.

    Eg:

    Code:

    ```
    int x = 3; switch(x) {
        default:
            System.out.println("default")
        case 0:
            System.out.println(0)
        case 1:
    ```

```
            System.out.println(1)
        case 2:
            System.out.println(2)
}
```

Output:

```
default
0
```

6.3 Iteration

Iterative statements allow you to repeat a block of code multiple times. Below are the important iterative statements in Java:

1. while
2. do-while
3. for
4. for-each

6.3.1 while()

- When number of iterations is not known in advance, you should use while loop.

Syntax:

```
while(condition) {
    Action
}
```

- The condition should be of boolean type.

Note:

In Java, "1" is not true or false.

Code:

```
while(1) { ✗
    System.out.println("Hello");
}
```

- Curly braces are optional and without curly you can take only one statement under while, and this statement **should not be declarative**.

- Below are some valid and invalid examples of while:

Code:

```
while(true) ✓
    System.out.println("Hello");
```

Code:

```
while(true); ✓
```

Code:

```
while(true)
    int x = 10; ✗
```

Code:

```
while(true) {
    int int x = 10; ✓
}
```

- Unreachable statement in while loop also results in compile-time error. Below are some examples showing unreachable statement:

Code:

```
while(true) {
    System.out.println("Hello");
}
System.out.println("Hi"); ✗// Compile-time error
```

Code:

```
while(false) {
    System.out.println("Hello");
}
System.out.println("Hi"); ✗// Compile-time error
```

Code:

```
int a=10, b=20;
while(a<b) { ✓
    System.out.println("Hello");
}
System.out.println("Hi");
```

6.3.2 do-while()

- If we want to execute loop body atleast once, then we should go for do-while loop.

Syntax:

```
do {
    action
} while(condition);
```

- The ";" after while is compulsory.
- The condition should be of boolean type.
- Curly braces are optional
- Without curly braces only one statement is allowed which should not be declarative statement.
- Below are some valid and invlaid example of do-while:

Code:

```
do
    System.out.println("Hello"); ✓
while(true);
```

Code:

```
do; ✓
while(true);
```

Code:

```
do
    int x = 10; ✗
while(true);
```

Code:

```
do {
int x = 10;
} while(true); ✓
```

Code:

```
do
while(true); ✗
```

Code:

```
do
    while(true) ✓
        System.out.println("Hello");
while(false);
```

- Unreachable statement in do-while loop also results in compile-time error. Below are some examples showing unreachable statement:

Code:

```
do {
    System.out.println("Hello");
} while(true);
System.out.println("Hi"); ✗// Unreachable statement error
```

Code:

```
do {
    System.out.println("Hello");
} while(false);
System.out.println("Hi"); ✓
```

Output:

```
Hello
Hi
```

- Every final variable will be replaced by value at compile-time only. If any variable is final in while's condition, then the condition will be evaluated at compile-time only.

Code:

```
final int a = 10, b = 20;
do {
    System.out.println("Hello");
} while(a < b);
System.out.println("Hi"); ✗      // Unreachable statement
```

Code:

```
final int a = 10, b = 20;
do {
    System.out.println("Hello");
} while(a > b);
System.out.println("Hi"); ✓
```

6.3.3 for()

- If you know number of iterations in advance then for loop is the best choice.

```
                    1          2    5    3          4  7
for(initialisation_section; conditional_check; increment/decrement)
{
                    loop_body  3  6  9
}
```

- Curly braces are optional, without curly braces only one statement is allowed, which should not be declarative statement.
- Egs:

Code:

```
for(int i = 0; true; i++)
    System.out.println("Hello"); ✓
```

Code:

```
for(int i = 0; i < 10; i++) ; ✓
```

Code:

```
for(int i = 0; i < 10; i++)
    int x = 10; ✗
```

- Let's see each section of for loop in detail:
  - **Initialisation section:**
    - This section will be executed only once in for loop lifecycle.
    - Use to declare and initialise local variables.
    - You can declare any number of variables, but should be of the same type.
    - If you are trying to declare different datatype variables, then you'll get compile time error.

Code:

```
for(int i = 0; i < 10; i++) {} ✓
for(int i = 0, j=0; i < 10; i++) {} ✓
for(int i = 0, String = "a"; i < 10; i++) {} ✗
for(int i = 0, int j = 0; i < 10; i++) {} ✗
```

* You can take any valid Java statement in this section.
* Eg:

Code:

```
int i = 0;
for( System.out.println("Hi"); i < 3; i++ ) {
    System.out.println("Hello");
}
```

Output:

```
Hi
Hello
Hello
Hello
```

– **Conditional section:**

* In this section you can take any valid Java expression, but it should be of the type boolean.
* Eg:

Code:

```
for(int i = 0; true ; i++)
```

* This section is optional, if nothing is added here, the compiler will always place true.

– **Increment/decrement section:**

* In this section, you can can take any valid Java statement.
* Eg:

Code:

```
int i = 0;
for(System.out.println("Hello"); i < 3; System.out.println("Hi")) {
    i++;
}
```

Output:

```
Hello Hi Hi Hi
```

Note:

All 3 parts of for loop are independent of each other and optional.

- Infinite loop examples:

Code:

```
for(;;) {
    System.out.println("Hello");
}
```

Code:

```
for(;;);
```

- Unreachable statement in for loop, results in compile-time error.

  Eg:

Code:

```
for(int i = 0; true ; i++) {
    System.out.println("Hello");
}
System.out.println("Hello"); ✗// Unreachable statement
```

6.3.4 for-each loop

- This is enhanced for loop introduced in Java 1.5 version.
- It is used to retrieve elements of arrays and collections.

Syntax:

```
for (type var : array) {
    statements using var;
}
```

- Eg: Print elements of 1-dimensional array -

Normal for loop	Enhanced for loop

Code:
`int[] x = {10,20,30};`
`for(int i=0;i<x.length;i++) {`
`System.out.println(x[i]);`
`}`
	Code:
	`int[] x = {10,20,30};`
	`for(int x1; x) {`
	`System.out.println(x1);`
	`}`

- Eg: Print elements of 2-dimensional array -

Normal for loop	Enhanced for loop

Code:
`int[][] x={{10,20},{40,50}};`
`for(int i=0;i<x.length; i++){`
`for(int j=0;j<x[i].length;j++){`
`System.out.println(x1);`
`}`
`}`
	Code:
	`int[][] x={{10,20},{40,50}};`
	`for(int[] x1: x) {`
	`for(int x2: x1) {`
	`System.out.println(x1);`
	`}`
	`}`

- Eg3: Print 3-dimensional array using for-each loop -

Code:

```
int[][][] x = {
    { {1, 2}, {3, 4}, {5, 6}, {7, 8} },
    { {9, 10}, {11, 12}, {13, 14}, {15, 16} },
    { {17, 18}, {19, 20}, {21, 22}, {23, 24} }
};

for(int[]][] x1:x) {
    for(int[] x2: x1 ) {
        for(int x3: x2) {
            System.out.println(x3);
        }
    }
}
```

- Drawback of for-each loop:
  - Applicable only for arrays and collections.
  - Using for-each loop, you can print array elements in original order but not in reverse order.

6.4 Transfer Statements

Transfer statements are used to alter the flow of control in a program. They allow you to jump from one part of the code to another based on certain conditions or requirements. Below are transfer statement:

- break
- continue
- return
- try..catch..finally
- assert

6.4.1 break

Use break statement in the following places:

- Inside switch to stop fall-through:

Code:

```
int x = 0;
switch(x) {
    case 0:
        System.out.print(0);
    case 1:
        System.out.print(1);
    case 2:
        System.out.print(2);
}
```

- Inside loop to break loop execution based on some condition.

Code:

```
for(int i=0; i < 10; i++) {
    if(i==5)
        break;          System.out.print(i);
}
```

- Inside labeled blocks to break block execution based on some condition:

Test.java

```
class Test {
    public static void main(String[] args) {
    int x = 10;
    l1: {
        System.out.print("begin");
        if(x==10)
            break l1;
        System.out.print("end");
    }
        System.out.print("Hello");
    }
}
```

6.4.2 continue

- Use continue statement inside loops to skip current iteration and continue for the next iteration.

Code:

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

Output:

```
1
3
5
7
9
```

6.4.3 Labeled break & continue

- Use labeled break and continue to break/continue a particular loop in nested loops.

Syntax:

```
label-1:
for(...)  {
      label-2:
      for(...)  {
      ..
      ..
            for(...)  {
                  break  label-1;
                  break  label-2;
                  break;
            }
      }
}
```

Eg 1:

Code:

```
l1:
for(int  i=0;  i<j;  i++)  {
     for(int  j=0;  j<3;  j++)  {
           if(i==j)
                 continue  l1;
           System.out.println(i+"..."+j);
     }
}
```

Output:

```
1...0
2...0
2...1
```

Eg 2:

Code:

```
l1:
for(int  i=0;  i<j;  i++)  {
      for(int  j=0;  j<3;  j++)  {
            if(i==j)
                  break  l1;
            System.out.println(i+"..."+j);
      }
}
```

Output:

```
—No  output—
```

6.4.4 return

6.4.5 try..catch..finally

6.4.6 assert

Don't fear failure.
Not failure, but low aim is the crime.

7. Functions

7.1 var_arg methods

In this section, you are going to learn:

Selection Statements	Iterative Statements	Transfer Statements
• if..else • switch()	• while() • do-while() • for() • for-each loop (Java 1.5)	• break • continue • return • try..catch.finally • assert

7.2 Selection Statements

Below are 2 selection statements:

- **if..else**
- **switch case**

Let's see each of these in detail.

Don't fear failure.

Not failure, but low aim is the crime.

8. Packages in Java

8.1 Packages in detail

8.1.1 import keyword

- import keyword is used to access classes, interfaces, and other types from external packages or libraries.
- Consider an example where you want to use **ArrayList** class from **java.util** package. For this, you will need to use fully qualified name:

Code:

```
class New {
    public static void main(String[] args) {
    java.util.ArrayList l1 = new java.util.ArrayList();
    }
}
```

With **import** keyword, the code would look like:

Code:

```
import java.util.ArrayList;
class New {
    public static void main(String[] args) {
        ArrayList l1 = new ArrayList();
    }
}
```

- Using import statement, it is not required to use fully qualified name everytime.

Note:

All classes and interfaces present in the following packages are by default available to every Java program. Hence we are not required to write import statement:

- java.lang
- default package (current working directory)

- There are 2 types of **import**. Let's see each of these in detail.

Explicit class import:

- Used to import a single class.
- It is highly recommended as improves readability of the code.

Syntax:

```
import packageName.ClassName;
```

- Eg:

Code:

```
import java.util.ArrayList; ✓
import java.util; ✗
```

Implicit class import:

- Used tp import all classes from a package.
- Not recommended to use as reduces readability of the code.

Syntax:

```
import packageName.*;
```

- Eg:

Code:

```
import java.util.ArrayList.*; ✗
import java.util.*; ✓
```

- Implicit declaration results in **ambiguity problem**. Eg:

Code:

```
import java.util.*;
import java.sql.*;
class New {
    public static void main(String[] args) {
        Date d = new Date(); ✗  Ambiguous error
    }
}
```

In above code, **Date** is available in both **java.util** as well as **java.sql** package.

- While resolving class names, compiler will give precedence in the following order:
  - Explicit class import
  - Classes present in current working directory (default package)
  - Implicit class import

Code:

```
import java.util.*;
import java.sql.*;
class New {
    public static void main(String[] args) {
        Date d = new Date();
        System.out.println(d.getClass().getName());
    }
}
```

Output:

```
java.util.Date
```

Some more things to note about import statement:

- **No subpackage import**: By importing a Java package, all classes and interfaces present in that package are available to Java program, but not the subpackage classes.
- **No effect on execution time:** Import statements is totally compile-time related concept. **If more number of imports, then more will be the compile-time.** But, there is no effect on execution time (runtime).
- Difference between C language **include** and Java **import** statement:

#include	import statement
All I/O header files will be loaded at the beginning (at translation time)	No **".class"** file will be loaded at the beginning. Whenever we are using a particular class, then only corresponding **".class"** file will be loaded.
It is static include	It is "dynamic include" or "load on fly" or "load on demand"

8.1.2 Packages

- Java package groups related classes and method can be grouped into a single unit.
- Eg: All classes and interfaces for File I/O operations are group into **java.io** package.
- Advantage of package:
  - **Resolve naming conflict**
  - **Modularity:** Modularises the application
  - **Maintainability:** Improves application maintainability.
  - **Security:** Provides security for our components

Naming for Package

- There is one universally accepted naming convention for packages, i.e to use internet domain name in reverse:
  Eg:

Test.java

```
package com.lavatech.www;
class Test {
    public static void main(String[] args) {
        int a=10;
        System.out.println(a);
    }
}
```

- Below command can be used to directory structure at valid location.

Syntax:

```
$ javac -d <location> <filename.java>
```

Eg:

Command:

```
# To create directory structure in current location
$ javac -d . Test.java

# To create directory structure under "D:" location
$ javac -d D: Test.java
```

Above command will create below directory structure:

- While running, make sure that you provide fully qualified name:

Command:

```
$ java com.lavatech.www.Test
10
```

Important pointers:

- There can be utmost one package statement in any Java source file.
  Eg 1:

Code:

```
package pack1;
package pack2; ✗      # Result in compile-time error
public class A
```

Eg 2:

Code:

```
package pack1; ✓
public class A {}
```

- The first non comment statement should be package statement (if it is available), otherwise we will get compile-time error.

Code:

```
import java.util.*; ✗      # Result in compile-time error
package pack1;
package pack2;
public class A {}
```

- The following is valid order in any Java source file:

```
package statement; ← Atmost one
import statements; ← Any number
class | interface | enum declarations ← Any number
```

- An empty source file is a valid java program. Hence the following are valid java source files:

Test.java

```
# Empty file
```

Test.java

```
pacakge  pack1;
```

Test.java

```
import  java.util.*;
```

Test.java

```
package  pack1;
import  java.util.*;
```

Test.java

```
class  Test  {}
```

Don't fear failure.

Not failure, but low aim is the crime.

9. Polymorphism & Inheritance

9.1 Class

- A class is a blueprint to create an object.
- A class consists of instance variable and methods.

In this section, we shall see class in detail.

9.1.1 Attributes in detail

- Represents the **state or characteristics** of an object.
- Declared **within a class but outside any methods**.
- Each object of the class has its own copy of instance variables.

Syntax:

```
modifier type identifier;
```

where,

- **access modifiers** can be public, private, protected, default, static, final, transient, volatile.
- **type** can be data-type, classname
- **identifier** is name of attribute

Let see each effect of the access modifier on instance variable in detail.

- **public attributes:**
  - Can be accessed from any other class or package.
  - Eg 1: Accessing from other class -

Code:

```
class A {
    public int no;
}

public class Test3 {
    public static void main(String[] args) {
        A a1 = new A();
        a1.no = 1;
        System.out.println(a1.no);
    }
}
```

Output:

```
1
```

– Eg 2: Accessing from other package -

Test1.java

```
package com.lavatech.www;
public class Test1 {
    public int code;
}
```

Test2.java

```
package com.lavatech.info;
import com.lavatech.www.Test1;
class Test2 {
    public static void main(String[] args) {
        Test1 test = new Test1();
        test.code = 12345678;
        System.out.println(test.code);
    }
}
```

Command:

```
$ javac -d . Test1.java
$ javac -d . Test2.java
$ java com.lavatech.info.Test2
12345678
```

- **private attributes**:
  - Can only be accessed within the same class.
  - Not visible to other classes or packages.
  - Eg: Accessing with the same class:

Test.java

```
class A {
    private int no;
    public void display(){
        no=100;
        System.out.println(no);
    } }
public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.display();
    } }
```

- **protected attribute**:
  - Accessible within the same package or subclass.
  - Accessed from subclasses even in different package.
  - Eg 1: Accessing from within subclass:

Test.java

```
class A {
    protected int no;
}
public class Test extends A {
    public static void main(String[] args) {
        Test3 t1 = new Test3();
        t1.no = 120;
        System.out.println(t1.no);
    } }
```

Output:

```
120
```

- Eg 2: Accessing from other package:

Test1.java

```
package com.lavatech.www;
public class Test1 {
    protected int no;
}
```

Test2.java

```
package com.lavatech.info;
import com.lavatech.www.Test1;
class Test2 extends Test1 {
    public static void main(String[] args) {
        Test2 test = new Test2();
        test.no = 150;
        System.out.println(test.no);
    }
}
```

Command:

```
$ javac -d . Test1.java
$ javac -d . Test2.java
$ java com.lavatech.info.Test2
150
```

- **default (package-private):**
  - No access modifier specified is considered as the default attribute.
  - Can be accessed within the same package but not from other packages.

- Eg: Accessing from same class -

Test.java

```
class A {       int no; }
public class Test3 {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.no);
    } }
```

Output:

```
0
```

- **static attribute**:
  - A static instance variable belongs to the class rather than an instance of the class.
  - It is shared among all instances of the class.
  - Eg: Accessing varaible using class name -

Test.java

```
class A {
    static int count;
}
public class Test3 {
    public static void main(String[] args) {
        System.out.println(A.count);
    }
}
```

Output:

```
0
```

- **final attribute**:
  - Can only be assigned a value once, and its value cannot be changed thereafter.
  - Onec value is assigned it becomes a constant and cannot be modified.
  - Key points:
    * **Initialization:** Must be initialized when it is declared or within the **constructor** of the class.
    * **Naming convention**: Final variable names are written in **uppercase** letters with underscores separating words (e.g., FINAL_VARIABLE).
    * **Primitive types:** For final variables of primitive types (e.g., int, double, boolean), the value assigned at initialization cannot be modified.
      Eg: Assigning final variable value using constructor:

Test.java

```
class A {
    final int COUNT;
    final int NO = 40;
    public A(int count) {
        this.COUNT = count;
    }
}
public class Test {
    public static void main(String[] args) {
        A a1 = new A(50);
        System.out.println(a1.COUNT);
        System.out.println(a1.NO);
    }
}
```

Output:

```
50
40
```

* **Reference types:** For final variables that are references to objects, the reference itself cannot be changed, but the state of the object it refers to can be modified.
Eg:

Test.java

```
class A {
    final StringBuilder NAME = new StringBuilder("Raman");
}
public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.NAME.append(" Verma");
        System.out.println(a1.NAME);
    }
}
```

Output:

```
Raman verma
```

* **Final and static:** It is also possible to declare a final variable as static, making it a class-level constant accessible without creating an instance of the class.
Eg:

Test.java

```
class Constant {
    public static final int MAX_VALUE = 500;
}
public class Test3 {
    public static void main(String[] args) {
                                                Sys-
tem.out.println(Constant.MAX_VALUE);
    }
}
```

Output:

```
500
```

* **protected final:** Provides a read-only variable accessible within the same package or subclass.

Test.java

```
class Constant {
    protected final int MAX_VALUE = 500;
}
public class Test3 {
    public static void main(String[] args) {
        Constant c1 = new Constant();
        System.out.println(c1.MAX_VALUE);
    }
}
```

Output:

```
500
```

- **transient**:
  - Transient keyword is applicable only on attributes.
  - Use transient keyword in serialisation context.
  - At the time of serialisation if we dont want to save the value of a particular variable to meet security constraint then we should declare that variable as trainsent.
  - At the time of serialsation JVM ignores original value of trainsent variables and save default value to the file.
  - Hence, trainsent means not to serialise.
  - More on this is in serialisation chapter

- **volatile attributes**:
  - A volatile instance variable is used in multithreaded programs to ensure that changes to the variable are visible to all threads.
  - It guarantees that reads and writes to the variable are atomic and consistent.
  - More on this in multi-threading chapter.

9.1.2 Methods in detail

- Defines the **behavior or actions** that objects.
- Declared within a class and can access class attributes.
- Invoked using object of the class.

Syntax:

```
modifier returnType methodName(type1 arg1, type2 arg2, ...)
```

where,

- **access modifier** can be public, private, protected, default, static, final, abstract, synchronized, native or strictfp.
- **returnType:** Specifies the type of value returned. Can be a **primitive type**, an **object type**, or **void** if the method does not return any value.
- **methodName:** Method name that follow the Java naming conventions.
- **type:** Data type of parameter passed to the method.
- arg: Name given to each parameter.

Let's see each part of method syntax in detail.

Access modifiers applicable on methods

- **public:**
  - Can be accessed from any other class or package.
  - Eg:

Test.java

```
class A {
    public void message() {
        System.out.println("Time is money");
    }
}
public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.message();
    }
}
```

Output:

```
Time is money
```

- **private**:
  - Can only be accessed within the same class.
  - It is not visible to other classes or packages.
  - Eg:

Test.java

```
class A {
    String msg;
    private void set() {
        msg = "Time is money";
    }
```

```
    public void get() {
        set();
        System.out.println(msg);
    }
}
public class Test3 {
    public static void main(String[] args) {
        A a1 = new A();
        a1.get();
    }
}
```

Output:

```
Time is money
```

protected:

- Accessible within the same package or subclass.
- It can be accessed from subclasses even if they are in a different package.
- Eg 1: Accessing from within subclass:

Test.java

```
class A {
    protected void msg() {
        System.out.println("Time is money");
    }
    public static void main(String[] args) {
        A a1 = new A();
        a1.msg();
    }
}
```

Output:

```
Time is money
```

- **default (package-private):**
  - No access modifier specified is considered as the default attribute.
  - Can be accessed within the same package but not from other packages.
  - Eg: Accessing from same class -

Test.java

```
class A {
    void msg() {
        System.out.println("Time is money");
    }
}
public class Test3 {
    public static void main(String[] args) {
        A a1 = new A();
        a1.msg();
    }
}
```

Output:

```
Time is money
```

- **static**:
  - A static method belongs to the class rather than an instance of the class.
  - Can be called directly using the class name without creating an object of the class.
  - Eg: Accessing method using class name -

Test.java

```
class A {
    static void msg() {
        System.out.println("Time is money");
    }
}
public class Test3 {
    public static void main(String[] args) {
        A.msg();
    } }
```

Output:

```
Time is money
```

- **final**:
  - Cannot be overridden by subclasses.
  - It provides the implementation that cannot be changed.
  - Eg:

Test3.java

```
class Parent {
    public final void display() {
    System.out.println("Parent class final method");
    }
}
class Child extends Parent {
    // public void display() { }
}
class FinalMethodExample {
    public static void main(String[] args) {
```

```
        Parent parent = new Parent();
        parent.display();
        Child child = new Child();
        child.display();
    }
}
```

Output:

```
Parent class final method
Parent class final method
```

- **abstract**:
  - An abstract method does not have an implementation and must be overridden by any concrete subclass.
  - More on this in abstract class chapter

- **synchronized:**
  - A synchronized method can be accessed by only one thread at a time, ensuring thread safety.
  - More on this in threading chapter

- **native:** A native method is implemented in a language other than Java, typically using JNI (Java Native Interface). It provides a bridge between Java and other languages like C or C++.

- **strictfp:**
  - Enforces strict floating-point precision for floating-point calculations.
  - It ensures consistent results across different platforms.
  - Eg:

A.java

```
strictfp class A {
    public strictfp double cal(double a, double b) {
        return a * b / Math.sqrt(a + b);
    }
    public static void main(String[] args) {
        A example = new A();
        double result = example.cal(10.5, 5.3);
        System.out.println("Result: " + result);
    }
```

Output:

```
Result: 14.000276895995912
```

Method argument types

- Method arguments specify the types of values that can be passed to a method when it is invoked.
- Method arguments define the parameters that a method expects to receive in order to perform its functionality. i
- Here are some common types of method arguments in Java:
  - **Primitive types**: Includes int, double, long, short, byte, char, float, boolean.

Code:

```
class Person {
    String name;
    int age;
    long phno;
    public void setDetails(String n, int a, long p) {
        name = n;
        age = a;
        phno = p;
    }
    public static void main(String[] args) {
        Person p = new Person();
        p.setDetails("Ravi",23,987123546L);
        System.out.println("Name : " + p.name);
        System.out.println("Age : " + p.age);
    System.out.println("Phone number : " + p.phno);
    }
}
```

Output:

```
Name : Ravi
Age : 23
Phone number : 987123546
```

– **Reference types**: Refer to objects or classes or arrays or interfaces or enums. Eg:

Code:

```
class Person {
    String name="Raman";
    int age=56;
    long phno=785642312L;
}
class Test {
    public static void displayPerson(Person person){
        System.out.println(person.name);
        System.out.println(person.age);
        System.out.println(person.phno);
    }

    public static void main(String[] args) {
        Person p = new Person();
        displayPerson(p);
    }
}
```

Output:

```
Raman
56
785642312
```

– **Varargs**:
  * Allows a method to accept a variable number of arguments of the same type.
  * It is denoted by an ellipsis (...) after the parameter type.
  * Eg:

Test.java

```
class Test {
    public static void nos(int... numbers) {
System.out.println("Arguments: " + numbers.length);
            System.out.print("Numbers: ");
            for (int number : numbers) {
                System.out.print(number + " ");
            }
            System.out.println();
    }

public static void main(String[] args) {
            nos(1, 2, 3);
            nos(10, 20, 30, 40, 50);
            nos();
    }
}
```

Output:

```
Arguments: 3
Numbers: 1 2 3
Arguments: 5
Numbers: 10 20 30 40 50
Arguments: 0
Numbers:
```

Method return types

- The return type of a method specifies the type of value that the method will return when it is executed.
- The return type is declared in the method syntax, immediately before the method name.
- Here are some common return types in Java:
  - **Primitive types**: Includes int, double, long, short, byte, char, float, boolean.

Code:

```
class Person {
    String name;
    int age;
    long phno;
    public void setDetails(String n, int a, long p) {
        name = n;
        age = a;
        phno = p;
    }
    public static void main(String[] args) {
        Person p = new Person();
        p.setDetails("Ravi",23,987123546L);
        System.out.println("Name : " + p.name);
        System.out.println("Age : " + p.age);
    System.out.println("Phone number : " + p.phno);
    }
}
```

Output:

```
Name : Ravi
Age : 23
Phone number : 987123546
```

– **Reference types**: Refer to objects or classes or arrays (int[] or String[]) or interfaces or enums. Eg:

Code:

```
class Person {
    String name;
    int age;
    long phno;
} class Test {
    public static Person setPerson(Person person){
        person.name="Raman";
        person.age=67;
        person.phno=785642312L;
        return person;
    }    public static void main(String[] args) {
        Person p;
        p = setPerson(new Person());
        System.out.println("Name:"+p.name);
        System.out.println("Age:"+p.age);
        System.out.println("Phone no:"+p.phno);
    }
}
```

Output:

```
Name:Raman
Age:67
Phone no:785642312
```

- **void**:
  - Indicates that the method does not return any value.
  - Eg:

Test.java

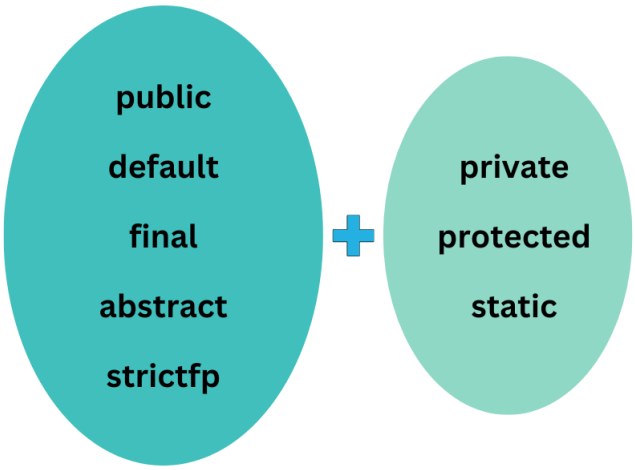
```
class Person {
    String name="Raman";
    int age=56;
    long phno=785642312L;
}
class Test {
    public static void displayPerson(Person person){
        System.out.println(person.name);
        System.out.println(person.age);
        System.out.println(person.phno);
    }

    public static void main(String[] args) {
        Person p = new Person();
        displayPerson(p);
    }
}
```

Output:

```
Raman
56
785642312
```

9.1.3 Class level access modifier

- Using access modifier, class can provide more information to the JVM like:
  - Whether the class is accessible from anywhere or not
  - Whether child class creation is possible or not
  - Whether object creation is possible or not
- The only applicable modifiers for top-level classes are:
  - public
  - default
  - final
  - abstract
  - strictfp
- But, for inner classes, the applicable modifiers are:
  - private
  - protected
  - static



Note:

In Java, there are only access modifiers, there is no word like access specifier.

Let's see these class level access modifiers in detail.

public classes:

- If a class is declared as public then we can access that class from anywhere.
- Eg:

Test1.java

```
package com.lavatech.www;
public class Test1 {
    public void message(){
        System.out.println("Hard work pays off!");
    }
}
```

Test2.java

```
package com.lavatech.info;
import com.lavatech.www.Test1;
public class Test2 {
    public static void main(String[] args) {
        Test1 t1 = new Test1();
        t1.message();
    }
}
```

Command:

```
$ javac -d . Test1.java
$ javac -d . Test2.java
$ java com.lavatech.info.Test2
Hard work pays off!
```

default classes:

- A default class **does not have an access modifier** specified.
- Also known as a **package-private class**
- It is accessible **only within the same package**.
- If no access modifier (such as public, private, or protected) is specified, the class is considered to have default access.
- Eg:

Test1.java

```
package com.lavatech.www;
class Test1 {
    public void message(){
        System.out.println("Hard work pays off!");
    }
}
```

Test2.java

```
package com.lavatech.info;
import com.lavatech.www.Test1;
```

Command:

```
$ javac -d . Test2.java
Test2.java:2: error: Test1 is not public in
com.lavatech.www; cannot be accessed from outside
package
import com.lavatech.www.Test1;
1 error
```

final class

- Final class can't be inherited
- You can't create child class for final class.
- More on final class is mentioned in inheritance section.

abstract class

- Abstract classes can be instantiated.
- You cannot create object of abstract class.
- More on abstract class is mentioned in Abstract class section.

strictfp class

- strictfp was introduced in Java1.2 version.
- Strictfp class ensures all methods in the class & its subclasses, adhere to strict floating-point precision rules.
- strictfp is used to ensure that floating points operations give the same result on any platform.
- As floating points precision may vary from one platform to another. strictfp keyword ensures the consistency across the platforms.
- Eg:

Test.java

```
strictfp class A {
    double num1 = 10e+102;
    double num2 = 6e+08;
    double calculate() {
        return num1 + num2;
    } }
public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println("Result: " + a1.calculate());
    } }
```

Output:

```
Result:  1.0E103
```

9.2 Constructors and garbage collection

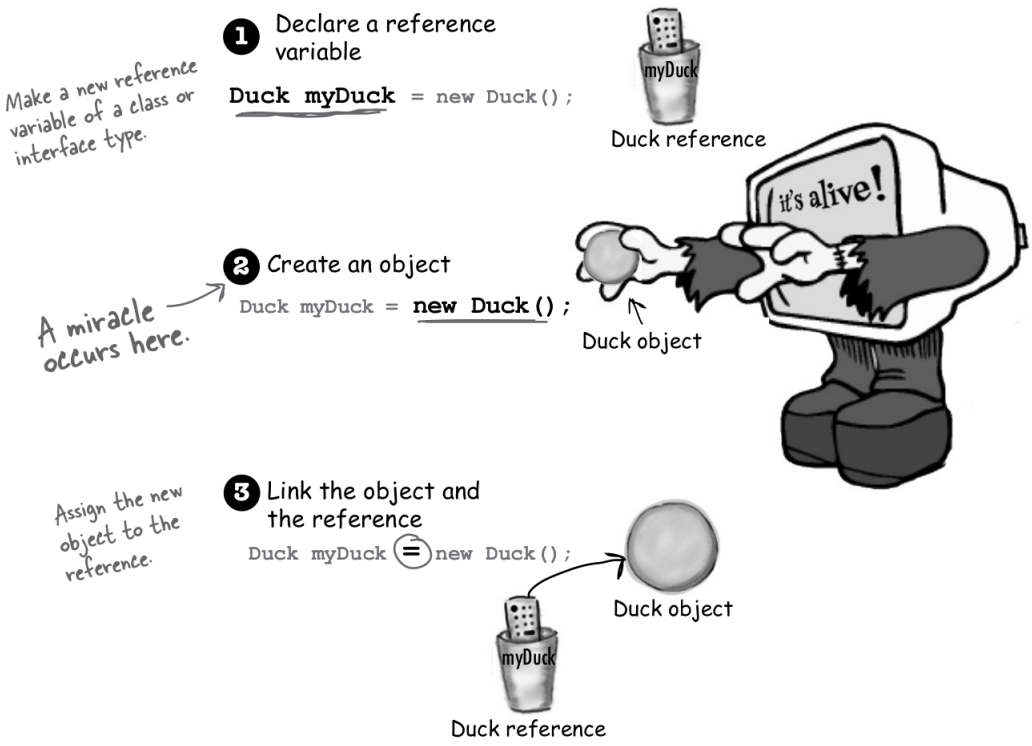
-

9.2.1 new operator

- The **new** operator is used to **create new objects**.
- The new operator:
  - Dynamically allocates memory for an object at runtime
  - Initializes object by calling the constructor

Syntax:

```
ClassName objectName = new ClassName();
```



- Objects created are stored in the **heap memory**, and their memory is managed by the **Java Virtual Machine (JVM) through garbage collection**.

9.2.2 Constructor

- A constructor has the code that runs when you use **new** operator.
- Main purpose of constructor is to perform **initialsation** of an object.
- Rules for creation of constructor:
  - Constructor has the **same name as the class.**
  - Access modifiers for constructors allowed are **public, private, protected and default or default (which means no access modifier at all)**.
  - Constructor **does not have a return type, not even void.**
  - Constructors can accept parameters to initialize the object's attributes.

Syntax:

```
public/private/protected/default Classname(args) {
    // Code here
}
```

- Eg:

Test.java

```
public class Test {
    public Test() {
        System.out.println("Constructor called");
    }

    public static void main(String[] args) {
        Test t1 = new Test();
    }
}
```

Output:

```
Constructor called
```

- Every class has a constructor, even if you don't write it yourself.

But where is the constructor? If we didn't write it, who did?

Ans:

- You can write a constructor for your class (we're about to do that), but if you don't, **the compiler writes one for you!**
- The default constructor is always a no-arg constructor as shown below:

Code:

```
class Test {
    // No-arg constructor
    public Test() {}
}
```

9.2.3 this keyword

- **this** keyword refers to the current object instance within a class.
- It is used to differentiate between class members and local variables.
- It is used to access or modify instance variables or invoke instance methods of the current object.
- **this** keyword can be used within constructors.
- Eg:

Test.java

```
class Rectangle {
    float l, w, h;
    public Rectangle(int l, int w, int h) {
        this.l = l;
        this.w = w;
        this.h = h;
    }
    public float getArea() {
        float area;
        area = 2*(l*w) + 2*(l*h) + 2*(h*w);
        return area;
    }
}
public class Test3 {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(3,4,5);
        float area;
        area = r1.getArea();
        System.out.println(area);
    }
}
```

9.2.4 Constructor Chaining.

- All the constructors in an object's inheritance tree must run when you make a new object.
- This all happens in a process called **Constructor Chaining**.

Test.java

```
class Animal {
    public Animal() {
        System.out.println("This is a Animal");
    }
}
class Duck extends Animal {
    public Duck() {
        System.out.println("This is a Duck");
    }
}

public class Test {
    public static void main(String[] args) {
        Duck d1 = new Duck();
    }
}
```

Output:

```
This is a Animal
This is a Duck
```

9.2.5 super()

- **super()** is used to invoke a superclass constructor from a subclass constructor.
- Key points about super():
  - Must be the **first statement in the subclass** constructor body.
  - The compiler automatically inserts a call to the superclass's default (no-argument) constructor.
  - super() can only be used only within a constructor.
- Eg 1:

Test.java

```
class Animal {
    public Animal() {
        System.out.println("This is a Animal");
    }
}
class Duck extends Animal {
    public Duck() {
        super();
        System.out.println("This is a Duck");
    }
}
```

Eg2: Superclass constructors with arguments

Test.java

```
class Animal {
    String name;
    Animal(String name) {
        this.name = name;
    }
}
```

```
class Duck extends Animal {
    int size;
    public Duck(String name, int size) {
        super(name);
        this.size = size;
    }
}

public class Test {
    public static void main(String[] args) {
        Duck d1 = new Duck("Duckling",4);
        System.out.println("Name : "+d1.name);
        System.out.println("Size : "+d1.size);
    }
}
```

9.2.6 super keyword

- **super keyword** is used to refer to the superclass or parent class of a subclass.
- It can be used to access or invoke the members (methods, variables, constructors) of the superclass from within the subclass.
- Eg:

Car.java

```
class Vehicle {
    int maxSpeed;
    public void displayInfo() {
    System.out.println("Max Speed: " + maxSpeed);
    }
}
public class Car extends Vehicle {
    int numWheels;
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Wheels: "+ numWheels);
    }
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.displayInfo();
    }
}
```

Output:

```
Max Speed: 0
Wheels: 0
```

9.2.7 Constructor overloading

- You can have more than one constructor in your class, as long as the argument lists are different.
- More than one constructor in a class means **overloaded constructors**.
- Eg:

Test.java

```
class Rectangle {
    float l, w, h;
    public Rectangle() {
        this.l=0.0f;
        this.w=0.0f;
        this.h=0.0f;
    }
    public Rectangle(int width, int height) {
        this.l = 0.0f;
        this.w = width;
        this.h = height;
    }
    public Rectangle(int length, int width, int height) {
        this.l = length;
        this.w = width;
        this.h = height;
    }
    public float getArea() {
        float area;
        area = 2*(l*w) + 2*(l*h) + 2*(h*w);
        return area;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        float area;
        area = r1.getArea();
        System.out.println(area);

        Rectangle r2 = new Rectangle(3,4);
        float area2;
        area2 = r2.getArea();
        System.out.println(area2);

        Rectangle r3 = new Rectangle(3,4,5);
        float area3;
        area3 = r3.getArea();
        System.out.println(area3);
    }
}
```

Output:

```
0.0
24.0
94.0
```

9.2.8 this()

- **this()** is used to invoke one constructor from another constructor within the same class.
- It allows constructors with different parameter lists to call each other and **reuse initialization code**.
- The this() invocation **must be the first statement in the constructor body**.
- It is used to **avoid code duplication when multiple constructors** in a class need to perform common initialization tasks.
- Eg:

Test.java

```
class Rectangle {
    float l, w, h;
    public Rectangle() {
        this(0,0,0);
}
    public Rectangle(int width, int height) {
        this(0,width,height);
    }
    public Rectangle(int length, int width, int height) {
        this.l = length;
        this.w = width;
        this.h = height;
    }
    public float getArea() {
        float area;
        area = 2*(l*w) + 2*(l*h) + 2*(h*w);
        return area;
    }
}
```

```
public class TestRect {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(3,4,5);
        float area;
        area = r1.getArea();
        System.out.println(area);

        Rectangle r2 = new Rectangle();
        float area2;
        area2 = r2.getArea();
        System.out.println(area2);
    }
}
```

Output:

```
94.0
0.0
```

Note:

- The first line inside every constructor should be either super() or this() but not both at a time.
- We can use super() or this() only inside constructor.

9.2.9 super(),this() V/S super,this

super(), this()	super, this
These are constructor calls to call super class and current class constructors	These are keywords to refer super class and current class isntance members
We can use these only in constructors as first line	We can use these anywhere except static area
We can use only once in constructors	We can use these any number of times

9.2.10 Access modifier and constructor

private constructor

Syntax	Accessibility	Uses
private Classname { }	Only accessible within the same class	Used to **prevent direct instantiation** of the class

- Eg 1:

A.java

```
class A {
    private A() {
        System.out.println("A constructor called");
    }
    public static void main(String[] args) {
        A a1 = new A(); ✓
    }
}
```

Output:

```
A constructor called
```

- Eg 2:

B.java

```
class A {
    private A() {
        System.out.println("A constructor called");
    } }
public class B extends A {
    public static void main(String[] args) {
        B b1 = new B(); ✗      // Cannot instantiate
    } }
```

protected constructor

Syntax	Accessibility	Uses
protected Classname { }	Accessible within • Same class • Subclasses • Other classes within the same package Cannot be accessed from different package where class is not subclass.	Used to allow sub-classes to access the constructor but restrict direct access from un-related classes.

- Eg 1:

B.java

```
class A {
    protected A() {
        System.out.println("A constructor called");
    }
}
public class B extends A {
    public B() {
        System.out.println("B constructor called");
    }
    public static void main(String[] args) {
        B b1 = new B(); ✓
    } }
```

Output:

```
A constructor called
B constructor called
```

public constructor

Syntax	Accessibility	Uses
public Classname {}	Accessible from anywhere like: • Other classes • Subclasses • Different packages	Used to create objects of the class from any context

- Eg 1:

B.java

```
class A {
    public A() {
        System.out.println("A constructor called");
    }
}
public class B extends A {
    public B() {
        System.out.println("B constructor called");
    }
    public static void main(String[] args) {
        B b1 = new B(); ✓
    } }
```

Output:

```
A constructor called
B constructor called
```

default constructor: Has no explicit access modifier specified.

Syntax	Accessibility	Uses
Classname { }	Accessible within the same package but not accessible from classes outside the package.	If no constructor is defined, a default constructor is automatically provided by the compiler.

- Eg 1:

B.java

```
class A {
    A() {
        System.out.println("A constructor called");
    }
}
public class B extends A {
    public B() {
        System.out.println("B constructor called");
    }
    public static void main(String[] args) {
        B b1 = new B(); ✓
    } }
```

Output:

```
A constructor called
B constructor called
```

9.2.11 Instance block

- An instance initializer block (or instance block) is a block of code defined within a class, executed when an instance of the class is created.
- It initialises instance variables or perform other initialization tasks for each object of the class.
- It is **executed before the constructor of the class**.
- It is used when you have multiple constructors or when you want to perform common initialization logic.
- Eg:

Demo.java

```
public class Demo {
    int value;
    {
        value = 8080;
    }
    public Demo() {}
    public Demo(int value) {
        this.value = value;
    }
    public static void main(String[] args) {
        Demo d1 = new Demo();
        Demo d2 = new Demo(10);
        System.out.println(d1.value);
        System.out.println(d2.value);
    }
}
```

Output:

```
8080
10
```

9.2.12 Static initializer

- Also known as a static initialization block
- Executed only once when the class is loaded into memory.
- It initialize static variables or perform other one-time initialization tasks.
- It is defined within a class and is marked with the static keyword.
- Within a class, you can declare any number of static blocks but all these static blocks will be executed from top to bottom
- It does not have a method name and is enclosed within curly braces .

Syntax:

```
class Classname {
    static {
        // codehere
    }
}
```

- Eg:

Test.java

```
class Test {
    static int count;
    static {
        count = 10;
        System.out.println("Count: " + count);
    }

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

Output:

```
Count: 10
Executed after static block
```

Without main method is it possible to print some statements to console?

Ans:

- Before Java 1.7, yes by using static blocks.
- Eg:

Code:

```
class Test {
    static int count;
    static {
        count = 10;
        System.out.println("Count: " + count);
    }
}
```

- After Java 1.7, this code will result in runtime error.

9.2.13 Constructor V/S Instance block V/S Static block

Constructor	Instance block	Static block
Called when objects are created	Executed for each instance of the class	Executed only once when the class is loaded
Used for object initialization	Used to initialize variables or execute static/instance-specific logic	Used to initialize variables or execute static/instance-specific logic

9.2.14 Destroying object

An object can be destroyed under below conditions:

- Reference goes out of scope, permanently.
- Assign the reference to another object.
- Explicitly set the object reference to **null**.

  Eg:

A.java

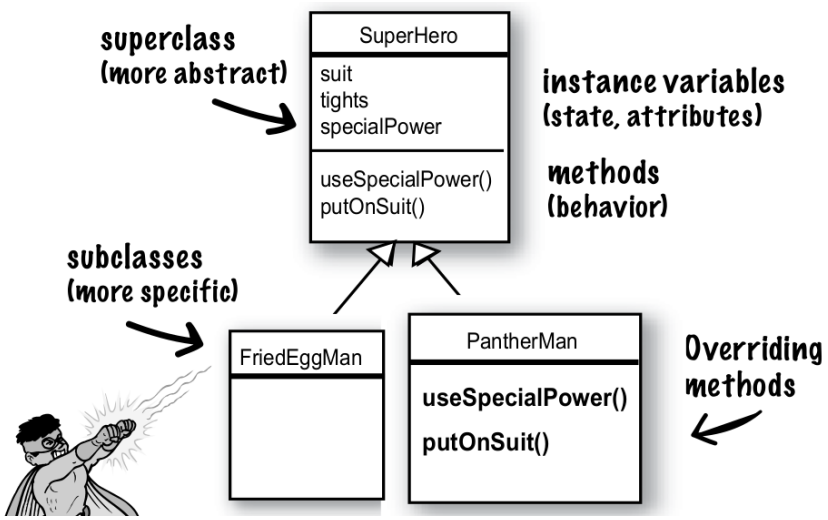
```
class A {
    public A() {
        System.out.println("A constructor called");
    }
    public static void main(String[] args) {
        A a1 = new A();
        a1 = null;
    }
}
```

9.2.15 The Stack and the Heap: where things live

-

9.3 Inheritance

- Inheritance means putting common code in a class(**superclass**) and tell other classes(**subclass**) that the common class is their superclass.
- The subclass **extends** the superclass.
- The subclass inherits the members (i.e instance variables and methods) of the superclass.
- Eg:



- Instance variables are not overridden.
- **IS-A relationship**:
  - Inheritance is also called **IS-A** relationship.
  - Eg: FriedEggMan IS-A SuperHero
- The **extends** keyword is used to implement IS-A relationship.

Syntax:

```
class A {}

class B extends A {}
```

- Superclass reference can be used to hold subclass object.
- Eg:

Code:

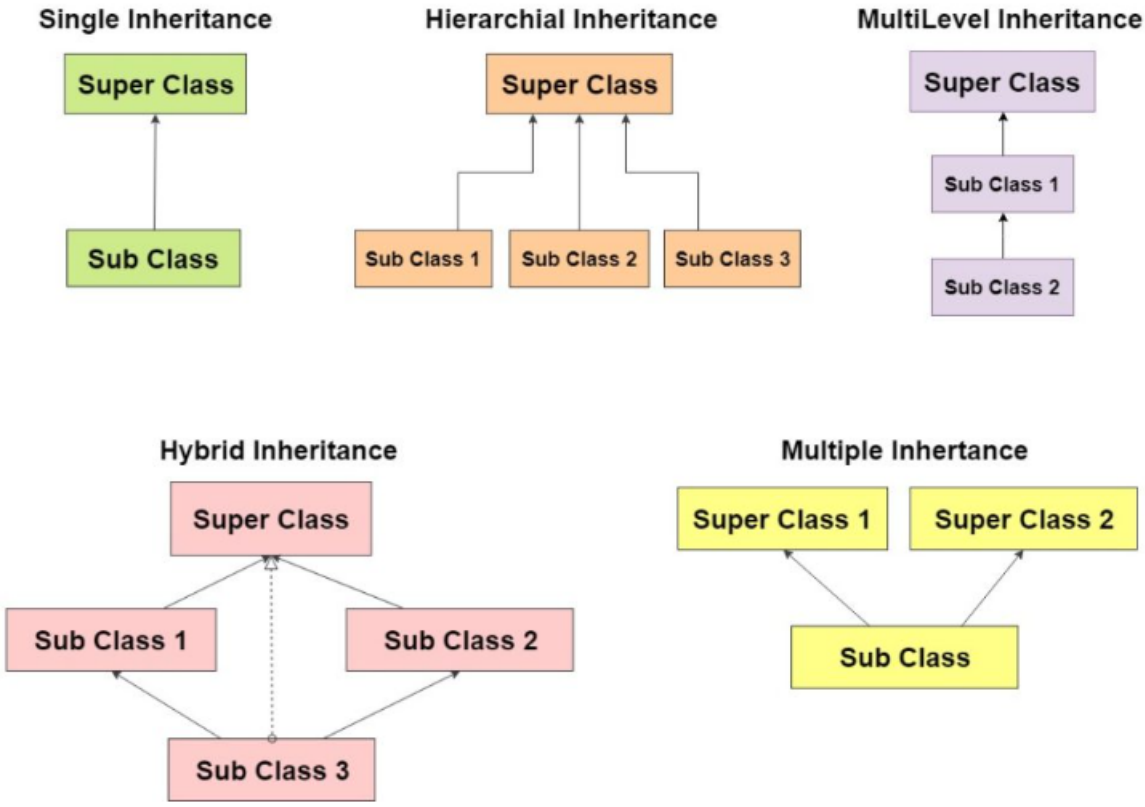
```
class A {
    void displayA() {
        System.out.println("This is A");
    }
}
class B extends A {
    void displayB() {
        System.out.println("This is B");
    }
}
class Test {
    public static void main(String[] args) {
        A a = new A();
        a.displayA();
        a.displayB(); ✗      //Results in error
```

```
        A a2 = new B();
        a2.displayA();
        a2.displayB(); ✗      //Results in error

        B b = new B();
        b.displayA();
        b.displayB();

    B b2 = new A(); ✗      //Results in error
    }
}
```

Note:

- Using parent refernece, you cant call child specific methods.
- Parent reference can be used to hold child object, but using that refernece, we cant call child specific methods.
- Child reference cannot be used to hold parent object.

9.3.1 Types of inheritance



- **Single inheritance:** In single inheritance, a class inherits from a single superclass.

Code:

```
class Superclass {
    // superclass members
}

class Subclass extends Superclass {
    // subclass members
}
```

- **Multilevel inheritance:** Multilevel inheritance refers to a situation where one class inherits from another class, and then a third class inherits from the second class.

Code:

```
class Grandparent {
    // grandparent members
}

class Parent extends Grandparent {
    // parent members
}

class Child extends Parent {
    // child members
}
```

- **Hierarchical inheritance:** Hierarchical inheritance occurs when multiple classes inherit from a single superclass.

Code:

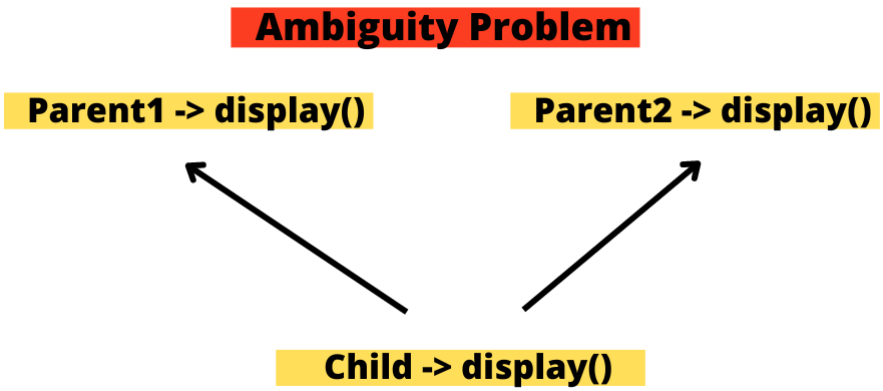
```
class Superclass {
    // superclass members
}

class Subclass1 extends Superclass {
    // subclass1 members
}

class Subclass2 extends Superclass {
    // subclass2 members
}
```

- **Multiple inheritance (through interfaces)**:
  - A java class cant extend more than one class at a time.
  - Multiple inheritance is not directly supported.
  - It can be achieved through interfaces.

Code:

```
interface Interface1 {
    // interface1 methods
}

interface Interface2 {
    // interface2 methods
}

class MyClass implements Interface1, Interface2 {
    // class members
}
```

  - Why Java does not provide support for multiple-inheritance?
    - There maybe a chance of ambiguity problem.



  - Why multiple-inheritance is supported by interface?
    - We shall see this in the interface chapter.

- **Cyclic inheritance:**
  - This is not allowed in java.
  - Eg 1:

Code:

```
class A extends B {} ✗

class B extends A {} ✗
```

  - Eg 2:

Code:

```
class A extends A {} ✗
```

9.3.2 Final class

- Final class can't be inherited
- You can't create child class for final class.
- Eg:

Test.java

```
final class A {}
class Test extends A {} ✗
```

- Every method present inside final class is always final by default(i.e method cannot be overridden).
- But every variable present inside final class need not be final.
- Advantage of final keyword:
  - Security
  - Unique implementation
- Disadvantage of final keyword:
  - Missing inheritance (due to final classes)
  - Polymorphism (due to final methods)
- Hence, if there is no specific requirement, it is not recommended to use final keyword.

9.3.3 Has-A relationship

- Advantage of has-a relationship is **reuseability** of the code.
- There are 2 types of Has-A relationship:
  - **Aggregation:**
    - Aggregation is a weaker form of the "has-a" relationship.
    - It is a one-way relationship and called unidirectional association.
    - For example, Bank can have employees but vice versa is not possible.

Code:

```
class Bank {
    String nameOfBank;
    Bank(String nameOfBank) {
        this.nameOfBank = nameOfBank;
    }
    public void displayAllDetails(Customer customer) {
        System.out.println("Bank = "+ nameOfBank);
        System.out.println("Customer = "+ customer.nameOfCustomer);
    }
}
class Customer {
    String nameOfCustomer;
    Customer(String nameOfCustomer) {
        this.nameOfCustomer = nameOfCustomer;
    }
}
```

```
class Branch {
    public static void main(String arg[])
    {
        Bank bank = new Bank("AXIS");
            Customer customer = new Cus-
tomer("Ram");
        bank.displayAllDetails(customer);
    }
}
```

– **Composition:**
  * Composition is a strong form of the "has-a" relationship.
  * In composition two entities are highly dependent on each other.
  * One entity cannot exist without the other.
  * It represents a **part-of** relationship.
  * Eg: a car has an engine

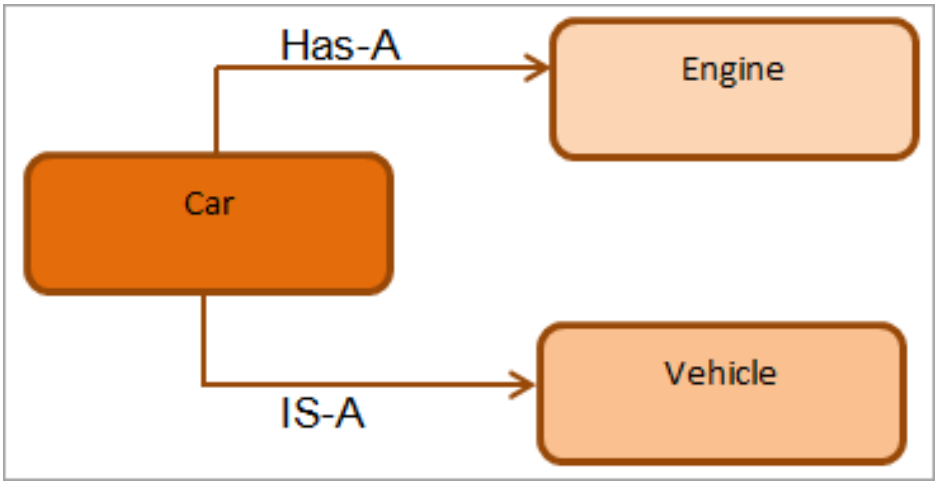
Code:

```
class Car {
    private final Engine engine;
    String nameOfCar;
    String model;
    public Car(String nameOfCar, String model)
{
        engine = new Engine("POWERHIGH",
"ABC");
        this.nameOfCar = nameOfCar;
        this.model = model;
    }
```

```
    public void showAlldetails() {
    System.out.println("Car ="+nameOfCar);
    System.out.println("Model ="+model);
        System.out.println("Engine used ="+en-
gine.typeOfEngine);
        System.out.println("Engine name ="+en-
gine.nameOfEngine);
    }
}

class Engine {
    String typeOfEngine;
    String nameOfEngine;
     Engine(String typeOfEngine, String name-
OfEngine) {
        this.typeOfEngine = typeOfEngine;
        this.nameOfEngine = nameOfEngine;
    }
}

public class Test {
    public static void main(String arg[]) {
        Car car = new Car("BMW", "5X");
        car.showAlldetails();
    }
}
```

9.3.4 Has-A V/S Is-A relationship

- If you want total functionality of a class automatically, then go for **IS-A** relationship.
- If you want part of the functionality, then go for **has-a** relationship.



9.4 Polymorphism

- Polymorphism refers to the ability of objects to take on multiple forms or behaviors.
- It allows objects of different classes to be treated as objects of a common superclass or interface.
- Eg:

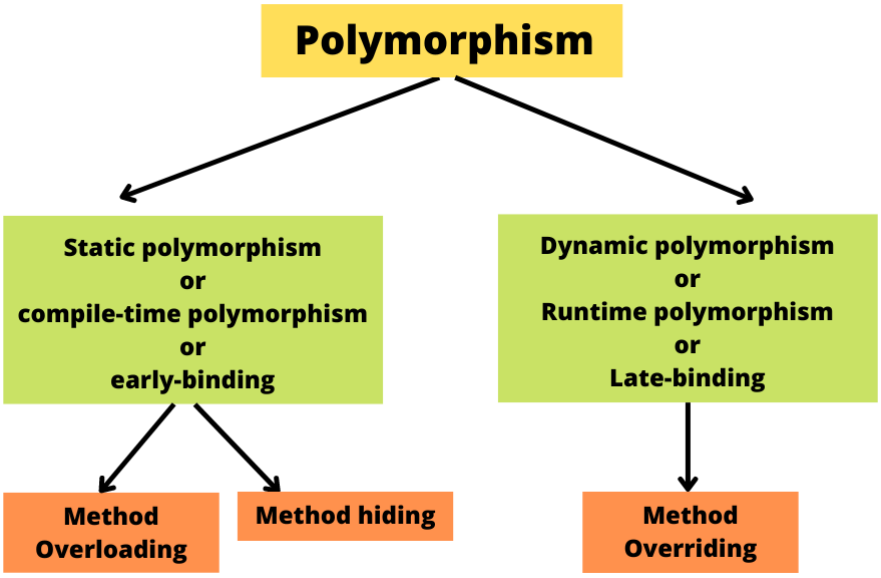
Test.java

```
class Animal {
    String sound;
    public void makenoise() {
        System.out.println(sound);
    } }
class Dog extends Animal {
    public Dog(String sound) {
        this.sound = sound;
    } }
class Cat extends Animal {
    public Cat(String sound) {
        this.sound = sound;
    } }

class Test {
    public static void main(String[] args) {
        Animal[] animals = new Animal[2];
        animals[0] = new Dog("Baw Baw!");
        animals[1] = new Dog("Meow!");
        for(int i=0; i<animals.length; i++) {
            animals[i].makenoise();
        }
    }
}
```

Output:

```
Baw  Baw!
Meow!
```

- Polymorphism in Java can be achieved through two main mechanisms:



Let's see each of these type of polymorphism in detail.

9.4.1 Method overloading

- Allows multiple methods with the same name but different parameters to coexist within the same class.
- When an overloaded method is called, the compiler determines the appropriate method to execute based on the arguments provided.
- Key aspects of method overloading:
  - **Method signature:**
    - Consists of the method name and the parameter list.

Syntax:

modifier returnType **methodName(type1 arg1, type2 arg2, ..)**

    - Overloaded methods must have the same name but different parameter lists.
    - The parameter lists can differ in terms of the number of parameters, their types, or both.
    - Compiler will use method signature while resolving method calls

Note:

- Method **returnType** and **modifier** are not considered as part of method signature and won't affect method overloading.
- Within a class 2 methods with same signature not allowed

- Eg:

Calculate.java

```
class Calculate {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    public String add(String str1, String str2) {
        return str1 + str2;
    }

    public static void main(String[] args) {
        Calculate math = new Calculate();
        int sum1 = math.add(5, 10);
        double sum2 = math.add(2.5, 3.7);
        int sum3 = math.add(1, 2, 3);
        String concatenated = math.add("Hello", "World");
        System.out.println("Sum1: " + sum1);
        System.out.println("Sum2: " + sum2);
        System.out.println("Sum3: " + sum3);
          System.out.println("Concatenated: " + concate-
nated);
    }
}
```

Output:

```
Sum1: 15
Sum2: 6.2
Sum3: 6
Concatenated: HelloWorld
```

9.4.2 Method overriding

- Allows a subclass to provide its own implementation of a method that is already defined in its superclass.
- When a method in the subclass has the **same name, return type, and parameter list** as a method in the superclass, it is said to override the superclass method.
- Key points about method overriding:
  - **Inheritance:** Based on the concept of inheritance, where a subclass inherits methods and fields from its superclass.
  - **Signature:** The overriding method must have the same method signature (name, return type, and parameter list) as the method it is overriding.
  - **Access modifier:** The overriding method cannot have a more restrictive access modifier than the method it is overriding. It can have the same or a more permissive access modifier.
  - **@Override annotation:** It is a good practice to use the @Override annotation when overriding a method.

- Eg:

Test.java

```
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Test {
    public static void main(String[] args) {
        Animal animal1 = new Cat();
        Animal animal2 = new Dog();

        animal1.makeSound();
        animal2.makeSound();
    }
}
```

Output:

```
Cat meows
Dog barks
```

Method overriding with respect to var_arg

- For overriding wrt var_arg method, the overriding method must have the **same method name, return type, and parameter types** as the overridden method.
- Eg:

Main.java

```
class BaseClass {
    public void printValues(String... values) {
        System.out.println("BaseClass - printValues:");
        for (String value : values) {
            System.out.println(value);
        }
    } }
class SubClass extends BaseClass {
    @Override
    public void printValues(String... values) {
        System.out.println("SubClass - printValues:");
        for (String value : values) {
            System.out.println(value);
        }
    } }

class Main {
    public static void main(String[] args) {
        BaseClass base = new BaseClass();
        base.printValues("Hello", "World");
```

```
        SubClass sub = new SubClass();
        sub.printValues("Hello", "World");
        BaseClass polymorphic = new SubClass();
        polymorphic.printValues("Hello", "World");
    } }
```

Output:

```
BaseClass - printValues:
Hello
World
SubClass - printValues:
Hello
World
SubClass - printValues:
Hello
World
```

9.4.3 Overloading V/S Overriding

Property	Overloading	Overriding
Method names Argument types	Must be same Must be different (atleast order)	Must be same Must be same (including order)
Method signature	Must be different	Must be same
Return types	No restrictions	Must be same until Java1.4 version , from Java1.5 version co-varient return types also allowed
Private, static, final methods	Can be overloaded	Cannot be overloaded
Access modifiers	No restrictions	The scope of access modifiers cannot be reduced but we can increase
Method resolution	Always takes care by compiler based on reference types	Always takes care by JVM based on runtime object
It is also known as	Compile time polymorphism, static polymorpohim or early binding	Runtime polymorphism, dynamic polymorphism, or late binding

9.4.4 Method hiding

All rules of method hiding are exactly same overriding except the following difference:

Method hiding	Method overriding
Both parent and child class method should be static	Both parent and child class method should be non-static
Compiler is responsible for method resolution based on reference type	JVM is always responsible for method resolution based on run-time object
It is also known as compile-time polymorphism or static polymorphism or early binding	It is also known as runtime polymorphism or dynamic polymorphism or late binding

Eg:

Test.java

```
class Parent {
    public static void m1() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    public static void m1() {
        System.out.println("Child");
    }
}
class Test {
    public static void main(String[] args) {
        Parent p1 = new Parent();
        p1.m1();
        Child c1 = new Child();
```

```
        c1.m1();
        Parent p2 = new Child();
        p2.m1();
    }
}
```

Output:

```
Parent
Child
Parent
```

Note:

Method hiding is like sticking poster on black board, while method overriding is like erasing board and writing new content.

9.5 Data Encapsulation

- The process of binding data and corresponding methods (behavior) together into a single unit is called encapsulation in Java.

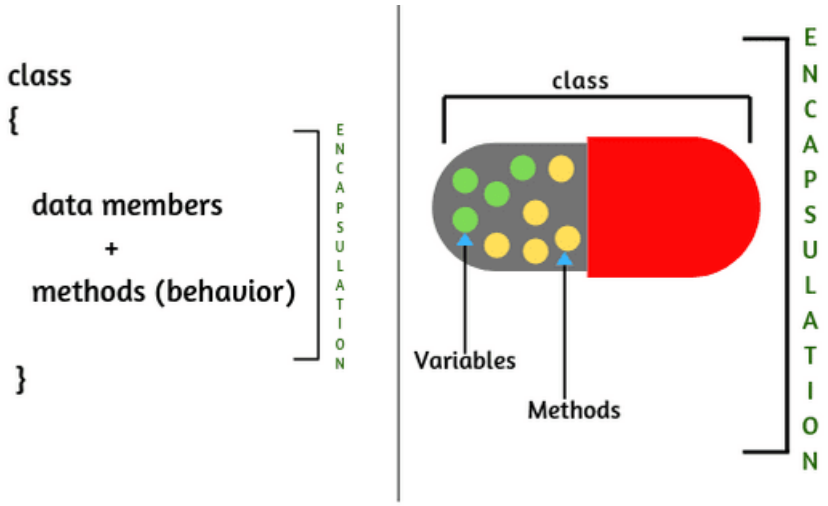


Fig: Encapsulation

- It makes variables and methods safe from outside interference and misuse.
- Data encapsulation = **data hiding** + **data abstraction**.
- Let see each of these in detail.

9.5.1 Data Hiding

- Data hiding refers to encapsulating data within a class and controlling its visibility and accessibility from outside the class.
- Below code shows data being directly accessible to the objects and it's not data hidiing:

Test.java

```
class Sample {
    int no;
    String name;
}
class Test {
    public static void main(String[] args) {
        Sample s1 = new Sample();
        s1.no = 1; // No Data hiding
        s1.name = "Apples"; // No Data hiding
    }
}
```

- In order to hide the data, the instance variable should be declared as **private**.
- Eg: You can create public methods **setter()** and **getter()** to get and set private instance variables as shown below:

Test.java

```
class Sample {
    private int no;
    private String name;
    public void setter(int no, String name) {
        this.no = no;
        this.name = name;
    }
```

```
        public void getter() {
                System.out.println(no + ": " + name);
        }
}
class Test {
        public static void main(String[] args) {
                Sample s1 = new Sample();
                s1.setter(0,"Apple"); // Data hidden
                s1.getter();
        }
}
```

Output:

```
0: Apple
```

- It makes variables and methods safe from outside interference and misuse.
- Data encapsulation is closely related to **data hiding**.

9.5.2 Data Abstraction

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).

9.5.3 Tightly encapsulated class

- A class is said to be tightly encapsulated if and only if each variable declared is **private**.
- Tightly encapsulate class may or may not contain corresponding getter and setter method or these methods may or may not be declared as public.
- Eg:

Code:

```
//Below is tightly encapsulated class
class A {
    private int a;
}

//Below is not tightly encapsulated class
class B extends A {
    int b;
}
```

9.5.4 Coupling

- The degree of dependency between the components is called **coupling**.
- If dependency is more, then it is considered as **tightly coupling**.
- If dependency is less, then it is considered as **lossly coupling**.
- Eg:

A.java

```
class A {
    static int i = B.j;
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i);
    }
}
class B {
    static int j = C.k;
}
class C {
    static int k = D.m1();
}
class D {
    public static int m1() {
        return 10;
    }
}
```

Output:

```
10
```

- **Tightly coupling is not good** programming practise.
- Problem with tighly coupling:
  - Without affecting remaining components we can modify any component and hence enhaacement will become difficult.
  - It suppresses reuseability.
  - It reduces mainitainabilty of the application.
  - Hence we have to maintain dependency between the components as less as possible i.e lossly coupling is a good programming practice.

9.5.5 Cohesion

- Refers to the degree to which working of a module (classes or a packages) are related on a single purpose.
- It measures how closely the elements within a module are related & how well they work together to achieve a common objective.
- **High cohesion is considered desirable** in software design because it leads to more modular, maintainable, and understandable code.
- When a module exhibits high cohesion, it means that its members (variables, methods, and classes) are closely related and contribute to a single, well-defined purpose.

Don't fear failure.

Not failure, but low aim is the crime.

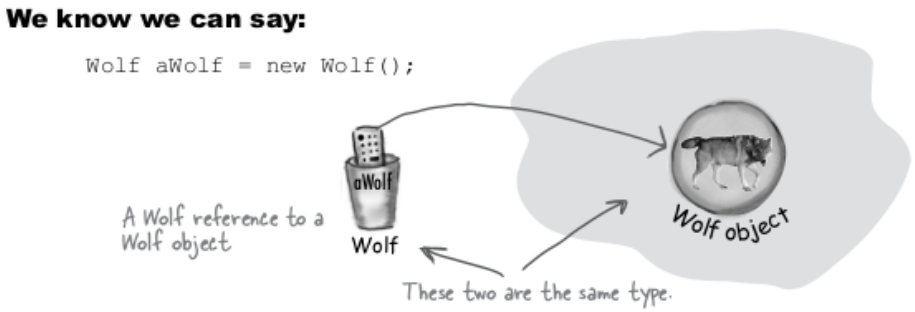
10. Abstract Class & Interface

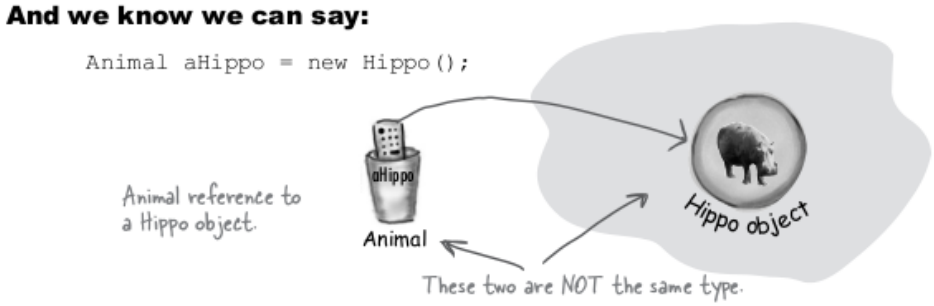
10.1 Abstract class

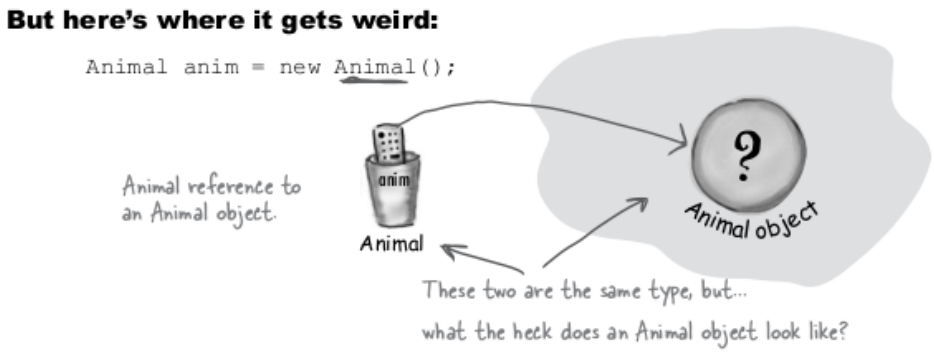
10.1.1 What is abstract class?

- An **abstract class cannot be instantiated directly**.
- It **serves as a blueprint** or template for its subclasses.
- It defines **common attributes and behaviors** that can be shared among multiple related classes.

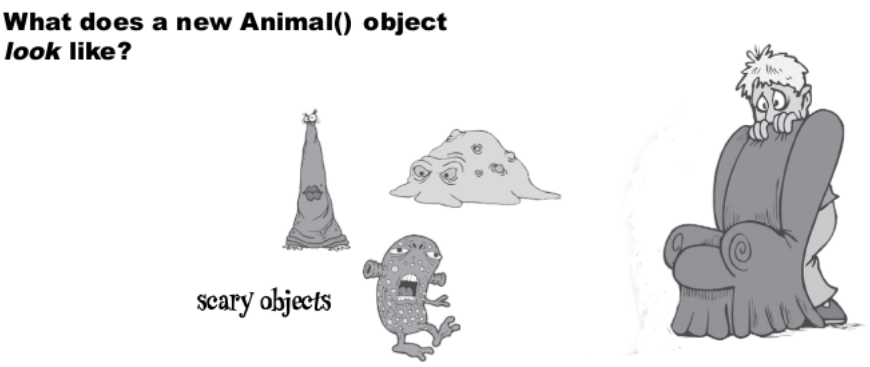
- **What is the need of abstract class?**







- Problem is -



- Similarly, there can be situations where there is no need to create parent class object.
- The **compiler** won't let you instantiate an abstract class.

Syntax:

```
abstract public class Classname {}
```

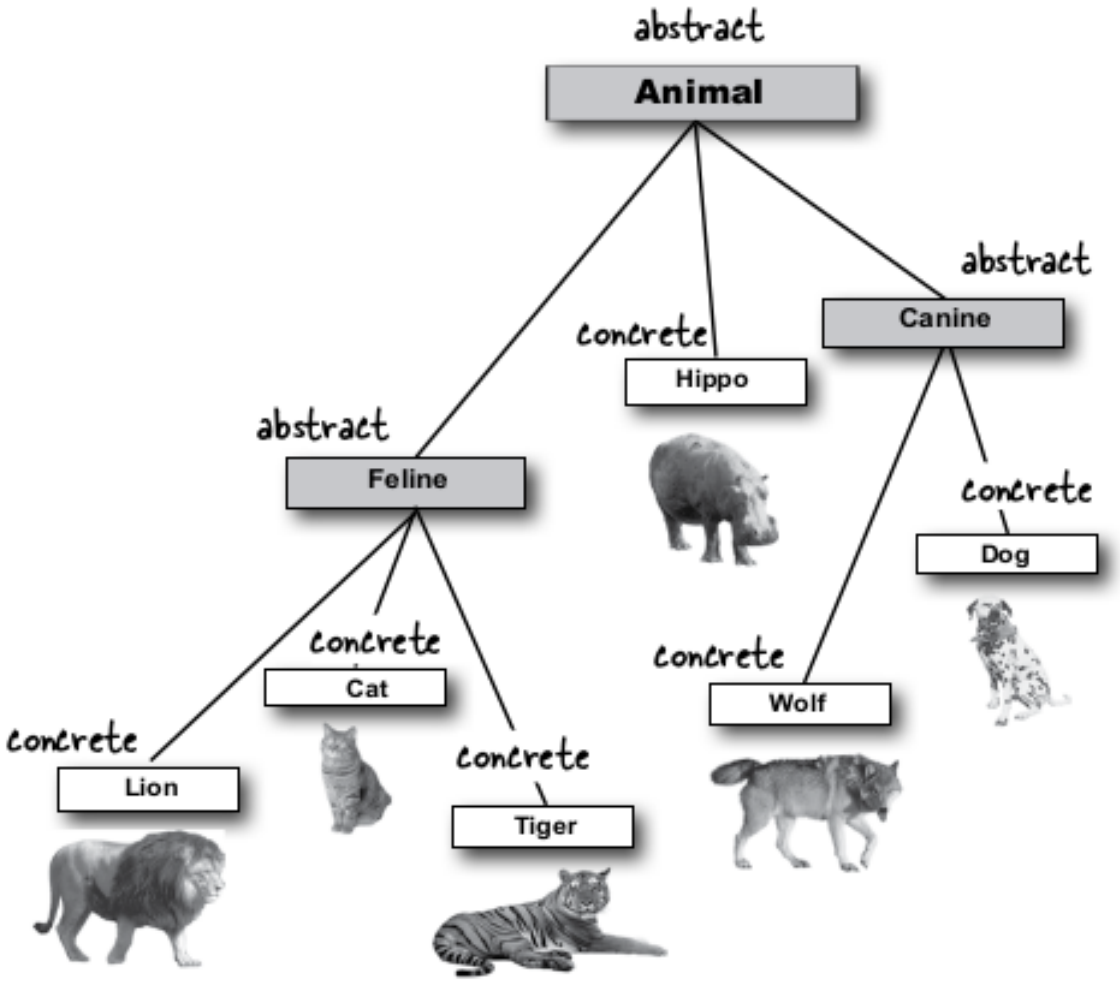
- Eg:

Test.java

```
abstract public class Canine extends Animal {
    public void roam() {}
}

public class MakeCanine {
    public void go() {
        Canine c;
        c = new Dog(); ✓
        c = new Canine(); ✗
    }
}
```

10.1.2 What is concrete class?

- A class that's not abstract is called a **concrete** class.
- Eg:



10.1.3 What is abstract method?

- An abstract method has no body!
- If you declare an abstract method, you MUST mark the class abstract as well.
- You can't have an abstract method in a non-abstract class.
- Abstract methods are declared using the abstract keyword, and they end with a semicolon instead of a method body.

Syntax:

```
abstract class Classname() {
    public abstract void methodname1();
    public abstract void methodname2();
}
```

- These methods are intended to be overridden by the subclasses.
- Subclasses **must provide the implementation** for all the abstract methods.
- Eg:

Main.java

```
abstract class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public abstract void sound();
}
```

```
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
    public void sound() {
        System.out.println("Woof!");
    }
}
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }
    public void sound() {
        System.out.println("Meow!");
    }
}

class Main {
    public static void main(String[] args) {
        Animal dog = new Dog("Buddy");
        Animal cat = new Cat("Whiskers");

        System.out.println(dog.getName());
        dog.sound();

        System.out.println(cat.getName());
        cat.sound();
    }
}
```

Output:

```
Buddy
Woof!
Whiskers
Meow!
```

10.1.4 Illegal modifier: final abstract

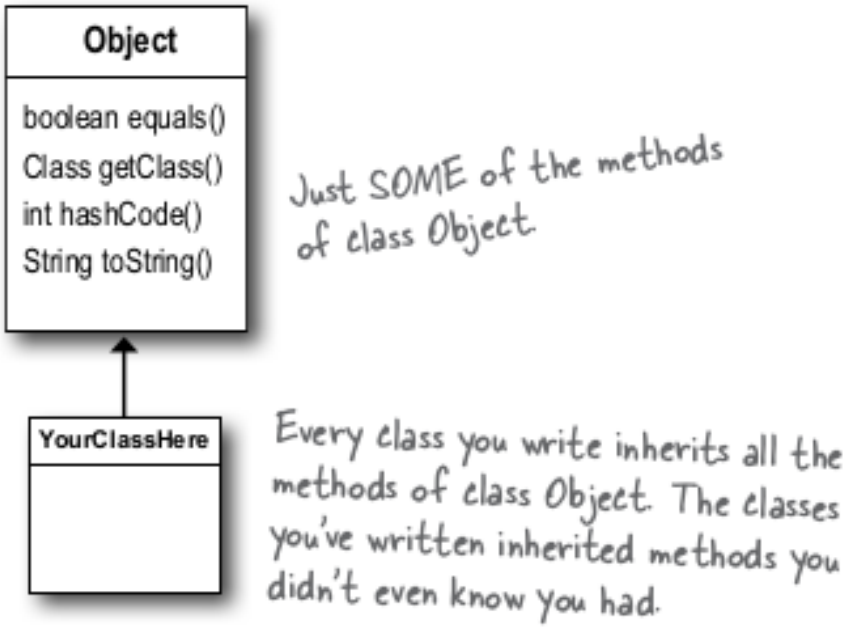
- Abstract methods compulsory we should override in child classes to provide implementation.
- Whereas, we can't override final methods.
- Hence final abstract combination is illegal combination for methods.

10.2 The ultimate superclass: Object

- Every class in Java extends class **Object**.
- Class Object is the mother of all classes; it's the superclass of everything.
- Every class you write extends Object, without your ever having to say it.

So what's in this Object class?

- Class Object does indeed have methods for below four things:



– **equals(Object o)**: Used to check if 2 objects are considered "equal".

Test.java

```
class Dog {}
class Cat {}

class Test {
    public static void main(String[] args) {
        Dog a = new Dog();
        Cat b = new Cat();
        if (a.equals(b)) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
    }
}
```

Output:

```
false
```

– **getClass()**: Used to display the class that object was instantiated from.

Test.java

```
class Cat {}

class Test {
    public static void main(String[] args) {
        Cat a = new Cat();
        System.out.println(a.getClass());
    }
}
```

Output:

```
class Cat
```

– **hashCode()**: Prints out a hashcode for the object (think of it as a unique code).

Test.java

```
class Cat {}
class Test {
    public static void main(String[] args) {
        Cat a = new Cat();
        System.out.println(a.hashCode());
    }
}
```

Output:

```
1178945
```

– **toString()**: Prints out a string message with the name of the class and random number.

Test.java

```
class Cat {}

class Test {
    public static void main(String[] args) {
        Cat a = new Cat();
        System.out.println(a.toString());
    }
}
```

Output:

```
Cat@515f550a
```

10.2.1 Object type-casting

10.3 Interface

- An interface is like a 100% pure abstract class.
- Inside interface, every method is always abstract whether we are declaring it or not.
- Any service requirement specification(SRS) or any contract between client and service provider are 100% pure abstract class, is nothing but interface.
- To declare an interface, use the interface keyword followed by the name of the interface.

Syntax:

```
public interface InterfaceName {}
```

- To implement an interface, a class must use the implements keyword followed by the name of the interface(s) it is implementing.

Syntax:

```
public class ClassName implements InterfaceName {}
```

- A class can implement any number of interfaces solving the multiple inheritance problem.

10.3.1 Interface methods

- Implementing an interface requires implementing each method of interface.
- Interface methods should always by declared as **public**.
- Interface methods are abstract whether we are declaring or not.
- Hence inside interface the ofllowing method declaration are equal:

Code:

```
void m1();
public void m1();
abstract void m1();
public abstract void m1();
```

- Eg:

Circle.java

```
public interface Drawable {
    void draw();
    double calculateArea();
}
public class Circle implements Drawable {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }
    @Override
    public double calculateArea() {
```

```
            return Math.PI * radius * radius;
        }
        public static void main(String[] args) {
            Circle circle = new Circle(4);
            circle.draw();
            System.out.println(circle.calculateArea());
        }
}
```

Output:

```
Drawing a circle.
50.26548245743669
```

10.3.2 extends V/S implements

<table>
<tr><th>extends</th><th>implements</th></tr>
<tr><td>A class can extend only one class at a time:
Code:
class B {}
class A extends B {}</td><td>A class can implement any number of interfaces:
Code:
interface A {}
interface B {}
class C implements A, B {}</td></tr>
<tr><td>A class can extend another class and can implement any number of interfaces:
Code:
class B {}
interface C {}
interface D {}
class A extends B implements C,D {}</td><td>An interface can extend any number of interfaces:
Code:
interface A {}
interface B {}
interface C extends A, B {}</td></tr>
</table>

10.3.3 Interface variables

- Every interface variable is always **public, static, final whether we are declaring or not**.
- For interface variables, you should perform initialisation at the time of declaration else you will get compile-time error.

Code:

```
interface Test1   int x = 10;
```

- Why interface variable is public static and final always?
  - **public**: To make variable available to every implementing class.
  - **static**: Without any object also, implementing class should be able to access the variable.
  - **final**: If one implementing class changes value then remaining implementation class will be affected. To restrict this, every interface variable is final.
- Hence within the interface, the following variable declarations are equal:

Code:

```
interface Test1 {
    int x = 10;
    public int x = 10;
    public static int x = 10;
    public static final int x = 10;
    static final int x = 10;
}
```

10.3.4 Interface method naming conflicts

- **Case 1:** If 2 interfaces contains method **with same signature and same return type**, then in the implementation class, you have to provide implementation for only one method.
  Eg:

Test.java

```
interface A {
    public void m1();
}
interface B {
    public void m1();
}
class Test implements A, B {
    public void m1() {
        System.out.println("Method implemented");
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        t1.m1();
    }
}
```

Output:

```
Method implemented
```

- **Case 2**: If 2 interface contians methods with **same name but different argument types**, then in implementation class we have to provide implementation for both methods and these methods acts as overload methods.

Eg:

Test.java

```
interface A {
    public void m1();
}
interface B {
    public void m1(int i);
}
class Test implements A, B {
    public void m1() {
        System.out.println("Method implemented");
    }
    public void m1(int i) {
        System.out.println("Value passed:"+i);
}
    public static void main(String[] args) {
        Test t1 = new Test();
        t1.m1();
        t1.m1(50);
    }
}
```

Output:

```
Method implemented
Value passed:50
```

- **Case 3**: If 2 interfaces contains a method with **same signature but different return types**, then it is **impossible to implement** both interfaces simultaneously.

Code:

```
// Below interface method cannot be overriden:
interface A {
    public void m1();
} interface B {
    public int m1();
}
```

Can java class be implemented any number of interfaces sumultanoutsly?

Ans: Yes, except a particular case. If 2 interfaces contains a method with same signature but different return types then it is impossible to implement both interfaces simultaneously.

10.3.5 Interface variable naming conflicts

- Multiple interfaces can have variable naming conflicts.
- But we can solve this problem by using interface names:
- Eg:

Test.java

```
interface A {
    int x = 777;
}
interface B {
    int x = 888;
}
class Test implements A, B {
    public static void main(String[] args) {
        System.out.println(A.x); // Output: 777
        System.out.println(B.x); // Output: 888
    } }
```

10.3.6 Marker interface

- A marker interface is an interface that does not declare any methods.
- Its sole purpose is to mark or tag a class, indicating that the class has a certain property or behavior.
- A marker interface is essentially an empty interface, but by implementing that interface, a class can convey some additional information to the compiler.
- Here's an example of a marker interface named Serializable in Java:

Code:

```
public interface Serializable {
    // Empty interface
}
```

- Below are some markers interfaces, these are marked for some ability:
  - Serialisable (I)
  - Cloneable (I)
  - RandomAccess (I)

Without having any methods, how the objects will get some ability in marker interfaces?

Ans: Internally JVM is responsible to provide required ability.

Why JVM is providing required ability in marker interfaces?

Ans:

- To reduce complexity of programming.
- And to make java language as simple.

Is it possible to create our own marker interface?

Ans:

- Yes, but customisation of JVM is required.
- For this we will have to design our own JVM.

10.3.7 Adapter Classes

- Adapter class is a simple Java class that implements an interface with only empty implementation.
- Using adapter class, you can extend the adapter class and override only the methods they need, instead of implementing all the methods of the interface.
- Eg:

AudioPlayer.java

```
public interface MediaPlayer {
    void play();
    void pause();
    void stop();
}

abstract class MediaPlayerAdapter implements MediaPlayer {
    @Override
    public void play() {
        // Default implementation
    }

    @Override
    public void pause() {
        // Default implementation
    }
```

```
    @Override
    public void stop() {
        // Default implementation
    }
}

public class AudioPlayer extends MediaPlayerAdapter {
    @Override
    public void play() {
        // Implementation specific to AudioPlayer
        System.out.println("Playing audio.");
    }
    public static void main(String[] args) {
        AudioPlayer player = new AudioPlayer();
        player.play();
    }
}
```

Output:

```
Playing audio.
```

10.3.8 Interface V/S Abstract class

Interface	Abstract Class
Used when you dont know anything about implementation, you just have requirement specification	Used when you know about partial implementation
Every method is always public and abstract whether we are declaring or not. It is 100% abstract class	Methods need not be public and abstract. Methods can be concrete
You can't declare with the following modifiers • Private • Protected • Static • Final • Native • Syncronised • Strictfp	There are no restrictions on abstract class method modifiers.
Variable is always "public static final" whether we are declaring it or not	Variable present inside abstract class need not be public static final.
Variables should perform initialization at the time of declaration	Variables are not required to perform initialsation at the time of declaration
Interface can't declare static and instance blocks	C contain static and instance blocks
Cannot contain constructors	Can declare constructors

Why abstract class can contain constructor but interface does not contain constructor?

Ans:

- The main purpose of constructor is to perform initialisation for the instance variables.
- Abstract class can contain instance variables, (for child object to perform initialsation of those instance variable) hence constructor is required for abstract.
- But, every variable present inside interface is always **public static final** whether you are declaring or not.
- Also there are no instance variables inside interface, hence constructor concept is not required for interface.

Inside interface, every method is always abstract and abstract class also contains abstract methods. Is it possible to replace interface with abstract class?

Ans:

- You can replace interface with abstract class, but it is not a good programming practise.
- If everything is abstract, then it is highly recommended to go interface & not abstract class.

11.1 Overview of java API

In this section, you are going to learn text processing commands like:

- **find & grep**
- **head & tail**
- **more & wc**
- **sort, cut & uniq**

There will be a **small excerise** on these topics to check your knowledge.

So let's get started....

11.1.1 String, String Buffer and String Builder

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.2 Exception Handling

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.3 Threads and multithreading

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.4 Wrapper Classes

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.5 Data Structures

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.6 JAVA COLLECTION FRAMEWORKS

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.7 File Handling

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```

11.1.8 Serialization

- Pandas is a Python library used to analyze data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Pandas Installation:

- Install pandas package using below command:

Syntax:

```
pip install pandas
```

- Pandas is usually imported under the pd alias.

Code:

```
import pandas as pd
```