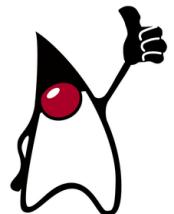




Core Java



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

Copyright © 2022 Lavatech Technology

The contents of this course and all its modules and related materials, including handouts are Copyright ©

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Lavatech Technology.

If you believe Lavatech Technology training materials are being used, copied, or otherwise improperly distributed please e-mail:

info@lavatechtechnology.com

PUBLISHED BY LAVATECH TECHNOLOGY

lavatechtechnology.com

January 2022



YOU CAN

TOTALLY

DO THIS!

Contents

1	Introduction to Java	13
1.1	Getting started with Java	13
1.1.1	What is Java?	13
1.1.2	Uses of Java	15
1.1.3	Problem with C and C++	16
1.1.4	Advantages of Java over C++	16
1.1.5	How does program execution works?	17
1.1.6	The way Java works	18
1.1.7	Compiler V/S JVM	19
1.1.8	JVM components	20
1.1.9	History of Java	21
1.2	JDK v/s JRE v/s JVM	25
1.3	Installing Java and IDE	26
1.3.1	Java installation	26
1.3.2	IDE installation	31
1.3.3	Our first Java program	31
1.3.4	Executing our first Java program	34
1.3.5	Using JShell	38

2 Java Language Fundamentals	41
 2.1 Identifiers, variables and more	41
2.1.1 Java identifiers	41
2.1.2 Java grammar	43
2.1.3 Java variable	43
2.1.4 Literals	44
2.1.5 Java Comments	44
2.1.6 Java reserved words or keywords	46
2.1.7 How Objects Can Change Your Life?	47
 2.2 Introductions to OOPs	50
2.2.1 Java source file	51
2.2.2 main() method	54
2.2.3 Command-line argument	58
3 Data types in Java	61
 3.1 Getting started with data type	61
3.1.1 Java is strongly typed	61
3.1.2 Types of data types	61
 3.2 Integer data type	62
3.2.1 byte	63
3.2.2 short	64
3.2.3 int	64
3.2.4 long	65
3.2.5 Integer literals	66
 3.3 Floating-point	69
3.3.1 Float	69
3.3.2 Double	69
3.3.3 Floating-point literals	70
 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	75

3.5 Boolean	76
3.6 Type conversion	77
4 String, StringBuffer, StringBuilder	81
4.1 String	81
4.1.1 Mutable and Immutable objects	81
4.1.2 String constructors	83
4.1.3 String indexing	85
4.1.4 String methods	86
4.2 StringBuffer	90
4.2.1 StringBuffer constructors	91
4.2.2 StringBuffer methods	92
4.3 StringBuilder	98
4.3.1 StringBuffer V/S StringBuilder	99
4.4 String method chaining	99
4.5 == V/S equals()	100
4.6 String memory management	101
4.6.1 String Constant Pool(SCP)	102
4.6.2 Heap	104
5 Arrays	105
5.1 Arrays in detail	105
5.1.1 Array introduction	105
5.1.2 Array declaration	106
5.1.3 Array creation	108
5.1.4 Array initialisation	113
5.1.5 Array declaration, creation and initialisation in one line	116
5.1.6 length variable	117
5.1.7 Anonymous Arrays	119
5.1.8 Array element assignments	120
5.1.9 Array variable assignments	121

6 Operators	123
 6.1 Operators in Java	124
6.1.1 Arithmetic Operator	124
6.1.2 String Concatenation Operator	128
6.1.3 Increment/Decrement Operator	129
6.1.4 Relational Operator	132
6.1.5 Bitwise Operator	134
6.1.6 Boolean complement operator	136
6.1.7 Short circuit Operator	137
6.1.8 Assignment Operator	138
6.1.9 Conditional Operator	140
6.1.10 new Operator	140
6.1.11 [] Operator	140
6.1.12 Operator Precedence	141
7 Flow Control	143
 7.1 Flow control statement	143
 7.2 Selection Statements	144
7.2.1 if...else	144
7.2.2 switch case	147
 7.3 Iteration	151
7.3.1 while()	151
7.3.2 do-while()	153
7.3.3 for()	155
7.3.4 for-each loop	157
 7.4 Transfer Statements	159
7.4.1 break	159
7.4.2 continue	160
7.4.3 Labeled break & continue	161
8 Wrapper classes	163
 8.1 Introduction	163
 8.2 Wrapper class constructors	164

8.3 Integer wrapper class methods	166
8.4 Utility methods	169
8.5 Autoboxing and AutoUnboxing	174
9 Packages in Java	177
9.1 import keyword	177
9.2 Packages	180
9.3 JAR files	187
10 OOPs in detail	191
10.1 Class	191
10.1.1 Attributes in detail	192
10.1.2 Methods in detail	199
10.1.3 Class level access modifier	208
10.2 Constructor in detail	212
10.2.1 new operator	212
10.2.2 Constructor	213
10.2.3 this keyword	214
10.2.4 Constructor Chaining..	215
10.2.5 super()	216
10.2.6 super keyword	217
10.2.7 Constructor overloading	218
10.2.8 this()	219
10.2.9 super(),this() V/S super,this	220
10.2.10 Constructor's Access modifier	220
10.2.11 Instance block	224
10.2.12 static initializer	225
10.2.13 Constructor V/S Instance block V/S Static block	226
10.2.14 Destroying object	227
10.3 Inheritance	227
10.3.1 Types of inheritance	229
10.3.2 Has-A relationship	231
10.3.3 Has-A V/S Is-A relationship	232

10.4 Polymorphism	233
10.4.1 Method overloading	235
10.4.2 Method overriding	236
10.4.3 Method hiding	237
10.4.4 Overloading V/S Overriding	238
10.5 Data Encapsulation	239
10.5.1 Data Hiding	239
10.5.2 Data Abstraction	240
10.5.3 Tightly encapsulated class	241
10.5.4 Coupling	241
10.5.5 Cohesion	242
11 Abstract Class & Interface	243
11.1 Abstract class	243
11.1.1 What is abstract class?	243
11.1.2 What is concrete class?	245
11.1.3 What is abstract method?	246
11.1.4 Illegal modifier: final abstract	246
11.2 The ultimate superclass: Object	247
11.3 Interface	249
11.3.1 Interface methods	249
11.3.2 extends V/S implements	250
11.3.3 Interface variables	251
11.3.4 Interface method naming conflicts	252
11.3.5 Interface variable naming conflicts	254
11.3.6 Marker interface	255
11.3.7 Adapter Classes	256
11.3.8 Interface V/S Abstract class	257
12 File Handling in Java	259
12.1 File IO	259
12.1.1 File class	260
12.1.2 FileWriter class	264

12.1.3 FileReader class	267
12.1.4 Drawback of FileWriter & FileReader	268
12.1.5 BufferedWriter class	268
12.1.6 BufferedReader class	270
12.1.7 PrintWriter class	271

13 Exception Handling 275

13.1 What is exception? 275
--

13.1.1 What is Exception Handling?	275
13.1.2 Default Exception Handling	276

13.2 Exception Hierarchy 278

13.2.1 Throwable class	278
13.2.2 Exception	279
13.2.3 Error	280
13.2.4 Try..Catch..Finally	280
13.2.5 Rules for try..catch..finally	283
13.2.6 Methods to print exception	283
13.2.7 Try with multiple catch blocks	285

13.3 Java 1.7 Exception Handling Enhancement 287

13.3.1 try-with resources	287
13.3.2 Multi-catch block	290

13.4 Throwing exception 291
--

13.4.1 throw keyword	291
13.4.2 Checked & unchecked exception	293
13.4.3 FullyChecked v/s ParitallyChecked	295
13.4.4 User-defined exceptions	297
13.4.5 throws keyword	299
13.4.6 Exception propagation	301

13.5 Top 10 Exceptions 303

14 Collection Framework 311

14.1 Introduction 311

14.1.1 Drawback of arrays	311
---------------------------------	-----

14.1.2 What is Collection?	312
14.1.3 Difference between Array & Collection	313
14.1.4 What is Collection Framework?	314
14.1.5 Collection Interface	315
14.2 List	317
14.2.1 ArrayList	319
14.2.2 LinkedList	323
14.2.3 Vector	327
14.2.4 Stack	332
14.3 Queue	335
14.3.1 Comparator & Comparable	337
14.3.2 PriorityQueue	339
14.3.3 Deque	342
14.3.4 ArrayDeque	343
14.4 Set	344
14.4.1 HashSet	345
14.4.2 LinkedHashSet	347
14.4.3 SortedSet	348
14.4.4 NavigableSet	349
14.4.5 TreeSet	350
14.5 Map	352
14.5.1 HashMap	354
14.5.2 LinkedHashMap	357
14.5.3 IdentityHashMap	358
14.5.4 SortedMap	359
14.5.5 NavigableMap	360
14.5.6 TreeMap	361
14.5.7 Properties	363
14.6 Cursors	363
14.6.1 Enumeration	363
14.6.2 Iterator	366
14.6.3 ListIterator	369
14.7 Collections class	372
14.7.1 Sorting	373

14.7.2 Searching	374
14.7.3 Reversing	375
14.8 Advance array operations	376
14.8.1 Sorting	377
14.8.2 Searching	378
15 Multithreading	379
15.1 Introduction to Multi-threading	379
15.1.1 What is Multi-tasking?	379
15.1.2 What is Multi-threading?	380
15.1.3 Thread scheduler	381
15.1.4 Thread life-cycle	381
15.2 Thread creation	381
15.2.1 Extending Thread class	382
15.2.2 Implementing Runnable Interface	382
15.2.3 Getting and Setting name of a thread	384
15.2.4 Thread Priorities	385
15.2.5 start() & run() methods	387
15.2.6 Preventing thread from execution	391
15.3 Synchronization	400
15.3.1 Synchronised methods using object lock	400
15.3.2 Synchronised methods using class lock	405
15.3.3 Synchronised Block	408
15.3.4 Inter Thread Communication	412
15.4 More on Thread	415
15.4.1 DeadLock	415
15.4.2 Deadlock v/s Starvation	417
15.4.3 Daemon Thread	417
15.4.4 Default nature of a thread	418
15.4.5 Green Thread	418

16 Serialisation 419

16.1 Introduction 419

16.1.1 What is serialisation?	419
16.1.2 What is deserialisation?	420
16.1.3 Rules of serialisation	420
16.1.4 Transient Modifer	422
16.1.5 Object order in serialisation	424

16.2 Customised Serialisation 425

16.3 Externalisation 427

16.4 SerialVersionUID 429

16.4.1 Custom SerialVersionUID	430
--------------------------------------	-----

17 Garbage Collection 433

17.1 Introduction 433

17.1.1 Ways to make an object	434
17.1.2 JVM Methods for running Garbage collection	435
17.1.3 Finalisation	437
17.1.4 Memory Leaks	438

JAVA PADHO!

JOB KI CHINTA MAAT KARO

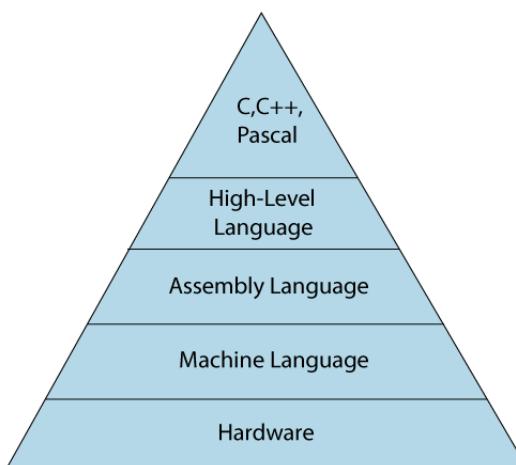


1. Introduction to Java

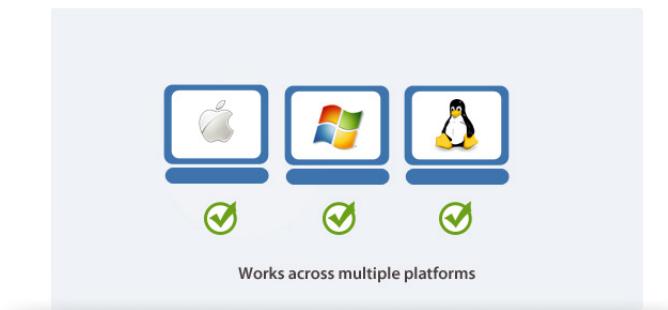
1.1 Getting started with Java

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 electronic devices, eg: **microwave ovens and remote controls.**
- Problem with C and C++ is they are **compiled for a specific target**.
- 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.
- **Simplicity:**
 - No pointers, unions, templates, structures, multiple inheritance.
- **Support multithreaded application**
- **Portability:** Support WORA (Write it once, run it anywhere), using Java virtual machine (JVM).

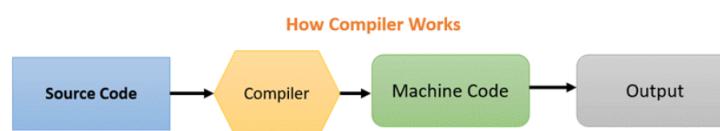
1.1.5 How does program execution works?

- **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.

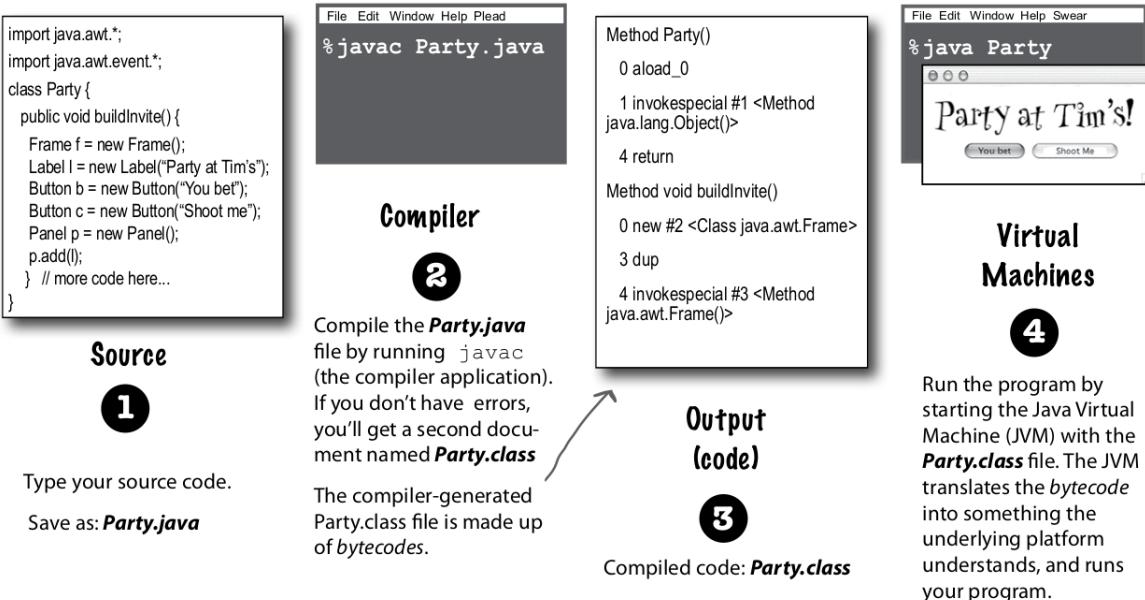
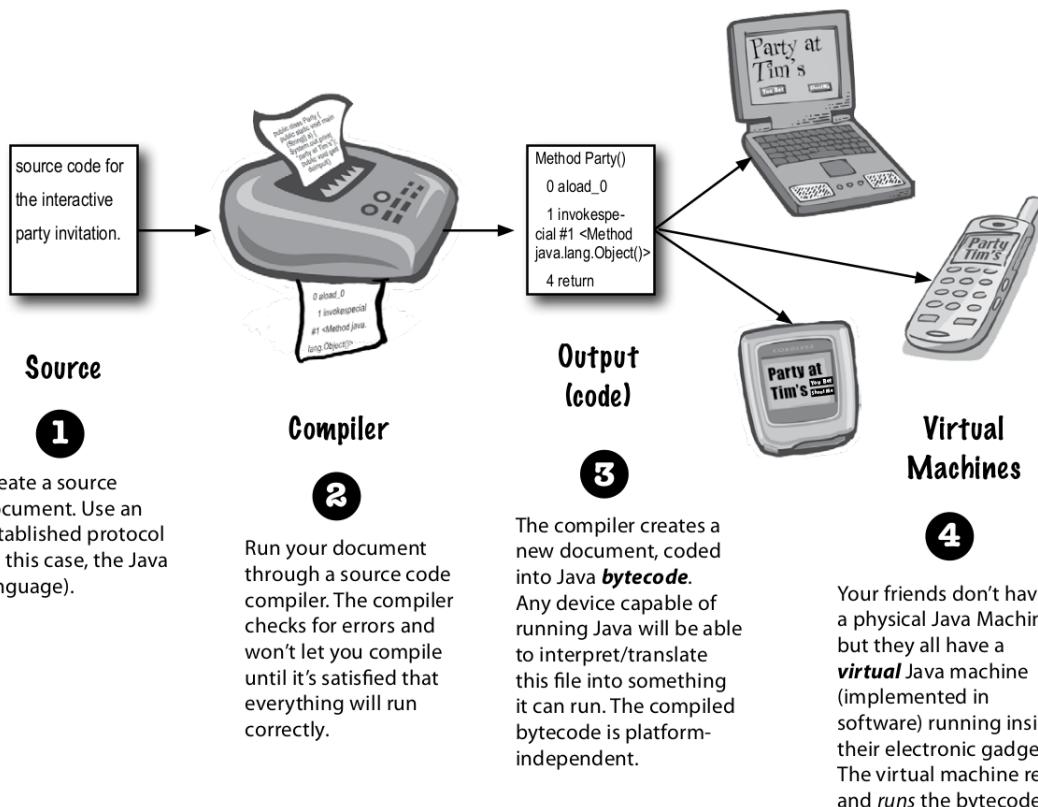


- **Java Virtual Machine (JVM):**

- 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?



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 its 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.

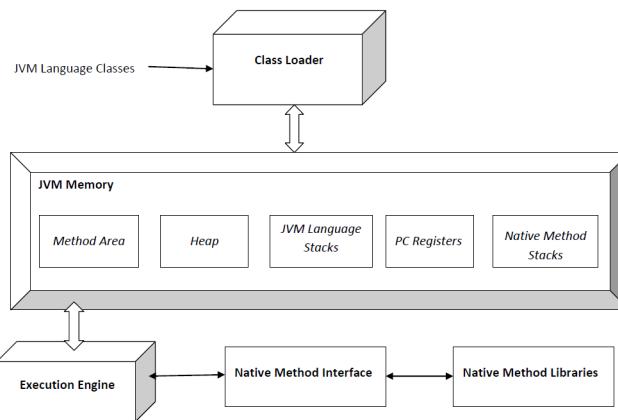
Compiler: Yes, that can emerge at runtime that only you can catch to allow dynamic binding. But my job is to stop anything that would never succeed at runtime.

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:** Loads class files into memory of JVM.
- **Execution Engine:** Executes bytecode loaded in memory. It includes:
 - **Interpreter:** Executes bytecode line-by-line.
 - **JIT compiler:** Compiles the bytecode into machine code for fast execution.
- **Garbage Collector:** Periodically frees memory for Java application.
- **Runtime Data Area:** Memory space assigned by JVM. Includes: Method area, Heap, Stack, PC registers.
- **Native Method Interface (JNI):** Allows Java code to call other languages code 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

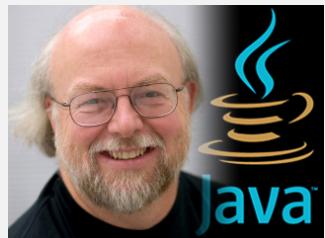
What company invented Java?

Ans: Sun Microsystems



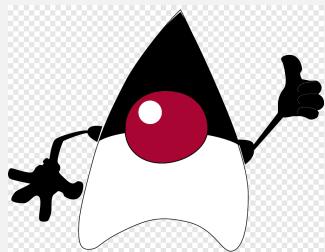
Who is founder of Java?

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 9	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.

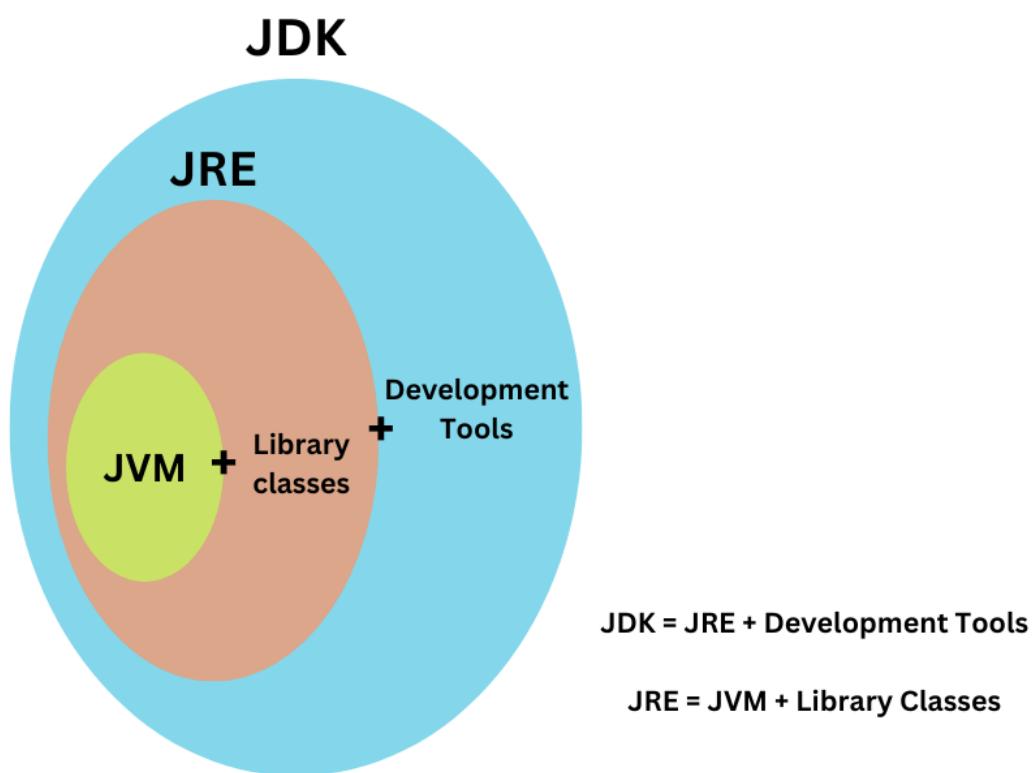
I see Java 2 and Java 5.0, but was there a Java 3 and 4? And why is it Java 5.0 but not Java 2.0?

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 JDK v/s JRE v/s JVM

- **JDK(Java Development Kit)** provides environment to development and run java application.
- **JRE(Java Runtime Environment)** provides environment to run java application.
- **JVM (Java virtual machine)** is responsible to run Java program line by line. Hence it is an interpreter.



Note:

On the developer's machine, install JDK whereas on the client machine install JRE.

1.3 Installing Java and IDE

1.3.1 Java installation

On Ubuntu, open terminal & follow below steps:

- First check if Java is already installed:

Command:

```
$ java -version
```

```
Command 'java' not found
```

- Install the Java Development Kit (JDK) with the following

command:

Command:

```
$ sudo apt install default-jdk -y
```

```
Setting up default-jdk-headless (2:1.11-72build2) ...
```

- Install JRE with the following command:

Command:

```
$ sudo apt install default-jre -y
```

```
Setting up default-jre (2:1.11-72build2) ...
```

- Optionally, you can install source code of Java:

Command:

```
$ sudo apt-get install openjdk-11-source -y
```

- Configure JAVA_HOME on Ubuntu, for this locate your Java installation on Ubuntu using below command:

Command:

```
$ update-alternatives --config java
```

```
There is only one alternative in link group java (providing /usr/bin/java): /usr/lib/jvm/java-11-openjdk-amd64/bin/java
```

- Add JAVA_HOME to the environment:

Command:

```
$ sudo vi /etc/environment  
JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64/bin/java"
```

- Reload the environment configuration file:

Command:

```
source /etc/environment
```

- Confirm Java is installed:

Command:

```
$ java -version  
openjdk version "11.0.20" 2023-07-18
```

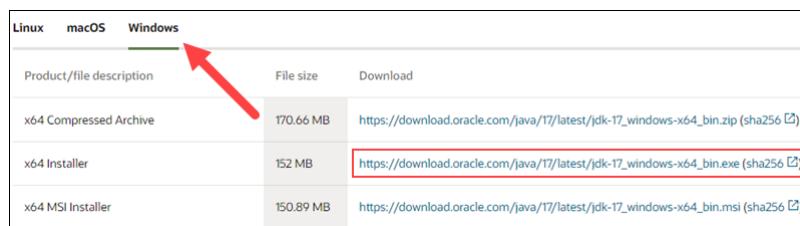
On Windows, follow below steps:

- Open a command prompt by typing **cmd** in the search bar and press **Enter**.
- Check if Java is installed by running following command:

```
C:\Users\boskom>java -version  
'java' is not recognized as an internal or external command,  
operable program or batch file.
```

```
C:\Users\boskom>
```

- Navigate to the <https://www.oracle.com/java/technologies/downloads/#jdk17-windows> & click the x64 Installer download link:



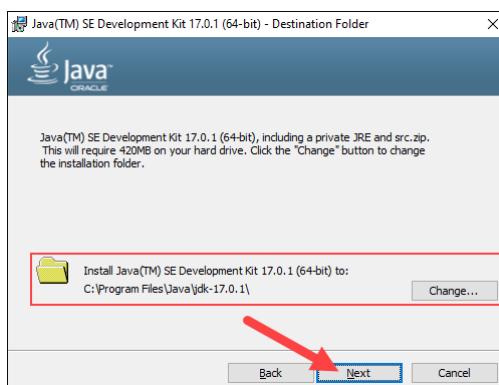
Linux	macOS	Windows
Product/file description	File size	Download
x64 Compressed Archive	170.66 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.zip (sha256)
x64 Installer	152 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.exe (sha256)
x64 MSI Installer	150.89 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.msi (sha256)

Double-click the downloaded file to start the installation.

- Configure the Installation Wizard:



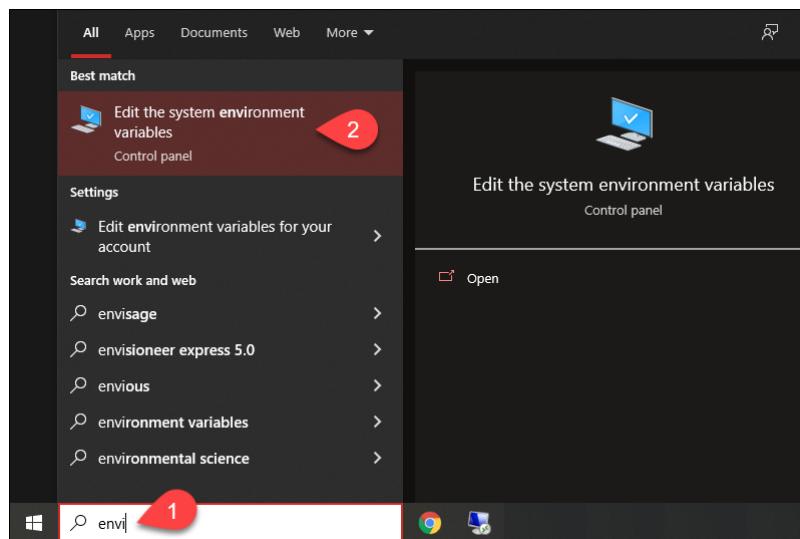
- Choose the destination folder for the Java installation files or stick to the default path. Click Next to proceed.



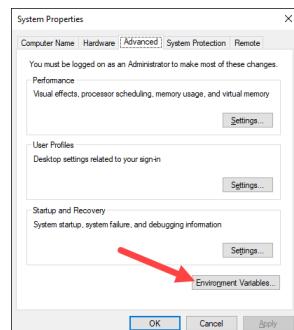
- Click Close to exit the wizard.



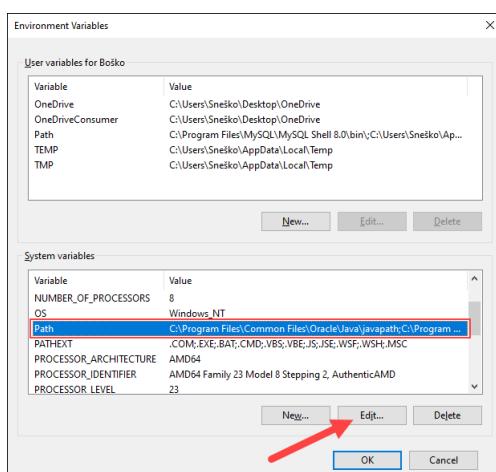
- Add Java to system variables. Open the Start menu and search for environment variables. Select the Edit the system environment variables result.



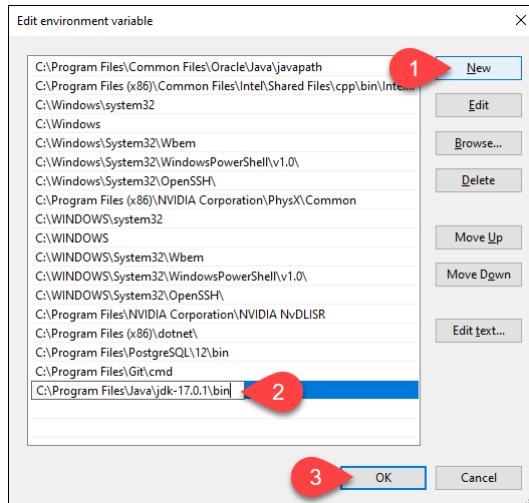
- In the System Properties window, under the Advanced tab, click Environment Variables...



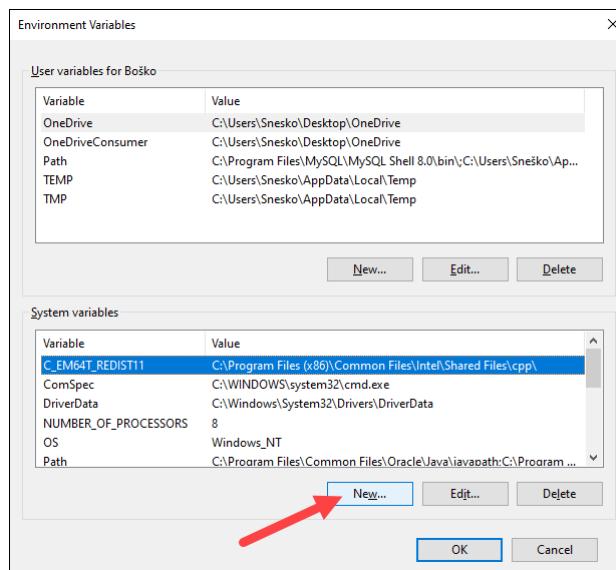
- Under the System variables category, select the Path variable and click Edit:



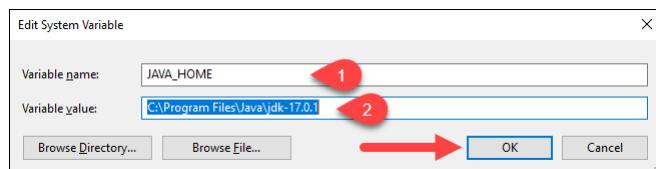
- Click the New button and enter the path to the Java bin directory:



- Add **JAVA_HOME** variable. In the Environment Variables window, under the System variables category, click the New... button to create a new variable.



- Name the variable as **JAVA_HOME**.
- In the variable value field, paste the path to your Java jdk directory and click OK.



- Confirm the changes by clicking OK in the Environment Variables and System properties windows.
- Test the Java Installation:

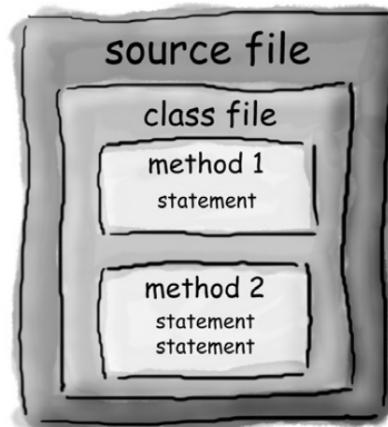
```
C:\Users\boskom>java -version
java version "17.0.1" 2021-10-19 LTS
Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)
```

1.3.2 IDE installation

Download and install VScode using this link on Microsoft Windows:
<https://code.visualstudio.com/download>

1.3.3 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()

```
public class Dog {  
    void bark() {  
    }  
}  
method
```

What goes in a method?

Ans:

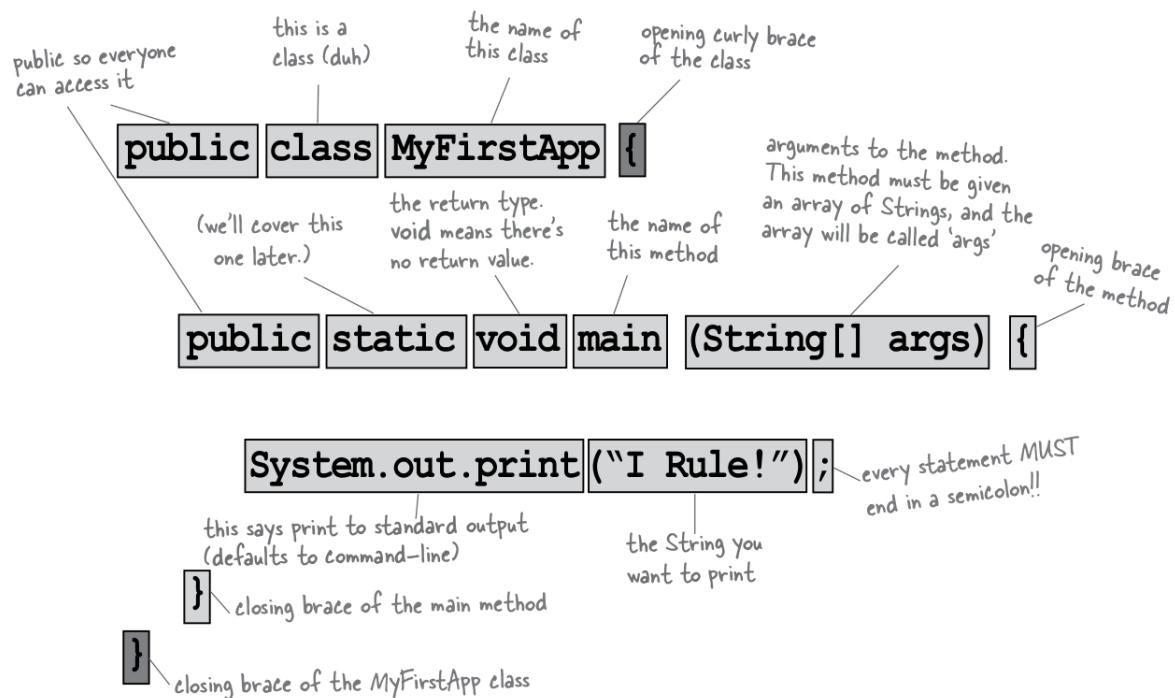
- Method code is basically a set of statements.
- Method kind of like a function or procedure.
- When the JVM starts running, it looks for the method inside the class that looks exactly like:

Code:

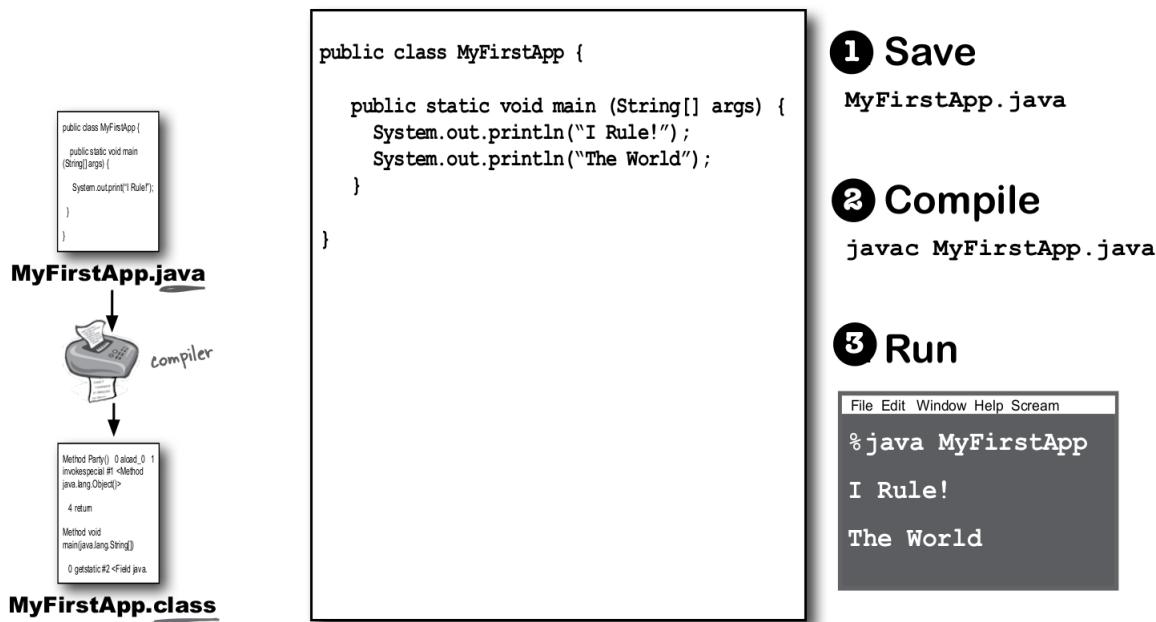
```
public static void main (String[] args) {
    // your code goes here
}
```

- 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:



1.3.4 Executing our first Java program

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

```

Understanding javac command

- **javac** command is used to compile a single or group of java source files.

Syntax:

```
javac [options] code1.java code2.java ....
```

- Eg:

Command:

```
javac Test.java  
javac A.java B.java C.java  
javac *.java
```

- Important options:

- -version: Display java compiler version.

Command:

```
$ javac -version  
javac 11.0.20
```

- -d: Specify where to place generated class files.

Command:

```
$ mkdir code  
$ javac -d code Test.java  
$ ls code  
apply.class
```

- -source: Provide specific Java release.

Command:

```
$ javac -source 11 Test.java
```

- -verbose: Output messages about what the compiler is doing.

Command:

```
$ javac -verbose Test.java
```

Understanding java command

- Use java command to run a single class file. You can run only one “.class” file at a time.

Syntax:

```
java [options] classname arg1 arg2 arg3 ....
```

- Eg:

Command:

```
$ java Test
```

- Important options:

- -version: Print product version.

Command:

```
$ java -version
openjdk version "11.0.20" 2023-07-18
```

- -D: Used to set the system property to customise behaviour program.

- * To display all properties of system, execute below code:

info.java

```
import java.util.*;
public class info {
    public static void main(String[] args) {
        Properties p = System.getProperties();
        p.list(System.out);
    }
}
```

- * You can set the property using **-D** option and use it in program as below:

info.java

```
public class info {  
    public static void main(String[] args) {  
        String fp = System.getProperty("fname");  
        System.out.println("File name is:"+fp);  
    }  
}
```

Command:

```
$ javac info.java  
$ java -Dfname=app.txt info  
File name is:app.txt  
  
$ java info  
File name is:null
```

- **-cp**: Classpath describes the location where the required “.class” files are available. By default, JVM searches in current working directory for the required “.class” file. Classpath can be set in the following 3 ways:

- * **Using environment variable**: This is permanent and preserved across system restarts. Recommended when you are installing a permanent software.
- * **Using “set” command**: Preserved only for particular command prompt.

Command:

```
$ set classpath=C:\lavatech_classes
```

- * **Using “-cp” option**: Preserved only for particular command. Eg:

apply.java

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

Command:

```
$ mkdir app  
$ javac -d app/ apply.java  
$ java -cp app apply  
Welcome
```

1.3.5 Using JShell

- Introduced in Java9, 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.
- To use jshell, type “jshell” command in command prompt:

Command:

```
$ jshell  
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"
```

- 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

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)

Valid Identifiers

String
_total
S1U1M
var_name
Integer
Int
_one_two
an\$

Invalid Identifiers

1abc
if
for
&ns
class
#one
int

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

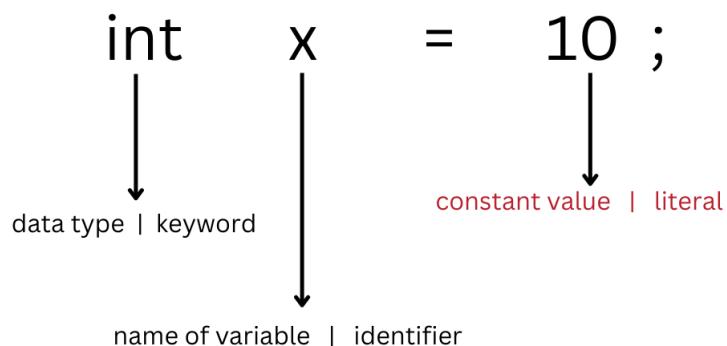
To declare more than one variable of the specified type, use a comma-separated list.

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 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;  
}
```

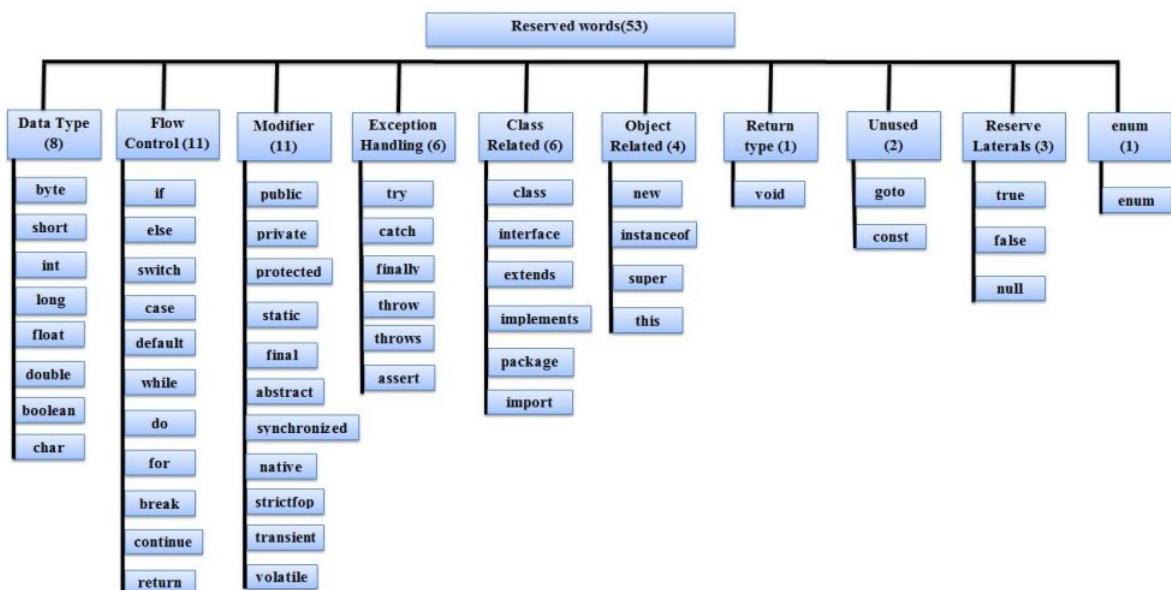
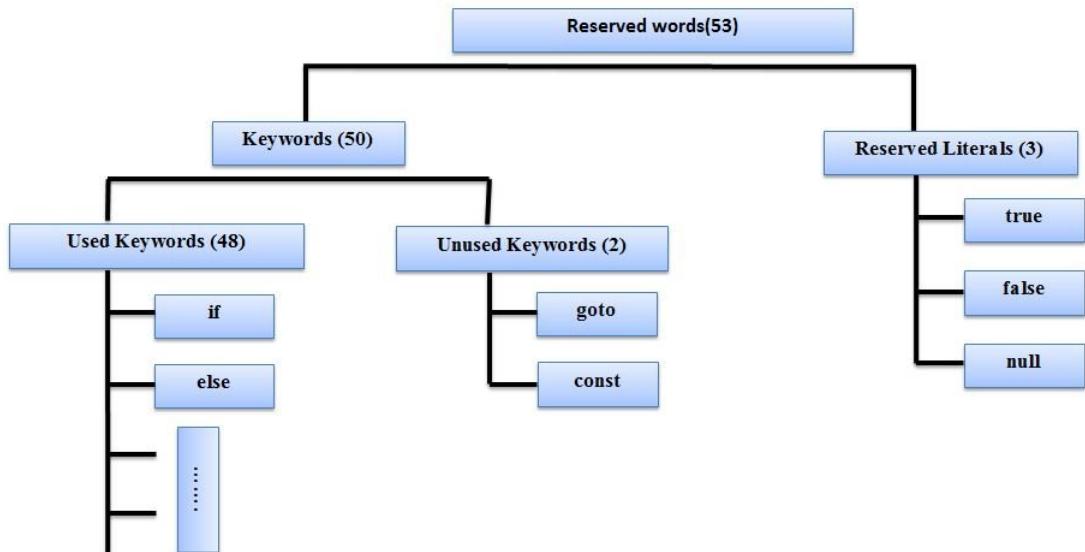
- /* ... */ : used for multiline comments

Code:

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

2.1.6 Java reserved words or keywords

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



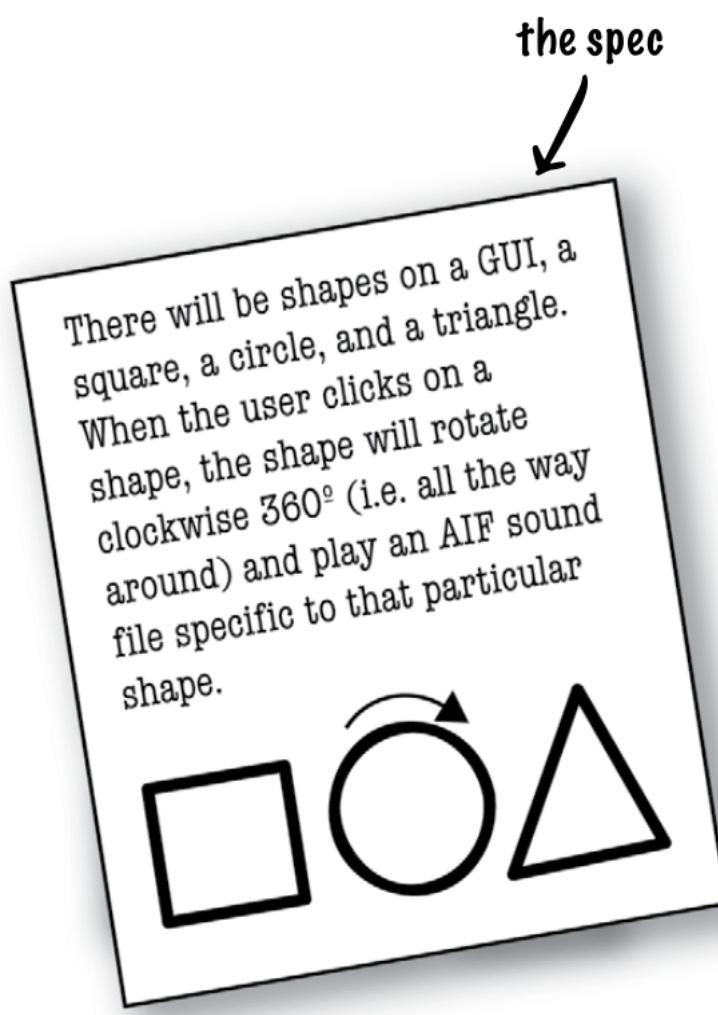
Note:

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

2.1.7 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:

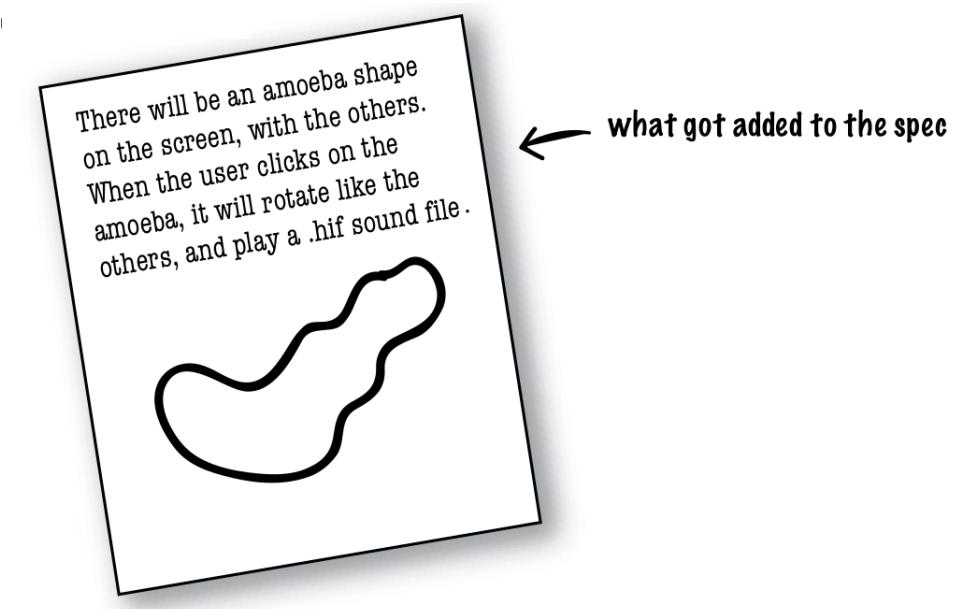
Create a Shape Application with below requirement:



- 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
<p>Code:</p> <pre>rotate(shapeNum) { //code here } playSound(shapeNum) { //code here }</pre>	<pre>Square rotate() { // code to rotate a square } playSound() { // code to play the AIFF file // for a square } Circle rotate() { // code to rotate a circle } playSound() { // code to play the AIFF file // for a circle } Triangle rotate() { // code to rotate a triangle } playSound() { // code to play the AIFF file // for a triangle }</pre>

- 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
<p>Raj will need to make changes in existing code and perform the testing again:</p> <div style="border: 1px solid black; padding: 10px;"> <p>Code:</p> <pre>rotate(shapeNum) { //new changes to add amoeba } playsound(shapeNum) { //add change for amoeba }</pre> </div>	<p>Ram just need to create one more class called "Amoeba":</p> <div style="border: 1px solid black; padding: 10px; background-color: #f0f0f0;"> <p>Amoeba</p> <pre>rotate() { // code to rotate an amoeba } playSound() { // code to play the new // .hif file for an amoeba }</pre> </div>

So what do you like about OO?

Ans:

- Design software as per real world usage.
- New changes can be incorporated 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

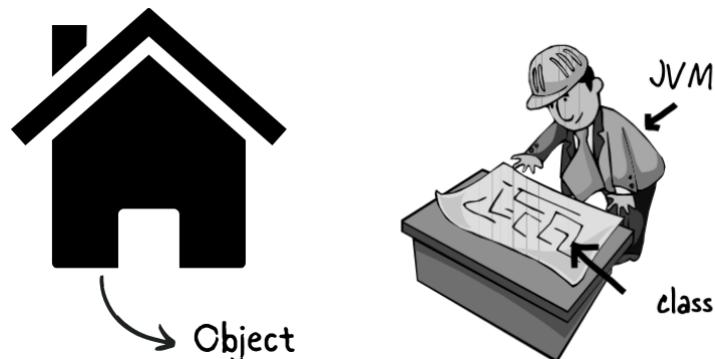
2.2 Introductions to OOPs

Object-Oriented Programming (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.

Instance variable

brick
sand
paint
steel
door
fan

Methods

Build wall
Remove door
Add door
Remove wall
Build floor
Destroy floor

- 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!");
    }
}

class DogTestDrive {
    public static void main (String[] args) {
        Dog d = new Dog(); ← make a Dog object
        d.size = 40; ← use the dot operator (.)
        d.bark(); ← to set the size of the Dog
                    and to call its bark() method
    }
}
```

instance variables

a method

dot operator

2.2.1 Java source file

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

Lava.java

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

Command:

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

- **Case II:** Program name should be same as public class name.

Lava.java

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

B.java

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

- **Case III:** There can be only 1 public class in single program.

B.java

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

- **Class IV:** Multiple class with main() means 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");  
    }  
}
```

Command:

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

```
$ java A  
A class main
```

```
$ java B  
B class main
```

```
$ java Lava  
RuntimeError: NoClassDefFoundError: Lava
```

Conclusion:

- For every class in program, a separate “**.class**” is generated.
- ".java" file is compiled to generate ".class" file. ".class" file is executed.
- Executing a ".class" file executes it's main().
- Executing a ".class" file without main() results in runtime exception.
- Multiple classes in a single ".java" file is not recommended.

2.2.2 main() method

- **main()** is entry point for a Java program.
- JVM looks for main() in the class & executes it.
- JVM searches for main() with below prototype:

Syntax:

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

- **public:** Modifier to call main() from anywhere
- **static:** without any object, JVM can call this method
- **void:** Return nothing to JVM
- **main:** Method name JVM calls by default
- **String[] args:** command line argument

Note:

main() syntax is strict & any change in this will result in runtime error: "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 identifier can change:
Eg: **public static void main(String temp[])**
 - 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
 - * strictfp
- Eg:

New.java

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

- Multiple main() methods i.e 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"); // Output: Starting
    }
    public static void main(int[] args){
        System.out.println("Sample 2");
    }
}
```

- **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
```

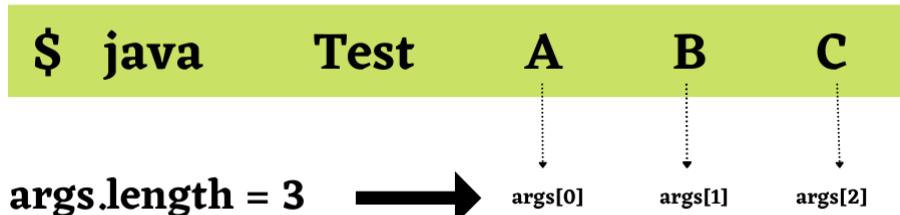
Note:

- 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"
```




Small steps every day.....

3. Data types in Java

3.1 Getting started with data type

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:

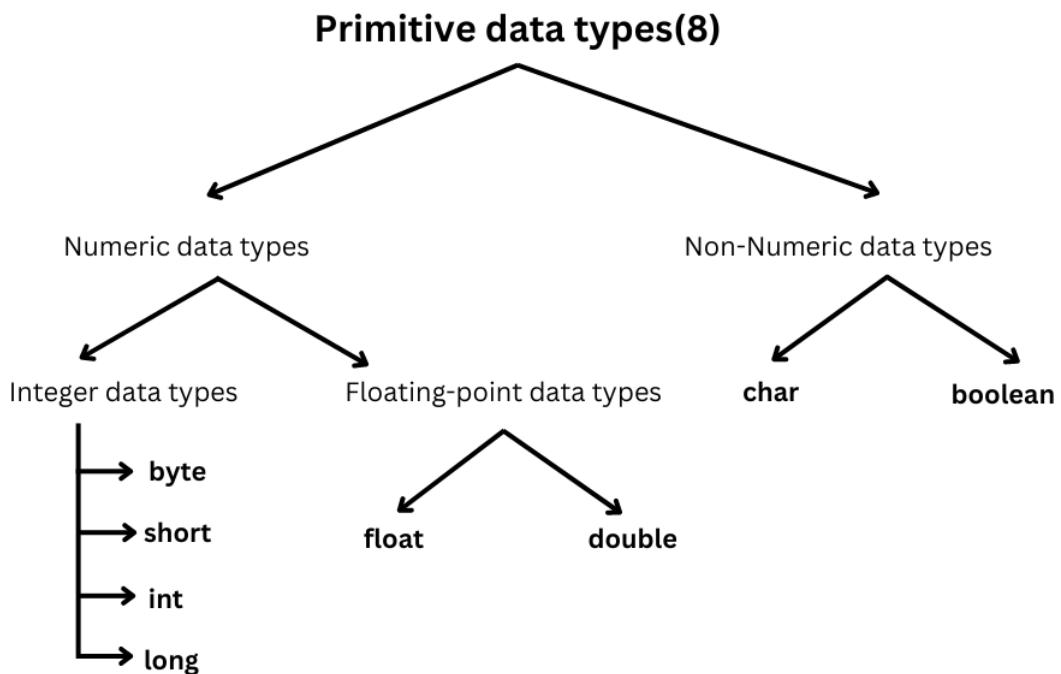
- **Primitive data types:** Includes boolean, char, byte, short, int, long, float and double.

- **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 data type

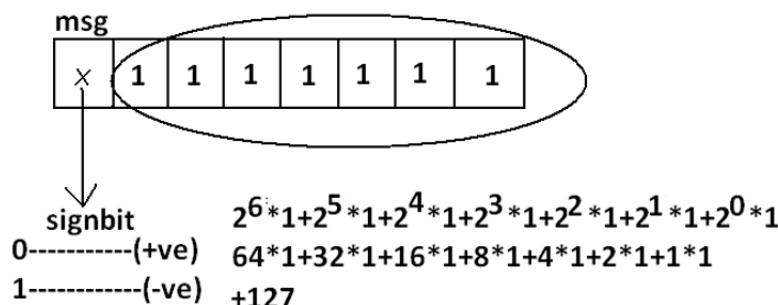


There are 4 types of integer in Java: byte, short, int & long

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 **most significant bit**) 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	Invalid short values
<code>short s = 32767; short s = -32767;</code>	<code>short s=10.5; short s=true;</code>

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	Invalid int values
<code>int x = 2147483647; int x = -90;</code>	<code>int b=10.5; int b=true; int b="lava";</code>

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.

$$\text{long } l = 1,26,000 \times 60 \times 60 \times 24 \times 1000 \text{ miles.}$$
- 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:

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

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

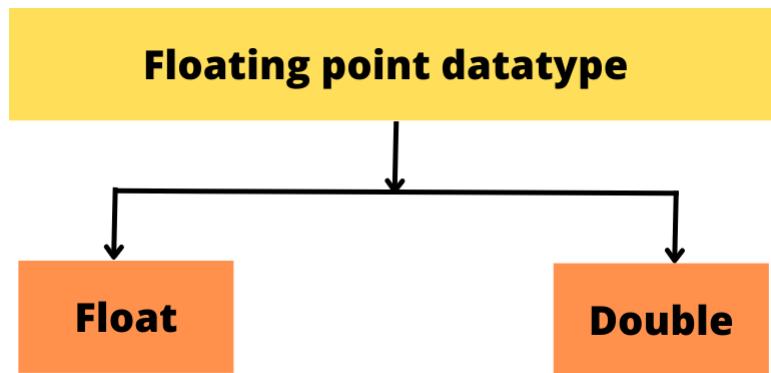
- Hexadecimal to decimal form:

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

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

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 X
```

- 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; X  
float b = 0X56.9; X  
double c = 0X56.9; X
```

- 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; ✓
```

- **Exponential 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; ✓
```

- **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

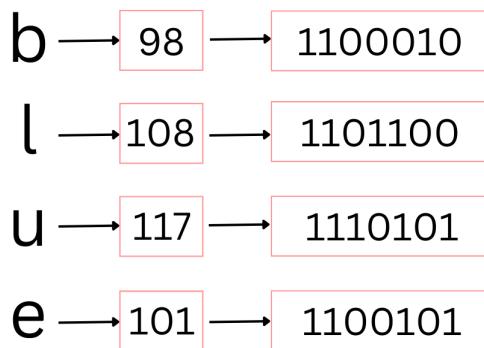
- In order to understand char data type, we need to understand what is "ASCII" and "UNICODE"

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'; ✓
```

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");
}
```

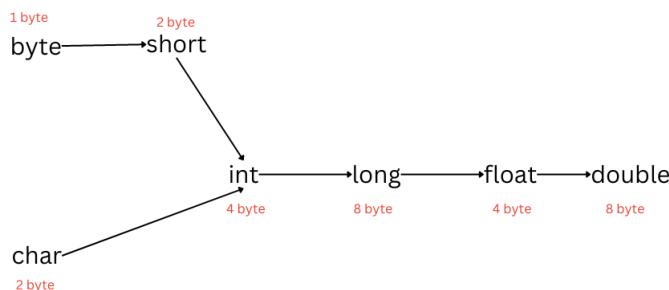
```
while(1)
{
    System.out.println("Hello");
}
```

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 implicit 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

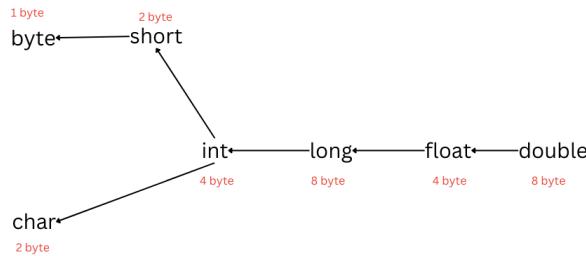
double d = 10;
System.out.println(d); // output: 10.0
  
```

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 explanation:

- 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
```




Small steps every day.....

4. String, StringBuffer, StringBuilder

4.1 String

- A string is a **sequence of characters**.
- It is represented by Java Standard Library **java.lang.String** class.
- Strings in Java are immutable, meaning their values cannot be changed once they are created.

4.1.1 Mutable and Immutable objects

Immutable Object

- An immutable object is an object whose **data** cannot be changed after it is created.
- Once an immutable object is created, its internal state remains constant throughout its lifetime.
- Eg: **String class**
- If you want to modify a String, a new String object is created with the modified value.

Code:

```
String str1 = "Hello";
String str2 = str1; // str2 refers to the same "Hello"
string object as str1
```

str1 = str1 + ", World!"; // A new string object
is created with the concatenated value

```
System.out.println(str1); // Output: "Hello, World!"
System.out.println(str2); // Output: "Hello"
```

- Benefits of immutability: **thread safety, caching, and security.**

Mmutable Object

- A mutable object is an object whose **data** can be modified after it is created.
- Eg: **StringBuffer class**

Code:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(", World!"); // Modifying the internal state of
the StringBuilder
```

```
System.out.println(sb.toString()); // Output: "Hello,
World!"
```

4.1.2 String constructors

- Empty string object:

Syntax:

```
String s = new String();
```

Code:

```
String s1 = new String();
System.out.println(s1);
```

- String object using string literal:

Syntax:

```
String s = new String(String literal);
```

Code:

```
String s1 = new String("lavatech technology");
System.out.println(s1);
```

- String object using StringBuffer object:

Syntax:

```
String s = new String(StringBufer sb);
```

Code:

```
StringBuffer sb = new StringBuffer("Hello, world");
String s1 = new String(sb);
System.out.println(s1);
```

- String object using StringBuilder object:

Syntax:

```
String s = new String(StringBuilder sb);
```

Code:

```
StringBuilder sb = new StringBuilder("Hello, world");
String s1 = new String(sb);
System.out.println(s1);
```

- String object using character array:

Syntax:

```
String s = new String(char[] ch);
```

Code:

```
char[] ch = {'a','b','c','d','e'};
String s1 = new String(ch);
System.out.println(s1);
```

- String object using byte array:

Syntax:

```
String s = new String(byte[] b);
```

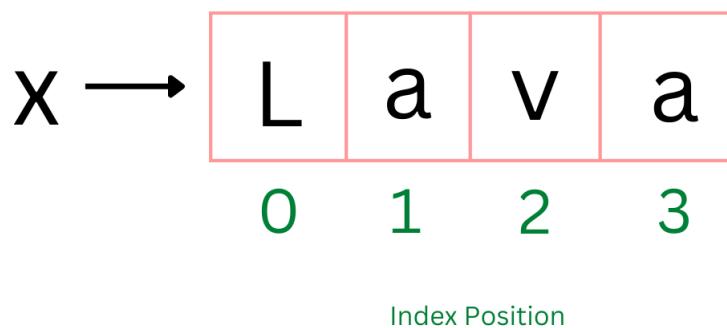
Code:

```
byte[] b = {97,98,99,100};
String s1 = new String(b);
System.out.println(s1);
```

4.1.3 String indexing

Indexing:

- Strings are sequences of characters.
- Each character in a string has an index starting from 0.

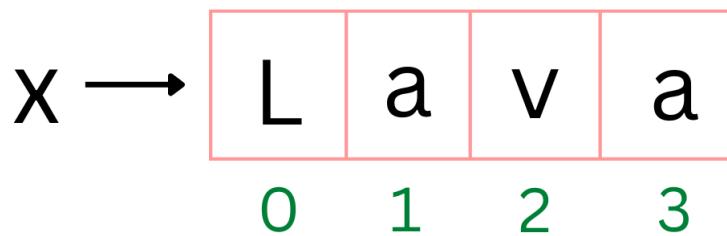


Slicing:

- You can perform string slicing or substring extraction using the index:

Syntax:

[startIndex,endIndex] => [startIndex,endIndex-1]



Slicing
[startIndex , endIndex] => [startIndex, endIndex-1]
Eg: [0, 3] => [0, 2] => Lav

4.1.4 String methods

Function	Syntax & Example
charAt()	<p>public char charAt(int index)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "hello world"; System.out.println(s1.charAt(3)); // Output: l</pre> </div>
concat()	<p>public String concat(String)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; s1 = s1.concat("tech"); System.out.println(s1); // Output: Lavatech s1 = s1 + " Techo"; System.out.println(s1); // Output: Lavatech Techo s1 += "logy"; System.out.println(s1); // Output: Lavatech Technology</pre> </div>
equals()	<p>public boolean equals(Object o)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.equals("lava")); // Output: false</pre> </div>

Function	Syntax & Example
toCharArray()	<p>public char[] toCharArray()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String result = "Lava"; char[] ch2 = result.toCharArray(); System.out.println(ch2); // Output: Lava</pre> </div>
isempty():	<p>public boolean isEmpty()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.isEmpty()); // false</pre> </div>
length()	<p>public int length()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.length()); // Output: 4</pre> </div>
replace()	<p>public String replace(char old, char new)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.replace('a', 'A')); // Output: LAvA</pre> </div>

Function	Syntax & Example
substring()	<p>public String substring(int begin)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.substring(1)); // Output: ava</pre> </div> <p>public String substring(int begin, int end)</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.substring(0,2)); // Output: La</pre> </div>
indexof()	<p>public int indexof(char ch);</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.indexOf("L")); // Output: 0 System.out.println(s1.indexOf("z")); // Output: -1</pre> </div> <p>It always will return first occurrence index only. Returns -1 if character not found.</p>
lastIndexof()	<p>public int lastIndexOf(char ch)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.lastIndexOf("L")); // Output: ava</pre> </div>

Function	Syntax & Example
toUpperCase()	<p>public String toUpperCase()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.toUpperCase()); //</pre> <p>Output: LAVA</p> </div>
toLowerCase()	<p>public String toLowerCase()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = "Lava"; System.out.println(s1.toLowerCase()); //</pre> <p>Output: lava</p> </div>
trim()	<p>public String trim()</p> <p>Used to remove space from beginning and end of a string</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>String s1 = " Lava "; System.out.println(s1.trim()); // Output: Lava</pre> </div>

4.2 StringBuffer

- The StringBuffer class is a **mutable** sequence of characters.
- It is part of the **java.lang** package.
- StringBuffer allows you to modify the contents of a string without creating a new object.
- length:**
 - Length is total characters already present.

Code:

```
StringBuffer s1 = new StringBuffer("Hello");
System.out.println(s1.length()); // Output: 5
```

- capacity:**
 - The capacity represents the maximum number of characters that the StringBuffer can hold without resizing.
 - Capacity of empty StringBuffer object:
 - Default capacity is 16 characters.
 - Once 16th character is also used, new capacity will be:

Syntax:

New capacity = (current capacity + 1) * 2

* Eg:

Code:

```
StringBuffer s1 = new StringBuffer();
System.out.println(s1.capacity()); // Output: 16
```

```
s1.append("hello");
```

```
System.out.println(s1.capacity()); // Output: 16
```

```
s1.append("world by Java and testing");
```

```
System.out.println(s1.capacity()); // Output: 34
```

- Capacity of non-empty StringBuffer object:

Syntax:

capacity = current length of characters + 16

- Eg:

Code:

```
StringBuffer s2 = new StringBuffer("Lavatech Technology");
System.out.println(s2.capacity()); // Output: 35
```

4.2.1 StringBuffer constructors

- Empty StringBuffer object:

Syntax:

```
StringBuffer sb = new StringBuffer();
```

Code:

```
StringBuffer s1 = new StringBuffer();
System.out.println(s1);
```

- StringBuffer object using string literal:

Syntax:

```
StringBuffer sb = new StringBuffer(String s);
```

Code:

```
String s1 = "hello world";
StringBuffer s2 = new StringBuffer(s1);
System.out.println(s2);
```

- StringBuffer object using initial capacity:

Syntax:

```
String s = new String(integer initCapacity);
```

Code:

```
StringBuffer sb = new StringBuffer(5);
System.out.println(sb.capacity());
```

4.2.2 StringBuffer methods

Function	Syntax & Example
length()	<p>public int length();</p> <p>Eg:</p> <p>Code:</p> <pre>StringBuffer sb = new StringBuffer("hello world"); System.out.println(sb.length()); // Output: 11</pre>
capacity()	<p>public int capacity();</p> <p>Eg:</p> <p>Code:</p> <pre>StringBuffer sb = new StringBuffer("hello world"); System.out.println(sb.capacity()); // Output: 27</pre>

Function	Syntax & Example
append()	<pre>public StringBuffer append(String s) public StringBuffer append(byte b) public StringBuffer append(int i) public StringBuffer append(long i) public StringBuffer append(float f) public StringBuffer append(double d) public StringBuffer append(short s) public StringBuffer append(char ch) public StringBuffer append(boolean b)</pre> <p>Eg:</p> <div style="background-color: #ccc; padding: 5px; border-radius: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("hello world"); sb.append("again"); System.out.println(sb); // Output: hello world again sb.append(10); System.out.println(sb); // Output: hello world again10 sb.append(2.0f); System.out.println(sb); // Output: hello world again102.0 sb.append('*'); System.out.println(sb); // Output: hello world again102.0* sb.append(true); System.out.println(sb); // Output: hello world again</pre> </div>

Function	Syntax & Example
insert()	<p>public StringBuffer insert(int index, String s) public StringBuffer insert(int index, byte b) public StringBuffer insert(int index, int i) public StringBuffer insert(int index, long i) public StringBuffer insert(int index, float f) public StringBuffer insert(int index, double d) public StringBuffer insert(int index, short s) public StringBuffer insert(int index, char ch) public StringBuffer insert(int index, boolean b)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("Hello"); sb.insert(5, " world!"); System.out.println(sb); // Output: Hello world! sb.insert(0, 1); System.out.println(sb); // Output: 1Hello world! sb.insert(0,1.0f); System.out.println(sb); // Output: 1.01Hello world! sb.insert(0,'A'); System.out.println(sb); // Output: A1.01Hello world! sb.insert(0,true); System.out.println(sb); // Output: trueA1.01Hello world!</pre> </div>

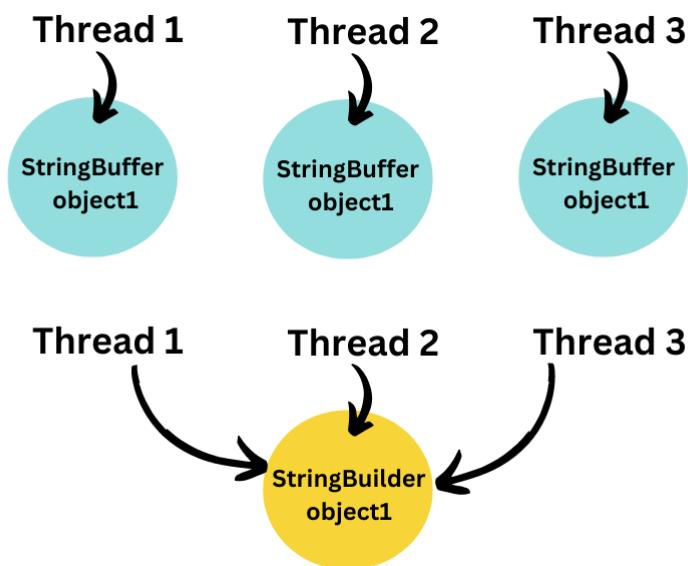
Function	Syntax & Example
delete()	<p>public StringBuffer delete(int begin, int end) This would delete characters from begin to end-1. Eg:</p> <div data-bbox="568 489 684 527" style="background-color: #333; color: white; padding: 2px;">Code:</div> <pre data-bbox="579 543 1270 759">StringBuffer sb = new StringBuffer("Hello world!"); sb.delete(0,6); System.out.println(sb); // Output: world!</pre>
deleteCharAt()	<p>public StringBuffer deleteCharAt(int index)</p> <p>Eg:</p> <div data-bbox="568 961 684 999" style="background-color: #333; color: white; padding: 2px;">Code:</div> <pre data-bbox="579 1015 1270 1291">StringBuffer sb = new StringBuffer("Hello world!"); sb.deleteCharAt(5); System.out.println(sb); // Output: Hel- loworld!</pre>
reverse()	<p>public StringBuffer reverse()</p> <p>Eg:</p> <div data-bbox="568 1500 684 1538" style="background-color: #333; color: white; padding: 2px;">Code:</div> <pre data-bbox="579 1554 1270 1718">StringBuffer sb = new StringBuffer("Hello world!"); System.out.println(sb.reverse()); // Output: !dlrow olleH</pre>

Function	Syntax & Example
setLength()	<p>public void setLength(int length)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("Hello world!"); sb.setLength(5); System.out.println(sb); // Output: Hello</pre> </div>
ensureCapacity()	<p>public void ensureCapacity(int length)</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("Hello world!"); sb.ensureCapacity(50); System.out.println(sb.capacity()); // Output: 58</pre> </div>
trimToSize()	<p>Extra allocated free memory, can you please deallocate</p> <p>public void trimToSize()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("Hello world!"); sb.ensureCapacity(50); sb.trimToSize(); System.out.println(sb.capacity()); // Output: 12</pre> </div>

Function	Syntax & Example
toString()	<p>public String toString()</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px; border-radius: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("Hello world!"); String test = sb.toString(); System.out.println(test); // Output: Hello world!</pre> </div>
setCharAt()	<p>public char setAt(int integer);</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px; border-radius: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("hello world"); sb.setCharAt(0,'H'); System.out.println(sb); // Output: Hello world</pre> </div>
charAt()	<p>public char charAt(int integer);</p> <p>Eg:</p> <div style="background-color: #f0f0f0; padding: 5px; border-radius: 5px;"> Code: <pre>StringBuffer sb = new StringBuffer("hello world"); System.out.println(sb.charAt(3)); // Output: l</pre> </div>

4.3 StringBuilder

- The StringBuilder class is a **mutable** sequence of characters, similar to the StringBuffer class.
- It is part of the `java.lang` package
- Unlike StringBuffer, StringBuilder is **not thread-safe**, meaning it is not designed to be used in multi-threaded environments.
- StringBuilder is more efficient than StringBuffer in single-threaded scenarios due to its lack of synchronization.
- If you don't need thread safety, it's recommended to use StringBuilder over StringBuffer.
- All methods of StringBuffer object are applicable on StringBuilder objects as well.



4.3.1 StringBuffer V/S StringBuilder

StringBuffer	StringBuilder
Every method present in StringBuffer is synchronized.	No method present in StringBuilder is synchronized.
At a time, only one thread is allowed to operate on StringBuffer object and hence it is thread safe.	At a time, multiple threads are allowed to operate on StringBuilder object and hence it is not thread safe.
Threads are required to wait to operate on StringBuffer object and hence relatively performance is slow.	Threads are not required to wait to operate on StringBuilder object and hence relatively performance is high.
Introduced in 1.0 version	Introduced in 1.5 version.

4.4 String method chaining

- String method chaining refers to the technique of chaining multiple method calls on a String object in a single line of code.
- This technique is possible because most methods in the String class return a new String object as result of the operation.
- Eg:

Code:

```
String text = " Hello, World! ";
String result = text.trim().toUpperCase().replace("WORLD",
    "Java").substring(7);
System.out.println(result); // Output: JAVA!
```

- This makes your code **readable and concise**.
- Method chaining is possible for String, StringBuffer and StringBuilder object methods.

4.5 == V/S equals()

==

- For String & StringBuffer: Returns true when both object point to the same object.

Code:

```
String s1 = "Lava";
String s2 = "Lava";
System.out.println(s1 == s2); // Output: true
```

```
String s3 = new String("Lava");
String s4 = new String("Lava");
System.out.println(s3 == s4); // Output: false
```

```
StringBuffer s5 = new StringBuffer("Lava");
StringBuffer s6 = new StringBuffer("Lava");
StringBuffer s7 = s5;
System.out.println(s5 == s6); // Output: false
System.out.println(s5 == s7); // Output: true
```

equals()

- For **String** object: Returns if content of both the object are same.

Code:

```
String s1 = "Lava";
String s2 = "Lava";
System.out.println(s1.equals(s2)); // Output: true
```

```
String s3 = new String("Lava");
String s4 = new String("Lava");
System.out.println(s3.equals(s4)); // Output: true
```

- For **StringBuffer** object: Return true when both object point to the same object.

Code:

```
StringBuffer s1 = new StringBuffer("Lava");
StringBuffer s2 = new StringBuffer("Lava");
System.out.println(s1.equals(s2)); // Output: false
```

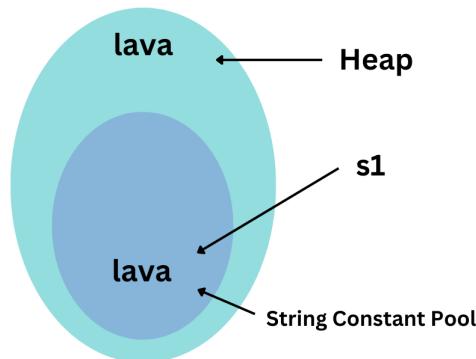
4.6 String memory management

Here are some key points regarding string memory management:

- **String Constant Pool:** The string object created using "=" operator, is created in string constant pool.
- Eg:

Code:

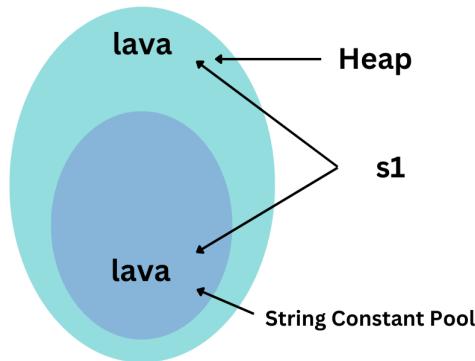
```
String s1 = "lava";
```



- **Heap:** The string object created using "new" operator, is created in heap.
- Eg:

Code:

```
String s1 = new String("lava");
```



Let's see each of these in detail.

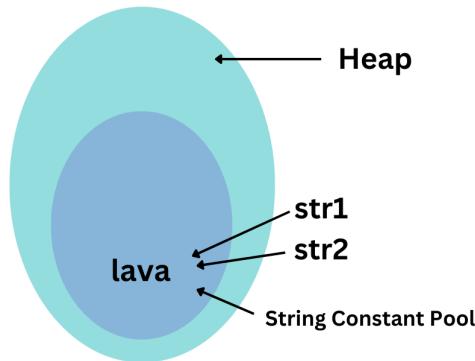
4.6.1 String Constant Pool(SCP)

- The string constant pool (SCP) (or string pool), is a **special area of memory in the Java heap** that stores string objects.
- Using SCP, Java compiler checks if a new string is already exists in the pool.
- If not, a new string object is created and added to SCP.
- If it does, the reference to the existing string is returned.
- Eg:

Code:

```
String str1 = "lava"; // Added to the string constant pool
String str2 = "lava"; // Reuses the existing string from
                     the pool
```

```
System.out.println(str1 == str2); // Output: true (both
                                 strings refer to the same object)
```



- Use intern() method to explicitly add string to the string constant pool:
- Eg:

Code:

```
String str3 = new String("Hello").intern(); // Explicitly  
interns the string  
String str4 = "Hello";
```

```
System.out.println(str3 == str4); // Output: true
```

In this case, str3.intern() adds the string to the string constant pool, allowing str3 and str4 to refer to the same string object.

Note:

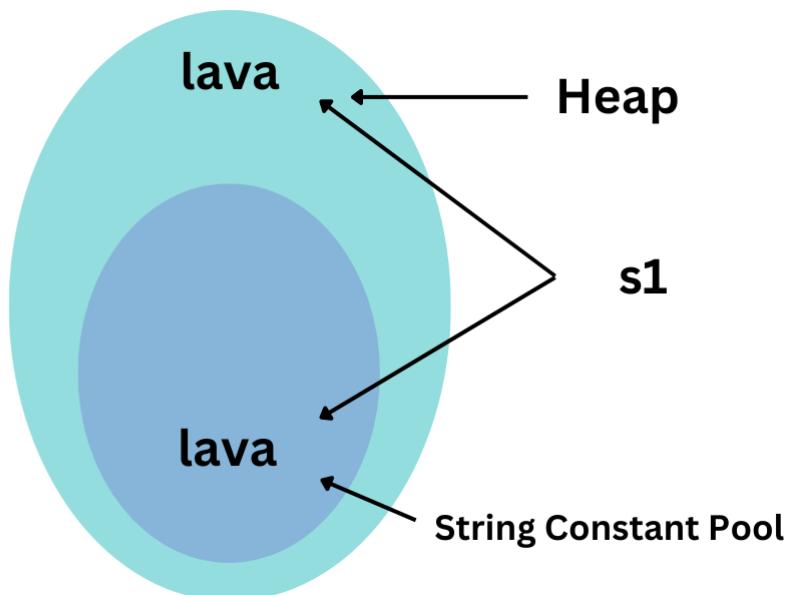
Upto Java1.6, scp is in method area. After java1.7, scp is present in heap itself.

4.6.2 Heap

- When you create a string using the **new** keyword, it is stored in the heap.
- Eg:

Code:

```
String s1 = new String("lava");
String s2 = new String("lava");
String s3 = "lava";
String s4 = "lava";
```



If not now, when?

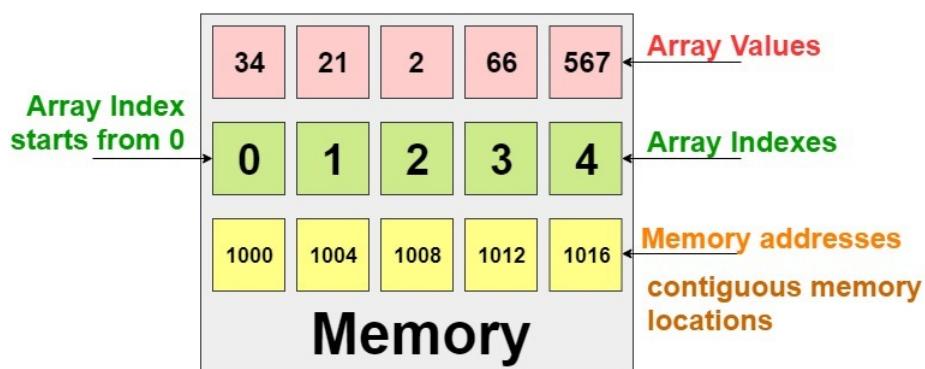
5. Arrays

5.1 Arrays in detail

5.1.1 Array introduction

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

```
int x[ ] = new int[ ] {34, 21, 2, 66, 567};
```



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.

5.1.2 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; X
```

- Two-dimensional array declaration:

Syntax:

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

```
int [][]x;
```

```
int x[][];
```

- **3-dimensional array declaration:**

Syntax:

```
int[][][] x; (Recommended)  
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; X
int[]    []a,[]b,[]c; X
```

5.1.3 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];
```

Memory



a

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.

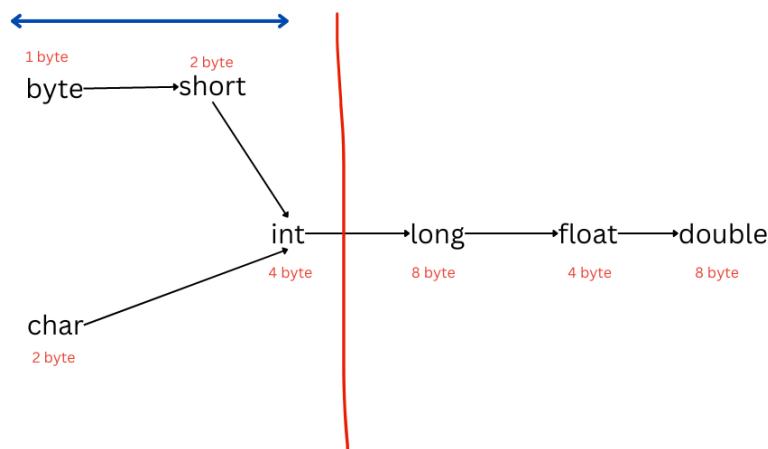
However, Java Virtual Machine will throw runtime error:

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[10]; X
int[] x = new int[3.5]; X
```

- Maximum size of array can be 2147483647:

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

- 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

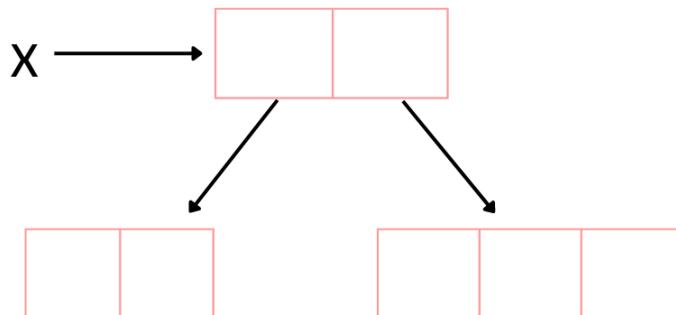
- **Two-dimensional array creation:**

- In Java, two dimensional array is not implemented 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.

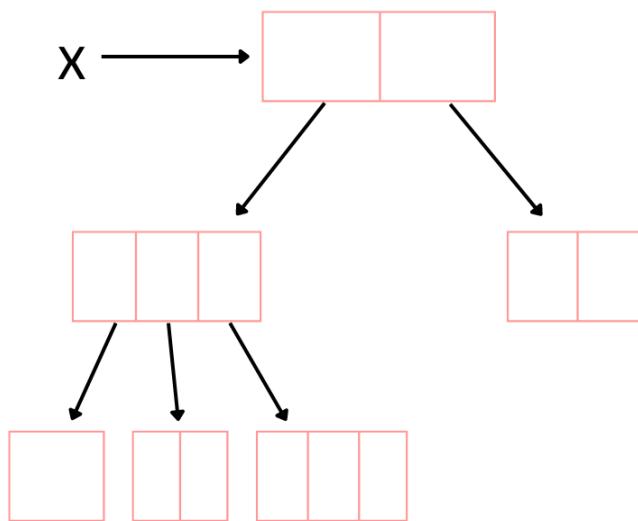
- **Base size:** In this we specify the size of first dimension at the time of array creation.

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



- Three-dimensional array creation:

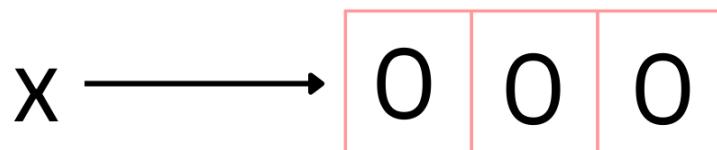
```
int[][][] x = new int[2][][];
x[0] = new int[3][];
x[0][0] = new int[1];
x[0][1] = new int[2];
x[0][2] = new int[3];
x[1] = new int[2][2];
```



5.1.4 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:

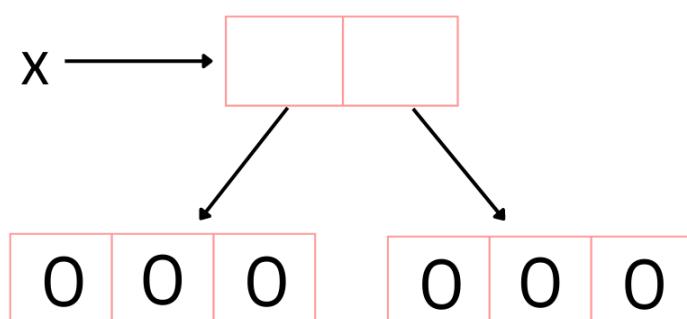
```
[I@422a8473
0
```

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:

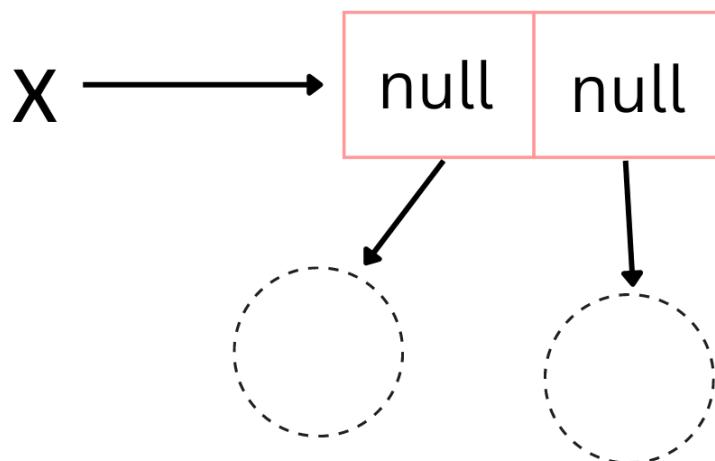


```
int[][] a = new int[2][3];
System.out.println(a);
System.out.println(a[0]);
System.out.println(a[0][0]);
```

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]);
```

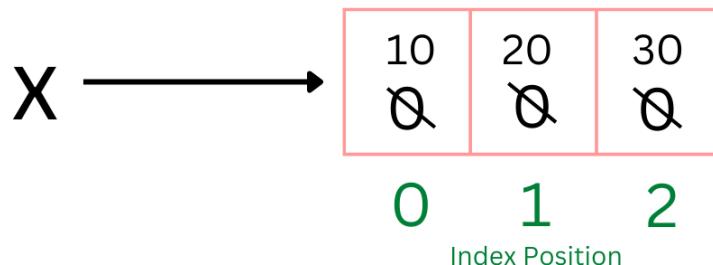
Output:

```
[[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:

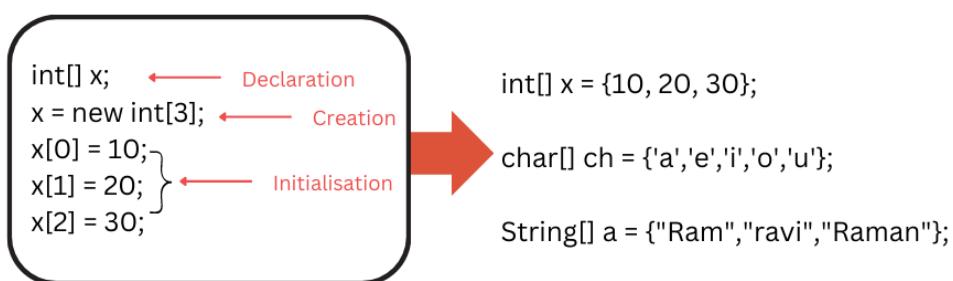
```
10
20
30
```

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

5.1.5 Array declaration, creation and initialisation in one line

- **One dimensional array:**

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



Code:

```

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

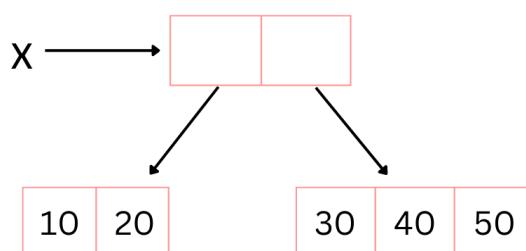
You can declare the array & provide it's value as shown below:

Code:

```

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

- **Multi-dimensional array:**



Code:

```

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

5.1.6 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
```

5.1.7 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 its size:

Code:

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

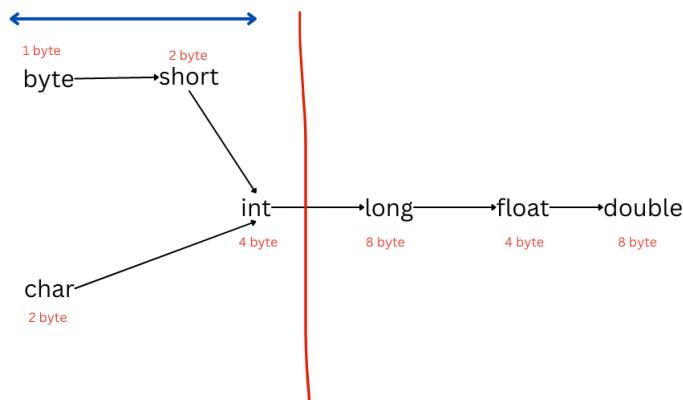
- In below example, main() is calling sum() using an anonymous arrays:

Test.java

```
class Test {  
    public static void main(String[] args) {  
        sum(new int[]{1, 2, 3});  
    }  
    public static void sum(int[] x) {  
        int total = 0;  
        for(int x1: x)  
            total += x1;  
        System.out.println(total); // 6  
    } }
```

5.1.8 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

5.1.9 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.

6. Operators

6.1 Operators in Java

6.1.1 Arithematic Operator

Operator	Example
Addition(+)	<pre>int a = 5; int b = 10; int c = a + b; // c will be 15</pre>
Subtraction(-)	<pre>int a = 10; int b = 5; int c = a - b; // c will be 5</pre>

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); X
```

- 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); X
```

- 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
```

6.1.2 String Concatenation Operator

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

6.1.3 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); X
```

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

Code:

```
int x=10;
int y = ++(++x); X
```

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

Code:

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

6.1.4 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

Let's see these in detail.

Comparison operators: > , >= , < , <=

- Comparison operator is **applicable 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); X
```

- Comparison operator is **not applicable on object types**:

Code:

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

- Chaining of comparison operators is not allowed.

Code:

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

Equality operators: == , !=

- Equality operators are **applicable on all primitive type**.

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 are **applicable for object types**. Returns true, if both object reference pointing to the same object.

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
```

If child & parent are not of same type, 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); X
```

6.1.5 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	Explanation				
<pre>int a=4; int b=5; System.out.println(a & b)</pre>	4	<table style="margin-left: auto; margin-right: auto;"> <tr><td>100</td></tr> <tr><td>101</td></tr> <tr><td>—</td></tr> <tr><td>100</td></tr> </table>	100	101	—	100
100						
101						
—						
100						

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

Sample Code	Output	Explanation				
<pre>int a=4; int b=5; System.out.println(a b)</pre>	5	<table style="margin-left: auto; margin-right: auto;"> <tr><td>100</td></tr> <tr><td>101</td></tr> <tr><td>—</td></tr> <tr><td>101</td></tr> </table>	100	101	—	101
100						
101						
—						
101						

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

Sample Code	Output	Explanation
<pre>int a=4; int b=5; System.out.println(a ^ b)</pre>	1	$ \begin{array}{r} 100 \\ 101 \\ \hline 001 \end{array} $

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

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

Bitwise » - Bitwise right shift.

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

Sample Code	Output	Explanation
<pre>int a=10; int x=2; System.out.println(a >> x)</pre>	2	$ \begin{array}{l} 000...1010 \gg 2 \\ 000...0010 \end{array} $

Bitwise << - Bitwise left shift.

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

Sample Code	Output	Explanation
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.

6.1.6 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.

6.1.7 Short circuit Operator

In Java, the short-circuit operators are:

- **&& (logical AND)**: If LHS evaluates to false, RHS is not evaluated at all. Note that && requires both condition to be true.
- **|| (logical OR)**: If the LHS evaluates to true, the RHS is not evaluated at all. Note that || requires atleast one condition to be true.

Eg:

Code:

```
String user="Ram";
String password="ram@123";
System.out.println(user=="Ram" && password=="ram@123"); // output: true
System.out.println(user=="Ram" || password=="ram@123"); // output: true
```

Difference between Bitwise "& |" and Logical "&& ||"

& ,	&& ,
Both LHS & RHS are evaluated	RHS evaluation is optional
Performance is low	Performance is high
Applicable on boolean & integral types	Applicable only for boolean.

6.1.8 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; X  
  
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
```

6.1.9 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");
```

6.1.10 new Operator

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

Syntax:

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

Eg:

Code:

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

6.1.11 [] Operator

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

Code:

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

6.1.12 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 explanation:

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



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

7. Flow Control

7.1 Flow control statement

In this section, you are going to learn:

Selection Statements

- if..else
- switch()

Iterative Statements

- while()
- do-while()
- for()
- for-each loop (Java 1.5)

Transfer Statements

- 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.

7.2.1 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) { X// 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");
```

7.2.2 switch case

- If several options are 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 version **byte, 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(); X // Compile-time error!
}
```

- Case argument should be compile-time constant or declared as final.

Code:

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

- 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: X      // Compille-time error!
        System.out.println(1000);
        break;
}
```

- Duplicate case labels are not allowed.

Code:

```
int x = 10;
switch(x) {
    case 97:
        System.out.println(97);
        break;
    case 'a': x // 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

Default case:

- Default case will be executed if and only if, there is no case matched.
- 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);
        break;
```

Output:

```
default  
0
```

7.3 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

7.3.1 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) { X  
    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
```

7.3.2 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 invalid 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  
    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"); X// Unreachable statement error
```

- Final variable will be evaluated compile-time only. Thus for non-reachable condition, it will result in compile-time error.

Code:

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

7.3.3 for()

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

```

❶           ❷   ❸   ❹           ❺   ❻
for(initialisation_section; conditional_check; increment/decrement)
{
    loop_body ❷ ❸ ❹
}

```

- Curly braces are optional, without curly braces only one statement is allowed, which should not be declarative statement.
- All 3 parts of for loop are independent of each other and optional.
- 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.
- It declares and initialises local variables.
- Any valid Java statement is allowed in this section.

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++) {} ✗
```

- Conditional section:

- * It contains expression of the type boolean.
- * If nothing is added here, the compiler will always place true.
- * Eg:

Code:

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

- Increment/decrement section:

- * It contains any valid Java statement, mostly increment and decrement operation.
- * Eg:

Code:

```
for(int i=0; i < 3; i++)
    System.out.print("Hi ");
```

Output:

Hi Hi Hi

• Infinite loop examples:**Code:**

```
for(;;) ✓
    System.out.println("Hello");
or
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
```

7.3.4 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 1: Print elements of 1-dimensional array -

Normal for loop	Enhanced for loop
<p>Code:</p> <pre>int[] x = {10,20,30}; for(int i=0;i<x.length;i++) System.out.println(x[i]);</pre>	<p>Code:</p> <pre>int[] x = {10,20,30}; for(int x1: x) System.out.println(x1);</pre>

- Eg 2: Print elements of 2-dimensional array -

Normal for loop	Enhanced for loop
<p>Code:</p> <pre>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(x[i][j]);</pre>	<p>Code:</p> <pre>int[][] x={{ {10,20},{40,50}}; for(int[] x1: x) for(int x2: x1) System.out.println(x1[x2]);</pre>

- Eg 3: 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);
```

- Eg 4: Print String characters using for-each -

Normal for loop	Enhanced for loop
<p>Code:</p> <pre>String n = "lavatech"; for(int i = 0; i < n.length(); i++) System.out.print(n.charAt(i));</pre>	<p>Code:</p> <pre>String s = "lavatech"; char[] ch = s.toCharArray(); for(char c: ch) System.out.print(c);</pre>

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.

7.4 Transfer Statements

Transfer statements are used to alter the flow of program according to some conditions. Below are transfer statement:

- break
- continue

7.4.1 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);
        break;
    case 1:
        System.out.print(1);
        break;
    case 2:
        System.out.print(2);
        break; }
```

- 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:

Code:

```
int x = 10;  
l1: {  
    System.out.print("begin");  
    if(x==10)  
        break l1;  
    System.out.print("end");  
}  
System.out.print("Hello");
```

7.4.2 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.print(i); // Output: 13579  
}
```

7.4.3 Labeled break & continue

- Use labeled break/continue to break/continue a particular loop.

Syntax:

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

Eg 1:

Code:

```
String str = "elephant";  
char[] ch = str.toCharArray();  
l1:  
for(char c : ch) {  
    if(c == 'p')  
        break l1;  
    System.out.print(c); // Output: ele  
}
```

Eg 2:

Code:

```
String str = "elephant";  
char[] ch = str.toCharArray();  
l1:  
for(char c : ch) {  
    if(c == 'p')  
        continue l1;  
    System.out.print(c); // Output: elehant  
}
```


Don't fear failure.



Not failure, but low aim is the crime.

8. Wrapper classes

8.1 Introduction

- Wrapper class allows you to convert primitive data types (such as int, char, boolean, etc.) into objects.
- Used when you need to treat primitive types as objects.
- Wrapper classes for each primitive type:
 - **Integer** for int
 - **Long** for long
 - **Float** for float
 - **Double** for double
 - **Byte** for byte
 - **Short** for short
 - **Character** for char
 - **Boolean** for boolean

8.2 Wrapper class constructors

- Wrapper class constructor contain argument as shown below:

Wrapper class	Constructor arguments
Byte	byte, String
Short	short, String
Integer	int, String
Long	long, String
Float	float,double, String
Double	double, String
Boolean	boolean, String
Character	char

Note:

Since Java 1.9 version, above constructors are deprecated.

Instead use valueOf() for better performance.

- Things to note for Boolean constructor:

- If you are passing boolean primitive as argument the only allowed values are: **true** or **false**
- If you are passing String argument then case is not important and content is also important.

Code:

```
Boolean b1 = new Boolean(true); // true
Boolean b2 = new Boolean(false); // false
//Boolean b3 = new Boolean(True); // invalid
Boolean b4 = new Boolean("lavatech"); // false
Boolean b5 = new Boolean("yes"); // false
Boolean b6 = new Boolean("no"); // false
Boolean b7 = new Boolean("true"); // true
Boolean b8 = new Boolean("True"); //
true
```

- Eg: Converting byte to Byte object:

Code:

```
byte b1 = 23;  
Byte b2 = new Byte(b1);  
Byte b3 = new Byte("67");  
System.out.println(b2); // 23  
System.out.println(b3); // 67
```

- Eg: Converting int to Integer object:

Code:

```
int i1 = 451;  
Integer I1 = new Integer(i1);  
Integer I2 = new Integer("451");  
System.out.println(I1); // 451  
System.out.println(I2); // 451
```

8.3 Integer wrapper class methods

Integer wrapper class contains important methods like:

- parseInt()
- toBinaryString()
- toOctalString()
- toHexString()

Let's see each of these in detail.

- **Integer.parseInt(String s)** - Parses the string argument as a signed decimal integer.

Code:

```
String s1 = "101101";
int i1 = Integer.parseInt(s1,2);
System.out.println(i1); // 45

String s2 = "34";
int i2 = Integer.parseInt(s2,8);
System.out.println(i2); // 28

String s3 = "aaff11";
int i3 = Integer.parseInt(s3,16);
System.out.println(i3); // 11206417

String s4 = "45";
int i4 = Integer.parseInt(s4,10);
System.out.println(i4); // 45
```

- **Integer.toBinaryString(int i)** - Converts a decimal, octal or hexadecimal number to its binary (base 2) representation.

Code:

```
int i5 = 45;
String s5 = Integer.toBinaryString(i5);
System.out.println(s5); // 101101
```

```
int i6 = 0xaa;
String s6 = Integer.toBinaryString(i6);
System.out.println(s6); // 10101010
```

```
int i7 = 034;
String s7 = Integer.toBinaryString(i7);
System.out.println(s7); // 11100
```

- **Integer.toOctalString(int i)** - Converts a decimal, binary or hexadecimal number to its octal (base 8) representation.

Code:

```
int i8 = 0B10101;
String s8 = Integer.toOctalString(i8);
System.out.println(s8); // 25
```

```
int i9 = 0xaaff44;
String s9 = Integer.toOctalString(i9);
System.out.println(s9); // 52577504
```

```
int i10 = 34;
String s10 = Integer.toOctalString(i10);
System.out.println(s10); // 42
```

- **Integer.toHexString(int i)** - Converts a decimal, binary or octal number to its Hex (base 8) representation.

Code:

```
int i11 = 0B10101;  
String s11 = Integer.toHexString(i11);  
System.out.println(s11); // 15  
  
int i12 = 045;  
String s12 = Integer.toHexString(i12);  
System.out.println(s12); // 25  
  
int i13 = 3489423;  
String s13 = Integer.toHexString(i13);  
System.out.println(s13); // 353e8f
```

8.4 Utility methods

- **valueOf():**

- Create wrapper class object for the given Primitives and Strings.
- All wrapper classes contains the valueOf() method:
- Eg:

Code:

```
Integer I = Integer.valueOf(10);
Float F = Float.valueOf(10.5f);
Double D = Double.valueOf(10.5);
Boolean B = Boolean.valueOf(true);
Character C = Character.valueOf('a');
```

- Constructor V/S valueOf():

- * Using constructor, "==" operator check for object reference equality.
- * valueOf() creates only 1 object and references same object using == operator.

Code:

```
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
System.out.println(i1 == i2); // false
```

```
Integer i3 = Integer.valueOf(10);
Integer i4 = Integer.valueOf(10);
System.out.println(i3 == i4); // true
```

- Forms of valueOf():

- * Form 1: All wrapper classes except Character class contains static factory valueOf() method to create wrapper object for given string.
- * Eg:

Code:

```
Integer I = Integer.valueOf("10");
Float F = Float.valueOf("10.5f");
Double D = Double.valueOf("10.5");
Boolean B = Boolean.valueOf("true");
```

- * Form 2: Only Integral Type wrapper classes(Byte, Short, Integer, Long)

Syntax:

```
public static wrapper valueOf(String s, int radix)
```

Allowed radix range is 2 to 36, where:

- 2 -> 0,1
- 8 -> 0 to 7
- 10 -> 0 to 9
- 11 -> 0 to 9, a
- 12 -> 0 to 9, a to b
- 16 -> 0 to 9, a to f
- 36 -> 0 to 9, a to z

- * Eg:

Code:

```
Integer I = Integer.valueOf("1110",2);
```

Predict the output for I in below code:

Code:

```
Integer I = Integer.valueOf("1111",15);
```

Ans: $(1111)_{10} = 15^3 + 15^2 + 15^1 + 15^0$

$(1111)_{10} = 3616$

Hence, I = 3616

- **xxxValue():**

- Used to find primitive value for a given wrapper object.

- Below are available methods:

- * byteValue()
 - * shortValue()
 - * intValue()
 - * longValue()
 - * floatValue()
 - * doubleValue()

- Eg:

demo.java

```
Integer I = new Integer(130);
System.out.println(I.byteValue()); // -126
System.out.println(I.shortValue()); // 130
System.out.println(I.intValue()); // 130
System.out.println(I.longValue()); // 130
System.out.println(I.floatValue()); // 130.0
System.out.println(I.doubleValue()); // 130.0
```

```
Character ch = new Character('a');
char c = ch.charValue(); // c = 'a'
```

Why is the byte output showing -126 for 130 integer value?

Ans: 130(Integer size 32 bits) == 00000...010000010

Byte 8-size – 10000010

As first digit is 1, the number is represented in 2's complement form.

2's complement = 1's complement + 1

i.e $01111101 + 1 = 11111110$

Hence $(11111110)_{10} = -126$

- **parseXxxx():**

- Used to find primitive value for given String object.
 - Available parseXxxxx() methods:
 - * **Integer.parseInt(String s)** - Parses the string argument as a signed decimal integer.
 - * **Long.parseLong(String s)** - Parses the string argument as a signed decimal long.
 - * **Float.parseFloat(String s)** - Parses the string argument as a floating-point number.
 - * **Double.parseDouble(String s)** - Parses the string argument as a double-precision floating-point number.
 - * **Boolean.parseBoolean(String s)** - Parses the string argument as a boolean.

- Eg:

Code:

```
int i = Integer.parseInt("-45");
long l = Long.parseLong("983452");
float f = Float.parseFloat("45.4");
double d = Double.parseDouble("23423.65");
boolean b = Boolean.parseBoolean("true");
System.out.println(i); // -45
System.out.println(l); // 983452
System.out.println(f); // 45.4
System.out.println(d); // 23423.65
System.out.println(b); // true
```

- **toString()** - Converts primitive datatype to String object.

Eg: Below code converts primitive datatype to String Object.

Code:

```
int i = 13;  
System.out.println(Integer.toString(i)); // 13
```

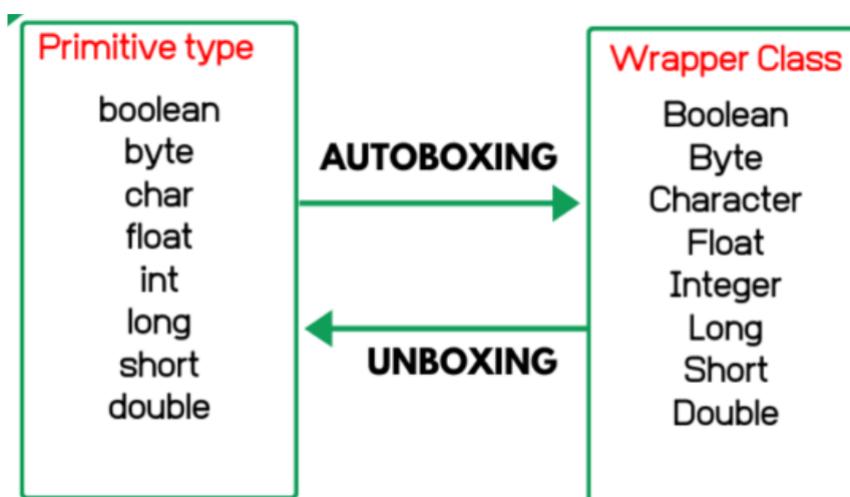
```
float f = 1.0f;  
System.out.println(Float.toString(f)); // 1.0
```

```
long l = 5634;  
System.out.println(Long.toString(l)); // 5634
```

```
double d = 783.3;  
System.out.println(Double.toString(d)); // 783.3
```

8.5 Autoboxing and AutoUnboxing

- Until Java 1.4 version, primitive values can't be provided in place of wrapper object & vice versa is not possible.
- Starting Java 1.5 version, all required conversions will be performed automatically by compiler.
- These automatic conversions are called **autoboxing and auto-unboxing**.



Autoboxing:

- At time of compilation, compiler will automatically use **valueOf()** to perform autoboxing.
- Eg:

Code:

```
int i = 45;
Integer I1 = i; // autoboxing
System.out.println(I1); // 45
```

AutoUnboxing:

- At time of compilation, compiler will internally use **xxxValue()** to perform autounboxing.
- Eg:

Code:

```
Integer I1 = new Integer(56);
int i = I1;
System.out.println(i); // 56
```


Don't fear failure.



Not failure, but low aim is the crime.

9. Packages in Java

9.1 import keyword

- **import** keyword is used to access classes & interfaces from **external packages or libraries**.
- Import keyword is not required for following packages as they are by default available to every Java program:
 - **java.lang**
 - **default package (current working directory)**
- There are 2 types of import:
 - **Explicit import**
 - **Implicit import**
- Let's see each of these in detail.

Explicit class import:

- Used to import a single class.

Syntax:

```
import packageName.ClassName;
```

- Eg:

Code:

```
import java.time.LocalDate;  
class demo {  
    public static void main(String[] args) {  
        LocalDate date = LocalDate.now();  
        System.out.println(date); // Ouptut: 2023-08-15  
    } }
```

Implicit class import:

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

Syntax:

```
import packageName.*;
```

- Eg:

Code:

```
import java.util.*;  
class New {  
    public static void main(String[] args) {  
        Date date = new Date();  
        System.out.println(date);  
        Calendar cal = Calendar.getInstance();  
        System.out.println(cal);  
    } }
```

- Drawback of implicit declaration: **Ambiguity problem.** Eg:

Code:

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

Note about import statement:

- **No subpackage import:** Importing a Java package, imports only classes and interfaces but not it's subpackage classes.
- **No effect on execution time: More import statements results in more compile-time. There is no effect on execution time (runtime).**
- Compiler resolves class names in the below order:
 - Explicit import
 - Classes present in current working directory (default package)
 - Implicit class import

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"

9.2 Packages

Package is group of Java ".class" files in a folder.

Package naming convention

- Universally accepted naming convention for packages is **internet domain name in reverse**.
- Eg: For domain name **www.lavatech.com**, package name is **com.lavatech.www**.
- Eg: Create program with package name **com.lavatech.www**:

Test.java

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

- Command to create package and place compiled ".class" file in it:

Syntax:

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

Eg:

Command:

```
# Create package in current location
$ javac -d . Test.java
```

```
# Create package under "D:" location
$ javac -d D: Test.java
```

Above command will create below directory structure:

- While executing, use package name and class name:

**Command:**

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

Important pointers:

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

Code:

```
package pack1; ✓
package pack2; ✗      # Result in compile-time error
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
```

Example of creating and importing package:

- Eg 1: Create "Test.class" in package **com.lavatech.www**.

Test.java

```
package com.lavatech.www;
public class Test {
    public int i=23;
}
```

Create program named **Apply.java** that would import
com.lavatech.www.Test:

Apply.java

```
import com.lavatech.www.Test;
public class Apply {
    public static void main(String[] args) {
        Test t1 = new Test();
        System.out.println(t1.i);
    }
}
```

Compile both **Test.java** and **Apply.java**:

Command:

```
$ javac -d . Test.java
$ javac Apply.java
```

Execute the **Apply.java** file:

Command:

```
$ java Apply
23
```

Example of Java code importing internally in package

- Eg2: Create below directory structure with 2 empty programs **App1.java** and **App2.java** as shown below:



- Create program **App1.java**:

App1.java

```
package com.lavatech.www;
public class App1 {
    public String name="Lavatech-App1";
}
```

- Create program **App2.java**:

App2.java

```
package com.lavatech.www;
public class App2 {
    public String name="Lavatech-App2";
    public static void main(String[] args) {
        App1 app1 = new App1();
        System.out.println(app1.name);
    }
}
```

- Compile the **App1.java & App2.java**

Command:

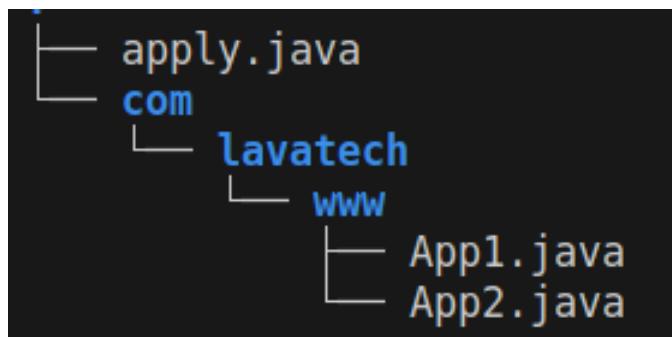
```
$ javac com/lavatech/www/App1.java
$ javac com/lavatech/www/App2.java
```

- Execute **App2.java**:

Command:

```
$ java com.lavatech.www.App2  
Lavatech-App1
```

- Create **apply.java** outside the **com/lavatech/www** package:



apply.java

```
import com.lavatech.www.App1;  
public class apply {  
    public static void main(String[] args) {  
        App1 app = new App1();  
        System.out.println(app.name);  
    }  
}
```

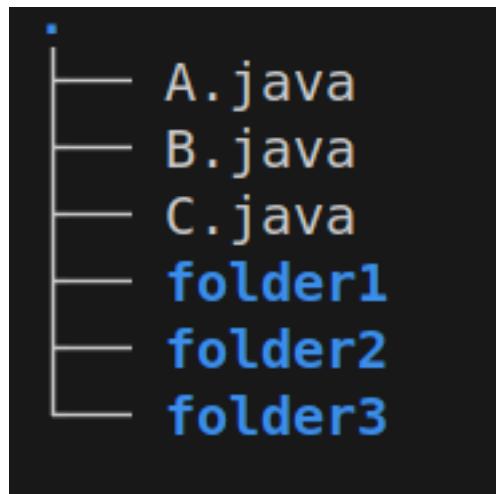
- Execute **apply.java**:

Command:

```
$ javac apply.java  
$ java apply  
Lavatech-App1
```

Example of using classpath and having Java code in different directory structure:

- Create 3 empty folders and 3 java code as shown below:



- Content of A.java should be:

A.java

```
package level1.level2;
public class A {
    public int a=10;
}
```

Content of B.java should be:

B.java

```
package level3.level4;
public class B {
    public int b=20;
}
```

Content of C.java should be:

C.java

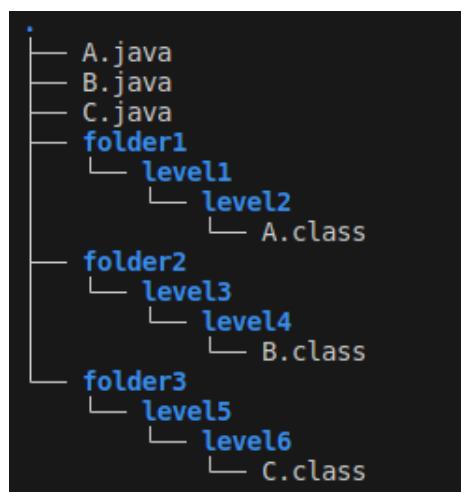
```
package level5.level6;
import level1.level2.A;
import level3.level4.B;
public class C {
    public static void main(String[] args){
        A a1 = new A();
        B b1 = new B();
        System.out.println(a1.a);
        System.out.println(b1.b);
    }
}
```

- Compile all the 3 codes such that "A.class" file is placed under folder1, "B.class" file is placed under folder2, "C.class" file is placed under folder3.

Command:

```
$ javac -d folder1 A.java
$ javac -d folder2 B.java
$ javac -d folder3 -cp folder1:folder2 C.java
```

Above commands should create below directory structure:



- Execute class "C" by setting classpath as shown:

Command:

```
$ java -cp folder1:folder2:folder3 level5.level6.C
```

```
10
```

```
20
```

9.3 JAR files

- A zip file containing group of “.class” files a **jar** file.
- Mostly, third party software plugins are available as jar file.
- Create a Manifest File in jar file (Optional):
 - Manifest contains information such as the entry point class and classpath.
 - If you don't specify a manifest, Java will automatically create a default one.
 - Eg: Create Manifest.txt, with content like this:

```
Main-Class: classname <press enter>
```

- Command to create jar file without manifest:

Syntax:

```
jar -cvf filename.jar code1.class code2.class....
```

```
jar -cvf filename.jar *.class
```

```
jar -cvf filename.jar *.*
```

- Command to create jar file with manifest:

Syntax:

```
jar -cvfm filename.jar Manifest.txt code1.class code2.class
```

```
.....
```

- Extract a jar file:

Syntax:

```
jar -xvf filename.jar
```

- Display table of content of jar file:

Syntax:

```
$ jar -tvf filename.jar
```

- Execute a jar file:

Syntax:

```
java -jar filename.jar
```

- Eg: Create 2 Java programs with manifest file, compress them in jar file & execute the jar file:

A.java

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

B.java

```
public class B {
    public String secret="Choomantar";
}
```

Manifest.txt

```
Main-Class: A
```

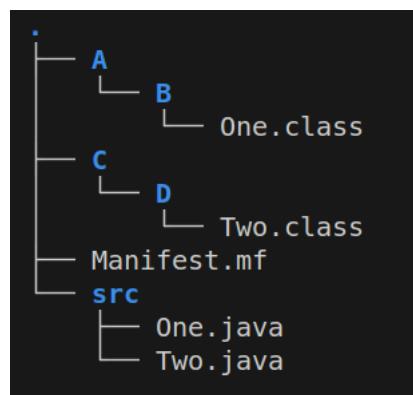
Create and execute the jar file:

Command:

```
$ javac A.java B.java  
$ jar -cvfm code.jar Manifest.txt A.class B.class  
$ java -jar code.jar
```

Choomantar

- Eg: Create jar with ".class" files in different packages, source file and Manifest for below directory structure:

**Command:**

```
$ jar -cvfm code.jar Manifest.mf A/B/One.class  
C/D/Two.class src/*.java  
$ java -jar code.jar
```

Executing a JAR file from anywhere

- **Windows: By creating a batch file.**

- A batch file contains a group of commands.
- You can double click a batch file, to execute commands in it one by one.
- Create a batch file to execute JAR file:

```
file.bat
```

```
java -jar full-path-of-jar-file\code.jar
```

- **Ubuntu: By creating a shell script.**

- Create a shell script to execute JAR file.

```
file.sh
```

```
#!/bin/bash
java -jar /full-path-of-jar-file/code.jar
```

- Set executable permission to JAR file:

```
Command:
```

```
chmod +x file.sh
```

- Add script location in the path system variable by adding below code:

```
Command:
```

```
$ vi /.bashrc
export PATH=<code-location>:$PATH
```

```
$ source /.bashrc
```

- Now you can execute file.sh from anywhere:

```
Command:
```

```
$ file.sh Choomantar
```

Don't fear failure.

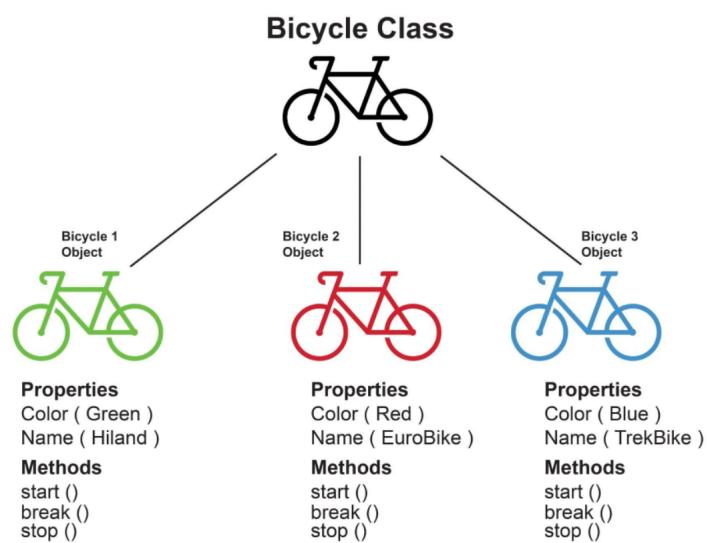


Not failure, but low aim is the crime.

10. OOPs in detail

10.1 Class

- A class is a blueprint to create an object.
- A class consists of:
 - Attributes (or properties)
 - Methods



In this section, we shall see class in detail.

10.1.1 Attributes in detail

- Attributes represents the **characteristics** of an object.
- They are declared **inside a class**.
- Each object has its own copy of attributes.

Syntax:

```
access-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

Access modifier 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:

A.java

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

- **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 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:**

- A static attribute belongs to the class rather than object.
- It is shared among all instances of the class.
- Eg: Accessing variable 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:**

- Can be assigned a value once, and its value cannot be changed.
- Key points:
 - * **Initialization:** Declared & initialized together or within the **constructor** of the class.
 - * **Naming convention:** Should be in **uppercase**, separated with underscores (e.g., FINAL_VARIABLE).
 - * **For primitive type:** Value assigned at initialization cannot be modified. Eg:

A.java

```
public class A {  
    final int count=100;  
    public static void main(String[] args) {  
        A a1 = new A();  
        System.out.println(a1.count); // Output: 100  
    }  
}
```

- * **For reference type:** Reference itself cannot be changed, but the state of the object it refers to can be modified. Eg:

Test.java

```
public class A {  
    final StringBuffer b1=new StringBuffer("Lava");  
    public static void main(String[] args) {  
        A a1 = new A();  
        a1.b1.append("tech");  
        System.out.println(a1.b1); // Output: Lavatech  
    }  
}
```

- * **final static:** Creates class-level constant accessible without an object. Eg:

A.java

```
public class A {  
    static final int count=100;  
    public static void main(String[] args) {  
        System.out.println(count); // Output: 100  
    }  
}
```

- * **protected final:** Creates read-only attribute accessible within the same package or subclass.

Test.java

```
class Constant {  
    protected final int MAX_VALUE = 500;  
}
```

- **transient:**
 - Use transient keyword in serialisation context.
 - More on this is in serialisation chapter.
- **volatile attributes:**
 - A volatile instance variable is used in multithreaded programs.
 - More on this in multi-threading chapter.

10.1.2 Methods in detail

- Methods defines the **actions** objects.
- They are declared within a class and can access attributes.
- They are invoked using object of class.

Syntax:

```
access-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:** Name of method.
- **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 and not other classes or package. Eg:

Test.java

```
class A {
    private void msg() {
        System.out.println("Top secret");
    }

    public void display() {
        msg();
    }
}

public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        a1.display(); // Output: Top secret
    }
}
```

- **protected:**

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

Test.java

```
class A {  
    protected void msg() {  
        System.out.println("Top secret");  
    }  
}  
  
class Test extends A {  
    public static void main(String[] args) {  
        Test t1 = new Test();  
        t1.msg(); // Output: Top secret  
    } }
```

- **default (package-private):**

- No access modifier specified means attribute is default.
- Can be accessed within the same package but not from other packages. Eg:

Test.java

```
class A {  
    void msg() {  
        System.out.println("Time is money");  
    } }  
  
public class Test {  
    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 object.
- Accessed using the class name and not object. Eg:

Test.java

```
class A {
    static void msg() {
        System.out.println("Time is money");
    }
}

public class Test {
    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:

Test.java

```
class Parent {
    public final void display() {
        System.out.println("Parent class");
    }
}

class Child extends Parent { }

class Test {
    public static void main(String[] args) {
        Parent parent = new Parent();
        parent.display();
        Child child = new Child();
        child.display();
    }
}
```

- **abstract:**

- An abstract method does not have body and must be overridden by any inherited class.
 - 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 multi-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 values that can be passed to a method when it is invoked.
- Types of method arguments:

- **Primitive types:**

Code:

```
class Person {
    int age;
    public void setDetails(int a) {
        age = a;
    }
    public static void main(String[] args) {
        Person p = new Person();
        p.setDetails(23);
        System.out.println(p.age); // Output: 23
    }
}
```

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

Code:

```
class Person {
    String name="Raman";
}
class Test {
    public static void displayPerson(Person person){
        System.out.println(person.name);
    }
    public static void main(String[] args) {
        Person p = new Person();
        displayPerson(p); // Output: Raman
    }
}
```

- **varargs:** A variable-length argument parameter (denoted by "...") accepts a variable number of arguments of same type. Eg:

Test.java

```
public class Test {  
    public static void main(String[] args) {  
        print(10, 20, 30, 40, 50);  
    }  
    public static void print(int... numbers) {  
        System.out.print("Numbers-");  
        for (int num : numbers) {  
            System.out.print(" " + num);  
        }  
    }  
}
```

return keyword

- Used to exit a method and provide a value (or no value) back to the caller of the method.

Syntax:

```
public return-type methodName(args) {  
    // body  
    return value;  
}
```

- It has two primary usages:
 - Returning a Value from a Method
 - Exiting a Method

Method return types

- The return type specifies the type of value that the method will return after method execution.
- Method return types:

- Primitive type:** Return primitive type. Eg:

Code:

```
class A {
    public int msg() {
        return 10;
    }
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.msg()); // Output: 10
    }
}
```

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

Code:

```
class Person {
    String name;
}
class Test {
    public static Person setPerson(Person person){
        person.name="Raman";
        return person;
    }
    public static void main(String[] args) {
        Person p;
        p = setPerson(new Person());
        System.out.println(p.name); // Output: Raman
    }
}
```

- **void:**

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

Test.java

```
class Person {  
    String name="Raman";  
}  
class Test {  
    public static void displayPerson(Person person){  
        System.out.println(person.name);  
    }  
  
    public static void main(String[] args) {  
        Person p = new Person();  
        displayPerson(p);  
    }  
}
```

10.1.3 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

 content/chapter11/images/class.png**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 is specified, the class 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.
- Final class method are always final ie. method cannot be overridden.
- Final class attributes need not be final.

- Eg:

Test.java

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

- Disadvantage of final keyword:
 - Missing inheritance (due to final classes)
 - Polymorphism (due to final methods)

abstract class

- Abstract classes cannot have any objects.
- More on abstract class is mentioned in Abstract class chapter.

strictfp class

- Introduced in Java1.2 version, strictfp class ensures all methods in the class & its subclasses provide same result for the floating points operations on any platform.

- 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(a1.calculate()); // Output: 1.0E103  
    } }
```

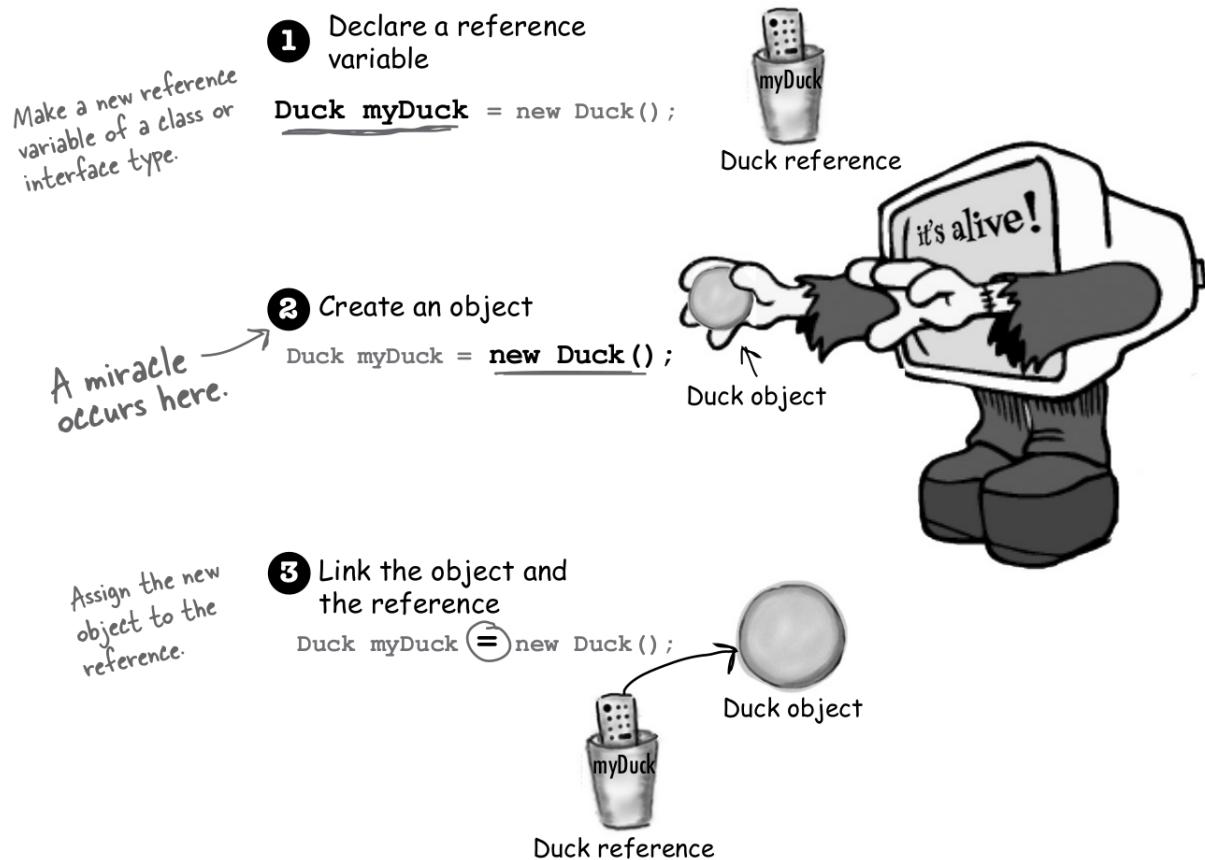
10.2 Constructor in detail

10.2.1 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**.

10.2.2 Constructor

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

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  
    }  
}
```

Who write the constructor if you hav'nt?

Ans: You can write a constructor for your class, but if you don't, **the compiler writes one for you!**. Compiler write a no-arg constructor as shown below:

Code:

```
class Test {
    public Test() {}
```

10.2.3 this keyword

- **this** keyword refers to the current object instance within a class.
- It is used to access attributes or invoke methods of the current object.

Eg:

Test.java

```
class Circle {
    float r;
    public Circle(float r) {
        this.r = r;
    }
    public float getArea() {
        float area = 3.14*r*r;
        return area;
    }
    public class Test {
        public static void main(String[] args) {
            Circle c1 = new Circle(3);
            float area = c1.getArea();
            System.out.println(area); // Output: 28.26
        }
    }
}
```

10.2.4 Constructor Chaining.

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

Test.java

```
class Animal {  
    public Animal() {  
        System.out.print("Animal");  
    }  
}  
  
class Duck extends Animal {  
    public Duck() {  
        System.out.print("Duck");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Duck d1 = new Duck(); // Output: Animal Duck  
    }  
}
```

10.2.5 super()

- **super()** method 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: 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(d1.name); // Output: Duckling
        System.out.println(d1.size); // Output: 4
    }
}

```

10.2.6 super keyword

- **super keyword** is used to refer parent class.
- It can be used to access members of parent class from child class.
- 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
```

10.2.7 Constructor overloading

- More than one constructor in your class, with different argument lists is called constructor overloading.
- Eg:

Rectangle.java

```
class Rectangle {  
    float l, w, h;  
    public Rectangle() {  
        this.l=0.0f;  
        this.w=0.0f;  
        this.h=0.0f;  
    }  
    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 static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        System.out.println(r1.getArea()); // Output: 0.0  
        Rectangle r3 = new Rectangle(3,4,5);  
        System.out.println(r3.getArea()); // Output: 94.0  
    }  
}
```

10.2.8 this()

- **this()** invokes one constructor from another constructor.
- **this()** needs **to be first statement in the constructor body**.
- Eg:

Test.java

```
class Rectangle {  
    float l, w, h;  
    public Rectangle() {  
        this(0,0,0);  
    }  
    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 static void main(String[] args) {  
        Rectangle r1 = new Rectangle(3,4,5);  
        System.out.println(r1.getArea()); // Output: 94.0  
        Rectangle r2 = new Rectangle();  
        System.out.println(r2.getArea()); // Output: 0.0  
    }  
}
```

Note:

- Constructor can have **super()** or **this()**, not both at a time.
- Use **super()** or **this()** only inside constructor.

10.2.9 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 instance 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

10.2.10 Constructor's Access modifier

private constructor

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

Eg:

A.java

```
class A {
    private A() {
        System.out.println("Working");
    }
    public static void main(String[] args) {
        A a1 = new A(); // Output: Working
    }
}
public class B extends A {
    public static void main(String[] args) {
        B b1 = new B(); X // Cannot instantiate
    }
}
```

protected constructor

Syntax	Accessibility
<code>protected Classname {}</code>	<ul style="list-style-type: none"> • Can be access from same class & subclasses or classes within the same package • Cannot be accessed from different package where class is not subclass.

Eg:

B.java

```
class A {
    protected A() {
        System.out.print("One");
    }
}

public class B extends A {
    public B() {
        System.out.print("Two");
    }
}

public static void main(String[] args) {
    B b1 = new B(); // Output: OneTwo
}
```

public constructor

Syntax	Accessibility
public Classname {}	Accessible from: <ul style="list-style-type: none">• Other classes• Subclasses• Different packages

- Eg 1:

B.java

```
class A {  
    public A() {  
        System.out.print("A");  
    }  
}  
  
public class B extends A {  
    public B() {  
        System.out.print("B");  
    }  
    public static void main(String[] args) {  
        B b1 = new B(); // Output: AB  
    }  
}
```

default constructor: Means no 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:

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: AB  
    }  
}
```

10.2.11 Instance block

- Instance block is defined within a class & executed object is created.
- It can initialises object attributes.
- It is **executed before the constructor of the class.**
- 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); // Output: 8080  
        System.out.println(d2.value); // Output: 10  
    }  
}
```

10.2.12 Static initializer

- Static initializer is used to initialize static attributes.
- It is executed only once when the class is loaded into memory.
- Multiple static blocks are allowed & will execute from top to bottom.

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("End");  
    }  
}
```

Output:

```
Count: 10  
End
```

Did you know?

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.

10.2.13 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

10.2.14 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**.

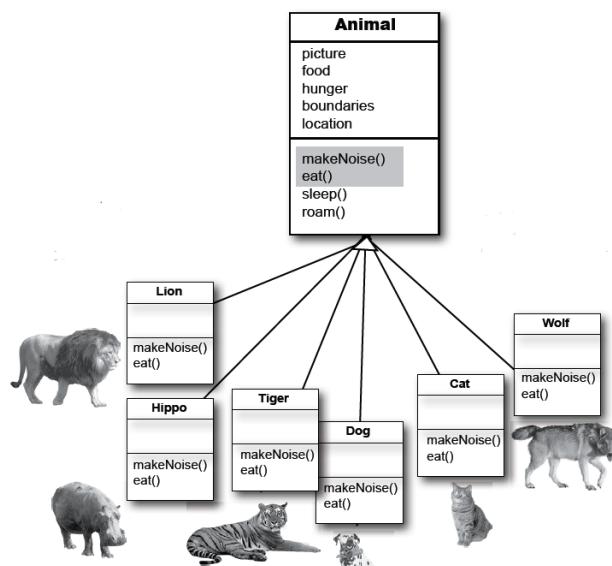
Eg:

A.java

```
class A {
    public static void main(String[] args) {
        A a1 = new A();
        a1 = null;
    }
}
```

10.3 Inheritance

- Inheritance makes putting common code in a parent class and inherit the code in child class.
- The child class inherits the members (i.e attributes & methods) of the parent class.



- **IS-A relationship:**

- Inheritance is also called **IS-A** relationship.
- Eg: SuperMan IS-A SuperHero

- The **extends** keyword is used to implement IS-A relationship.

Syntax:

```
class A {}  
class B extends A {}
```

- Parent class reference can hold child class object, but vice versa is not possible. 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
```

```

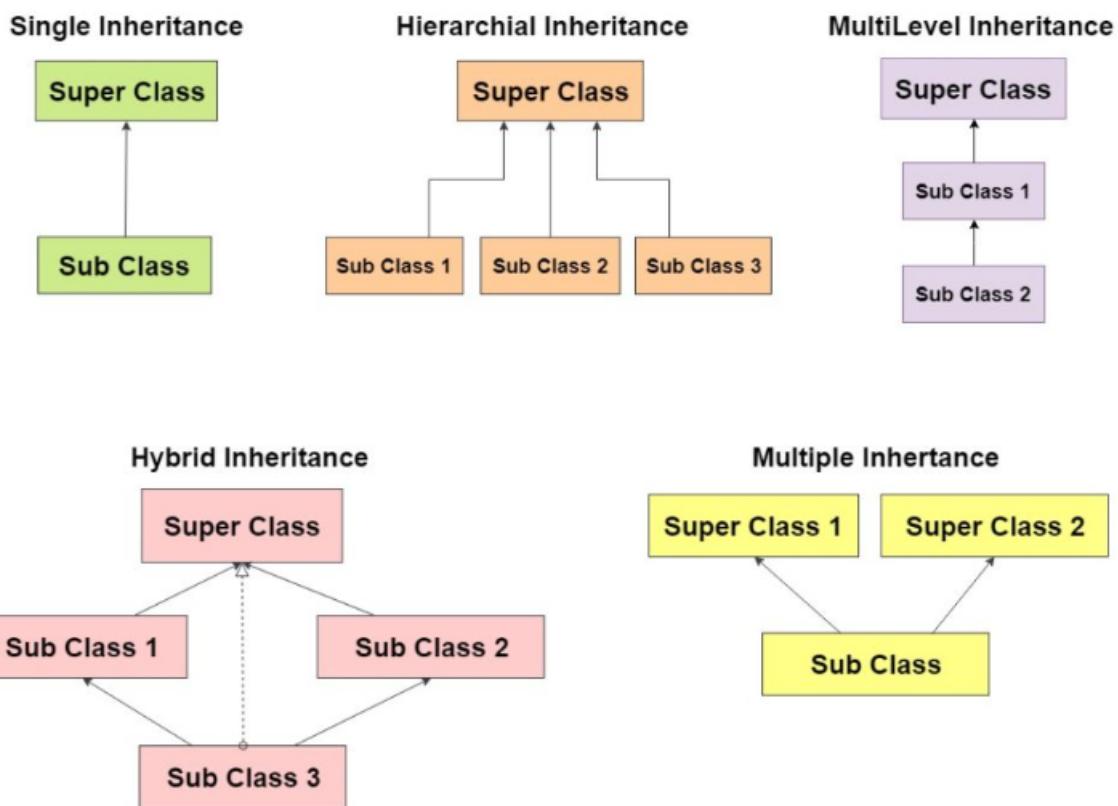
B b = new B();
b.displayA();
b.displayB();

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

}

```

10.3.1 Types of inheritance



- Single inheritance:

Code:

```

class A { }
class B extends A { }

```

- **Multilevel inheritance:**

Code:

```
class Grandparent { }
class Parent extends Grandparent { }
class Child extends Parent { }
```

- **Hierarchical inheritance:**

Code:

```
class Superclass { }
class Subclass1 extends Superclass { }
class Subclass2 extends Superclass { }
```

- **Multiple inheritance (through interfaces):**

- A java class can't extend more than one class at a time.
- Multiple inheritance is not directly supported.
- It can be achieved through interfaces.

Code:

```
interface Interface1 { }
interface Interface2 { }
class MyClass implements Interface1, Interface2 { }
```

Why Java does not provide support for multiple-inheritance?

Ans: There maybe a chance of ambiguity problem.

Why multiple-inheritance is supported by interface?

Ans: We shall see this in the interface chapter.

- **Cyclic inheritance:** Not allowed in java.

Code:

```
class A extends B {} X
class B extends A {} X
class A extends A {} X
```

10.3.2 Has-A relationship

Has-a relationship causes **reusability** of the code. There are 2 types of Has-A relationship:

- Aggregation (Weaker form of "has-a")
- Composition (Strong form of "has-a")

Aggregation:

- Aggregation is a one-way relationship(unidirectional association).
- For eg: Bank can have employees but vice versa is not possible.

Code:

```
class Bank {  
    String bname;  
    Bank(String bname) {  
        this.bname = bname;  
    }  
    public void display(Customer c) {  
        System.out.println(bname); // Output: AXIS  
        System.out.println(name); // Output: Ram  
    } }  
class Customer {  
    String name;  
    Customer(String name) {  
        this.name = name;  
    } }  
class Branch {  
    public static void main(String[] args) {  
        Bank b = new Bank("AXIS");  
        Customer c = new Customer("Ram");  
        b.display(c);  
    } }
```

Composition:

- In composition, two entities are highly dependent on each other.
- One entity cannot exist without the other. Eg: A car has an engine.

Code:

```

class Car {
    private final Engine engine;
    String model;
    public Car(String model) {
        engine = new Engine("POWERHIGH");
        this.model = model;
    }
    public void show() {
        System.out.print(model+",");
        System.out.print(engine.type);
    }
}
class Engine {
    String type;
    Engine(String type) {
        this.type = type;
    }
}
public class Test {
    public static void main(String arg[]) {
        Car car = new Car("5X");
        car.show(); // Output: 5X,POWERHIGH
    }
}

```

10.3.3 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.

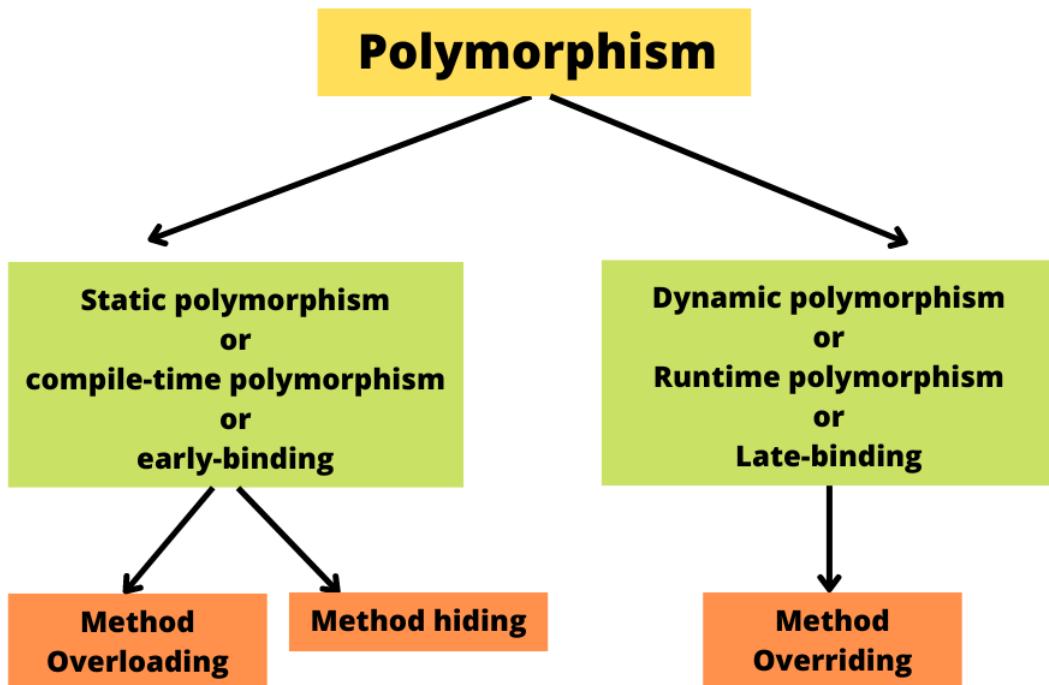
10.4 Polymorphism

- Polymorphism refers to the ability of objects to take on multiple forms or behaviors. Eg:

Test.java

```
class Animal {  
    String sound;  
    public void makenoise() {  
        System.out.print(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");  
        animals[1] = new Cat("Meow");  
        for(int i=0; i<animals.length; i++) {  
            animals[i].makenoise(); // Output: Baw Meow  
        }  
    } }
```

- Polymorphism can be achieved through two main mechanisms:



Let's see each of these in detail.

10.4.1 Method overloading

- Method overloading means multiple methods with the **same name but different arguments** in a class.
- Compiler will use method signature while resolving method calls.
- Method signature:

Syntax:

```
modifier returnType methodName(type arg, ...)
```

Note:

- Method **returnType** and **modifier** are not part of method signature and won't affect method overloading.
- Within a class two 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 static void main(String[] args) {  
        Calculate math = new Calculate();  
        int sum1 = math.add(5, 10);  
        double sum2 = math.add(2.5, 3.7);  
        System.out.println("Sum1: " + sum1); // Output: 15  
        System.out.println("Sum2: " + sum2); // Output: 6.2  
    } }
```

10.4.2 Method overriding

- Method overriding means method in the child class has the **same name, return type, arguments** as a method in the parent class.
- Key points:
 - Signature:** **Same signature** (name, return type, arguments) is required for overriding & overridden method.
 - Access modifier:** Overriding method **cannot have a more restrictive access modifier** than overridden method.
 - @Override annotation:** Optionally, use **@Override** annotation.
- 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 Test {  
    public static void main(String[] args) {  
        Animal animal1 = new Cat();  
        animal1.makeSound(); // Output: Cat meows  
    }  
}
```

10.4.3 Method hiding

Method hiding is same overriding except:

Method hiding	Method overriding
It is like sticking poster on black board	It is like erasing board and writing new content
Require static parent and child method	Require non-static parent and child method
Compiler does method resolution	JVM does method resolution
Known as compile-time/static polymorphism or early binding	Known as runtime/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();      // Output: Parent
        Child c1 = new Child();
        c1.m1();      // Output: Child
        Parent p2 = new Child();
        p2.m1();      // Output: Parent
    }
}

```

10.4.4 Overloading V/S Overriding

Property	Overloading	Overriding
Method names	Must be same	Must be same
Argument types	Must be different (atleast order)	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	Scope cannot be reduced but we can increase
Method resolution	Done by compiler based on reference types	Done by JVM based on runtime object
Known as	Compile-time/static polymorphism or early binding	Runtime/dynamic polymorphism or late binding

10.5 Data Encapsulation

- Encapsulation is process of binding attributes & methods (behavior) into a class.

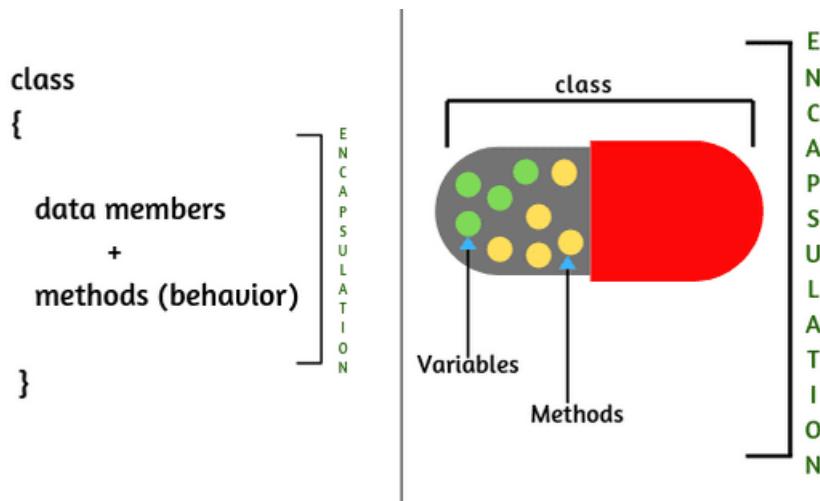


Fig: Encapsulation

- Data encapsulation = **data hiding + data abstraction**.
- Let see each of these in detail.

10.5.1 Data Hiding

- Data hiding refers to binding attributes within a class.
- It controls data visibility and accessibility from outside the class.
- Below code shows data not being hidden:

Test.java

```
class Sample {
    int no;
}

class Test {
    public static void main(String[] args) {
        Sample s1 = new Sample();
        s1.no = 1; // No Data hiding
    }
}
```

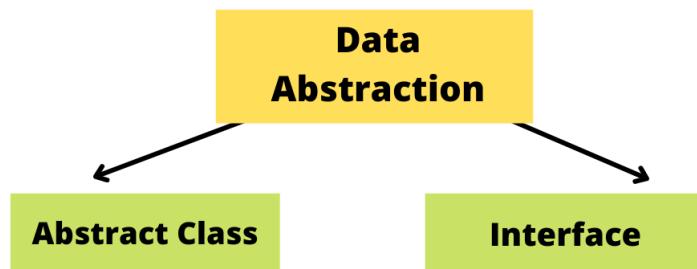
- Data hiding is done using **private** attributes.
- Eg: You can create public methods **setter()** and **getter()** to get and set private instance variables as shown below:

Sample.java

```
class Sample {
    private int no;
    public void setter(int no) {
        this.no = no;
    }
    public void getter() {
        System.out.println(no);
    }
    public static void main(String[] args) {
        Sample s1 = new Sample();
        s1.setter(10); // Data hidden
        s1.getter(); // Output: 10
    }
}
```

10.5.2 Data Abstraction

- Data abstraction is the process of showing essential information & hiding other details.
- Types of abstraction:



10.5.3 Tightly encapsulated class

- A class is said to be tightly encapsulated if and only if each variable declared is **private**.
- Eg:

Code:

```
class A {  
    private int a; //Tightly encapsulated  
}  
  
class B extends A {  
    int b; // Not tightly encapsulated  
}
```

10.5.4 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 B {  
    static int i=100;  
}  
  
class A {  
    static int i = B.i; // Tightly coupled  
    public static void main(String[] args) {  
        System.out.println(A.i); // Output: 100  
    }  
}
```

- **Tightly coupling is not good** programming practise.
- Lossly coupling is a good programming practice.

- Problem with tightly coupling:
 - Cannot modify component without dependency problem.
 - Reduce reuseability.
 - Reduce code mainitainabilty.

10.5.5 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.

11. Abstract Class & Interface

11.1 Abstract class

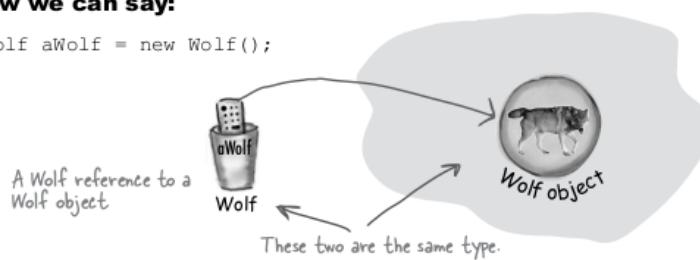
11.1.1 What is abstract class?

- 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?

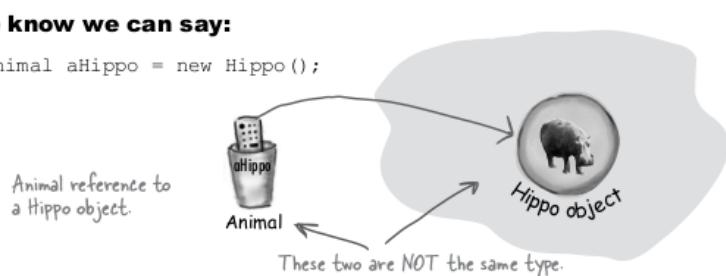
We know we can say:

```
Wolf aWolf = new Wolf();
```



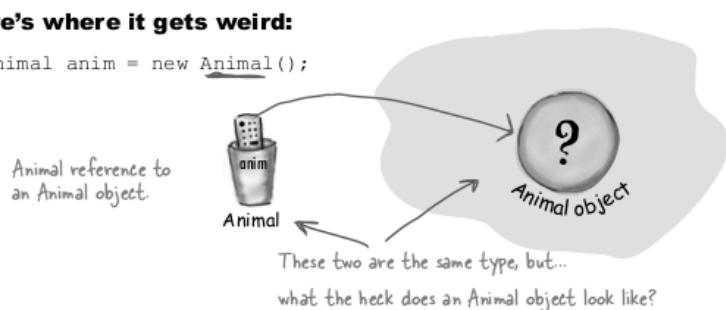
And we know we can say:

```
Animal aHippo = new Hippo();
```



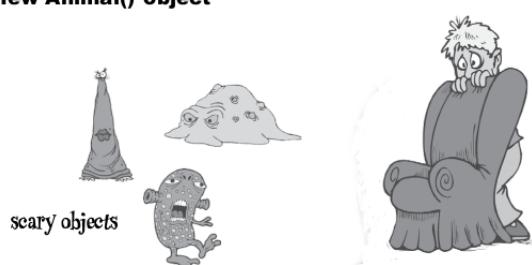
But here's where it gets weird:

```
Animal anim = new Animal();
```



- Problem is -

What does a new Animal() object look like?



- In this situation, there is no need to create parent class object.

- Solution: The **compiler** won't let you instantiate an abstract class.

Syntax:

```
abstract public class Classname { }
```

- Eg:

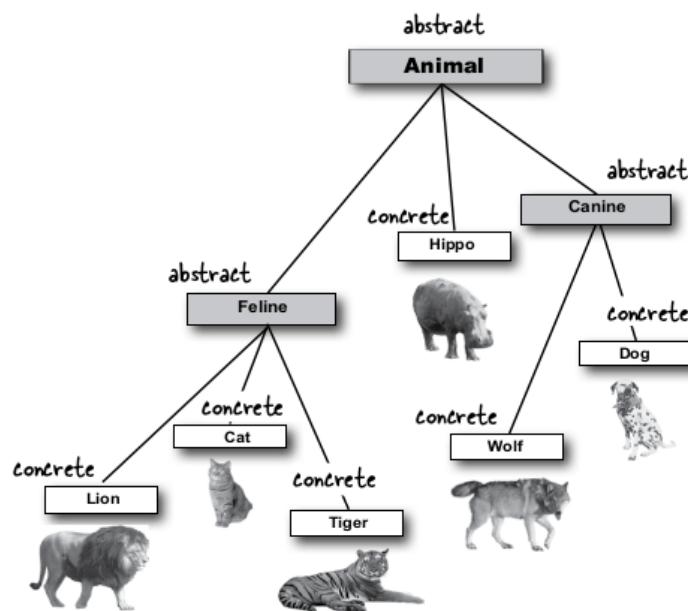
Test.java

```
abstract public class Canine extends Animal {
    public void roam() {}
}

public class MakeCanine {
    public void go() {
        Canine c;
        c = new Dog(); ✓
        c = new Canine(); ✗
    }
}
```

11.1.2 What is concrete class?

- A class that's not abstract is called a **concrete** class. Eg:



11.1.3 What is abstract method?

- An abstract method has no body!
- Abstract method are created in abstract class abstract only.

Syntax:

```
abstract class Classname() {
    modifier abstract return-type methodname(args);
}
```

- Abstract method are compulsorily overridden by child class.
- Eg:

Dog.java

```
abstract class Animal {
    public abstract void sound();
}

class Dog extends Animal {
    public void sound() {
        System.out.println("Bow");
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Output: Bow
    }
}
```

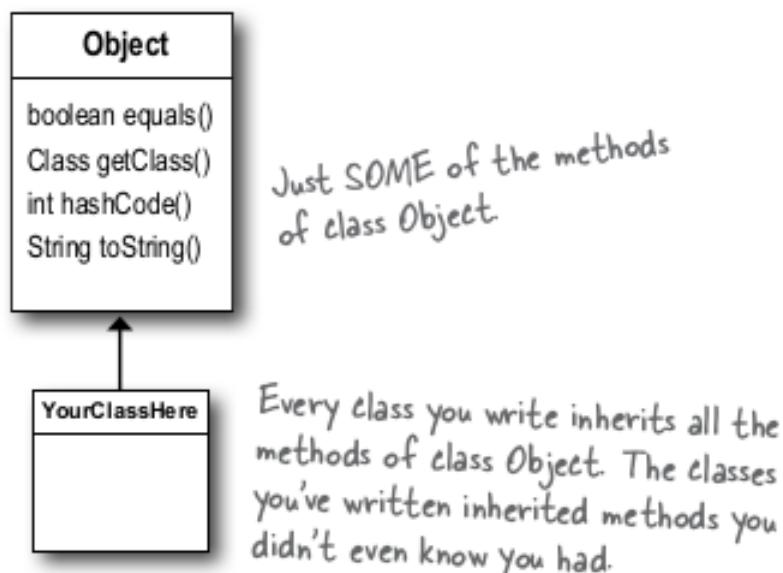
11.1.4 Illegal modifier: final abstract

- Abstract methods should be overriden in child classes.
- Whereas, you can't override final methods.
- Hence final abstract combination is illegal combination for methods.

11.2 The ultimate superclass: Object

- Every class in Java extends **Object** class.
- Object class is the mother of all classes.

So what's in this **Object** class?



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 of the object.

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: 117845  
    } }
```

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  
    } }
```

11.3 Interface

- Interface is a 100% pure abstract class as it's every method is always abstract.

Syntax:

```
modifier interface InterfaceName{ }
```

- A class uses the **implements** keyword to implement interface:

Syntax:

```
modifier class Name implements Interface1, Interface2..{ }
```

- Interfaces **solves the multiple inheritance problem.**

11.3.1 Interface methods

- Interface methods are always **public & abstract** (even without declaring it explicitly). Below all declaration are valid:

Code:

```
void m1(); or public void m1();  
abstract void m1(); or public abstract void m1();
```

- Eg:

Dog.java

```
interface Animal {  
    void sound();  
}  
class Dog implements Animal {  
    public void sound() {  
        System.out.println("Bow"); }  
    public static void main(String[] args) {  
        Dog dog = new Dog();  
        dog.sound(); // Output: Bow  
    } }
```

11.3.2 extends V/S implements

extends	implements
<p>A class can extend only one class at a time:</p> <p>Code:</p> <pre>class B {} class A extends B {}</pre>	<p>A class can implement any number of interfaces:</p> <p>Code:</p> <pre>interface A {} interface B {} class C implements A, B {}</pre>
<p>A class can extend another class and can implement any number of interfaces:</p> <p>Code:</p> <pre>class B {} interface C {} interface D {} class A extends B implements C,D {}</pre>	<p>An interface can extend any number of interfaces:</p> <p>Code:</p> <pre>interface A {} interface B {} interface C extends A, B {}</pre>

11.3.3 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;  
}
```

11.3.4 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 contains 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 overridden:  
interface A {  
    public void m1();  
} interface B {  
    public int m1();  
}
```

Can java class be implemented any number of interfaces simultaneously?

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.

11.3.5 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  
    } }
```

11.3.6 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:
 - Serializable (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 & 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.

11.3.7 Adapter Classes

- Adapter class implements an interface with only empty implementation.
- You can extend the adapter class and override only the methods they need, instead of implementing all the methods of the interface.
- Eg:

Dog.java

```
interface Animal {  
    public abstract void sound();  
    public abstract void move();  
}  
  
class Cannian implements Animal {  
    public void sound() {}  
    public void move() {}  
}  
  
class Dog extends Cannian {  
    public void sound() {  
        System.out.println("Bow");  
    }  
    public static void main(String[] args) {  
        Dog d1 = new Dog();  
        d1.sound(); // Output: Bow  
    }  
}
```

11.3.8 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 <ul style="list-style-type: none"> • 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 initialisation at the time of declaration	Variables are not required to perform initialsation at the time of decaration
Interface can't declare static and instance blocks	Can 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 initialisation 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.

Don't focus on the pain ->



Focus on progress ->

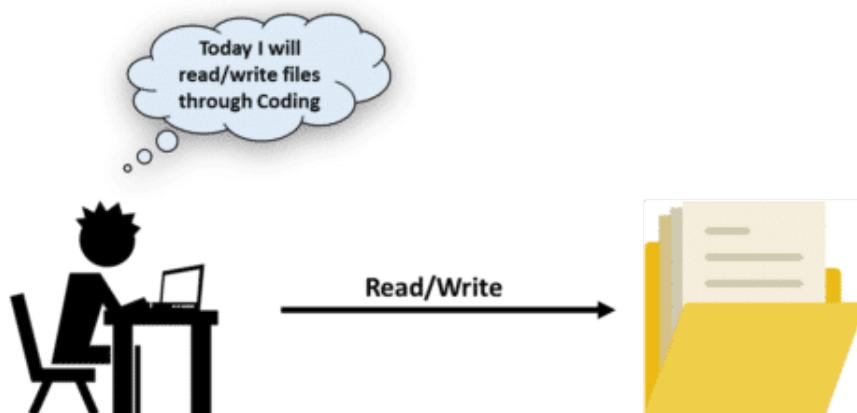


- Dwayne Johnson

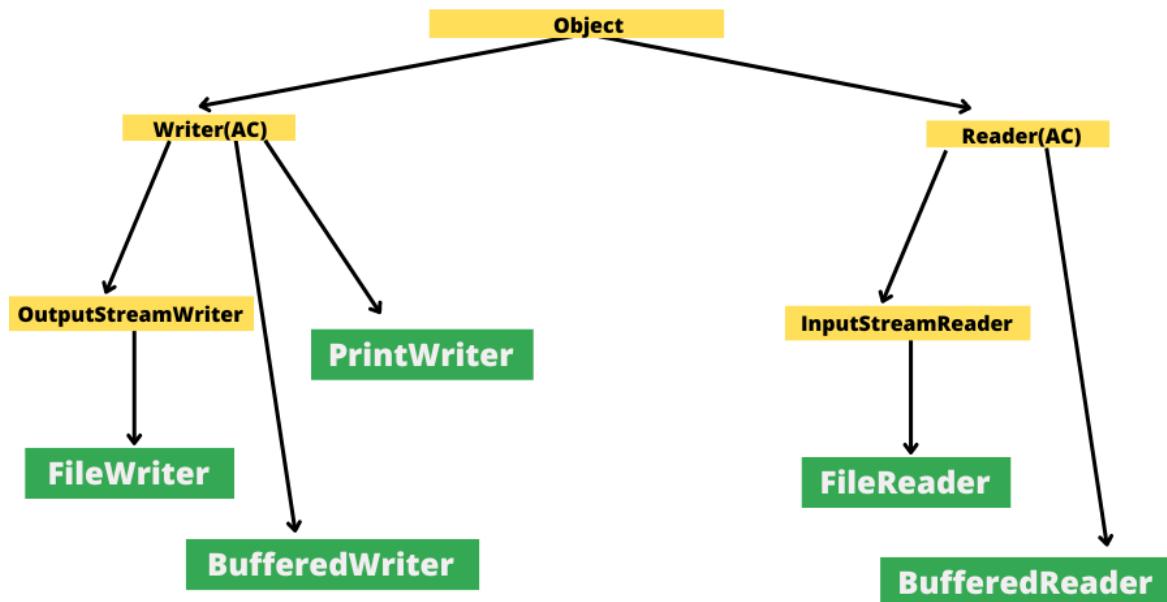
12. File Handling in Java

12.1 File IO

- Java I/O stream is used to perform read and write operations in file permanently.
- Java I/O stream is also called File Handling or File I/O.
- It is available in **java.io** package.



Below is the heirarchy of important FileIO classes:



Let's see each of the green highlighted classes in detail.

12.1.1 File class

File class provides methods for creating, deleting, renaming, and obtaining information about files and directories.

- Create file object in current working directory:

Syntax:

```
File f = new File(String name);
```

- Creates a file object for a specified sub directory:

Syntax:

```
File f = new File(String subdirname, String name);
File f = new File(File subdir, String name);
```

- **Methods**

- Check if file/directory is available:

Syntax:

```
boolean exists();
```

- Create a new file if not present:

Syntax:

```
boolean createNewFile();
```

- Create a new directory if not present:

Syntax:

```
boolean mkdir();
```

- Check if file object points to given file:

Syntax:

```
boolean isFile();
```

- Check if file object points to given directory:

Syntax:

```
boolean isDirectory();
```

- List the names of all file/diretory present in a directory:

Syntax:

```
String[] list();
```

- Returns number of characters present in specified file:

Syntax:

```
long length();
```

- Delete specified file or directory:

Syntax:

```
boolean delete();
```

Example programs:

- Create a file named "abc.txt" and a folder named "abc" in current working directory:

Test.java

```
import java.io.File;  
class Test {  
    public static void main(String[] args) throws Exception {  
        File f1 = new File("abc");  
        f1.mkdir(); // Create a directory  
        File f = new File("abc.txt");  
        f.createNewFile(); // Create a file  
    } }
```

- Create a directory named "inner" in folder named "outer", assuming "outer" already exists:

Test.java

```
import java.io.File;  
class Test {  
    public static void main(String[] args) throws Exception {  
        File f1 = new File("outer");  
        if (f1.exists()) {  
            File f2 = new File(f1, "inner");  
            f2.mkdir();  
        }  
        else  
            System.out.println("Outer directory does not exist");  
    } }
```

- Create a file named "abc.txt" only if it exists else display message "File exists".

Test.java

```
import java.io.File;
class B {
    public static void main(String[] args) throws Exception {
        File f = new File("abc.txt");
        if (f.exists())
            System.out.println("File exists");
        else
            f.createNewFile();
    }
}
```

- Display all files present in folder "outer/inner" folder:

Test.java

```
import java.io.File;
class Test {
    public static void main(String[] args) throws Exception {
        File f1 = new File("outer", "inner");
        String[] files = f1.list();
        for (String x: files)
            System.out.println(x);
    }
}
```

- Display only files present in "lavatech" folder, but not directory present under "lavatech" folder:

Test.java

```
import java.io.File;
class Test {
    public static void main(String[] args) throws Exception
    {
        File f1 = new File("lavatech");
        String[] files = f1.list();
        for (String x: files) {
            File f = new File(f1,x);
            if (f.isFile())
                System.out.println(x);
        }
    }
}
```

12.1.2 FileWriter class

- FileWriter class is used to **write characters to a file**.
 - To override existing data:

Syntax:

```
FileWriter fw = new FileWriter(String fname);
FileWriter fw = new FileWriter(File f);
```

- To append to existing data:

Syntax:

```
FileWriter fw = new FileWriter(String fname,
boolean append);
FileWriter fw = new FileWriter(File f, boolean ap-
pend);
```

Note:

If the specified file is not already available, then all the above constructors will create that file.

- **Methods**

- Write a single character to file:

Syntax:

```
void write(int ch);
```

- Write an array of character to file:

Syntax:

```
void write(char[] ch);
```

- Write a string to file:

Syntax:

```
void write(String s);
```

- To give the guarantee that total data including last character will be written to the file.

Syntax:

```
void flush();
```

- To close the writer:

Syntax:

```
void close();
```

Example Program:

- Create a file named "abc.txt" and write some string and characters to it. Close the file and open again to override and write some string and characters to it.

Test.java

```
import java.io.*;
class Test {
    public static void main(String[] args) throws Exception {
        FileWriter fw = new FileWriter("abc.txt");
        fw.write(108);
        fw.write("avatech\ntechnology");
        fw.write("\n");
        char[] c = {'a','b','c'};
        fw.write(c);
        fw.flush();
        fw.close();
        FileWriter fw = new FileWriter("abc.txt", true);
        fw.write("\nappending new line");
        fw.flush();
        fw.close();
    }
}
```

Content of "abc.txt":

Output:

```
lavatech
technology
abc
appending new line
```

12.1.3 FileReader class

- FileReader class is used to **read characters from the file**.

- Constructors:**

Syntax:

```
FileReader fr = new FileReader(String fname);  
FileReader fr = new FileReader(File f);
```

- Methods**

- Read character in unicode format from file:

Syntax:

```
int read();
```

- Read character in character format from file:

Syntax:

```
int read(char[] ch);
```

- To close the reader:

Syntax:

```
void close();
```

Example Program:

- Display all character in file "abc.txt".

Test.java

```
import java.io.*;  
class One {  
    public static void main(String[] args) throws Exception  
{  
        FileReader fr = new FileReader("abc.txt");  
        int ch;  
        while((ch = fr.read())!= -1) {  
            System.out.println((char)ch);  
        } } }
```

12.1.4 Drawback of FileWriter & FileReader

- FileWriter **require adding line separator "\n" manually.**
- FileReader **require reading data character by character & not line by line or word by word.**

To overcome these problems we should go for **BufferedWriter and BufferedReader**.

12.1.5 BufferedWriter class

- BufferedWriter **writes character data** to the file.
- It can't write directly in the file & require some writer object.
- Constructor:

Syntax:

```
BufferedWriter bw = new BufferedWriter(Writer w);  
BufferedWriter bw = new BufferedWriter(Writer w, int  
buffersize);
```

- **Methods**

- Write a single character to file:

Syntax:

```
void write(int ch);
```

- Write an array of character to file:

Syntax:

```
void write(char[] ch);
```

- Write a string to file:

Syntax:

```
void write(String s);
```

- To give the guarantee that total data including last character will be written to the file.

Syntax:

```
void flush();
```

- To close the writer. On closing BufferedWriter, the internal FileWriter object will be closed automatically.

Syntax:

```
void close();
```

- To insert a line separator:

Syntax:

```
void newline();
```

Example Program:

- Create a file "xyz.txt" and write string and characters to it.

Test.java

```
import java.io.*;  
class Test {  
    public static void main(String[] args) throws Exception  
{  
    FileWriter f = new FileWriter("xyz.txt");  
    BufferedWriter bw = new BufferedWriter(f);  
    bw.write(108);  
    bw.newLine();  
    char[] c = {'a','b','c'};  
    bw.write(c);  
    bw.newLine();  
    bw.write("Lavatech Technology");  
    bw.flush();  
    bw.close();  
}
```

12.1.6 BufferedReader class

- BufferedReader reads **character data line-by-line or character-to-character** from the file.
- **Constructor:**

Syntax:

```
BufferedReader br = new BufferedReader(Reader r);  
BufferedReader br = new BufferedReader(Reader r , in  
buffersize);
```

- **Method:**

- Read character in unicode format from file:

Syntax:

```
int read();
```

- Read character in character format from file:

Syntax:

```
int read(char[] ch);
```

- To read next line from the file and returns it. If the next line not available, then this method returns null:

Syntax:

```
String readLine();
```

- To close the reader:

Syntax:

```
void close();
```

Example Program:

- Read all lines in file "abc.txt":

Test.java

```
import java.io.*;
class Test {
    public static void main(String[] args)
throws Exception {
    FileReader fr = new FileReader("abc.txt");
    BufferedReader br = new BufferedReader(fr);
    String line;
    while( (line = br.readLine()) != null ) {
        System.out.println(line);
    }
    br.close();
}
```

12.1.7 PrintWriter class

- PrintWriter is the most enhanced writer.
- PrintWriter can **write primitive data directly to the file**.
- **Constructor:**

Syntax:

```
PrintWriter pw = new PrintWriter(String fname);
PrintWriter pw = new PrintWriter(File f);
PrintWriter pw = new PrintWriter(Writer w);
```

Note:

PrintWriter can communicate directly with the file and can communicate via some writer object also.

- **Method:**

- Write a single character to file:

Syntax:

```
void write(int ch);
```

- Write an array of character to file:

Syntax:

```
void write(char[] ch);
```

- Write a string to file:

Syntax:

```
void write(String s);
```

- To give the guarantee that total data including last character will be written to the file.

Syntax:

```
void flush();
```

- To close the writer:

Syntax:

```
void close();
```

- To print a single character to file:

Syntax:

```
void print(char ch);
```

- To print an integer to file:

Syntax:

```
void print(int i);
```

- To print a double to file:

Syntax:

```
void print(double d);
```

- To print a boolean to file:

Syntax:

```
void print(boolean b);
```

- To print a string to file:

Syntax:

```
void print(String s);
```

Example Program:

- Create a file named "mno.txt" and write some content to it:

Test.java

```
import java.io.*;
class Test {
    public static void main(String[] args) throws Exception
    {
        FileWriter fw = new FileWriter("mno.txt");
        PrintWriter out = new PrintWriter(fw);
        out.write(100);
        out.println(100);
        out.println(true);
        out.println('c');
        out.println("lavatech");
        out.flush();
        out.close();
    }
}
```

This will create file named "mno.txt" with below content:

Output:

d100
true
c
lavatech

Note:

write() v/s print()

- write(100) would result in 'd'
- print(100) would result in 100

Don't focus on the pain ->



Focus on progress ->



- Dwayne Johnson

13. Exception Handling

13.1 What is exception?

- **Errors or exceptional situations** that may occur during the execution of a program are called exceptions.
- Common egs:
 - Division by zero
 - Accessing an array out of bounds
 - File not found

13.1.1 What is Exception Handling?

- Exception handling handles runtime “exceptional situations”.
- It’s based on you knowing that the method you’re calling is risky (i.e. generate an exception), so that you can write code to deal with it.
- If the exception is not handled properly, it can terminate the program.
- Exception handling does not mean repairing an exception.

13.1.2 Default Exception Handling

- The method containing exception creates **exception object**.
- Exception object includes:
 - Exception name
 - Exception description
 - Stack trace containing exception location
- Exception results in abnormal program termination.
- **Exception handlers prints exception** information as shown below:

Exception in thread "xxxxxx" <Name of Exception>: <Description>

Stack Trace

- Eg 1: Below code result in ArithmeticException.

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        int i=5;  
        int j=0;  
        System.out.println(i/j);  
    }  
}
```

Error:

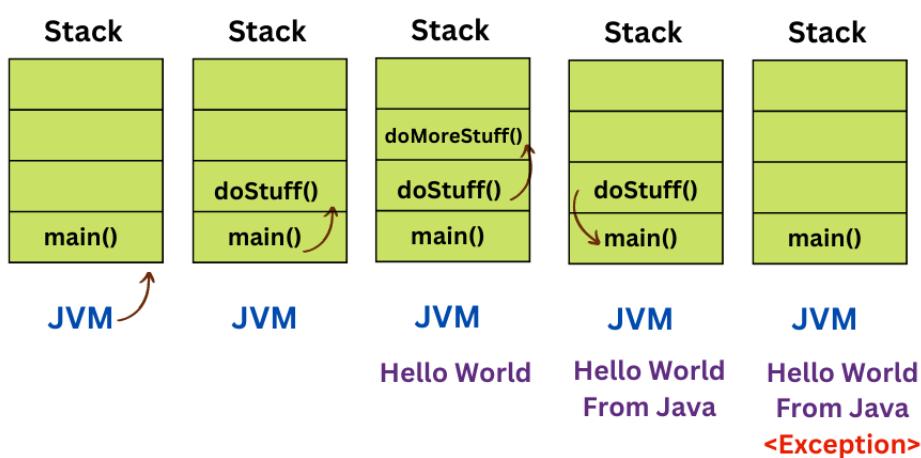
Exception in thread "main" java.lang.ArithmetricException:
/ by zero at demo.main(demo.java:5)

- Eg 2: Consider below example where main() exception works with multiple method calls.

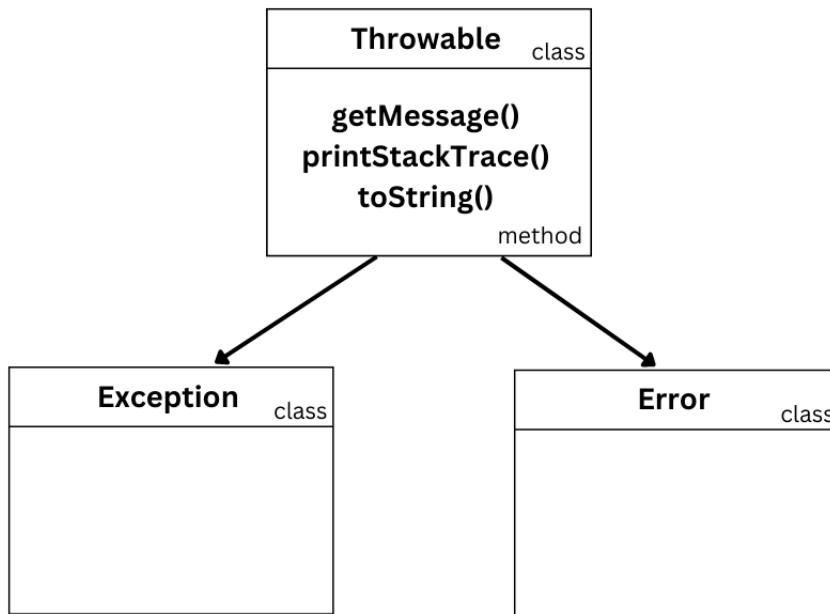
demo.java

```
public class demo {
    public static void main(String[] args) {
        doStuff();
        System.out.println(10/0);
    }
    public static void doStuff() {
        doMoreStuff();
        System.out.println("From Java");
    }
    public static void doMoreStuff() {
        System.out.println("Hello world");
    }
}
```

Output with stack trace and exception:



13.2 Exception Hierarchy



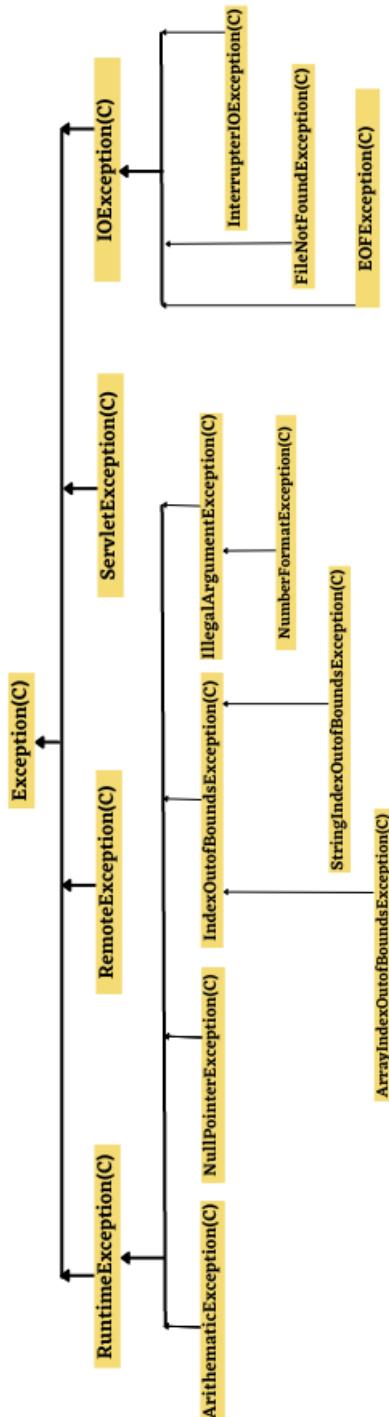
Let's see each of these class in detail.

13.2.1 Throwable class

- **Throwable** is the **root class** of Exception hierarchy.
- It represents an object that can be thrown and caught.
- It **can be subclassed to create custom exception**.
- It has two direct subclasses:
 - **Exception**
 - **Error**
- It contains 3 important methods:
 - **getMessage()**
 - **printStackTrace()**
 - **toString()**

13.2.2 Exception

- Exception is a subclass of Throwable.
- Exception class represents exceptional conditions that a program can catch and handle, allowing the program to recover from them.



13.2.3 Error

- Error subclass of Throwable.
- Errors are exceptional conditions that are not recoverable by the application.
- Eg: OutOfMemoryError, StackOverflowError, VirtualMachineError

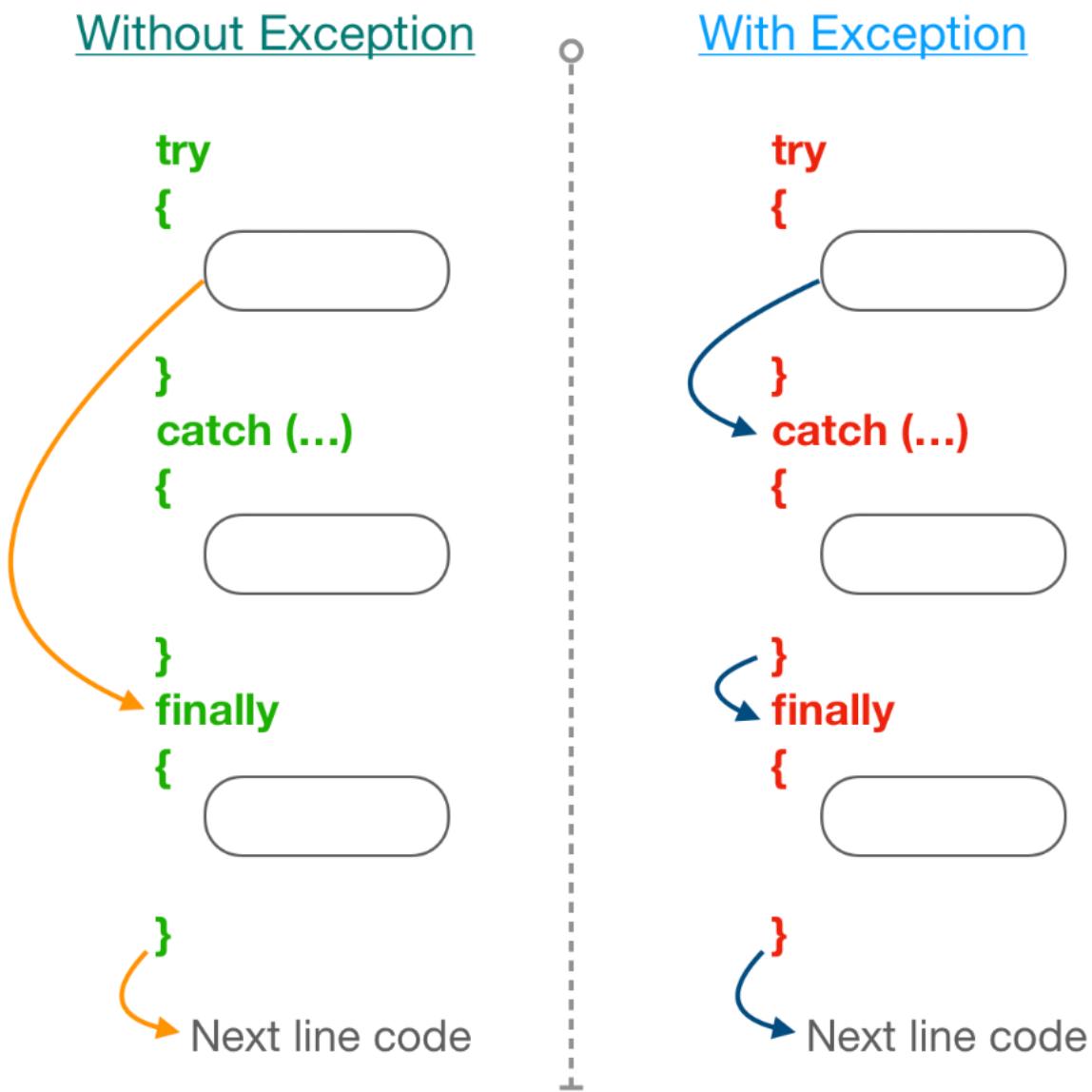
13.2.4 Try..Catch..Finally

Exception handling works using four keywords:

- try:
 - This block **contains code with exception**, also called risky code.
- catch:
 - This is optional block from Java 1.7.
 - It **catches exception and handles it**.
- finally:
 - This is optional block from Java 1.7.
 - It **defines code that executes regardless of exception**.
 - Eg: Close open files.

Syntax:

```
try {  
    // risky-code here  
}  
catch(Exception e) {  
    // handling-code here  
}  
finally {  
    // resource-to-be-closed-here  
}
```



- Without try..catch, below code would result in “ArithException: / by zero” exception:

demo.java

```
public class demo {
    public static void main(String[] args) {
        System.out.println(5/0); // ArithException: / by
        zero
    }
}
```

- Code handling “ArithmaticException: / by zero” exception:

demo.java

```
public class demo {
    public static void main(String[] args) {
        try {
            System.out.println(5/0);
        }
        catch (ArithmaticException e) {
            System.out.println("Oops! Problem.");
        }
        System.out.println("Ending demo");
    }
}
```

Output:

Oops! Problem.
Ending demo

- Code handling “ArithmaticException: / by zero” using try..catch..finally:

demo.java

```
public class demo {
    public static void main(String[] args) {
        try {
            System.out.println(5/0);
        }
        catch (ArithmaticException e) {
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("Ending demo");
        }
    }
}
```

Output:

```
/ by zero
Ending demo
```

13.2.5 Rules for try..catch..finally

① You cannot have a catch or finally without a try

```
void go() {
    Foo f = new Foo();
    f.foo();
    catch(FooException ex) { }
}
```

*NOT LEGAL!
Where's the try?*

② You cannot put code between the try and the catch

```
try {
    x.doStuff();
}
int y = 43;
} catch(Exception ex) { }
```

*NOT LEGAL! You can't put
code between the try and
the catch.*

③ A try MUST be followed by either a catch or a finally

```
try {
    x.doStuff();
} finally {
    // cleanup
}
```

*LEGAL because you
have a finally, even
though there's no catch.
But you cannot have a
try by itself.*

④ A try with only a finally (no catch) must still declare the exception.

```
void go() throws FooException {
    try {
        x.doStuff();
    } finally { }
}
```

*A try without a catch
doesn't satisfy the
handle or declare law*

13.2.6 Methods to print exception

Throwable class defines the following methods to print exception information:

Method	Printable format
printStackTrace()	Name of exception: Description Stack Trace Internally, default exception handler will use printStackTrace() method to print exception information to the console
toString()	Name of exception: Description
getMessage()	Description

Eg:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        try {  
            System.out.println(5/0);  
        }  
        catch (ArithmaticException e) {  
            // toString()  
            System.out.println("Output for: toString()");  
            System.out.println(e);  
            System.out.println(e.toString());  
            //printStackTrace()  
            System.out.println("Output for: printStackTrace()");  
            e.printStackTrace();  
            //getMessage()  
            System.out.println("Output for: getMessage()");  
            System.out.println(e.getMessage());  
        } } }
```

Output:

```
Output for: toString()  
java.lang.ArithmaticException: / by zero  
java.lang.ArithmaticException: / by zero  
Output for: printStackTrace()  
java.lang.ArithmaticException: / by zero  
at demo.main(demo.java:5)  
Output for: getMessage()  
/ by zero
```

13.2.7 Try with multiple catch blocks

You should take separate catch block for every exception type.

The order of catch block is very important:

- **Catch the child exception first and then parent**, else it shall result in error.
- Two catch blocks with the same exception are not allowed.

Handling multiple exceptions with common message:

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        try {
            FileReader r1 = new FileReader("abc.txt");
            BufferedReader r2 = new BufferedReader(r1);
            String line;
            while ( (line = r2.readLine()) != null ) {
                System.out.println(line);
            }
            System.out.println(5/0);
        }
        catch (Exception e) {
            System.out.println("Something went wrong!");
        } } }
```

Output:

Something went wrong!

Handling multiple exception with separate catch blocks:

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        try {
            FileReader r1 = new
            FileReader("abc.txt");
            BufferedReader r2 = new BufferedReader(r1);
            String line;
            while ( (line = r2.readLine()) != null ) {
                System.out.println(line);
            }
            System.out.println(5/0);
        }
        catch (FileNotFoundException e) {
            System.out.println("File abc.txt not found");
        }
        catch (ArithmaticException e) {
            System.out.println("You are dividing number with
0!");
        }
        catch (Exception e) {
            System.out.println("Something went wrong!");
        }
    }
}
```

Output:

File abc.txt not found

13.3 Java 1.7 Exception Handling Enhancement

With Java 1.7, below are the new enhancements in exception handling:

- Try with resources
- Multi-catch block

13.3.1 try-with resources

- Until Java 1.6 version, programmer used finally block to close resources like database connections or opened file as shown below:

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader("abc.txt"));
        }
        catch(IOException e) {
            System.out.println("File not found");
        }
        catch (Exception e) {
            System.out.println("Oops!");
        }
        finally {
            try {
                if (br != null)
                    br.close();
            }
            catch (IOException e) {
                System.out.println("File not found");
            } } } }
```

- Drawback: Programmer need to manually close the resources.
- try-with resources closes opened resources automatically on block end.

Syntax:

```
try(resource1; resource2; resource3,...) {
    code-here
}
catch(Exception) {
    code-here
}
```

Things to note:

- Multiple resources are separated with ";".
- Resources should be AutoCloseable(that implements java.lang.AutoClosable interface).
- Code that opens a file and close it as try block ends:

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        try(BufferedReader br = new BufferedReader(new
FileReader("abc.txt"))) {
            String line;
            while ( (line = br.readLine()) != null) {
                System.out.println(line);
            }
        }
        catch(IOException e) {
            System.out.println("File not found");
        } } }
```

- Within try block you can't perform reassignment of resource, as resources are final.
- Below code results into compile-time error:

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        try(FileReader r1=new FileReader("abc.txt")) {
            r1=new FileReader("mno.txt");
        } catch(IOException e) {
            System.out.println("File not found");
        }
    }
}
```

Error:

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

The resource r1 of a try-with-resources statement cannot be assigned

at demo.main(demo.java:5)

13.3.2 Multi-catch block

- Until Java 1.6 version, multiple exceptions cannot be caught in single catch block.
- Multi-catch block in Java 1.7 version allows you to write single catch block that can handle multiple exceptions.

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        int no1;
        int no2;
        try {
            no1=5;
            no2=0;
            System.out.println(no1/no2);
        }
        catch(NullPointerException | ArithmeticException
e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
java.lang.ArithmetricException: / by zero
at demo.main(demo.java:9)
```

13.4 Throwing exception

A programmer may have to purposely throw exception for conditions like:

- Age less than 0 or more than 120
- Invalid phone number
- Invalid email address
- Name containing non-alphabetical characters etc.

This is useful when you want to indicate that a particular condition should be treated as an exception.

13.4.1 throw keyword

- You can explicitly throw an exception using the throw keyword.

Syntax:

```
throw new ExceptionType("Error message");
```

Syntax explanation:

- **throw**: Use the throw keyword to throw the exception object.
 - **new**: Create exception object.
 - **ExceptionType**: Select correct exception classes such as RuntimeException, IOException, NullPointerException, IllegalArgumentException, etc.
 - **"Error message"**: Optionally provide an error message describing the exception.
- Consider below code that throws **ArithematicException** explicitly:

demo.java

```
public class demo {  
    public static int divide(int dividend, int divisor) {  
        if (divisor == 0) {  
            throw new ArithematicException("Cannot divide  
by zero.");  
        }  
    }  
}
```

```
    }
    return dividend/divisor;
}

public static void main(String[] args) {
    try {
        int result = divide(10, 0);
        System.out.println("Result: " + result);
    }
    catch (ArithmaticException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

Output:

Error: Cannot divide by zero

Note:

- throw keyword cannot be used to throw empty object. It will result in "NullPointerException".
- You cannot write any statement after throw keyword. It will result in "Unreachable statement".
- You can use throw keyword only for throwable types.

13.4.2 Checked & unchecked exception

- **Checked exception:**

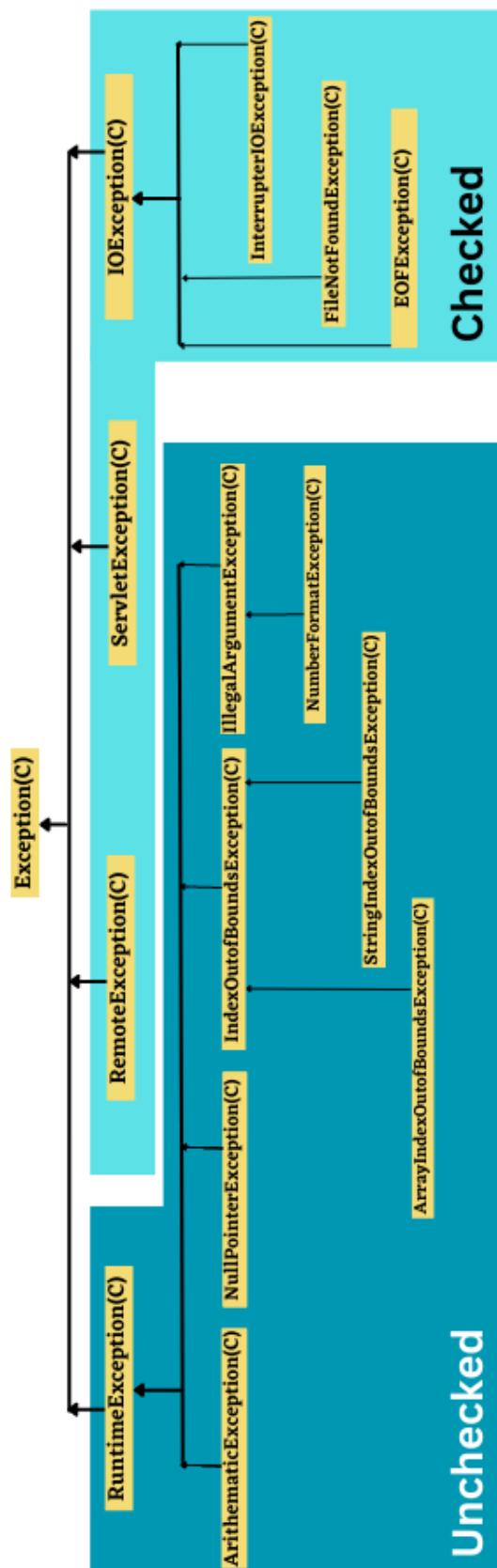
- Compiler checked exception for smooth execution are called checked exception
- Eg: IOException, FileNotFoundException, EOFException etc.
- Checked exception compulsory need to be handled either by try-catch or by throws keyword, otherwise it will result in compile-time error.
- **All exceptions are checked** (except RunTimeException and its child class & Error and its child class).

- **Unchecked exception:**

- The exceptions not checked by compiler are called unchecked exception.
- Eg: RuntimeException, ArithmeticException, ArrayIndexOutOfBoundsException etc.
- These exception may or may not be handled.
- **RunTimeException and its child classes, Error and its child classes are unchecked.**

Note:

Whether it is checked or unchecked, every exception occurs at runtime only.



13.4.3 FullyChecked v/s ParitallyChecked

FullyChecked:

- A checked exception is said to be fully checked, if and only if all its child classes also checked.
- Eg:
 - IOException
 - InterruptedException
 - FileNotFoundException
- In case of fully-check exception, you shall get compile-time error, if there is no chance of raising an exception in try block, and you still write catch block for it.
- Thus, below code will result in error:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        try {  
            System.out.println("Hello World!");  
        }  
        catch (FileNotFoundException e) {  
            System.out.println(e);  
        }  
    }  
}
```

Error:

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

Unreachable catch block for FileNotFoundException. This exception is never thrown from the try statement body at demo.main(demo.java:7)

PartiallyChecked:

- A checked exception is said to be partially checked, if and only if some of its child classes are unchecked.
- The only possible partially checked exception:
 - Exception
 - Throwable
- You won't get compile-time error if you write catch block for partially-checked exception & there is no chance of raising it in try block.
- Thus, below code will not result in error:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        try {  
            System.out.println("Hello World!");  
        }  
        catch (ArithmaticException e) {  
            System.out.println(e);  
        }  
    }  
}
```

Output:

Hello World!

13.4.4 User-defined exceptions

- User-defined exception are custom exceptions created by programmer to meet programming requirements.
- Eg:
 - InvalidAgeException
 - InvalidEmailException
 - InSufficientBalanceException
- Rules for creating custom exception:
 - Define customised exceptions as unchecked i.e extend RuntimeException but not Exception.
 - Inside every customised exception, super(arg) is needed to make description available to default exception handler.
 - You will need to explicitly throw custom exception using "throw" keyword.

Syntax:

```
class ExceptionName extends RuntimeException {  
    ExceptionName(String msg) {  
        super(msg);  
    }  
}
```

- Below code create custom exception named "InvalidAgeException":

demo.java

```
class InvalidAgeException extends RuntimeException {  
    InvalidAgeException(String msg) {  
        super(msg);  
    }  
}  
public class demo {  
    public static void main(String[] args) {
```

```
try {  
    int age = Integer.parseInt(args[0]);  
    if(age < 0 | age > 100) {  
        throw new InvalidAgeException("You seems have  
have entered invalid age!");  
    }  
    else {  
        System.out.println("Age: " + age + " is valid!");  
    }  
}  
catch (InvalidAgeException e) {  
    System.out.println(e.toString());  
}  
}
```

Output:

```
$ javac demo.java  
$ java demo 45  
Age: 45 is valid!  
$ java demo 450  
InvalidAgeException: You seems have have entered invalid  
age!
```

13.4.5 throws keyword

- Checked exceptions needs to be handled compulsorily, else it might result in compile-time error as below:

Error:

“Unreported exception XXX: must be caught or declared to be thrown”

- Below code results in **error** as FileNotFoundException exception needs to be handled:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        FileReader reader = new FileReader("abc.txt");  
    }  
}
```

Error:

Exception in thread "main" java.lang.Error: Unresolved compilation problems:
FileReader cannot be resolved to a type
FileReader cannot be resolved to a type
at demo.main(demo.java:5)

- There are 2 ways to handle above error:
 - Using the traditional try..catch block
 - Using throws keyword

- Using the traditional try..catch block:

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("abc.txt");
        }
        catch (FileNotFoundException e) {
            System.out.println(e);
        }
    }
}
```

- Using throws keyword:

- throws keyword delegates responsibility of exception handling to the caller (it maybe another method or JVM).

demo.java

```
import java.io.*;
public class demo {
    public static void main(String[] args) throws
FileNotFoundException {
    FileReader reader = new FileReader("abc.txt");
}
}
```

Important things to note for throws keyword:

- Throws keyword can be used for **methods and constructors**, but not for classes.
- You can use throws keyword only for throwable types.
- Throws keyword is **required only for checked exceptions**.
- Usage of throws keyword for **unchecked exceptions is of no use**.
- Throws keyword is required only for convenience of compiler however, **it does not prevent abnormal termination of program**.

13.4.6 Exception propagation

- Exception propagation is process by which an exception is passed from one method to another in the call stack until:
 - It is either caught and handled
 - Or causes the program to terminate if left uncaught
- When a method throws an exception but does not catch it, the exception is propagated up the call stack to the calling method.
- Consider below code where exception is handled by main() ultimately:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        try {  
            test1();  
        }  
        catch(Exception e) {  
            System.out.println("Caught exception:");  
            e.printStackTrace();  
        }  
    }  
}
```

```
public static void test1() {  
    test2();  
}  
  
public static void test2() {  
    test3();  
}  
  
static void test3() {  
    int arr[] = new int[2];  
    System.out.println(arr[9]);  
}  
}
```

Output:

Caught exception:
java.lang.ArrayIndexOutOfBoundsException: Index 9 out of
bounds for length 2
at demo.test3(demo.java:20)
at demo.test2(demo.java:16)
at demo.test1(demo.java:13)
at demo.main(demo.java:5)

Explanation:

- When method3() is called, it tries to access an element in the array with an index that is out of bounds.
- This results in an ArrayIndexOutOfBoundsException.
- Since method3() does not catch this exception, it is propagated up the call stack through method2() to method1() and finally caught in the main() method.

13.5 Top 10 Exceptions

Based on who is raising an exception, all exceptions are divided into 2 categories:

JVM exceptions

- These are raised automatically by JVM, whenever a particular event occurs.

Programmatic Exceptions

- These are raised explicitly either by programmer or API developer to handle exceptional situations.

Below are some classifications of exception:

JVM Exceptions	Programmatic Exceptions
ArrayIndexOutOfBoundsException	IllegalArgumentException
NullPointerException	NumberFormatException
ClassCastException	IllegalStateException
StackOverflowException	AssertionException
	NoClassDefFoundError
	ExceptionInInitializerError

ArrayIndexOutOfBoundsException

- Parent Class: **RuntimeClassException**
- Type: **Unchecked Exception**
- Raised By: **JVM**
- Reason: Trying to access array element with out of range index.
- Eg:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        int[] arr = new int[4];  
        System.out.println(arr[0]);  
        System.out.println(arr[5]);  
    }  
}
```

Output:

```
0  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: Index 5 out  
of bounds for length 4  
at demo.main(demo.java:7)
```

NullPointerException

- Parent Class: **RuntimeClassException**
- Type: **Unchecked Exception**
- Raised By: **JVM**
- Reason: Trying to perform any operation on null.
- Eg:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        String s=null;  
        System.out.println(s.length());  
    }  
}
```

Output:

Exception in thread "main" java.lang.NullPointerException
at demo.main(demo.java:6)

ClassCastException:

- Parent Class: **RuntimeClassException**
- Type: **Unchecked Exception**
- Raised By: **JVM**
- Reason: Trying to type cast parent object to child type.
- Eg:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        Object o = new Object();  
        String s = (String)o;  
    }  
}
```

Output:

Exception in thread "main" java.lang.ClassCastException...

StackOverflowException:

- Parent Class: **Error**
- Type: **Unchecked Exception**
- Raised By: **JVM**
- Reason: Trying to perform recursive method class.

demo.java

```
public class demo {  
    public static void test1() {  
        test2();  
    }  
    public static void test2() {  
        test1();  
    }  
    public static void main(String[] args) {  
        test1();  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.StackOverflowError  
at demo.test2(demo.java:7)  
at demo.test1(demo.java:4)  
at demo.test2(demo.java:7)  
..  
..  
..  
..  
..
```

ClassNotFoundException:

- Parent Class: **Error**
- Type: **Unchecked Exception**
- Raised By: **JVM**
- Reason: Raised when JVM unable to be find required ".class" file.
- Eg:

Code:

```
$ java Demo  
Error: Could not find or load main class Demo  
Caused by: java.lang.ClassNotFoundException: Demo
```

ExceptionInInitialiserError:

- Parent Class: **Error**
- Type: **Unchecked Exception**
- Raised By: **JVM**
- Reason: Raised when exception occurs while executing static variable assignments and static blocks.
- Eg:

demo.java

```
public class demo {  
    static int i = 10/0;  
}
```

Output:

Error: Main method not found in class demo, please define the main method as:
public static void main(String[] args)
or a JavaFX application class must extend
javafx.application.Application

IllegalArgumentException

- Parent Class: **RuntimeException**
- Type: **Unchecked Exception**
- Raised By: **Programmer or API developer**
- Reason: Indicate that a method has been invoked with illegal argument.
- Eg: The valid range of thread priorities is 1-10. Invalid thread priority will result in **IllegalArgumentException** exception:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        Thread t1 = new Thread();  
        t1.setPriority(7);  
        t1.setPriority(15);  
    }  
}
```

Output:

```
Exception           in          thread          "main"  
java.lang.IllegalArgumentException  
at java.base/java.lang.Thread.setPriority(Thread.java:1141)  
at demo.main(demo.java:7)
```

NumberFormatException:

- Parent Class: **IllegalArgumentException**
- Type: **Unchecked Exception**
- Raised By: **Programmer or API developer**
- Reason: Trying to convert string to number and the string is not properly formatted.

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        int i1 = Integer.parseInt("67");  
        int i2 = Integer.parseInt("one");  
    }  
}
```

Output:

```
Exception          in          thread          "main"  
java.lang.NumberFormatException:                   at  
java.base/java.lang.Integer.parseInt(Integer.java:652)  
at java.base/java.lang.Integer.parseInt(Integer.java:770)  
at demo.main(demo.java:6)
```

IllegalThreadStateException:

- Parent Class: **RuntimeException**
- Type: **Unchecked Exception**
- Raised By: **Programmer or API developer**
- Reason: Indicate that a method has been invoked at a wrong time.

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        Thread t1 = new Thread();  
        t1.start();  
        t1.start(); // Output: IllegalThreadStateException  
    }  
}
```

AssertionError

- Parent Class: **Error**
- Type: **Unchecked Exception**
- Raised By: **Programmer or API developer**
- Reason:
 - Used for debugging and testing purposes.
 - Indicates that certain condition is true during execution.
 - If the condition is false, the assert statement will throw an **AssertionError**.
 - To enable the assertion checks, JVM with the **-ea** or **-enableassertions** command-line option.
- Eg:

demo.java

```
public class demo {  
    public static void main(String[] args) {  
        int num = 14;  
        // Example 1: Simple assertion check  
        assert num >= 0; // No AssertionError as num > 0  
        // Example 2: Assertion & error message  
        int age = -5;  
        assert age >= 0 : "Age cannot be negative";  
    }  
}
```

Output:

```
$ javac demo.java  
$ java -ea demo  
Exception in thread "main" java.lang.AssertionError: Age  
cannot be negative
```

Don't focus on the pain ->



Focus on progress ->



- Dwayne Johnson

14. Collection Framework

14.1 Introduction

In this chapter, we shall see:

- Collection interface
- Collections class
- Entire collection framework

14.1.1 Drawback of arrays

- **Size:** Arrays once created cannot be increased or decreased in size.
- **Homogeneous data:** Array can hold only homogeneous data/
Eg:

Code:

```
Student[] s = new Student[100];
s[0] = new Student(); ✓
s[1] = new Customer(); ✗
```

Only way to solve this problem is by using **object** arrays:

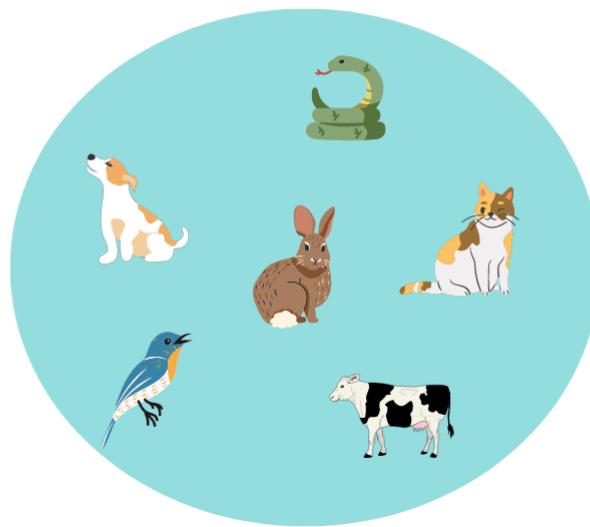
Code:

```
Object[] o = new Object[100];
o[0] = new Student(); ✓
o[1] = new Customer(); ✓
```

- **No data structure:** No standard data structure, hence builtin method support not available.

14.1.2 What is Collection?

- Collection is a group of individual objects as a single entity.

Collection of objects

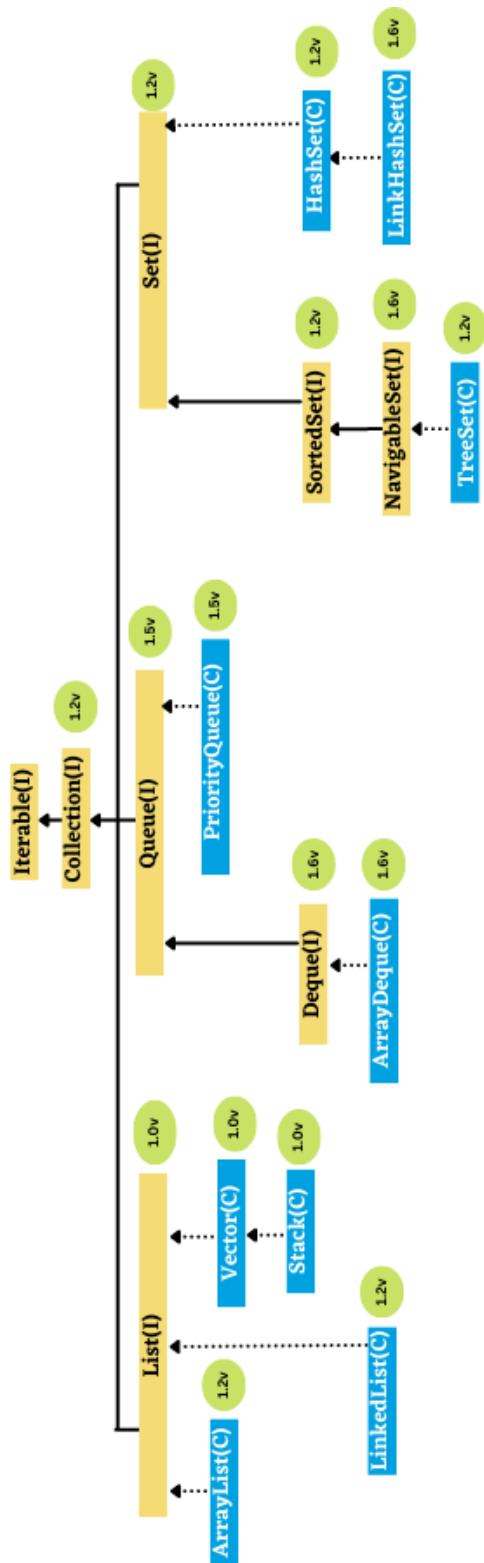
- Advantages of collection over array:
 - **Size:** Collections are growable in nature i.e based on our requirement we can increase or decrease it's size.
 - **Hetrogeneous data:** Collections can hold both homogeneous and hetrogeneous elements.
 - **Data structures:** Every collection class is implemented on some standard data structure.

14.1.3 Difference between Array & Collection

	Array	Collection
Size	Fixed	Flexible
Memory usage	Not very good	Highly recommended
Performance	Highly recommended	Not recommended
Data	Homogeneous data allowed	Hetrogeneous & homogeneous data allowed
Data structure	Not available	Available
Data types	Can hold primitives and object	Can hold only objects but not primitives

14.1.4 What is Collection Framework?

- Collection framework defines several **classes** and **interfaces** which can be used to group objects as a single entity.



14.1.5 Collection Interface

- Collection interface is **root interface of collection framework**.
- It defines common methods applicable for any collection objects.
- Important methods:

1. Add single object:

Syntax:

```
boolean add(Object o)
```

2. Add multiple object:

Syntax:

```
boolean addAll(Collection c)
```

3. Remove single object:

Syntax:

```
boolean remove(Object o)
```

4. Remove multiple object:

Syntax:

```
boolean removeAll(Collection c)
```

5. Remove all objects except those present in collection "c":

Syntax:

```
boolean retainAll(Collection c)
```

6. Clear all objects:

Syntax:

```
void clear()
```

7. Check if the object is present:

Syntax:

```
boolean contains(Object o)
```

8. Check if multiple object are present:

Syntax:

```
boolean containsAll(Collection c)
```

9. Check if collection is empty:

Syntax:

```
boolean isEmpty()
```

10. Check size of collection:

Syntax:

```
int size();
```

11. Convert collection to array:

Syntax:

```
Object[] toArray();
```

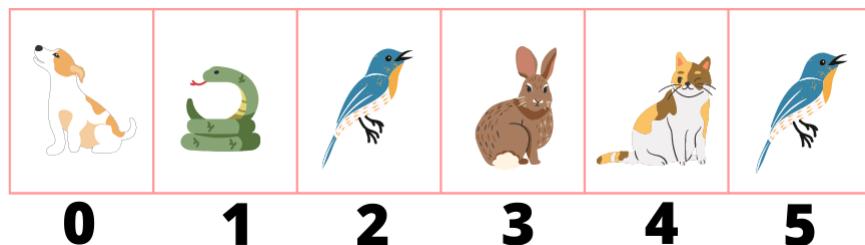
12. To iterate over collection:

Syntax:

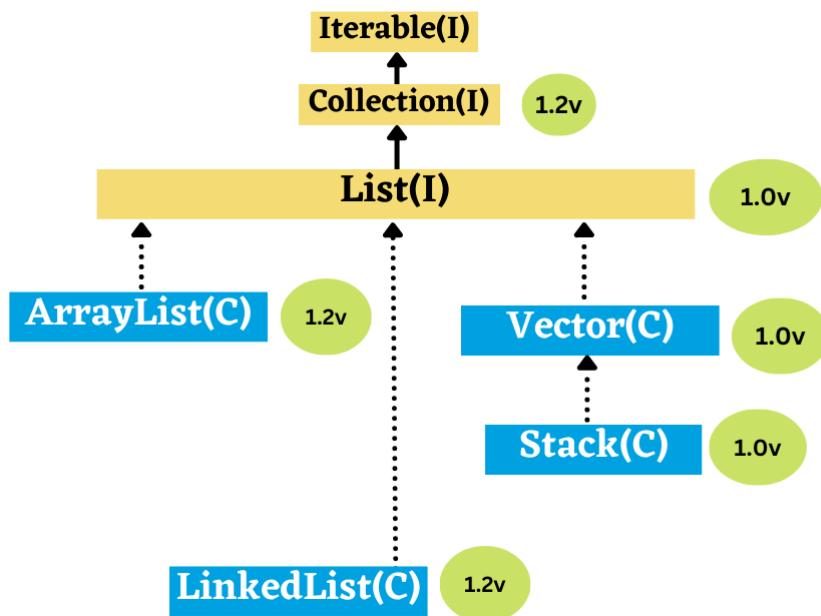
```
Iterator iterator()
```

14.2 List

- List is **ordered & indexed** collection of **heterogeneous objects** wherein **duplicate objects** are allowed.



- It is the child interface of Collection interface.



List interface methods

- Add an object at specific index position:

Syntax:

```
void add(int index, Object o)
```

- Add multiple objects at specific index position:

Syntax:

```
boolean addAll(int index, Collection c)
```

- Get object at a specific index position:

Syntax:

```
object get(int index)
```

- Remove an object at a specific index position:

Syntax:

```
object remove(int index)
```

- Replace an object at a specific index position:

Syntax:

```
object set(int index, Object new)
```

- Get index position of an object:

Syntax:

```
int indexOf(Object o)
```

- Get last index position of an object:

Syntax:

```
int lastIndexOf(Object o)
```

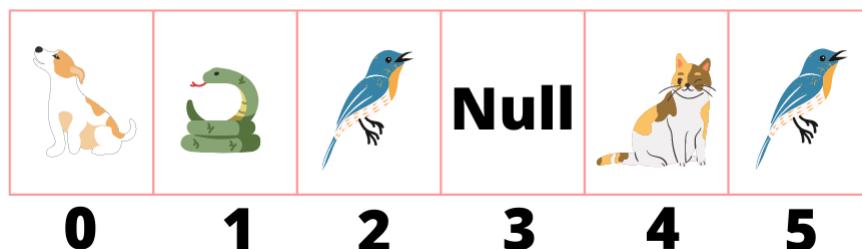
- Iterate over list collection:

Syntax:

```
ListIterator listIterator();
```

14.2.1 ArrayList

- ArrayList are **ordered & indexed** collection of **hetrogenous objects** wherein **duplicate objects & null** is allowed.



- ArrayList class implements List interface.

Ways of initializing ArrayList

- Creating am empty ArrayList:

Syntax:

```
// To create ArrayList of hetrogenous elements:  
ArrayList list = new ArrayList();
```

```
// To create ArrayList of homogenous elements:  
ArrayList<ClassName> list = new ArrayList<ClassName>();
```

Eg:

Code:

```
ArrayList list = new ArrayList();  
ArrayList<String> list = new ArrayList()<String>;
```

- ArrayList capacity:**

- Above syntax will create ArrayList with **default capacity of 10**.
- Once 10 objects are added, then a new array list object will be created with new capacity as shown below:

New capacity = (current capacity * 3/2) + 1

- Hence ArrayList capacity would be 10, 16, 25, 38,....

Note:

There is no method to check default initial capacity.

- Creating an empty ArrayList with initial capacity:

Syntax:

// To create ArrayList of heterogeneous elements:

```
ArrayList numbers = new ArrayList(capacity);
```

// To create ArrayList of homogenous elements:

```
ArrayList<ClassName> list = new ArrayList<ClassName>(capacity);
```

Eg:

Code:

```
ArrayList numbers = new ArrayList(20);
```

```
ArrayList<Integer> list = new ArrayList<Integer>(20);
```

- Initialization with data:
 - Using **Arrays.asList()** method:

Code:

```
import java.util.*;  
String[] namesArray = {"Jim", "Jane", "Alice"};  
ArrayList<String> namesList = new ArrayList<>(Arrays.asList(namesArray));
```

- Using **List.of** method:

Code:

```
ArrayList<String> colors = new ArrayList<>(List.of("Red", "Green", "Blue"));
```

- Using "**var**" and "**List.of**":

Code:

```
var fruits = new ArrayList<>(List.of("Apple", "Banana", "Orange"));
```

Where ArrayList is best choice:

- For retrieval operations, because ArrayList implements RandomAccess interface.

Where ArrayList is worst choice:

- For insertion or deletion operation in the middle.
- For this, LinkedList is best choice.

Example:

- Java program to create an empty ArrayList and perform add, remove operation and display it's size:

Test.java

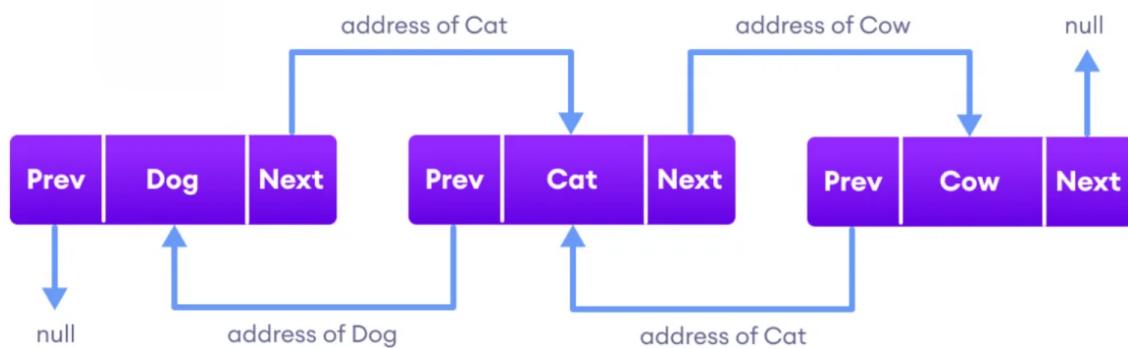
```
import java.util.ArrayList;
class Test {
    public static void main(String[] args) {
        ArrayList l1 = new ArrayList();
        l1.add("Jack");
        l1.add(10);
        l1.add('A');
        l1.add(null);
        l1.add("Jim");
        System.out.println(l1);
        l1.remove("Jack");
        System.out.println(l1);
        l1.add(2,"Jimmy");
        System.out.println(l1);
        System.out.println("ArrayList size: "+l1.size());
        boolean ans = l1.contains("Jack");
        System.out.println("Jack is present in ArrayList:
"+ans);
    }
}
```

Output:

```
[ Jack, 10, A, null, Jim ]
[ 10, A, null, Jim ]
[ 10, A, Jimmy, null, Jim ]
ArrayList size: 5
Jack is present in ArrayList: false
```

14.2.2 LinkedList

- LinkedList are **ordered & indexed** collection of **heterogeneous objects** wherein **duplicate objects & null** is allowed.



- LinkedList class implements List interface.
- LinkedList class is used to **develop stacks and queues**.

Difference between ArrayList and LinkedList

	ArrayList	LinkedList
Best choice for	Retrieval operation	Insertion/Deletion operation in the middle
Worst choice for	Insertion/Deletion operation in the middle	Retrieval operation
Object storage order	Consecutive memory location	Not stored in consecutive memory locations
Capacity	Allows mentioning capacity	Does not allow mentioning capacity

Ways of initializing LinkedList

- Creating an empty LinkedList:

Syntax:

```
// To create LinkedList of heterogeneous elements:
```

```
LinkedList numbers = new LinkedList();
```

```
// To create LinkedList of homogenous elements:
```

```
LinkedList<ClassName> list = new  
LinkedList<ClassName>();
```

Eg:

Code:

```
LinkedList numbers = new LinkedList();
```

```
LinkedList<Integer> list = new LinkedList<Integer>();
```

- Initialization with data:

- Using **Arrays.asList()** method:

Code:

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

```
String[] namesArray = {"Jim", "Jane", "Alice"};
```

```
LinkedList<String> namesList = new
```

```
LinkedList<>(Arrays.asList(namesArray));
```

- Using **List.of** method:

Code:

```
LinkedList<String> colors = new
```

```
LinkedList<>(List.of("Red", "Green", "Blue"));
```

- Using "var" and "List.of":

Code:

```
var fruits = new LinkedList<>(List.of("Apple", "Ba-  
nana", "Orange"));
```

Methods specific to LinkedList:

- Add first member to LinkedList:

Syntax:

```
void addFirst(Object o)
```

- Add last member to LinkedList:

Syntax:

```
void lastFirst(Object o)
```

- Get first member of LinkedList:

Syntax:

```
Object getFirst()
```

- Get last member of LinkedList:

Syntax:

```
Object getLast()
```

- Remove first member of LinkedList:

Syntax:

```
Object removeFirst()
```

- Remove last member of LinkedList:

Syntax:

```
Object removeLast()
```

Example

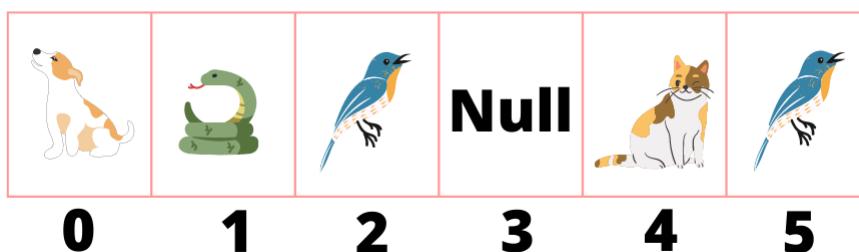
Create an empty LinkedList and perform add, remove operation and display it's size:

Test.java

```
import java.util.LinkedList;
class Test {
    public static void main(String[] args) {
        LinkedList l1 = new LinkedList();
        l1.add("Jack");
        l1.add(null);
        l1.remove("Jack");
        l1.add(2,"Jimmy");
        int count = l1.size();
        l1.set(0,"Jack Sparrow");
        l1.addFirst("First");
        l1.addLast("Last");
        l1.removeFirst();
        l1.removeLast();
    }
}
```

14.2.3 Vector

- Vector are **ordered & indexed** collection of **heterogeneous objects** wherein **duplicate objects & null** is allowed.



- Like ArrayList, Vector is best choice for retrieval operation.

Difference between ArrayList and Vector

ArrayList	Vector
Methods are non-synchronized	Methods are synchronized
At a time, multiple threads are allowed to operate on ArrayList Object and hence ArrayList is not thread safe	At a time, only one thread is allowed to operate on Vector Object and it is thread safe
Threads are not required to wait to operate on ArrayList, hence relatively performance is high	Threads are required to wait to operate on Vector Object and hence relatively performance is low .
Introduced in 1.2 version and it is non-legacy class	Introduced in 1.0 version and it is legacy class

Ways of initializing Vector

- Creating an empty Vector:

Syntax:

```
// To create Vector of heterogeneous elements:  
Vector list = new Vector();  
  
// To create Vector of homogenous elements:  
Vector<ClassName> list = new Vector<ClassName>();
```

Eg:

Code:

```
Vector list = new Vector();  
Vector<String> list = new Vector<String>();
```

- **Vector capacity:**

- Above syntax will create Vector with **default capacity of 10**.
- Once 10 objects are added, then a new Vector object will be created with new capacity as shown below:

New capacity = (current capacity * 2)

- Creating an empty Vector with initial capacity:

Syntax:

```
// To create Vector of heterogeneous elements:  
Vector v1 = new Vector(capacity);  
  
// To create Vector of homogenous elements:  
Vector<ClassName> list = new Vector<ClassName>(capacity);
```

Eg:

Code:

```
Vector numbers = new Vector(20);  
Vector<Integer> list = new Vector<Integer>(20);
```

- Initialization with data:

- Using **Arrays.asList()** method:

Code:

```
import java.util.*;  
String[] namesArray = {"Jim", "Jane", "Alice"};  
Vector<String> namesList = new Vector<>(Arrays.asList(namesArray));
```

- Using **List.of** method:

Code:

```
Vector<String> colors = new Vector<>(List.of("Red",  
"Green", "Blue"));
```

- Using "var" and "List.of":

Code:

```
var fruits = new Vector(List.of("Apple", "Banana",  
"Orange"));
```

Methods specific to Vector:

- Add member to Vector:

Syntax:

```
void addElement(Object o)
```

- Remove element at specific index of Vector:

Syntax:

```
void removeElementAt(int index)
```

- Remove all member of Vector:

Syntax:

```
void removeAllElements()
```

- Retrieve element at specific index of Vector:

Syntax:

```
Object elementAt(int index)
```

- Retrieve first element of Vector:

Syntax:

```
Object firstElement()
```

- Retrieve last element of Vector:

Syntax:

```
Object lastElement()
```

- Get Vector size:

Syntax:

```
int size()
```

- Get Vector capacity:

Syntax:

```
int capacity()
```

Example Program**Program to display vector capacity functioning:****Test.java**

```
import java.util.*;  
public class Test {  
    public static void main(String[] args) {  
        Vector v1 = new Vector(3);  
        v1.add("Apple");  
        v1.addElement("Mango");  
        v1.add(2,"Orange");  
        System.out.println(v1.capacity());  
        System.out.println(v1.capacity());  
        v1.add("Banana");  
        System.out.println(v1.capacity());  
    }  
}
```

Output:

```
[Apple, Mango, Orange]  
3  
[Apple, Mango, Orange, Banana]  
6
```

14.2.4 Stack

- Stack class is child class of Vector class.
- Stack is designed for **Last In First Out Order(LIFO)**
- Stack are **ordered & indexed** collection of **heterogeneous objects** wherein **duplicate objects & null** is allowed.



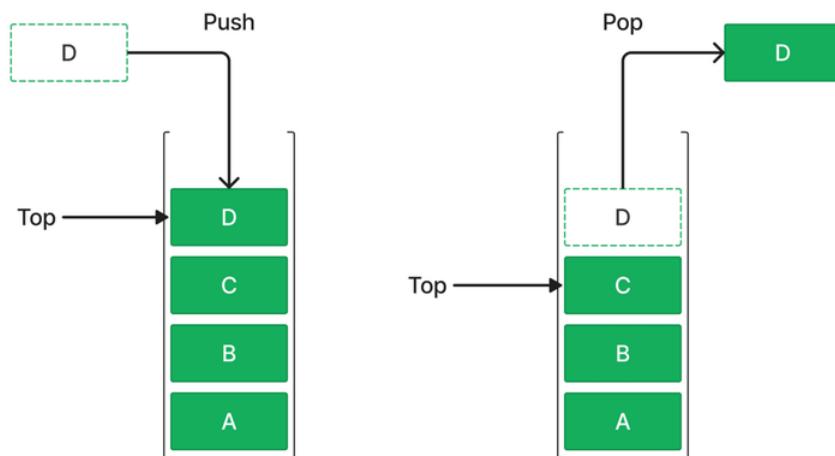
Real Life Example: Stack of Books

- Creating an empty Stack:

Code:

```
Stack s = new Stack();
```

Stack operations



Stack

Stack methods

- Inserting an object into stack:

Syntax:

```
Object push(Object obj);
```

- Remove and return top of stack:

Syntax:

```
Object pop();
```

- Return top of stack without removal of object:

Syntax:

```
Object peak();
```

- Search an object and return it's offset, if object is not present then return -1.



Syntax:

```
int search(Object obj);
```

- Check if stack is empty or not:

Syntax:

```
boolean empty();
```

Example Program:

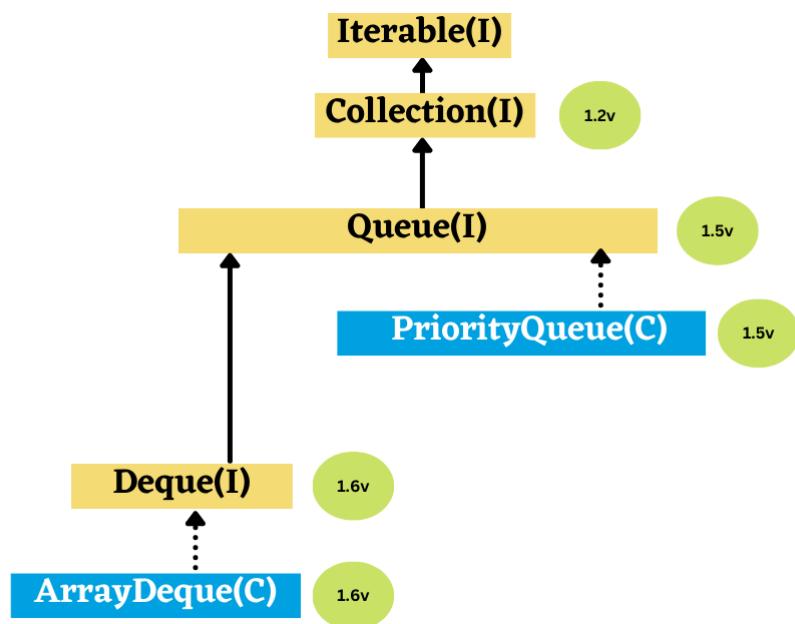
Display stack operations:

Test.java

```
import java.util.*;  
public class Test {  
    public static void main(String[] args) {  
        Stack s = new Stack<>();  
        s.push("Apple");  
        s.push("Chikoo");  
        s.push("Chikoo");  
        s.pop();  
        int pos = s.search("Banana");  
        Object val2 = s.peek();  
        System.out.println(s.empty());  
    }  
}
```

14.3 Queue

- Queue is an ordered list which enables **insert operations** to be performed at one end (**i.e REAR**) and **delete operations** to be performed at another end (**i.e FRONT**).
- Queue is referred to be as **First In First Out (FIFO)** list.
- Eg: People waiting in line in library form a queue.



- Queue is the child interface of Collection interface.

Queue interface specific methods

- Add object to queue:

Syntax:

```
boolean offer(Object o);
```

- Return head of the queue. If queue is empty, it will return null.

Syntax:

```
Object peek();
```

- Return head of the queue. If queue is empty, error is returned.

Syntax:

```
Object element();
```

- Return and remove head of the queue. If queue is empty, it will return null.

Syntax:

```
Object poll();
```

- Return and remove head of the queue. If queue is empty, error is returned.

Syntax:

```
Object remove();
```

- Check whether queue is empty or not:

Syntax:

```
boolean isEmpty();
```

14.3.1 Comparator & Comparable

Comparable

- Comparable interface is present in **java.lang** package.
- It contains only **compareTo()** method:

Syntax:

```
int compareTo(Object obj)
```

Eg:

Code:

```
int value = obj1.compareTo(obj2)
```

where value can be:

- 1 if and only if obj1 comes before obj2.
- +ve no if and only if obj1 comes after obj2.
- 0 if and only if obj1 and obj2 are equal.

Eg:

Test.java

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

Output:

```
-1  
25  
0
```

Comparator:

- The Comparator interface is part of the **java.util** package.
- It defines **custom ordering of objects**.
- It provides a way to compare two objects & find order.
- It contains a single method **compare()**, that takes two objects and returns an integer indicating their order.
- Eg: Java program to create custom comparator:

Test.java

```
import java.util.*;  
class Custom implements Comparator {  
    public int compare(Object o1, Object o2) {  
        Integer i1 = (Integer)o1;  
        Integer i2 = (Integer)o2;  
        if (i1 > i2)  
            return 1;  
        else if (i1 < i2)  
            return -1;  
        else  
            return 0;  
    } }  
public class Test {  
    public static void main(String[] args) {  
        Custom c = new Custom();  
        int value = c.compare(64,134);  
        if (value != 0)  
            if (value == 1)  
                System.out.println("First number is greater");  
            else  
                System.out.println("2nd number is greater");  
    } }
```

Output:

2nd number is greater

Difference between Comparable and Comparator

Comparable	Comparator
Present in java.lang package	Present in java.util package
Contains compareTo() method	Contains compare() & equals() method
Provides default natural sorting order	Allows sorting customisation

14.3.2 PriorityQueue

The PriorityQueue class implements Queue interface.

It provides a **priority-based ordering** of elements.

Elements are ordered as per **natural order** or a **custom comparator** that you can specify.

Hospital Emergency Queue

Features:

- The priority can be either **default natural sorting order or customised** sorting order defined by comparator.
- It is **resizable array**.
- **Duplicates are not allowed**.
- **Insertion order is not preserved**.
- With default natural sorting order, **homogenous** objects are **allowed**.
- If we are defining our own sorting by comparator then objects can be **hetrogenous**.
- Null insertion is **not allowed**.

Creating PriorityQueue

- Creating an empty PriorityQueue:

Syntax:

```
PriorityQueue queue = new PriorityQueue();
```

Capacity:

- Default initial capacity of 11 elements.
- Once 11 objects are added, then a new array list object will be created with new capacity as shown below:

New capacity = current capacity * 2

- Creating an empty PriorityQueue with initial capacity:

Syntax:

```
PriorityQueue queue = new PriorityQueue(int capacity);
```

Eg:

Code:

```
PriorityQueue queue = new PriorityQueue(20);
```

- Creating a PriorityQueue with custom capacity and Comparator:

Syntax:

```
PriorityQueue queue = new PriorityQueue(int capacity,  
Comparator c);
```

Example Program:

PriorityQueue with custom comparator that would add elements in descending order:

Test.java

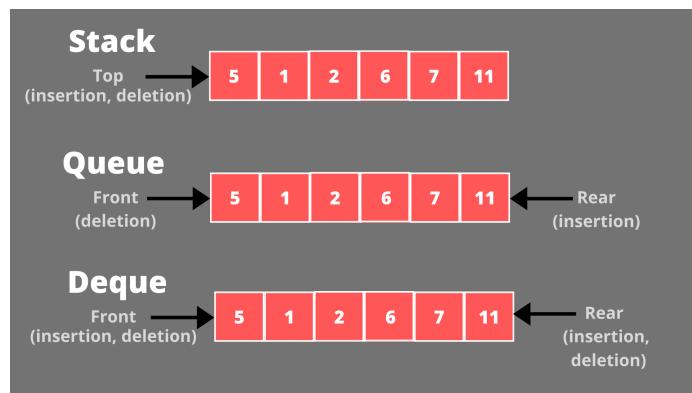
```
import java.util.*;  
class Custom implements Comparator<Integer>{  
    public int compare(Integer num1, Integer num2) {  
        return num2.compareTo(num1);  
    } }  
class CustomComparatorExample {  
    public static void main(String[] args) {  
        PriorityQueue<Integer> pq = new PriorityQueue<>(new  
Custom());  
        pq.offer(10);  
        pq.offer(5);  
                pq.offer(12);  
        while (!pq.isEmpty()) {  
            int element = pq.poll();  
            System.out.println(element);  
        } } }
```

Output:

```
10  
5
```

14.3.3 Deque

- The Deque interface **adds & remove elements from both ends** of queue.
- Deque can be used as a **stack** or a **queue**.
- Stack supports LIFO** operation, and **Queue supports FIFO**.
- Deque supports both LIFO and FIFO** and hence the acronym "**double ended queue**".



Deque specific methods

- Insert object to deque:

Syntax:

```
boolean offer(Object o)
```

- Inserts object at front & return true/false:

Syntax:

```
boolean offerFirst(Object o)
```

- Inserts object at last & return true/false:

Syntax:

```
boolean offerLast(Object o)
```

- Return first object & remove:

Syntax:

```
Object pollFirst()
```

- Return last object & remove:

Syntax:

Object pollLast()

- Return first object but not remove:

Syntax:

Object peekFirst()

- Return last object but not remove:

Syntax:

Object peekLast()

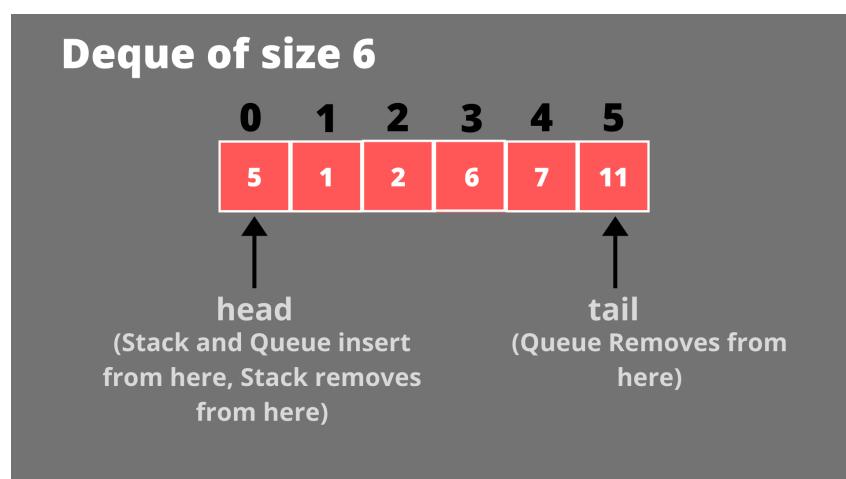
- Remove & return object from head:

Syntax:

Object remove()

14.3.4 ArrayDeque

- ArrayDeque class **implements the Deque and the Queue interface**.
- It uses **head** and **tail** where, **head takes care of insertion & deletion from front**, and the **tail takes care of insertion & deletion from end**.



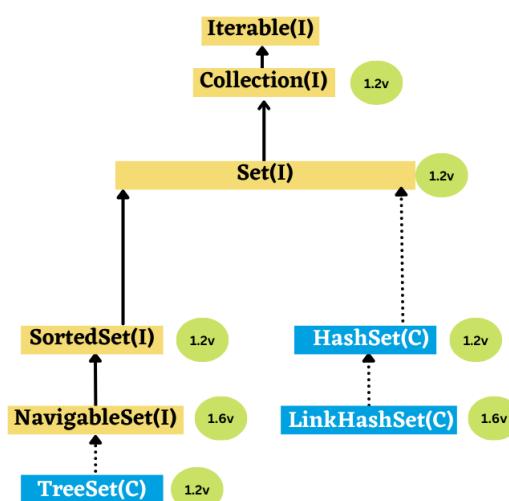
Example Program: Display ArrayDeque operations:

Test.java

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        Deque<Integer> queue = new ArrayDeque<Integer>();
        queue.offer(13);
        queue.offerFirst(12);
        queue.offerLast(10);
        int value1 = queue.peekFirst();
        int value2 = queue.peekLast();
        queue.pollFirst();
        queue.pollLast();
    }
}
```

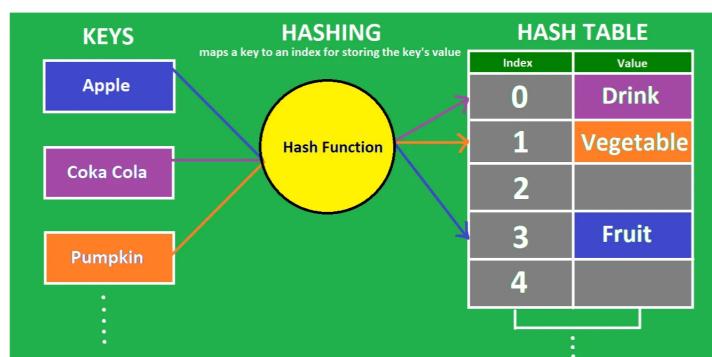
14.4 Set

- Set interface is **unordered** collection of **heterogeneous** objects wherein **duplicates are not allowed**.
- Set interface is child interface of collection.
- Set interface supports only collection interface methods.



14.4.1 HashSet

- HashSet class is **unordered** collection of **heterogeneous** objects wherein **duplicates are not allowed & null insertion is possible only once**.
- HashSet class implements Set interface.
- Best choice for **search** operation.
- Below is an example of **hash table**.



Ways of initializing HashSet

- Creating an empty HashSet:

Syntax:

```
HashSet s1 = new HashSet();
```

- **HashSet capacity & fill ratio:**

- Above syntax will create HashSet with **default capacity of 16**.
- Default fill ratio is **0.75**: Fill ratio is after filling 16th element, when 17th element is added, it's new capacity should be increased by 75%. Fill ratio is also called Load factor

- Creating am empty HashSet with initial capacity:

Syntax:

```
HashSet h = new HashSet(int initialcapacity);
```

- Creating an empty HashSet with initial capacity and fill ratio:

Syntax:

```
HashSet h = new HashSet(int initialcapacity, float fillratio);
```

Eg:

Code:

```
HashSet h = new HashSet(1000, 0.9);
```

Example Program

Java program to display HashSet operations:

Test.java

```
import java.util.*;  
class one {  
    public static void main(String[] args) {  
        HashSet set = new HashSet();  
        set.add("Apple");  
        set.add("Mango");  
        set.add("Apple");  
        System.out.println(set);  
        set.remove("Mango");  
        System.out.println(set);  
    }  
}
```

Output:

```
[Apple, Mango]
```

```
[Apple]
```

14.4.2 LinkedHashSet

- LinkedHashSet is child class of HashSet.
- It is **exactly same as HashSet**, except:

HashSet	LinkedHashSet
Datastructure is HashTable	Datastructure is combination of LinkedList + HashTable
Insertion order not preserved	Insertion order preserved
Introduced in 1.2 version	Introduced in 1.4 version

- Used to develop **cache based applications where duplicates are not allowed and insert order is preserved**.

Example Program:

Display LinkedHashSet operations:

Test.java

```
import java.util.*;  
class Test {  
    public static void main(String[] args) {  
        HashSet set = new LinkedHashSet();  
        set.add("Banana");  
        set.add("Apple");  
        set.add("Apple");  
        set.remove("Mango");  
        System.out.println(set);  
    }  
}
```

Output:

[Banana, Apple]

14.4.3 SortedSet

- SortedSet interface is child interface of **Set** interface.
- Represents **objects as per some sorting order without duplicates**.

SortedSet specific methods

- Return first element of SortedSet:

Syntax:

```
Object first()
```

- Return last element of SortedSet:

Syntax:

```
Object last()
```

- Return SortedSet whose elements are less than object

Syntax:

```
SortedSet headSet(Object o)
```

- Return SortedSet whose elements are greater than object

Syntax:

```
SortedSet tailSet(Object o)
```

- Return SortedSet whose elements are \geq obj1 and \leq obj2.

Syntax:

```
SortedSet subSet(Object o1, Object o2)
```

14.4.4 NavigableSet

- NavigableSet interface implements SortedSet interface.
- It defines several **methods for navigation purposes**.

NavigableSet specific methods

- Return highest element which is $\leq e$:

Syntax:

Object floor(e)

- Return highest element which is $< e$:

Syntax:

Object lower(e)

- Return lowest element which is $\geq e$:

Syntax:

Object ceiling(e)

- Return lowest element which is $> e$:

Syntax:

Object higher(e)

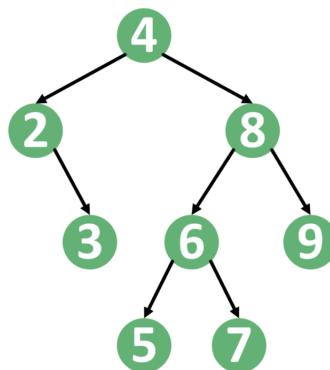
- Return Navigable set in reverse order:

Syntax:

NavigableSet descendingSet()

14.4.5 TreeSet

- TreeSet are **unordered** collection of **heterogenous objects** wherein **duplicate objects & null is not allowed**.
- TreeSet class implements NavigableSet interface.
- Objects are inserted based on some sorting order.** Order maybe natural or customised sorting order.
- Underlying data structure is balanced tree.



Ways of initializing TreeSet

- Creating an empty TreeSet:

Syntax:

```
TreeSet t1 = new TreeSet();
```

- Creating an empty TreeSet with custom Comparator:

Syntax:

```
TreeSet t = new TreeSet(Comparator c);
```

Example Program:

Java program to display TreeSet operations:

Test.java

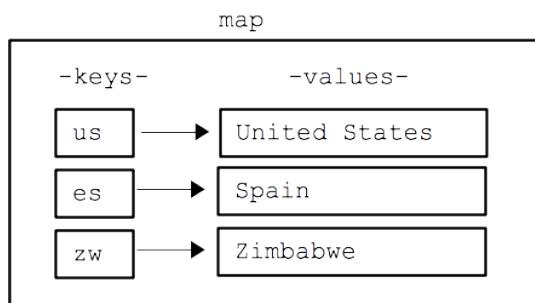
```
import java.util.*;  
class Test {  
    public static void main(String[] args) {  
        Integer[] nos = { 12,3,10,15,23,30,1,3,40 };  
        TreeSet t1 = new TreeSet(Arrays.asList(nos));  
        System.out.println(t1);  
        System.out.println(t1.first());  
        System.out.println(t1.last());  
        System.out.println(t1.headSet(34));  
        System.out.println(t1.tailSet(34));  
        System.out.println(t1.subSet(5,34));  
        System.out.println(t1.descendingSet());  
        System.out.println(t1.lower(23));  
        System.out.println(t1.higher(23));  
    }  
}
```

Output:

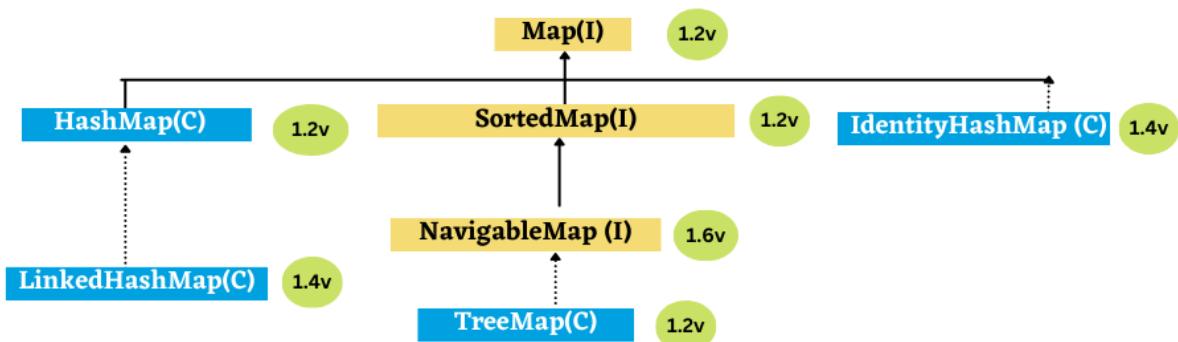
```
[1, 3, 10, 12, 15, 23, 30, 40]  
1  
40  
[1, 3, 10, 12, 15, 23, 30]  
[40]  
[10, 12, 15, 23, 30]  
[40, 30, 23, 15, 12, 10, 3, 1]  
15  
30
```

14.5 Map

- Map represents a group of objects as key-value pairs.



- Map is **not** child interface of collections.



- Both keys and values are objects.
- Duplicate keys are not allowed, but values can be duplicated.
- Each key-value pair is called **entry**, hence map is considered as a collection of entry objects.

Map interface methods:

- Add one key-value pair to the map. If the key is already present then, old value will be replaced with new value and returns old value:

Syntax:

Object put(Object key, Object value)

- Insert the specified map in the map.

Syntax:

```
void putAll(Map map)
```

- Returns the object that contains the value associated with the key.

Syntax:

```
Value get(Object key)
```

- Delete an entry for the specified key.

Syntax:

```
boolean remove(Object key, Object value)
```

- Returns true if some key equal to the key exists within the map, else return false.

Syntax:

```
boolean containsKey(Object key)
```

- Returns true if some value equal to the value exists within the map, else return false.

Syntax:

```
boolean containsValue(Object value)
```

- Returns true if the map is empty; returns false if it contains at least one key.

Syntax:

```
boolean isEmpty()
```

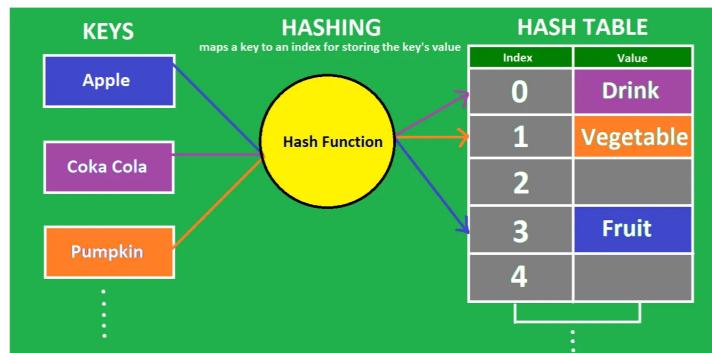
- Returns the number of entries in the map.

Syntax:

```
int size()
```

14.5.1 HashMap

- HashMap class implements Map interface.
- The underlying data structure is **hash table**.



- Features:
 - It is **resizable array**.
 - **Duplicate keys not allowed**, but duplicate values are allowed.
 - **Insertion order is not preserved** and it is based on hashCode of objects.
 - **Hetrogenous keys and values are allowed**.
 - **Null key is allowed (only once) , null value is allowed any number of times**.
- Best choice for frequent **search operation**.

Ways of initializing HashMap

- Creating am empty HashMap:

Syntax:

```
HashMap s1 = new HashMap();
```

- **HashMap capacity & fill ratio:**
 - Above syntax will create HashMap with **default capacity of 16**.
 - Default fill ratio is **0.75**: Fill ratio is after filling 16th element, when 17th element is added, it's new capacity should be

increased by 75%. Fill ratio is also called Load factor

- Creating an empty HashMap with initial capacity:

Syntax:

```
HashMap h = new HashMap(int initialcapacity);
```

- Creating an empty HashMap with initial capacity and fill ratio:

Syntax:

```
HashMap h = new HashMap(int initialcapacity, float fillratio);
```

Eg:

Code:

```
HashMap h = new HashMap(1000, 0.9);
```

HashMap specific methods

- Return a collection view of the mappings contained in this map.

Syntax:

```
Set entrySet()
```

- Return a set view of the keys contained in this map.

Syntax:

```
Set keySet()
```

- Return a set view of the keys contained in this map.

Syntax:

```
Set keySet()
```

- Returns a collection view of the values contained in the map.

Syntax:

```
Collection<V> values()
```

- Replaces the specified value for a specified key.

Syntax:

Value replace(K key, V value)

Example Program: Display HashMap operations:

Test.java

```
import java.util.*;
class one {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put("Raman",101);
        map.put("Ravi",102);
        map.put("Kavi",103);
        System.out.println(map);
        Set s = map.keySet();
        System.out.println(s);
        Collection c = map.values();
        System.out.println(c);
        Set s2 = map.entrySet();
        System.out.println(s2);
        System.out.println(map.replace("Ravi",106));
        System.out.println(map);
    }
}
```

Output:

```
{Ravi=102, Raman=101, Kavi=103}
[Ravi, Raman, Kavi]
[102, 101, 103]
[Ravi=102, Raman=101, Kavi=103]
102
{Ravi=106, Raman=101, Kavi=103}
```

14.5.2 LinkedHashMap

- LinkedHashMap is child class of HashMap class.
- It is exactly same as HashMap, except:

HashMap	LinkedHashMap
Data structure used is HashTable	Data structure is a combination of LinkedList + HashTable
Insertion order not preserved & based on keys	Insertion order preserved
Introduced in 1.2 version	Introduced in 1.4 version

Example Program: Display LinkedHashMap operations

Test.java

```
import java.util.*;  
class one {  
    public static void main(String[] args) {  
        HashMap map = new LinkedHashMap();  
        map.put("Raman",101);  
        map.put("Ravi",102);  
        Set s = map.keySet();  
        Collection c = map.values();  
        Set s2 = map.entrySet();  
        System.out.println(map.replace("Ravi",106));  
    } }
```

14.5.3 IdentityHashMap

- IdentityHashMap class implements Map interface.
- It is same as HashMaps except:
 - For HashMap, JVM uses ".equals()" to identify duplicate keys.
 - For IdentityHashMap, JVM uses "==" operator to identify duplicate keys.
- Notice below code for HashMap:

Test.java

```
import java.util.*;  
class one {  
    public static void main(String[] args) {  
        HashMap h1 = new HashMap();  
        h1.put(new Integer(10),"Pawan");  
        h1.put(new Integer(10),"Raman");  
        System.out.println(h1); // Output: {10=Raman}  
    }  
}
```

- Now, notice the same code for IdentityMap:

Test.java

```
import java.util.*;  
class one {  
    public static void main(String[] args) {  
        IdentityHashMap h1 = new IdentityHashMap();  
        h1.put(new Integer(10),"Pawan");  
        h1.put(new Integer(10),"Raman");  
        System.out.println(h1); // Output: {10=Pawan, 10=Raman}  
    }  
}
```

14.5.4 SortedMap

- SortedMap interface implements Map interface.
- It represent a group of **key,value pairs according to some sorting order of keys.**
- Sorting is based on the key, & not on value.

SortedMap specific methods:

- Returns the first key of the sorted map.

Syntax:

```
Object firstKey()
```

- Returns the last key of the sorted map.

Syntax:

```
Object lastKey()
```

- Returns all the entries of a map whose keys are less than the specified key.

Syntax:

```
SortedMap headMap(Object key)
```

- Returns all the entries of a map whose keys are greater than or equal to the specified key.

Syntax:

```
SortedMap tailMap(Object key)
```

- Returns all the entries of a map whose keys lies in between key1 and key2 including key1

Syntax:

```
SortedMap subMap(Object key1, Object key2)
```

14.5.5 NavigableMap

- NavigableMap interface implements SortedMap interface.
- It defines several methods for navigation purposes.

NavigableMap specific methods

- Return highest key element which is $\leq e$:

Syntax:

```
Object floorKey(e)
```

- Return highest key element which is $< e$:

Syntax:

```
Object lowerKey(e)
```

- Return lowest key element which is $\geq e$:

Syntax:

```
Object ceilingKey(e)
```

- Return lowest key element which is $> e$:

Syntax:

```
Object higherKey(e)
```

- Return Navigable Map in reverse order:

Syntax:

```
NavigableSet descendingMap()
```

- Return and remove first key, value pair:

Syntax:

```
pollFirstEntry()
```

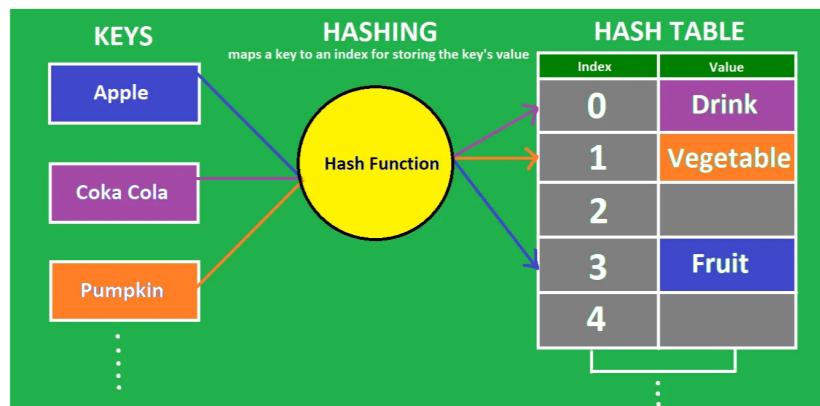
- Return and remove last key, value pair:

Syntax:

```
pollLastEntry()
```

14.5.6 TreeMap

- TreeMap class implements Map interface.
- The underlying data structure is **RED-BLACK tree**.



- Features:
 - It is **resizable array**.
 - Duplicate keys not allowed**, but duplicate values are allowed.
 - Insertion order is not preserved** and it is based on some sorting order of keys.
 - Homogenous keys** allowed for default **natural sorting order**.
 - Heterogenous keys** allowed if you are defining **custom sorting by comparator**.
 - Null key is not allowed**.

Ways of initializing TreeMap

- Creating an empty TreeMap with default natural sorting order:

Syntax:

```
TreeMap t1 = new TreeMap();
```

- Creating an empty TreeSet with custom Comparator:

Syntax:

```
TreeMap t = new TreeMap(Comparator c);
```

Example Program: Display TreeMap with custom comparator to add string keys in descending order:

Test.java

```
import java.util.*;  
class Custom implements Comparator {  
    public int compare(Object o1, Object o2) {  
        String s1 = o1.toString();  
        String s2 = o2.toString();  
        return s2.compareTo(s1);  
    }  
}  
public class two {  
    public static void main(String[] args) {  
        TreeMap t1 = new TreeMap(new Custom());  
        t1.put("Apple",101);  
        t1.put("Mango",102);  
        t1.put("Banana", 103);  
        t1.put("Orange",104);  
        System.out.println(t1);  
        System.out.println(t1.floorKey("Banana"));  
        System.out.println(t1.lowerKey("Banana"));  
        System.out.println(t1.descendingMap());  
    }  
}
```

Output:

```
{ Orange=104, Mango=102, Banana=103, Apple=101 }  
Banana  
Mango  
{Apple=101, Banana=103, Mango=102, Orange=104}
```

14.5.7 Properties**14.6 Cursors**

- Cursor is used to get objects one by one from the collection.
- There are 3 types of cursors available in Java:
 - Enumeration
 - Iterator
 - ListIterator
- Let's see each of these in detail.

14.6.1 Enumeration

- Enumeration interface is used to **enumerate** (or iterate over) elements in **certain legacy collection classes** like:
 - Vector
 - Stack
 - Hashtable
 - Properties
- It is part of the **java.util** package.

Enumeration specific methods:

- Check whether collection is empty or not:

Syntax:

```
boolean hasMoreElements()
```

- Check for the next element:

Syntax:

```
Object nextElement()
```

Note:

- Enumeration concept is applicable only for legacy classes.
- Only read access is allowed, and you can't perform remove operation.
- To overcome these limitations, go for iterator

Example:

Java program to iterate over vector object and display even numbers only:

Test.java

```
import java.util.*;
class Test{
    public static void main(String[] args) {
        Vector v1 = new Vector();
        v1.add(340);
        v1.add(123);
        v1.add(342);
        v1.add(541);
        Enumeration e = v1.elements();
        while (e.hasMoreElements()) {
            Integer i = (Integer) e.nextElement();
            if(i%2 == 0)
                System.out.println(i);
        }
    }
}
```

Output:

340
342

If Enumeration, Iterator and ListIterator is an interface, then how we are creating it's object?

Ans: Vector class internally implements Enumeration, Iterator and ListIterator as **anonymous class**.

14.6.2 Iterator

- Iterator is applicable for any collection object, hence it is universal cursor.
- It can perform both the read and remove operations.
- Iterator object is created using **iterator()** method of collection interface

Iterator specific methods:

- Create iterator object:

Syntax:

```
public Iterator iterator()
```

- Check for next element in collection:

Syntax:

```
public boolean hasNext()
```

- Get next element in collection:

Syntax:

```
public Object next()
```

- Remove element in collection:

Syntax:

```
public void remove()
```

Note:

- Enumeration and Iterator can move only in forward direction and not backward direction.
- These are single direction cursors but not bi-directional cursor.
- Iterator can perform only read & remove operations and not replace or addition of new objects.
- To overcome these limitations, go for **ListIterator**

Example:

Java program to iterate over vector object and display even numbers only:

Test.java

```
import java.util.*;  
class Test{  
    public static void main(String[] args) {  
        ArrayList a1 = new ArrayList();  
        a1.add(123);  
        a1.add(788);  
        a1.add(545);  
        a1.add(642);  
        Iterator t1 = a1.iterator();  
        while(t1.hasNext()) {  
            Integer i = (Integer)t1.next();  
            if (i%2 != 0)  
                t1.remove();  
        }  
        System.out.println(a1);  
    }  
}
```

Output:

[788, 642]

14.6.3 ListIterator

- ListIterator can move either to **forward direction** or to the **backward direction**.
- Hence it is **bi-directional** cursor.
- Using ListIterator, you can perform replacement and addition of new objects in addition to read and remove operations.

ListIterator specific methods: ListIterator is the child interface of Iterator, hence all methods of Iterator interface are applicable on ListIterator.

- Creating ListIterator object:

Syntax:

```
ListIterator l2 = collection.listIterator();
```

- **Forward movement:**

- Check for next element in collection:

Syntax:

```
public boolean hasNext()
```

- Get next element in collection:

Syntax:

```
public Object next()
```

- Return index of next element:

Syntax:

```
public int nextIndex()
```

- **Backward movement:**

- Check for previous element in collection:

Syntax:

```
public boolean hasPrevious()
```

- Get previous element in collection:

Syntax:

```
public Object previous()
```

- Return index of previous element:

Syntax:

```
public int previousIndex()
```

- **Extra operations:**

- Remove element in collection:

Syntax:

```
public void remove()
```

- Add element in collection:

Syntax:

```
public void add(Object o)
```

- Replaces the last element returned by **next()** or **previous()** with the specified element:

Syntax:

```
void set(Object o)
```

Example: Java program to display ListIterator operation by retaining only those name starting with vowel character and removing rest names:

Test.java

```
import java.util.*;
class Test{
    public static void main(String[] args) {
        ArrayList      vowels      =      new      Ar-
rayList(List.of("A","E","I","O","U"));

        LinkedList l1 = new LinkedList();
        l1.add("Apple");
        l1.add("Banana");
        l1.add("Grapes");
        l1.add("Chikoo");
        l1.add("Eggs");
        ListIterator l2 = l1.listIterator();
        while(l2.hasNext()) {
            String s = (String)l2.next();
            String value = Character.toString(s.charAt(0));
            if (! vowels.contains(value)) {
                l2.remove();
            }
        }
        System.out.println(l1);
    }
}
```

Output:

[Apple, Eggs]

14.7 Collections class

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 exercise** on these topics to check your knowledge.



So let's get started....

14.7.1 Sorting

- 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
```

14.7.2 Searching

- 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
```

14.7.3 Reversing

- 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
```

14.8 Advance array operations

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 exercise** on these topics to check your knowledge.



So let's get started....

14.8.1 Sorting

- 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
```

14.8.2 Searching

- 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
```

Don't focus on the pain ->



Focus on progress ->



- Dwayne Johnson

15. Multithreading

15.1 Introduction to Multi-threading

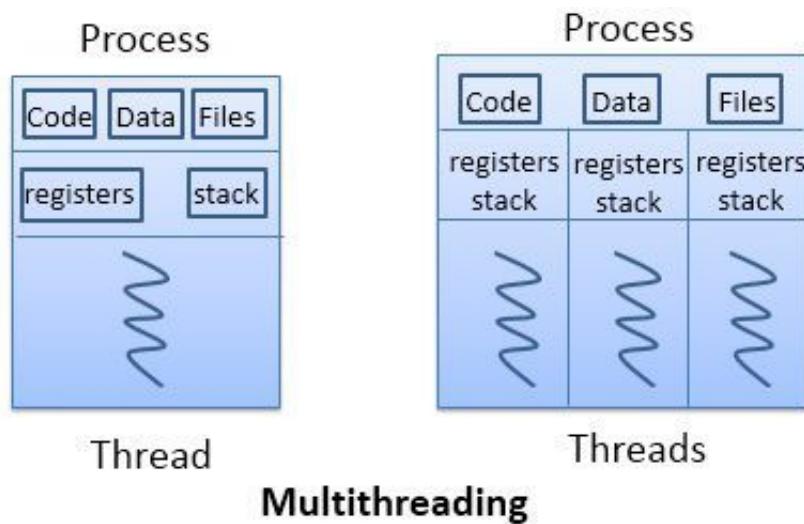
15.1.1 What is Multi-tasking?

Executing multiple task at a time is multi-tasking. Types of multi-tasking are:

- **Process-Based multi-tasking:** OS can perform multiple task independently at a time like:
 - Play audio songs
 - Execute Java program
 - Play movie etc.
- **Thread-Based multi-tasking:**
 - Task executed simultaneously where **each task is separate part** of a program is called thread-based multi-tasking.
 - **Each independent part is called a thread.**
 - Thread-based multi-tasking is best suitable at programmatic level.

15.1.2 What is Multi-threading?

- A thread is the smallest unit of execution within a process.
- **Multithreading refers to the concurrent execution of multiple threads** within a program.
- Multithreading makes **efficient use of resources** for responsiveness and performance.



Application of multi-threading

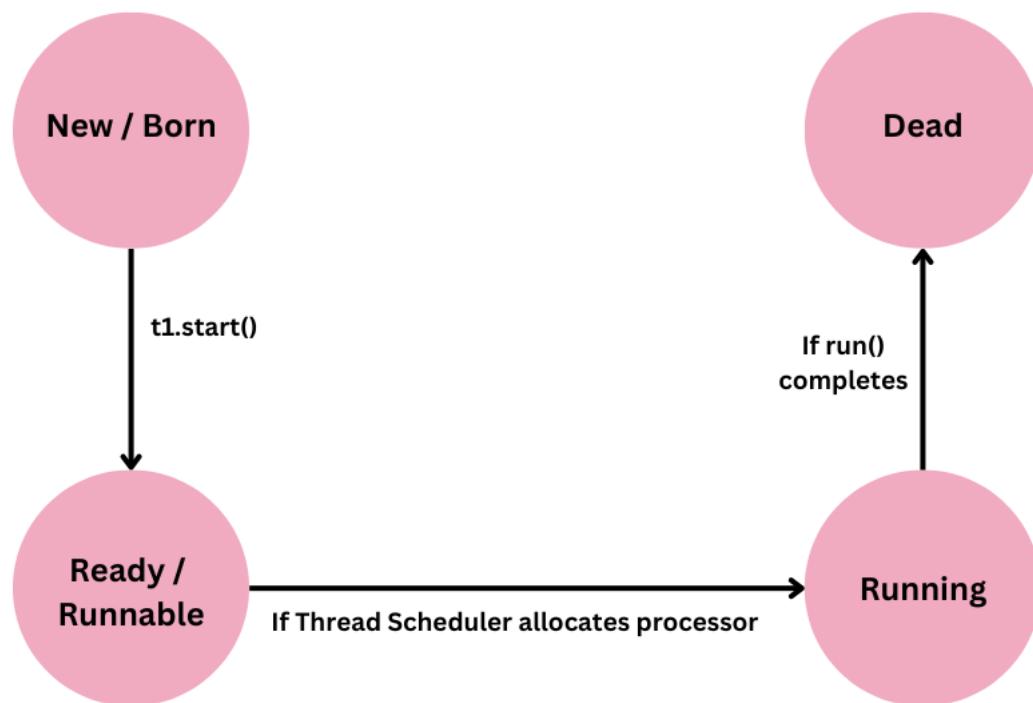
- Multi-media graphics
- Animations
- Video games
- Webservers and application servers etc.

15.1.3 Thread scheduler

- Thread scheduler is part of JVM.
- It **decides the order of threads** for execution.
- Thread scheduler **algorithm varies from JVM to JVM**.
- Thread's execution order & it's output can't be predicted.

15.1.4 Thread life-cycle

```
NewThread t1 = new NewThread()
```



15.2 Thread creation

- **Every thread has a separate job.**
- Ways of creating thread:
 - By extending a Thread class
 - By implementing Runnable interface

15.2.1 Extending Thread class

- Creating custom thread:

Syntax:

```
class ThreadName extends Thread {
    public void run() {
        // overriding thread code
    }
}
```

- Creating thread object and starting it:

Syntax:

```
ThreadName objname = new ThreadName();
obj.start()
```

- Eg:

demo.java

```
class DemoThread extends Thread {}
public class demo {
    public static void main(String[] args) {
        DemoThread demoThread = new DemoThread();
    }
}
```

15.2.2 Implementing Runnable Interface

- Runnable interface present in **java.lang** package.
- It contain only 1 method:

Syntax:

```
public void run()
```

- Creating custom thread:

Syntax:

```
class RunnableName implements Runnable {  
    public void run() {  
        // overriding thread code  
    } }
```

- Creating thread object and starting it:

Syntax:

```
RunnableName obj = new RunnableName();  
Thread t1 = new Thread(obj);  
t1.start();
```

- Eg:

demo.java

```
class ThreadX implements Runnable {  
    public void run() {}  
}  
public class demo {  
    public static void main(String[] args)  
throws InterruptedException {  
    ThreadX obj = new ThreadX();  
    Thread t1 = new Thread(obj);  
    t1.start();  
} }
```

Note:

start() cannot be executed for runnable object. Thus obj.start() in above code will result in compile-time error.

Which approach is better? Extending Thread class or implementing Runnable interface?

Ans: **Implementing Runnable interface is better.** Reason: Extending Thread class wont allow extending any other class. Thus missing inheritance benefit.

15.2.3 Getting and Setting name of a thread

- Thread has name given by **JVM or programmer.**
- Methods to get/set name of thread:

Syntax:

```
public final String getName()
public final void setName(String name)
```

- Get current thread object using **Thread.currentThread()**.
- Eg:

demo.java

```
class DemoThread extends Thread {}
public class demo {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()); // Output: main
        DemoThread t1 = new DemoThread();
        System.out.println(t1.getName()); // Output: Thread-0
        Thread.currentThread().setName("A");
        t1.setName("B");
        System.out.println(Thread.currentThread().getName()); // Output: A
        System.out.println(t1.getName()); // Output: B
    }
}
```

15.2.4 Thread Priorities

- Every thread has priority provided by **JVM or programmer**.
- Thread priority range: **1(minimum) to 10(maximum)**.
- **Thread scheduler & Thread priority:**
 - Thread scheduler uses priority while allocating processors.
 - **Highest priority thread gets first chance.**
 - Same priority thread execution order depends on thread scheduler.
- Thread class defined constants for priorities:

Code:

```
System.out.println(Thread.MIN_PRIORITY);
// Output: 1
System.out.println(Thread.MAX_PRIORITY);
// Output: 10
System.out.println(Thread.NORM_PRIORITY);
// Output: 5
```

- Thread class defined methods to get/set priority of a thread:

Syntax:

```
public final int getPriority()
public final void setPriority(int p)
```

- **Default Thread Priority:**
 - **main thread priority is 5.**
 - Default priority for normal thread is **inherited from parent to child.**

Eg: Create a thread with greater priority compared to main thread.

demo.java

```
class DemoThread extends Thread {  
    public void run(){  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Child Thread");  
        }  
    }  
    public class demo {  
        public static void main(String[] args) {  
            DemoThread demoThread = new DemoThread();  
            demoThread.setPriority(10);  
            demoThread.start();  
            for(int i = 0; i < 5; i++) {  
                System.out.println("Main Thread");  
            }  
        }  
    }  
}
```

Output:

```
Child Thread  
Child Thread  
Child Thread  
Child Thread  
Child Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread
```

15.2.5 start() & run() methods

start():

- start() comes from Thread class & is heart of multi-threading.
- **start() creates new thread and executes run().**
- start() can be overridden, however **it is not recommended**.
- Role of start():
 - **Register the thread** with thread scheduler
 - Perform related mandatory activities
 - **Invoke run()**

run():

- run() contains the code executed by start().
- By default, Thread class's run() is executed i.e empty.
- **Override this** method when you extend the Thread class or implement the **Runnable** interface.
- Overloaded run() will need to be call explicitly.
- start() always invoke no-argument run().

Note:

Difference between start() and run():

start() creates new thread & calls run().

run() will not create new thread. Calling run() explicitly will execute like a normal method call.

Eg: Below code shows overridden start() not executing run() explicitly.

demo.java

```
class DemoThread extends Thread {  
    public void start() {  
        System.out.println("Overridden Start");  
    }  
    public void run() {  
        System.out.println("This wont execute");  
    }  
}  
public class demo {  
    public static void main(String[] args) {  
        DemoThread demoThread = new DemoThread();  
        demoThread.start();  
    }  
}
```

Output:

Overridden Start

Eg: Below code shows overridden run() and no-arg run().

demo.java

```
class DemoThread extends Thread {  
    public void run() {  
        System.out.println("No-argument run executed");  
    }  
    public void run(int i) {  
        System.out.println("run with argument executed");  
    }  
}
```

```
    }
}

public class demo {
    public static void main(String[] args) {
        DemoThread demoThread = new DemoThread();
        demoThread.start();
    }
}
```

Output:

No-argument run executed

You can call super.start() in overridden start() to create new thread as usual:

Eg:

demo.java

```
class DemoThread extends Thread {
    public void start() {
        super.start();
        System.out.println("Overridden Start");
    }
    public void run() {
        System.out.println("This will execute");
    }
}

public class demo {
    public static void main(String[] args) {
        DemoThread demoThread = new DemoThread();
        demoThread.start();
    }
}
```

Output:

Overridden Start
This will execute

After starting a thread, if you are trying to restart the same thread, then it will result in runtime exception saying: **IllegalThreadStateException** exception.

Eg:

demo.java

```
class DemoThread extends Thread {}  
public class demo {  
    public static void main(String[] args) {  
        DemoThread demoThread = new DemoThread();  
        demoThread.start();  
        demoThread.start();  
    }  
}
```

Output:

Exception in thread "main"
java.lang.IllegalThreadStateException at
java.base/java.lang.Thread.start(Thread.java:794)
at demo.main(demo.java:7)

15.2.6 Preventing thread from execution

Prevent thread execution using below methods:

- yield()
- join()
- sleep()
- interrupt()
- stop() (deprecated)
- suspend() & resume() (deprecated)

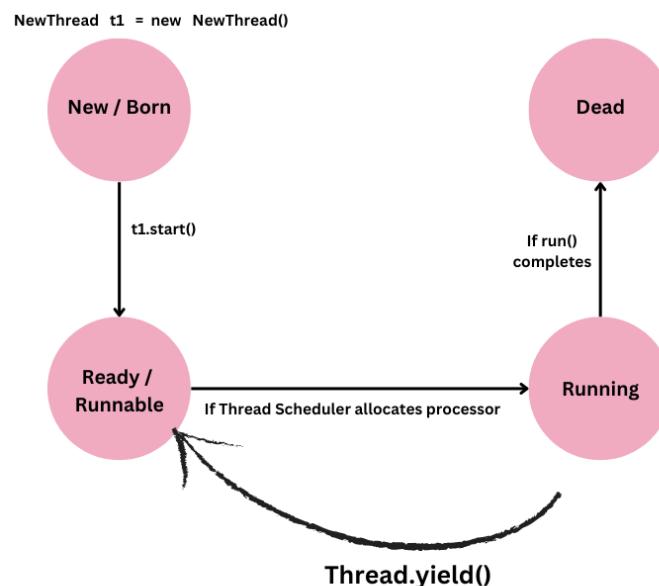
yield():

- **yield() pauses current thread & runs same priority waiting threads.**
- Low priority waiting threads causes same thread to run.

Syntax:

```
public static native void yield()
```

- Here, native means method is in non-java language.
- yield() & thread life cycle -



- Eg: In below code, if you uncomment "Line 1", then both threads (main & DemoThread) will be executed. As "Line 1" is commented, it first executes main thread and then DemoThread.

demo.java

```
class DemoThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Child Thread");  
            Thread.yield(); // Line 1  
        }  
    }  
}  
  
public class demo {  
    public static void main(String[] args) {  
        DemoThread t1 = new DemoThread();  
        t1.start();  
        for(int i = 0; i < 5; i++) {  
            System.out.println("Main Thread");  
        }  
    }  
}
```

Output:

Main Thread

..

Child Thread

..

Note:

Some platforms wont provide proper support for yield().

sleep():

- Pauses a thread for given time.

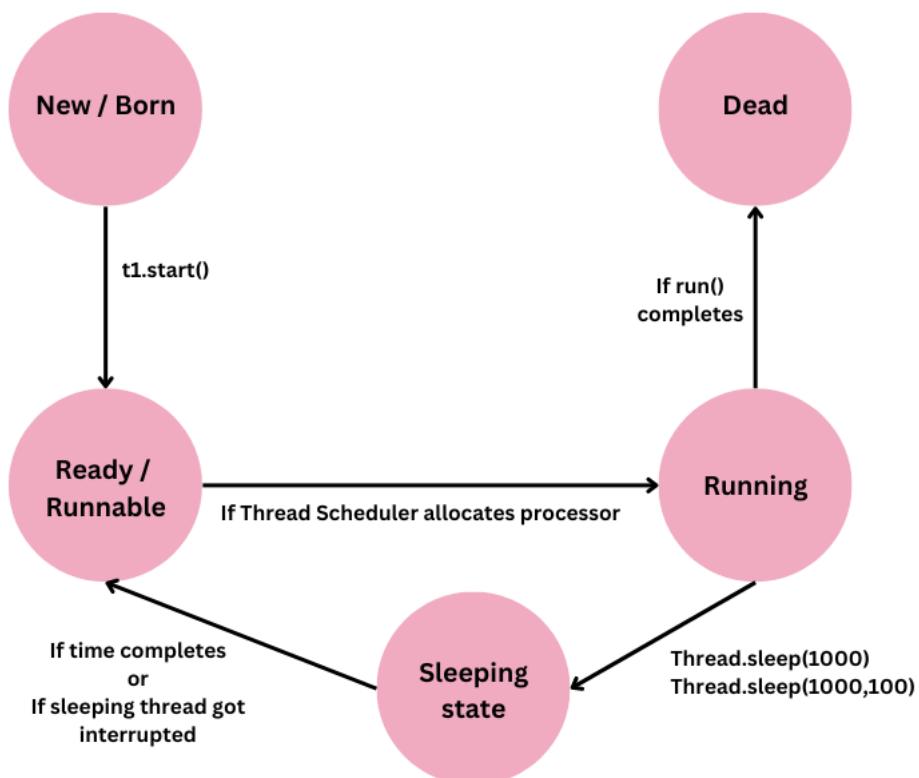
Syntax:

```
public static native void sleep(long ms) throws InterruptedException
```

```
public static native void sleep(long millisec, int nanosec)
throws InterruptedException
```

- sleep() & thread life-cycle:

`NewThread t1 = new NewThread()`



- Eg: Below code causes thread to sleep for 2000 seconds:

demo.java

```
public class demo {  
    public static void main(String[] args) throws InterruptedException{  
        for(int i=0; i<5; i++) {  
            System.out.println("Main-Thread");  
            Thread.sleep(2000);  
        }  
    }  
}
```

Output:

Main-Thread

..

join():

- **Cause a thread to wait until completing another thread.**
- ThreadA calls ThreadB.join(), if it wants to wait until ThreadB completes.

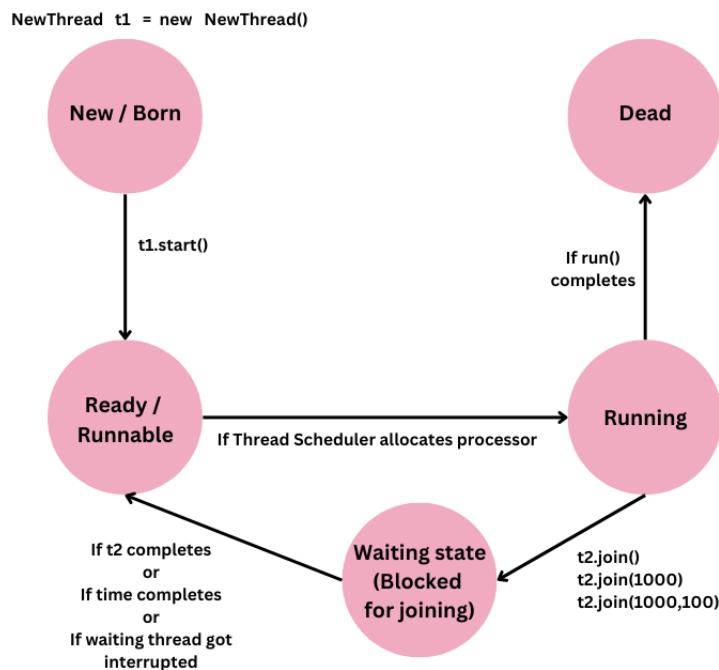
Syntax:

```
public final void join() throws InterruptedException
```

```
public final void join(long milliseconds) throws InterruptedException
```

```
public final void join(long milliseconds, int nanoseconds)  
throws InterruptedException
```

Impact of join() in thread lifecycle:



Eg: In below code, main & new thread are not in sync. There is no guarantee when the new thread's run() method will complete. Hence count is showing 0.

demo.java

```

class ThreadX extends Thread {
    int count;
    public void run() {
        count=10;
    }
}
public class demo {
    public static void main(String[] args) throws InterruptedException {
        ThreadX t1 = new ThreadX();
        t1.start();
        System.out.println(t1.count); // Output: 0
    }
}
  
```

Eg2: join() ensures that main thread waits for the new thread to finish.

demo.java

```
class ThreadX extends Thread {
    int count;
    public void run() {
        count=10;
    }
}
public class demo {
    public static void main(String[] args) throws InterruptedException {
        ThreadX t1 = new ThreadX();
        t1.start();
        t1.join();
        System.out.println(t1.count); // Output: 10
    }
}
```

Note:

Thread Deadlock: ThreadA calls join() on ThreadB & vice-versa results in wait forever. This is called **thread deadlock**.

stop():

- stop() causes thread to enter dead state.
- stop() is deprecated. Eg:

demo.java

```
class NewThread extends Thread {}
class demo {
    public static void main(String[] args) {
        NewThread t1 = new NewThread();
        t1.start(); t1.stop();
    }
}
```

interrupt():

- Interrupt a sleeping/waiting thread using interrupt().
- It works only on sleeping/waiting thread.

Syntax:

```
public void interrupt()
```

- Eg: In below code, if we comment "Line 1", then main thread wont interrupt child thread. Child thread will execute for loop 10 times. If we are not commenting "Line 1", then main thread interrupts child thread.

demo.java

```
class DemoThread extends Thread {  
    public void run() {  
        try{  
            for (int i = 0; i<5; i++) {  
                System.out.println("Child Thread-1");  
                Thread.sleep(2000);  
            } }  
        catch(InterruptedException e){  
            System.out.println("OOps thread interrupted");  
        } } }  
public class new3 {  
    public static void main(String[] args) throws InterruptedException{  
        DemoThread t1 = new DemoThread();  
        t1.start();  
        t1.interrupt(); // Line 1  
        System.out.println("End of Main Thread");  
    } }
```

Output:

End of Main Thread
Child Thread-1
OOps thread interrupted

suspend() & resume()

- You can suspend a thread by using suspend() from Thread class.
- The thread will be entered into suspended state.
- You can resume a suspended thread using resume() from Thread class.

Syntax:

```
public void suspend()  
public void resume()
```

- These methods are deprecated and not recommended to use.

Comparison tables for yield(), join() and sleep()

Property	yield()	join()	sleep()
Purpose	If a thread wants to pass its execution to give the chance for remaining threads of same priority, then we should go for yield()	If a thread wants to wait until completing some other thread, then we should go for join()	If a thread doesn't want to perform any operation, for a particular amount of time, then we should go for sleep()
Is it overloaded()	No	yes	yes
Is it final()	No	yes	No
Does it throw IE (interrupted exception)?	No	yes	yes
Is it native	yes	no	sleep(long m1) → native sleep(long milliseconds, int nanoseconds) → non-native
Is it static	yes	no	yes

15.3 Synchronization

Race Condition:

- Multiple threads accessing one object at a time, causes data inconsistency.
- This is called race condition.
- This can be solved using **synchronized** keyword.

What is synchronized keyword?

- synchronized modifier is **applicable for methods and blocks**.
- synchronized keyword allow **one thread to access given object** at a time.
- Disadvantage:
 - Increased thread waiting time
 - Performance problems

Types of synchronisation:

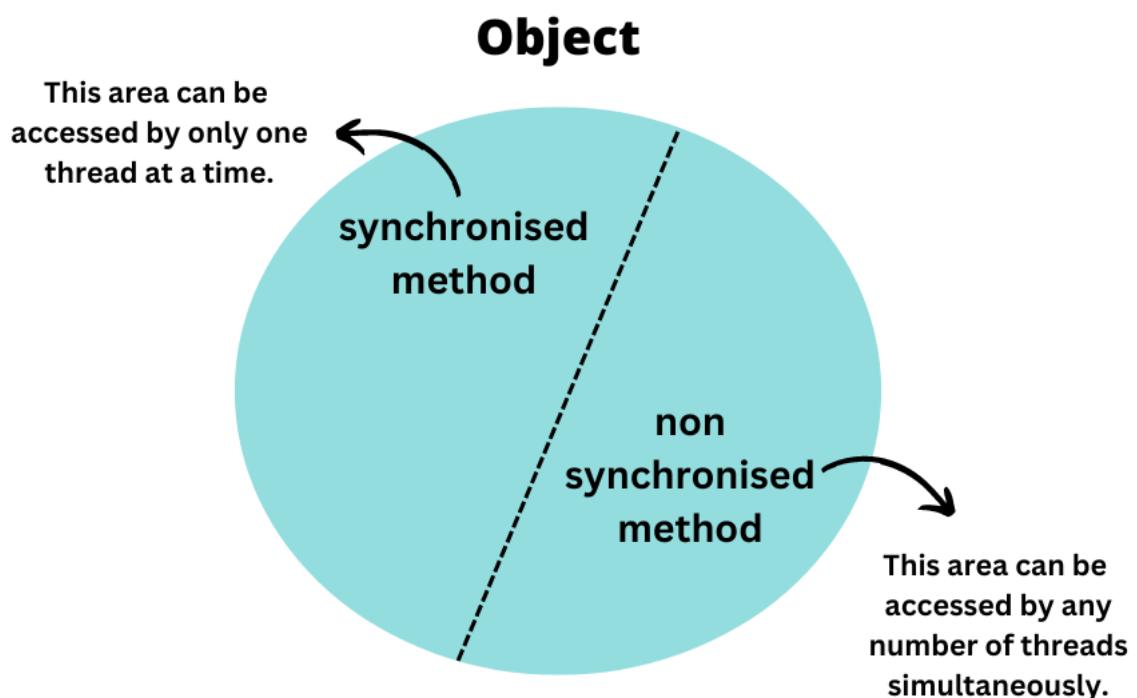
- Synchronised methods:
 - Using object lock
 - Using class lock
- Synchronised block

15.3.1 Synchronised methods using object lock

Synchronized method using object lock:

- **Step1: Choose an Object**
 - Select the Java object of your choice.
 - Every object in Java has a unique lock.
- **Step 2: Acquire the Lock**
 - A thread first gets lock of that object and enters the critical section.
 - Only one thread can hold the object lock at a time.
- **Step 3: Execute Critical Section**

- The thread executes code inside synchronized block/method.
 - This code needs to be protected from concurrent access.
 - Other threads are not allowed to execute synchronized method on the given object.
 - Other threads can execute only non-synchronized methods on the same object.
- **Step 4: Release the Lock**
 - The thread releases the lock, once method completes execution.



Eg: Below code updates Customer object 2 times, notice the output without synchronised method set()

Demo.java

```
class Customer {  
    String name;  
    Customer(String name) {  
        this.name = name;  
    }  
    public void set(String name) {  
        this.name = name;  
        try {  
            Thread.sleep(2000);  
        }  
        catch (InterruptedException e) {}  
        System.out.println(Thread.currentThread().getName() + "  
Updated the Customer");  
    }  
    public void get(){  
        System.out.println("Customer name: " + this.name);  
    }  
}  
  
class Demo extends Thread {  
    Customer c;  
    String name;  
    Demo(Customer c, String name) {  
        this.c = c;  
        this.name = name;  
    }  
}
```

```
public void run() {  
    c.set(name);  
    c.get();  
}  
public static void main(String[] args) throws InterruptedException{  
    Customer c1 = new Customer("Ramesh");  
    Demo t1 = new Demo(c1,"Ram");  
    Demo t2 = new Demo(c1,"Sham");  
    t1.start();  
    t2.start();  
    t1.join();  
    t2.join();  
}  
}
```

Output:

```
Thread-1 Updated the Customer  
Thread-0 Updated the Customer  
Customer name: Sham  
Customer name: Sham
```

Eg: Below code updates Customer object 2 times, notice the output with synchronised method set().

Demo.java

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

```
synchronized public void set(String name) {  
    this.name = name;  
    try {  
        Thread.sleep(2000);  
    }  
    catch (InterruptedException e) {}  
    System.out.println(Thread.currentThread().getName() + "  
Updated the Customer");  
}  
  
public void get(){  
    System.out.println("Customer name: " + this.name);  
}  
}  
  
class Demo extends Thread {  
    Customer c;  
    String name;  
    Demo(Customer c, String name) {  
        this.c = c;  
        this.name = name;  
    }  
    public void run() {  
        c.set(name);  
        c.get();  
    }  
    public static void main(String[] args) throws InterruptedException{  
        Customer c1 = new Customer("Ramesh");  
        Demo t1 = new Demo(c1,"Ram");  
        Demo t2 = new Demo(c1,"Sham");  
    }  
}
```

```
t1.start();
t2.start();
t1.join();
t2.join();
}
```

Output:

Thread-0 Updated the Customer

Customer name: Ram

Thread-1 Updated the Customer

Customer name: Sham

Notice

- If set() is not declared as synchronised, then both the threads will be executed simultaneously and hence, we will get irregular output.
- If set() is synchronised, then at a time, only one thread is allowed to execute set()

15.3.2 Synchronised methods using class lock

Follow below steps, to create synchronized method using class lock:

Step1: Choose the class

- Select the class of your choice.
- Class has a unique lock (class-level lock) used by static synchronised methods.

Step 2: Acquire the Lock

- A thread first gets class-level lock.
- Only one thread can hold the class lock at a time for class.

Step 3: Execute Critical Section

- One thread can execute static synchronised method at a time.
- Other threads can execute the below methods simultaneously:
 - Normal static methods
 - Synchronised instance methods
 - Normal instance methods

Step 4: Release the Lock

- The thread releases the lock, once method completes execution.

Eg: Consider below code wherein 2 threads are trying to increment static variable 5 times, however each thread is incrementing the static variable only once at a time.

demo.java

```
class MyClass {
    private static int count = 0;
    public static synchronized void incrementCount() {
        count=count+1;
    }
    public static synchronized int getCount() {
        return count;
    }
}
class ClassLevel implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            MyClass.incrementCount();
            System.out.println(Thread.currentThread().getName() + ":
Count is " + MyClass.getCount());
        }
    }
}
```

```
        catch (InterruptedException e) {
            e.printStackTrace();
        } } }

class demo {
    public static void main(String[] args) throws InterruptedException {
        ClassLevel c1 = new ClassLevel();
        Thread thread1 = new Thread(c1);
        Thread thread2 = new Thread(c1);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println("Final Count: " + MyClass.getCount());
    }
}
```

Output:

```
Thread-0: Count is 2
Thread-1: Count is 2
Thread-0: Count is 3
Thread-1: Count is 4
Thread-0: Count is 5
Thread-1: Count is 6
Thread-0: Count is 7
Thread-1: Count is 8
Thread-0: Count is 9
Thread-1: Count is 10
Final Count: 10
```

15.3.3 Synchronised Block

- Used if number of lines of the code to be synchronised is less.
- Pros of synchronised block over synchronised method:
 - Reduces thread waiting time.
 - Improves application performance.
- Ways of creating synchronised blocks:
 - Get lock of particular object:

Syntax:

```
synchronised(object)
```

- Get lock of class-level object:

Syntax:

```
synchronised(Class)
```

- Eg: Consider below code that uses synchronized block to obtain object lock and update custom name using multiple threads.

demo.java

```
class Customer {  
    String name;  
    Customer(String name) {  
        this.name = name;  
    }  
    public void set(String name) {  
        this.name = name;  
    }  
    public void get(){  
        System.out.println("Customer name: " + this.name);  
    }  
}
```

```
class Demo extends Thread {  
    Customer c;  
    String name;  
    Demo(Customer c, String name) {  
        this.c = c;  
        this.name = name;  
    }  
    public void run() {  
        synchronized (c) {  
            c.set(name);  
            try {  
                Thread.sleep(2000);  
            }  
            catch (InterruptedException e) {}  
            System.out.println(Thread.currentThread().getName() + "  
Updated the Customer");  
        }  
        c.get();  
    }  
    public static void main(String[] args) throws InterruptedException{  
        Customer c1 = new Customer("Ramesh");  
        Demo t1 = new Demo(c1,"Ram");  
        Demo t2 = new Demo(c1,"Sham");  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
    }  
}
```

Output:

Thread-0 Updated the Customer

Customer name: Ram

Thread-1 Updated the Customer

Customer name: Sham

- Eg: Consider below code that uses synchronized block to obtain class lock and update a variable using multiple threads.

demo.java

```
class MyClass {  
    private static int count = 0;  
    public static void incrementCount() {  
        synchronized (MyClass.class) {  
            count++;  
        }  
    }  
    public static int getCount() {  
        synchronized (MyClass.class) {  
            return count;  
        }  
    }  
}  
class ClassLevel implements Runnable {  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            MyClass.incrementCount();  
            System.out.println(Thread.currentThread().getName() + ":"  
                Count is " + MyClass.getCount());  
        }  
    }  
}
```

```
try {  
    Thread.sleep(100);  
}  
catch (InterruptedException e) {  
    e.printStackTrace();  
}  
}  
}  
  
class Main {  
    public static void main(String[] args) throws InterruptedException {  
        ClassLevel c1 = new ClassLevel();  
        Thread thread1 = new Thread(c1);  
        Thread thread2 = new Thread(c1);  
        thread1.start();  
        thread2.start();  
        thread1.join();  
        thread2.join();  
        System.out.println("Final Count: " + My-  
Class.getCount());  
    }  
}
```

Output:

```
Thread-0: Count is 2  
Thread-1: Count is 3  
Thread-0: Count is 4  
..  
Thread-1: Count is 8  
Thread-1: Count is 10  
Thread-0: Count is 10  
Final Count: 10
```

15.3.4 Inter Thread Communication

Methods for inter-thread communication and synchronization:

- wait()
- notify()
- notifyAll()

Features of these methods:

- They are part of the Object class.
- **Used with the synchronized keyword.**
- They implement thread-safe operations.
- **Used when you want a specific condition to meet for multiple threads to continue execution.**

Let's see each of these methods in detail:

- **wait():**
 - Causes the current thread to wait until another thread invokes notify() or notifyAll() method on the same object.
 - Must be called within synchronized context, as it releases the monitor lock on the object, allowing other synchronized methods to execute.

Syntax:

```
public final void wait() throws InterruptedException
```

```
public final native void wait(long milliseconds)  
throws InterruptedException
```

```
public final void wait(long milliseconds, int  
nanoseconds) throws InterruptedException
```

- **notify():**

- Wakes up a waiting thread that called wait() on the same object.
- The awakened thread will compete with other threads to reacquire the monitor lock.

Syntax:

```
public final native void notify()
```

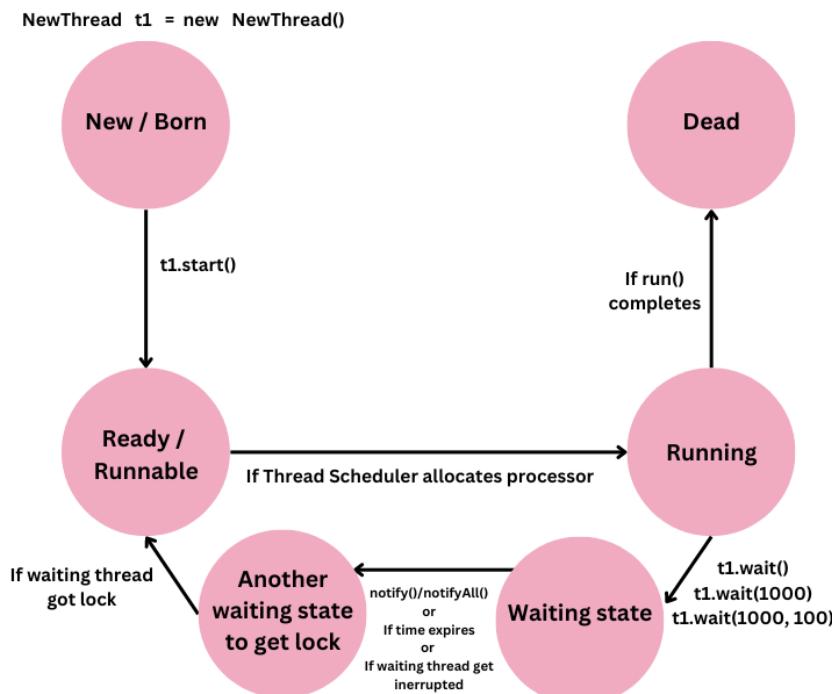
- **notifyAll():**

- Wakes up all waiting threads that called wait() on the same object.
- Each thread will then compete for the monitor lock to proceed with its execution.

Syntax:

```
public final native void notifyAll()
```

Impact of wait(), notify() and notifyAll() on Thread lifecycle:



Eg:

demo.java

```
class ThreadX extends Thread {  
    int count;  
    public void run() {  
        synchronized (this) {  
            for(int i=0;i<=100;i++){  
                count = count+i;  
            }  
            this.notify();  
        }  
    }  
}  
public class demo {  
    public static void main(String[] args) throws InterruptedException {  
        ThreadX t1 = new ThreadX();  
        synchronized (t1) {  
            t1.start();  
            t1.wait();  
            System.out.println(t1.count); // Output: 5050  
        }  
    }  
}
```

15.4 More on Thread

15.4.1 DeadLock

- A thread waiting for other thread forever is called deadlock.
- **Synchronized** keyword is **reason for deadlock** situation.
- There is no solution for deadlock, only prevention is possible.
- Eg: Both functions are synchronized, hence the deadlock situation:

demo.java

```
class A {  
    public synchronized void d1(B b) {  
        System.out.println("Thread 1 starts execution of  
d1()");  
        try {  
            Thread.sleep(2000);  
        }  
        catch (InterruptedException e) {}  
        System.out.println("Thread 1 trying to call B's  
last()");  
        b.last();  
    }  
    public synchronized void last() {  
        System.out.println("Inside A, this is last()");  
    }  
}  
public synchronized void d2(A a) {  
    System.out.println("Thread 2 starts execution of  
d2()");  
    try {  
        Thread.sleep(2000);  
    }
```

```

class B {
    catch (InterruptedException e) {}
    System.out.println("Thread 2 trying to call A's last()");
    a.last();
}
public synchronized void last() {
    System.out.println("Inside B, this is last()");
}
class Deadlock1 extends Thread {
    A a = new A();
    B b = new B();
    public void m1() {
        this.start();
        a.d1(b);
    }
    public void run() {
        b.d2(a);
    }
    public static void main(String[] args) {
        Deadlock1 d = new Deadlock1();
        d.m1();
    }
}

```

Output:

Thread 1 starts execution of d1()
 Thread 2 starts execution of d2()
 Thread 1 trying to call B's last()
 Thread 2 trying to call A's last()
 Press CTRL+C to break

If atleast 1 synchrnoinsed keyword is removed, the program wont enter into deadlock.

15.4.2 Deadlock v/s Starvation

Deadlock	Starvation
Long waiting of a thread where waiting never ends	Whereas long waiting of a thread, where waiting ends at certain point

15.4.3 Daemon Thread

- Daemon threads are threads executing in the background.
- Eg:
 - Garbage Collector
 - Signal Dispatcher
 - Attach Listener etc.
- Daemon threads support non-daemon threads (like main thread).
- Eg: Main thread running with low memory, causes JVM to run garbage collector to free memory.
- Usually daemon threads have low priority, but can have high priority also.
- Method to check daemon nature of a thread:

Syntax:

```
public boolean isDaemon()
```

- Method to change daemon nature of a thread:

Syntax:

```
public void setDaemon(boolean b)
```

- Daemon nature can be changed before starting of a thread.
- After starting a thread, nature of thread can't be changed.

15.4.4 Default nature of a thread

- Main thread is always non-daemon.
- Default nature of other threads is inherited from their parent threads.

Is it possible to change non-daemon behaviour of main thread?

Ans: No. JVM already starts the main thread at the beginning.

Eg:

demo.java

```
class NewThread extends Thread {}  
class demo {  
    public static void main(String[] args) {  
        System.out.println(Thread.currentThread().isDaemon()); // false  
        NewThread t1 = new NewThread();  
        System.out.println(t1.isDaemon()); // false  
        t1.setDaemon(true);  
        System.out.println(t1.isDaemon()); // true  
    } }
```

15.4.5 Green Thread

Multi-threading concept is implemented using 2 models:

- **Green thread model:**
 - Thread managed by JVM without OS support called Green thread.
 - OS like Sun Solaris provide support for Green Thread model.
 - Green thread model is deprecated.
- **Native OS model:**
 - Thread managed by the JVM with the help of OS is called Native OS model.
 - All Windows based OS provide support for native OS model.

Don't focus on the pain ->



Focus on progress ->



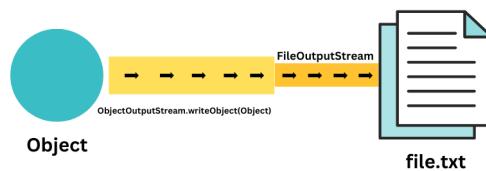
- Dwayne Johnson

16. Serialisation

16.1 Introduction

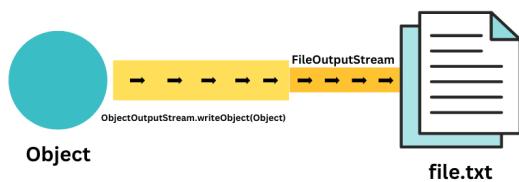
16.1.1 What is serialisation?

- Serialisation means **writing state of an object to a file**.
- It converts an object from Java supported form into file or network supported form.
- **FileOutputStream** and **ObjectOutputStream** classes implements serialisation.



16.1.2 What is deserialisation?

- **Reading state of an object from the file** is deserialisation.
- Process of converting an object from either file supported form or network supported form into java supported form. Eg:
FileInputStream and ObjectInputStream classes implement deserialisation.



16.1.3 Rules of serialisation

- **You can serialise only serialisable objects.**
- An object is serializable if class implements serialisable interface.
- Serializable is **marker interface** present in **java.io** package with no methods.
- Serialising a non-serialisable object, results in runtime exception saying: **NotSerializableException**.

Eg: Serialising and deserialising object:

demo.java

```
import java.io.*;
class New implements Serializable {
    int i= 10;
    int j= 20;
}
public class demo {
    public static void main(String[] args) throws Exception {
        New c1 = new New();
        FileOutputStream fout = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fout);
        oos.writeObject(c1);
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        New c2 = (New)ois.readObject();
        System.out.println("Deserialized Student: " + c2);
        System.out.println(c2.i+","+c2.j);
    }
}
```

Output:

Deserialized Student: New@13a57a3b
10,20

16.1.4 Transient Modifier

- During serialisation, if you dont want to save value of variable then declare it **transient**.
- JVM ignores the original value of transient variable & save default value to the file. Hence transient means not to serialize.

Syntax:

```
transient varname;
```

- Eg: Declaring a variable as transient hides its value to 0:

demo.java

```
import java.io.*;
class New implements Serializable {
    String username= "Ram";
    transient String password="ram@12345";
}
public class demo {
    public static void main(String[] args) throws Exception
    {
        New c1 = new New();
        FileOutputStream fout = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fout);
        oos.writeObject(c1);
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        New c2 = (New)ois.readObject();
        System.out.println(c2.username+","+c2.password); // Output: Ram,null
    }
}
```

static V/S final V/S transient:

- **Static variable** is not part of object state and hence it won't participate in serialisation. Due to this, declaring static variable as **transient**, **is of no use**.
- **Final variables** will be participated in serialisation directly by the value. Hence, declaring a final variable as a **transient**, **is of no impact**.
- Eg:

Declaration	Output
int i = 10; int j = 20;	10..20
transient int i = 10; int j = 20;	0..20
transient static int i = 10; transient int j = 20;	10..0
transient int i = 10; transient final int j = 20;	0..20
transient static int i = 10; transient final int j = 20;	10..20

16.1.5 Object order in serialisation

- Object serialise & deserialise order should be same.
- If object order is not same, **ClassCastException** occurs. Eg:

abc.java

```

import java.io.*;
class Cat implements Serializable { int a=10; }
class Dog implements Serializable { int b=20; }
class Rat implements Serializable { int c=30; }
public class abc {
    public static void main(String[] args)
throws Exception {
    Cat c1 = new Cat();
    Dog d1 = new Dog();
    Rat r1 = new Rat();
    FileOutputStream     fout      =      new      FileOutputStream("abc.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fout);
    oos.writeObject(c1);
    oos.writeObject(d1);
    oos.writeObject(r1);
    FileInputStream fis = new FileInputStream("abc.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Cat c2 = (Cat) ois.readObject();
    Dog d2 = (Dog) ois.readObject();
    Rat r2 = (Rat) ois.readObject();
    System.out.println("Deserialized:");
    System.out.println(c2.a); // Output: 10
    System.out.println(d2.b); // Output: 20
    System.out.println(r2.c); // Output: 30
}
}

```

16.2 Customised Serialisation

Default serialisation cause loss of information due to transient keyword.

Customised serialisation avoids this loss of information.

Methods:

- **writeObject()** executes automatically for serialisation.

Syntax:

```
private void writeObject(ObjectOutputStream os) throws  
Exception
```

readObject() executes automatically for deserialisation.

Syntax:

```
private void readObject(ObjectInputStream is) throws Ex-  
ception
```

- Eg: Below code overrides the writeObject() and readObject() using serialisation:

demo.java

```
import java.io.*;  
class Account implements Serializable {  
    String username = "Lavatech";  
    transient String pwd = "lava@1234";  
    private void writeObject(ObjectOutputStream os) throws  
    Exception {  
        os.defaultWriteObject();  
        String epwd = "678"+pwd;  
        os.writeObject(epwd);  
    }  
    private void readObject(ObjectInputStream is) throws Ex-  
    ception {  
        is.defaultReadObject();
```

```
String epwd = (String)is.readObject();
pwd = epwd.substring(3);
}

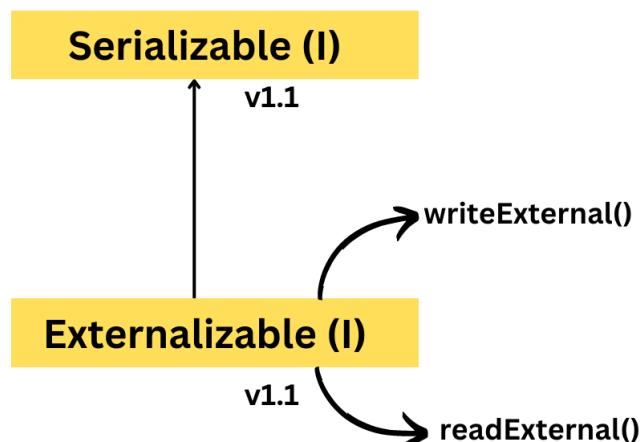
public class demo {
    public static void main(String[] args) throws Exception
{
    Account a1 = new Account();           System.out.println(a1.username+"..."+a1.pwd);
    FileOutputStream fos = new FileOutputStream("abc.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(a1);
    FileInputStream fis = new FileInputStream("abc.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Account a2 = (Account)ois.readObject();
    System.out.println(a2.username+"..."+a2.pwd);
}
}
```

Output:

Lavatech...lava@1234
Lavatech...lava@1234

16.3 Externalisation

- JVM can save **complete & not part of object** to the file during serialisation.
- **Using externalisation, programmer can save part or complete object to a file.**
- Performance wise, externalisation is best choice.
- Externalizable is child interface of Serializable.



- Rules for externalisation:
 - **Override `readExternal()` & `writeExternal()`** as they execute automatically during serialisation & deserialisation.
 - Externalisable implemented class should contain “public no-arg” constructor.
- Eg: Creates an object and overrides `readExternal()` and `writeExternal()` to write part of object to file "abc.ser":

demo.java

```
import java.io.*;
class External implements Externalizable {
    String s;
    int i, j;
    public External() {}
    public External(String s, int i, int j) {
        this.s = s; this.i = i; this.j = j;
    }
    public void writeExternal(ObjectOutput out) throws
IOException {
        out.writeObject(s);
        out.writeInt(i);
    }
    public void readExternal(ObjectInput in) throws
IOException, ClassNotFoundException {
        s = (String)in.readObject();
        i = in.readInt();
    }
    public static void main(String[] args) throws Exception {
        External t1 = new External("lavatech",10,20);
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(t1);
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        External t2 = (External)ois.readObject();
        System.out.println(t2.s+"..."+t2.i+"..."+t2.j);
    }
}
```

Output:

lavatech...10...0

Serialisation V/S Externalisation

Serialisation	Externalisation
Meant for default serialisation	Meant for customised serialisation
JVM has control	Programmer has control
Save complete object to the file	Save either complete or part of the object to the file
Performance is low	Performance is high
It is a marker interface & does'nt contain any methods	Contains 2 methods: writeExternal() and readExternal()
Does not contain "public no-argument" constructor	Contain public no-argument constructor
Transient used in serialisation	Transient cant be used in externalisation

16.4 SerialVersionUID

- During serialisation, **object state travels from sender to receiver & not it's ".class" file.**
- **While serialising, JVM creates and saves a unique identifier with every object** on sender side based on ".class" file.
- At the time of deserialisation, **recevier side JVM compares sender side unique identifier with local class unique identifier.**
- If both are same, then deserialisation will be performed.
- Otherwise it will result in runtime exception:
InvalidClassException
- This unqiue identifier is **SerialVersionUID**.

Problem with JVM generated default SerialVersionUID:

- Sender and receiver's **JVM version should be same**.
- Sender and receiver should use **same ".class" file version**.
- After serialisation, if there is any change in ".class" file, at receiver side then receiver will be unable to deserialise.
- JVM uses **complex algorithm to generate SerialVersionUID** causing performance problems.

16.4.1 Custom SerialVersionUID

- Solve the problems of default SerialVersionUID using custom SerialVersionUID.
- Create a custom serialVersionUID as shown below:

Code:

```
private static final long serialVersionUID=1L;
```

- Eg: In below program, after serialisation if you perform any change to the ".class" file at receiver side, you wont get any problem at the time of deserialisation. In this case, sender and receiver not required to maintain same JVM versions.

Sample.java

```
import java.io.*;
class Sample implements Serializable {
    private static final long serialVersionUID = 1L;
    int i = 10;
    int j = 20;
}
```

Sender.java

```
import java.io.*;
public class Sender {
    public static void main(String[] args) throws Exception
    {
        Sample d1 = new Sample();
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
    }
}
```

Receiver.java

```
import java.io.*;
public class Receiver {
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Sample d2 = (Sample)ois.readObject();
        System.out.println(d2.i+"..." +d2.j);
    }
}
```

Command:

```
$ javac Sample.java
$ javac Sender.java
$ javac Receiver.java
$ java Receiver
10...20
```


Don't focus on the pain ->



Focus on progress ->



- Dwayne Johnson

17. Garbage Collection

17.1 Introduction

- C & C++ require creating & destroying objects manually.
- Programmers neglects destruction of objects causing insufficient memory & poor application performance.
- Out of memory error is common problem in old languages like C++.
- Java requires programmer to create but not destroy the objects.
- Java calls garbage collector(GC) in background (i.e daemon thread) & destroys useless objects.
- Memory issues in Java is very less.

17.1.1 Ways to make an object

- Make an object eligible for GC, if it is no longer.
- Object is eligible for GC, if there's no reference variable.
- **Case 1: Nullify the reference variable**

- Assign null to all object's reference variable.

demo.java

```
import java.io.*;
class Test {}
public class demo {
    public static void main(String[] args) throws Exception {
        Test t1 = new Test();
        t1=null;      // eligible for GC
    }
}
```

- **Case 2: Reassigning the reference variable**

- Re-assign objectA reference variable to objectB, makes objectA eligible for GC.

demo.java

```
import java.io.*;
class Test {}
public class demo {
    public static void main(String[] args) throws Exception {
        Test t1 = new Test();
        Test t2 = new Test();
        t1 = new Test(); // eligible for GC
        t2=t1; // eligible for GC
    }
}
```

- **Case 3: Objects inside method**

- Objects created inside method are eligible for GC once the method completes.

demo.java

```
import java.io.*;
class Test {}
public class demo {
    public static void main(String[] args) throws Exception {
        test1(); // t1,t2 eligible for GC
    }
    public static void test1() {
        Test t1 = new Test();
        Test t2 = new Test();
    }
}
```

17.1.2 JVM Methods for running Garbage collection

- When JVM executes GC to destroy objects can't be predicted.
- **You can request JVM to run GC programmatically.**
- JVM may or may not accept the request. Mostly JVM accepts the request.
- Ways for requesting JVM to run GC:
 - Using System class
 - Using Runtime class

Using System class:

Syntax:

```
System.gc();
```

Using Runtime class:

- Runtime singleton class is present in **java.lang** package.
- Create Runtime object using **getRuntime()** factory method:

Syntax:

```
Runtime obj = Runtime.getRuntime();
```

What is a singleton class?

Ans: A class whose's only 1 object can be created is called singleton class.

What is factory method?

Ans: A method that returns objects of same class is called a factory method. Eg:

Code:

```
Runtime r = Runtime.getRuntime();
DateFormat df = DateFormat.getInstance();
```

• Methods of Runtime class:

1. **totalMemory()**: Returns **total heap memory** in bytes. (i.e **heapSize**).
2. **freeMemory()**: Returns **free heap memory** in bytes.
3. **gc()**: Requesting JVM to run GC.

Note:

It is recommended to use Runtime class **gc()** compared to System class **gc()** as System class **gc()** internally calls Runtime class **gc()**.

demo.java

```
import java.util.Date;  
public class demo {  
    public static void main(String[] args) {  
        Runtime r = Runtime.getRuntime();  
        System.out.println(r.totalMemory());  
        System.out.println(r.freeMemory());  
        for(int i=0; i<10000; i++){  
            Date d = new Date();  
            d = null;  
        }  
        System.out.println(r.freeMemory());  
        r.gc();  
        System.out.println(r.freeMemory());  
    } }
```

17.1.3 Finalisation

- **GC calls finalise()** before object destroy for cleanup activities.
- finalise() is present in **Object** class.

Syntax:

```
protected void finalise() throws Throwable
```

- Override finalise() to define custom cleanup activities.
- If you call finalize() method explicitly, then it would be executed like normal method call.
- **GC calls finalize() once, even if object is eligible for GC multiple times.**
- JVM runs GC whenever program is low on memory..
- Most GC uses **Mark and Sweep Algorithm**.

- Eg:

DemoTest.java

```
class DemoTest {
    public static void main(String[] args) {
        DemoTest test = new DemoTest();
        test = null;
        System.gc();
        System.out.println("End of main");
    }
    public void finalize() {
        System.out.println("DemoTest finalize");
    }
}
```

Output:

End of main
DemoTest finalize

Or

Output:

DemoTest finalize
End of main

17.1.4 Memory Leaks

- Object not used in program & not eligible for GC are called **memory leaks**.
- Memory leaks results in **OutOfMemoryError** exception.
- Make the objects eligible for GC to avoid memory leak.
- Memory leaks are result of programmer's mistake.
- Memory management tools to detect memory leaks:
 - HP-J-Meter
 - IBM-TIVOLI

- J-PROBE
- HP-PATROL

Island of Isolation

- "Island of Isolation" refers to **two or more objects reference each other**, making them inaccessible.
- This causes memory leaks because these objects won't be garbage collected when they're no longer needed.

One.java

```
public class One {  
    static class Node {  
        Node n;  
    }  
    public static void main(String[] args) {  
        Node n1 = new Node();  
        Node n2 = new Node();  
        n1.n = n2;  
        n2.n = n1; // Creates island of isolation  
        // Island of isolation remains in memory causing  
        memory leak  
        n1=null;  
        n2=null;  
    }  
}
```