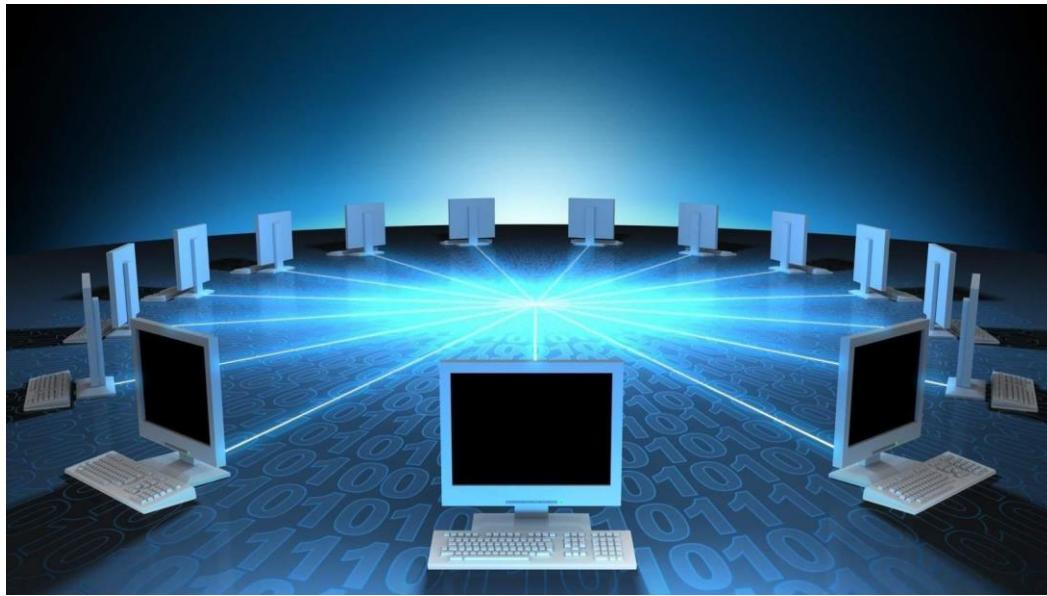


Introduction to Computer

INVENTION OF COMPUTER

Charles Babbage an English Mathematician invented world's first computer in 1940's.

He is also called as **Father of Computer**.



Definition of Computer.

Computer is an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program. It is also a collection of hardware and software components.

For example:

Hardware Components: CPU, RAM, ROM, Keyboard, Printer, Monitor, etc.

Software Components: Operating System, Applications etc.

Microprocessor or CPU

A microprocessor is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.

They are created using a technology called as Semiconductor technology

Types of Transistors

Any device which is made up of transistors is referred to as working in *Semiconductor Technology*. These devices are called as semiconductor devices.

A transistor is a device that regulates current or voltage flow and acts as a switch or gate for electronic signals. The transistors have three terminals emitter, base and collector.

Transistors can only store voltages i.e., High level voltage (5V) and Low-level voltage (0V).

What is HLL, ALL & MLL? What is Assembler and Compiler?

As we already know that microprocessor can store either **5v or 0v**.

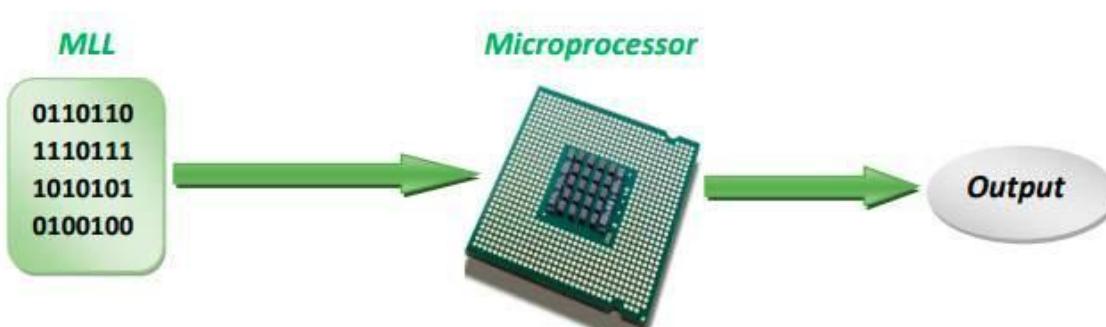
In software engineer's view, they may look the two levels as Low level as '**0**' and High level as '**1**'.

Machine Level Language

To program for first computer was not simple task for the programmer's because microprocessor only understand 0's and 1's. Programmers must remember the combination of 0's and 1's code to perform simple operations like addition, subtraction.

This combination codes are called as "*Machine Level Language*".

It is also referred as "*Low Level Language*".

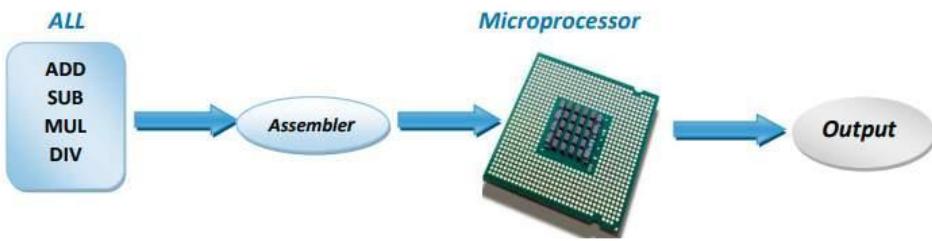


Assembly Level Language

The problem with Machine level code approach was decided to be changed in the year 1950's. They thought that instead of writing a long sequence of 0's and 1's a single instruction can be given.

This approach of writing code is what called as "*Assembly Level Language*".

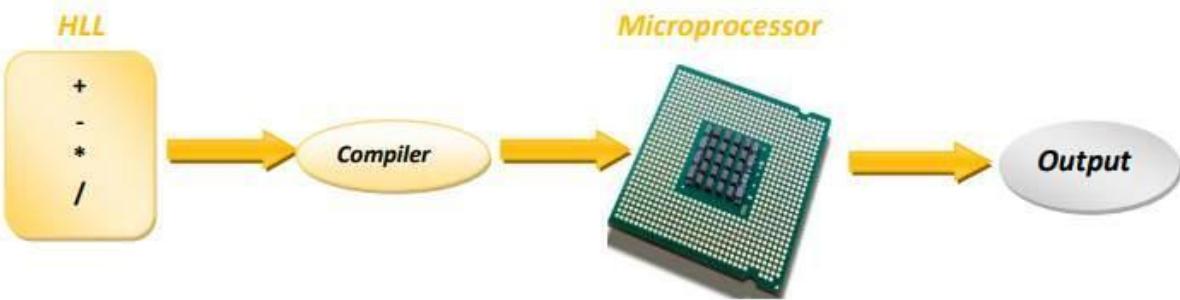
Instead of using numbers like in Machine languages here we use words or names in English forms.



An Assembler is software which takes Assembly Level Language (ALL) programs as input and converts it into Machine Level Language (MLL) program.

High Level Programming Language

In 1960's they came up with next type of language called "*High Level Programming Language*". High Level Languages are written in a form that is close to our human language, enabling the programmer to just focus on the problem being solved.



A compiler is software which takes High Level Language (HLL) programs as input and converts it into Machine Level Language (MLL) program.

Storage Devices

A storage device is a type of hardware that is used for storing, porting or extracting data files and objects.

Storage devices can hold and store information both temporarily and permanently.

They may be internal or external to a computer, server or computing device.

For example: RAM, ROM and Hard Disk.

BUS CONNECTION

The collection of instructions is called as code. These codes are called as **PROGRAMS**.

Hence, these programs must be stored somewhere because microprocessor can't store. Here arises the requirement of Storage Device that is called as "*HARD DISK*".

The connection between hard disk and microprocessor is achieved by bunch of wires called as "*BUS CONNECTION*".

WORKING:

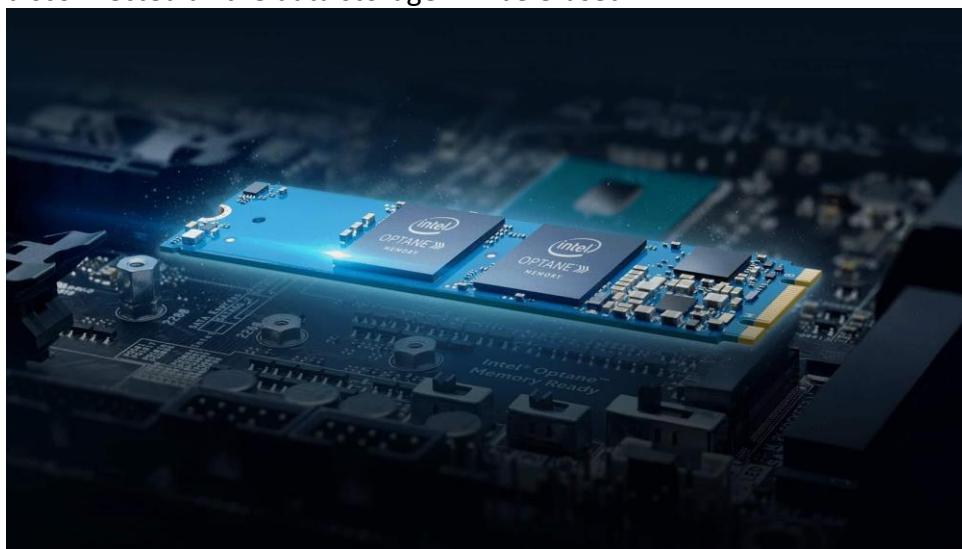
Hard disk sends instructions one by one via bus connection to microprocessor. Microprocessor has to wait for next instruction, this process was very slow because hard disk and microprocessor uses different technology.

- Microprocessor works on the principle of semiconductor technology i.e., on basics of current and voltage.
- Hard Disk works on the principle of magnetism i.e., on basics of magnetic tapes.

Random Access Memory

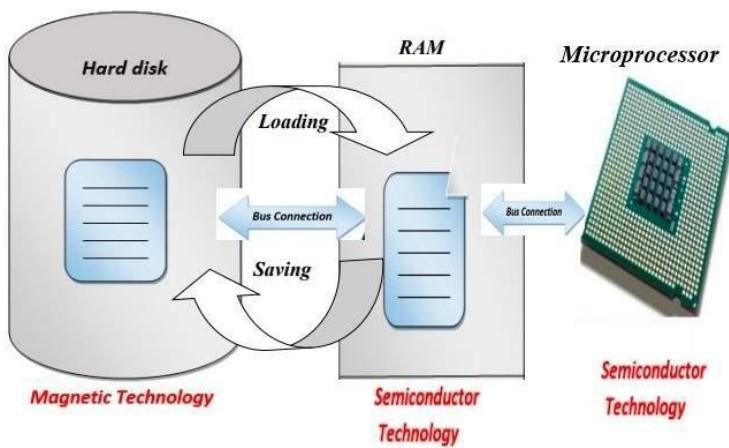
To overcome the speed issue RAM was invented which uses semiconductor technology.

RAM was volatile memory device it requires constant electricity supply. If the electricity is disconnected all the data storage will be erased.



Here comes the requirement of permanent storage device to store data or program. So, Hard Disk was used for storage.

	ADVANTAGES	DISADVANTAGES
RANDOM ACCESS MEMORY (RAM)	Fast	Expensive
	Compact	Volatile
READ ONLY MEMORY (ROM)	Cheap	Slow
	Non - Volatile	Bulky



Success of Java

1. What are the reasons for Java's success?

- a) Portable
- b) Freely downloadable
- c) Open source

2. What is WORA?

Write Once Run Anywhere feature of java.

3. Why is Java called an Internet Programming language?

Because of the portable features of java.

4. What is meant by platform independence?

It refers to the portable feature of java. i.e. java programs can be executed on any system irrespective of the OS.

5. What is meant by open source?

Open source software is such a software for which modifications and enhancements can not only be performed by the Engineers of the company but can also be performed by freelancers or independent thinkers or Engineers of other companies etc.

6. What is an assembler?

Assembler is a software which converts an assembly level language code to machine level language code.

7. What is a compiler?

Compiler is a software which converts high level language code to machine level code (C, C++) or high level language code to bytecode (java compiler).

8. What was Java initially known as?

C++++-, green and oak

9. Who were the inventors of Java?

James Gosling, Mike Sheridan, Patrick Naughton and 8 more members.

10. Which company's product was Java initially?

Sun Micro Systems.

11. Which company's product is Java currently?

Oracle

12. What is JVM?

JVM stands for Java Virtual Machine. It is the software which accepts byte code as its input and produces machine level code as its output.

13. Is the JVM platform dependent or independent? Why?

Platform dependent, because it is coded in C language to increase the execution speed of JVM.

14. What is a class file?

Class file is a file which contains byte code.

15. What is an intermediate code?

Intermediate code is such a code which is neither in high level nor in machine level. It is also called bytecode.

16. What is a class loader?

Class Loader is the part of Java Runtime Environment (JRE) that dynamically loads java classes into the Java Virtual Machine.

17. What is a byte code verifier?

Byte code verifier is part of JRE. It ensures the security of the system. It verifies all the bytecodes before it gets executed.

18. What is an interpreter?

Interpreter is a software which converts high level code to machine level code. Until and unless a statement is interpreted and executed the next high level statement would not get interpreted.

19. What is the difference between compiler and interpreter?

Compiler	Interpreter
All the statements present in the program are compiled	Only the current statement is interpreted
All the statements in program are loaded onto the ram	Only the current interpreted statement would be loaded onto the ram.
Program execution is fast	Program execution is slow

20. What is JIT?

JIT stands for Just In Time Compiler. It is the part of JVM. Compilation would be done during the execution of a program at runtime. It usually runs more quickly in the computer because it compiles the bytecode into platform-specific executable code that is immediately executed. JIT compiler is enabled by default. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.

21. What is JRE?

JRE stands for Java Runtime Environment. It is a portion of the memory allocated on the RAM by the OS for the execution of a java program.

22. Is Java a compiled or interpreted language?

Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into bytecode. At the run time, Java Virtual Machine interprets this bytecode and generates machine code which will be directly executed by the machine in which java programs runs. So java is both compiled and interpreted language.

23. What is Run Time System?

RTS stands for Run Time System ,it is a part of JVM. It is involved in Exception Handling.

24. What is JDK?

JDK stands for Java Development Kit. It consists of all the required software such as Java Compiler, Class Loader, Byte Code Verifier, Java Class Libraries, Java Interpreter, Just In Time Compiler and Runtime System for the development of a java project.

25. What is the latest version of Java available and when was it released?

Java SE 11.0.1, September 2018

26. Which is the next version of Java and when would it be released?

Java SE 12, March 2019

27. How did Java drastically grow in features?

Because Java was open source, many freelancers contributed their ideas and added many features and hence Java grew rich in features.

28. What is the disadvantage of java?

Execution of a Java program is relatively slow when compared to C and C++ because in Java, High level code would be converted into byte code first and then byte code would be converted into machine level code.

29. When was first version of Java officially released?

1994

30. Since when was Java unofficially available?

1992

31. Explain the architecture of Java.

Java combines both the approaches of compilation and interpretation. Java source file is given as input to java compiler, which gives class file which contains byte codes. This class file is given as input to class loader, the duty of class loader is to take all the library files in java class libraries and make a complete class file. This complete class file should be given as input to JVM and JVM is going to convert bytecodes into machine level code. This machine level code is given to processor and hence it gives output.

(For diagram, refer class notes)

32. What is meant by architecture neutrality?

If a software is capable of executing on any OS microprocessor combination then it is said to be architecture neutral.

33. How is object file different from class file?

Object file contains code in machine level whereas Class file contains code in intermediate level i.e. byte code

Object file is machine dependent whereas class file is machine independent.

34. Is class file platform dependent?

No. If a source file has more than one class, each class is compiled into a separate class file. JVMs are available for many platforms, and a class file compiled on one platform will execute on a JVM of another platform. This makes Java applications platform-independent.

35. Is object file platform dependent?

Yes. An object file is a file which contains the program in machine level and hence it is platform dependent

36. Can we have single JVM on a system which has multiple OS?

No, JVM is platform dependent and a single JVM cannot work with many OS. Rather each OS would be having a JVM of its own.

37. What is meant by a platform?

The combination of OS and Hardware(Microprocessor) is called as platform

38. Comment on the speed of Java.

It is relatively slow in execution.

39. What category of applications can be developed using Java?

Internet applications (Portable applications) which demand architecture neutrality can be developed using Java.

40. What is the single most important feature of Java that lead to its success?

Portability

41. What is the difference between JDK and JVM?

JVM is a part of JDK.

For a java program to be successfully executed a collection of software are required. This collection of software which can be freely downloadable from internet is referred to as “Java Development Kit” or JDK, whereas JVM stands for Java Virtual Machine. It is the software which accepts byte code as its input and produces machine level code as its output.

42. What is meant by loading?

Loading is the process of taking a copy of the data present on the hard disk and placing it on the RAM.

43. What is meant by saving?

Saving is the process of taking a copy of the data present on the RAM and placing it on the Hard disk.

44. Why do we have two types of memories in our computer?

Because there is no single memory which can satisfy all the 4 expectations of the user i.e.

- 1) Inexpensive
- 2) Fast
- 3) Non-volatile and
- 4) Compact.

Hence, we have two memories in our computer which can satisfy 2 expectations each.

45. Why is the primary memory called as “main memory”?

Because it is directly connected to the microprocessor.

46. Why is hard-disk called as the “secondary memory”?

Because it is not directly connected to the microprocessor.

47. What is a pointer and does Java support pointers?

Java does not support pointers. However, java has reference referring to an object.

48. What is difference between Path and Classpath?

Path is an environmental variable which is used by the OS to find the executables. Classpath is an environment variable which is used by the Java compiler to find the path of classes. i.e. in J2EE we give the path of jar files.

49. What environment variables do I need to set on my machine in order to be able to run Java programs?

CLASSPATH and PATH

50. What is an object file?

An object file is a file which contains the program in machine level code. It is incomplete and hence cannot be executed.

51. What is an executable file?

An executable file is a file which contains program in machine level code. It is complete because linking has been performed and hence it can be executed.

52. Is an object file executable?

No, because it is incomplete

53. Is an executable file executable?

Yes, because it is complete

54. What is a linkage editor?

It is a software which links the object file with the required library files and produces an executable file.

55. What is linking loader?

It is a software which performs linking of object file with the required library files to produce an executable file and also loads it onto the RAM.

56. What is an executable image?

An executable file when loaded onto the ram, it is called as an executable image.

57. What are library files?

Library files are the files which are created on the hard disk during the installation of a programming language and it would contain bodies to inbuilt functions present in machine level language.

58. Do library files contain code in HLL?

No.

59. Are library files machine dependent?

Yes, because they contain code in machine level language.

60. Why can't we execute an object file?

Because object file is incomplete.

61. Can we send HLL code over internet? Comment.

Yes, we can send it over the internet but it is not recommended due to security reasons.

62. Can we send MLL code over internet? Comment.

Yes, we can send it over the internet. But since MLL code is machine dependent, the portability feature of Java cannot be achieved.

63. Can we send IL code over internet? Comment.

Yes, and it is recommended to send Intermediate level code as it is both secured and machine independent as well.

64. Why can't we have a single JVM to work for all OS?

Because JVM is platform dependent. There is a specific JVM for a specific OS.

65. What are the reasons for java's slowness?

A java program does not directly produce the machine code. Rather, upon compilation it produces intermediate code. This intermediate code must be converted into machine code by JVM. This slows down the execution process of java program.

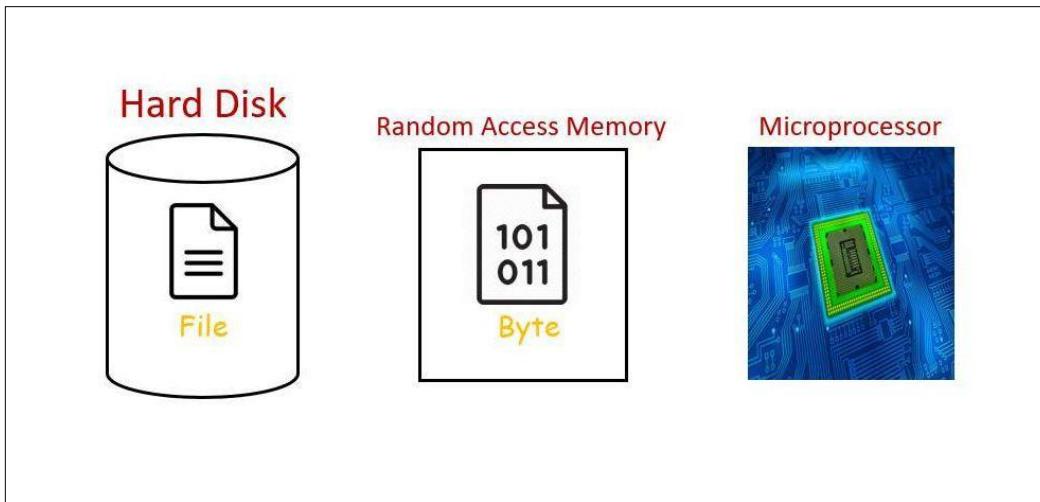
Features of JAVA

OPERATING SYSTEM

The heart of computer in terms of hardware is *Microprocessor*. But there must be another 2 memory devices are *Hard disk* and *RAM*.

All the hardware devices don't have ability to work on their own.

There is a requirement of software to provide instructions which referred to as "*Operating System*".



Platform Dependence

What is Platform?

Computer is a combination of hardware and software this is called as "*PLATFORM*". Hardware mostly refers to microprocessor. Software mostly refers to Operating System.

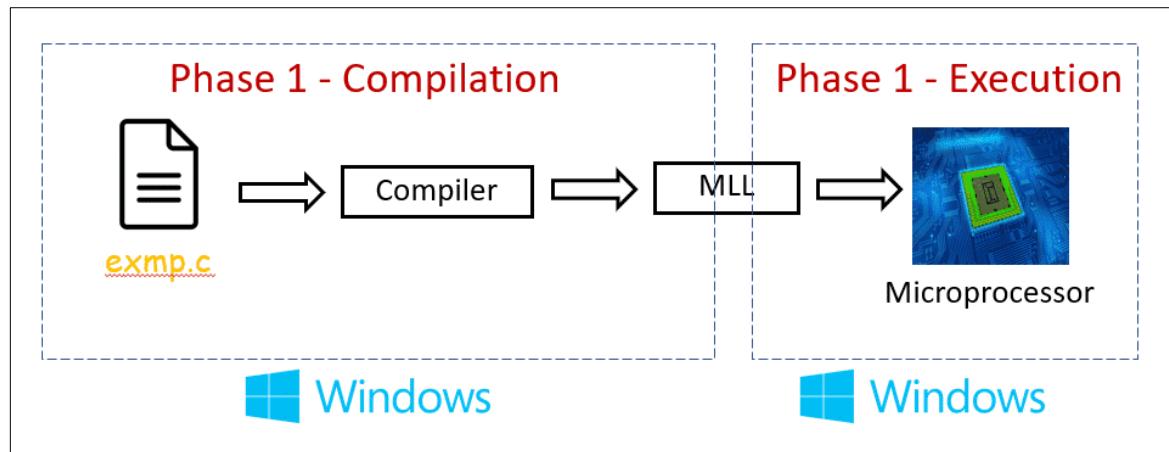
The combination of microprocessor and OS is called as platform.

For example:

- i5 is processor and Windows 10 is OS. This is platform.
- i3 is processor and Mac is OS. This is platform.
- i3 is processor and Linux/Unix is OS. This is platform.

DIFFERENT CASES IN PLATFORM DEPENDENCY

CASE 1



There are few programming languages are platform dependent / OS dependent.

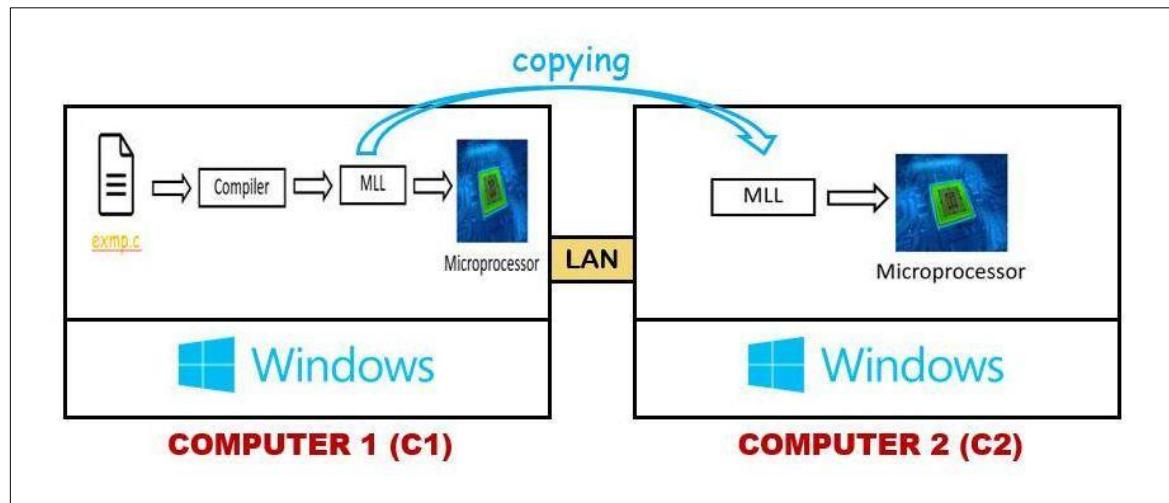
For example: C language & C++ language

The process from typing a program to get a output can be divided into 2 Phase i.e

1. Compilation phase
2. Execution phase.

C programming languages are platform dependent because such programming languages which expects that the platform of compilation and platform of execution must be same.

CASE 2

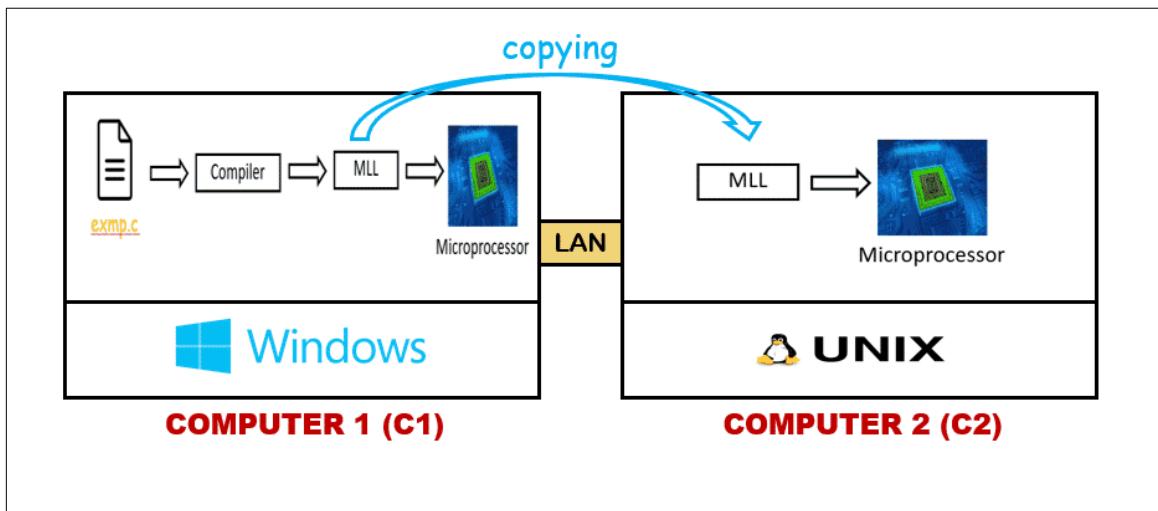


Here we have 2 computers i.e., Computer 1(C1) and Computer 2(C2) are connected to each other using LAN connection [Local Area Network] because data can be transferred from C1 to C2.

C1 will execute and give output, what if copy of machine level code is given to C2. Will we get the output?

If the platform is same, we will get the output and MLL will be executed on C2.

CASE 3

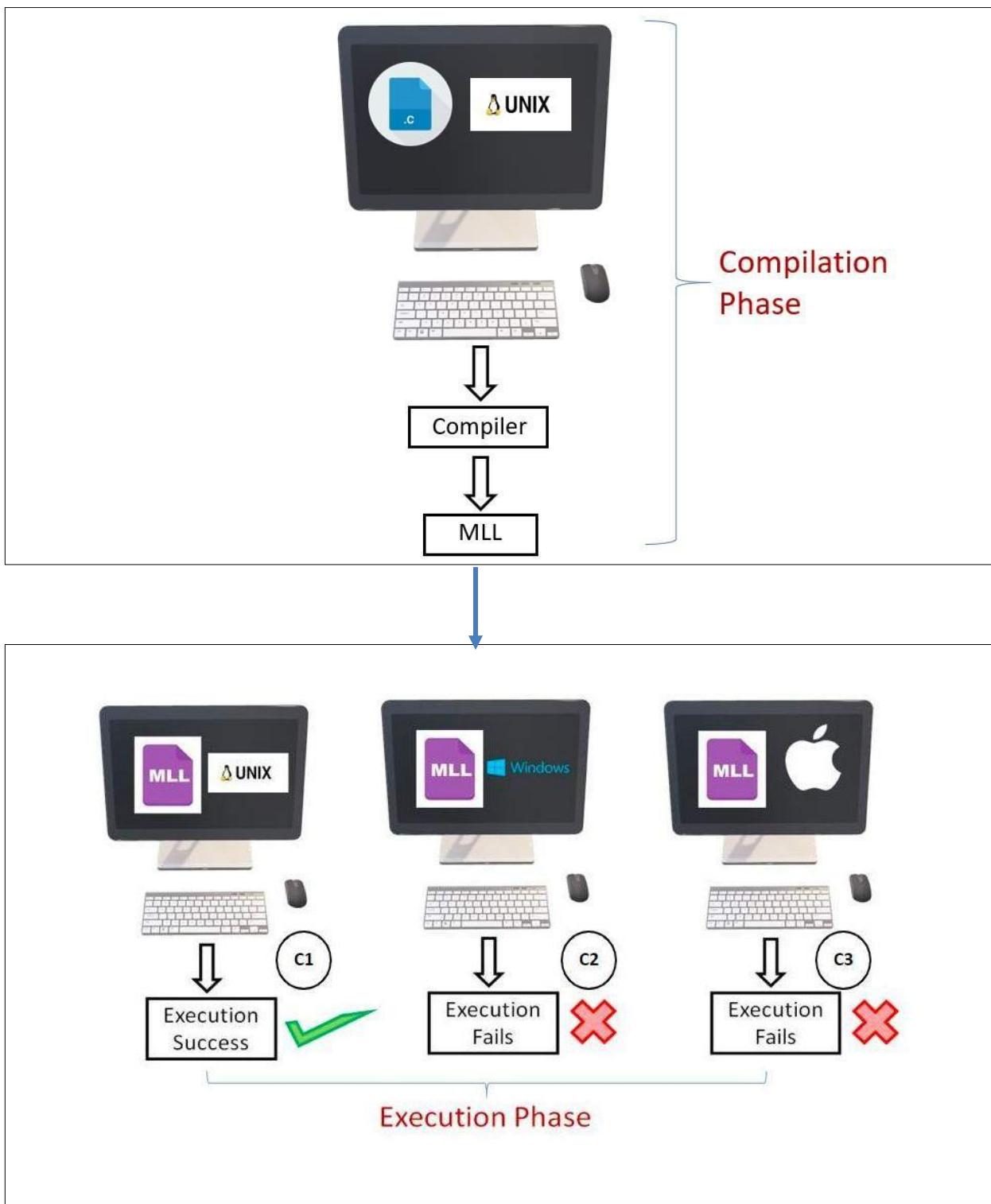


Here Computer 1 has windows operating system and Computer 2 has Linux operating system. The platform of two computers is different.

So, machine level language code in Computer 2 (C2) will not be executed.

Hence, platform dependency means the operating system of compilation and operating system of execution should be same.

Failure of platform dependent programming language



In C2 and C3 platform of compilation and platform of execution is different. So, the execution fails because usage of platform dependent programming language.

JAVA

James Gosling invented a platform independent programming language called as JAVA in the year **1994**.

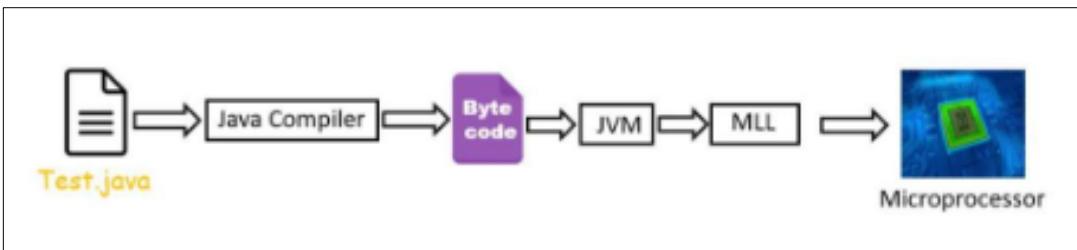
At first Java was called as **OAK**.

Java was originally designed for interactive TV, but the digital cable industry wasn't ready for the technology.

Java achieving platform independence

PLATFORM INDEPENDENCE

JAVA designed its own compiler called as Java Compiler which converts HLL into byte codes. This byte code is given to Java Virtual Machine (JVM) which converts it to MLL.



WORKING:

Let us assume you are writing code using java in your computer which has windows OS. User should save the file with the extension ".java" so that it is consider to be a java file.

For example, consider **Test.java** is a file which consist of HLL code. Since Machine understands **Machine Level code [MLL]** not your **high-level code [HLL]**, conversion must happen.

Let us see how exactly conversion happens in java.

Initially your HLL code is given as input to compiler but java compiler will not give MLL code as output like C and C++ compiler rather it takes **HLL** as input and gives a special type of code as output called as "**BYTE CODE**" which is platform independent.

Byte code is neither **HLL** code nor **MLL** code, hence it is also referred to as *intermediate code*.

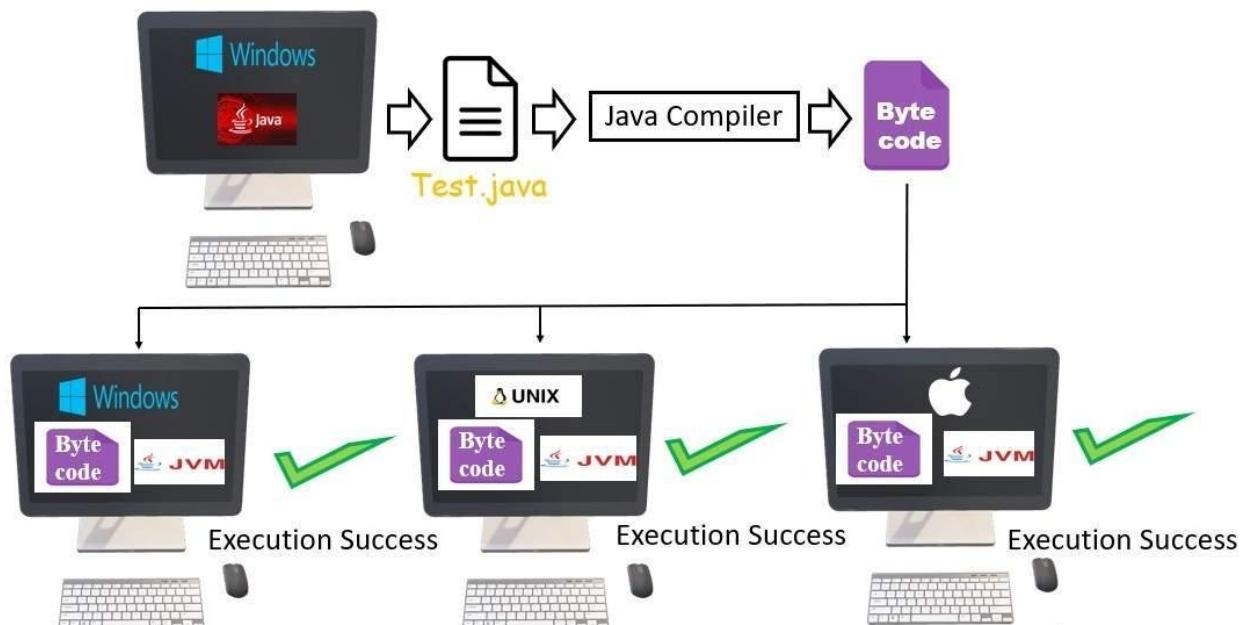
If you can recollect machine understands only **MLL** code but **java compiler gave you byte code**.

To resolve this, **James Gosling** provided a software called as "**Java virtual Machine**"(**JVM**) which was platform dependent that is different OS have different **JVM**.

Since you are writing code on windows OS, you will have to download windows *compatible JVM*.

JVM will now convert byte code to machine level code which machine can easily understand.

In this way, java achieved platform independence using a special type of code which is byte code.



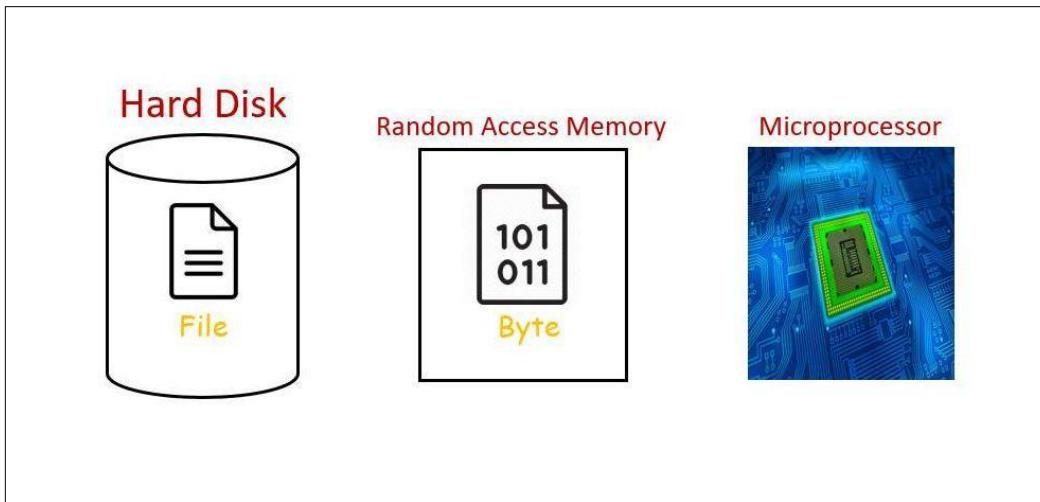
Features of JAVA

OPERATING SYSTEM

The heart of computer in terms of hardware is *Microprocessor*. But there must be another 2 memory devices are *Hard disk* and *RAM*.

All the hardware devices don't have ability to work on their own.

There is a requirement of software to provide instructions which referred to as "*Operating System*".



Platform Dependence

What is Platform?

Computer is a combination of hardware and software this is called as "*PLATFORM*". Hardware mostly refers to microprocessor. Software mostly refers to Operating System.

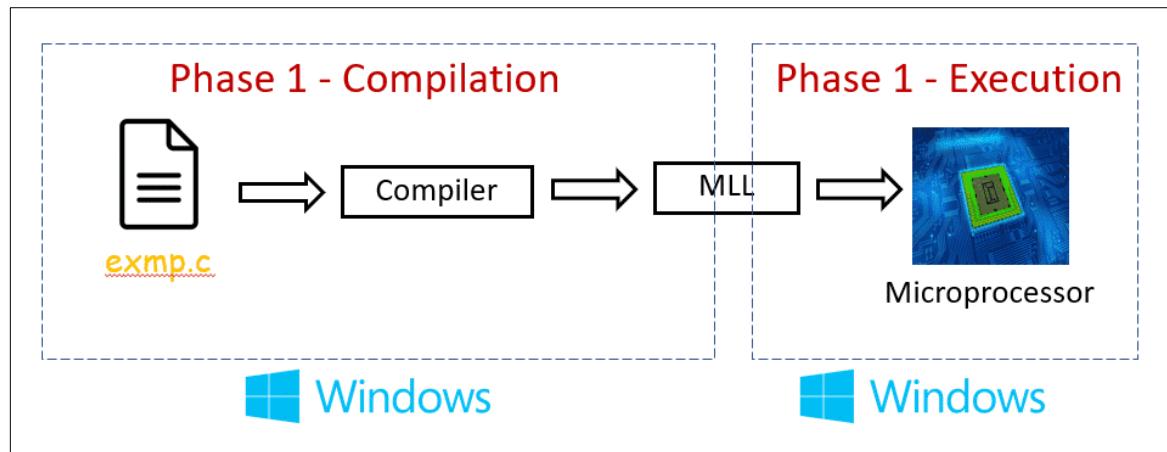
The combination of microprocessor and OS is called as platform.

For example:

- i5 is processor and Windows 10 is OS. This is platform.
- i3 is processor and Mac is OS. This is platform.
- i3 is processor and Linux/Unix is OS. This is platform.

DIFFERENT CASES IN PLATFORM DEPENDENCY

CASE 1



There are few programming languages are platform dependent / OS dependent.

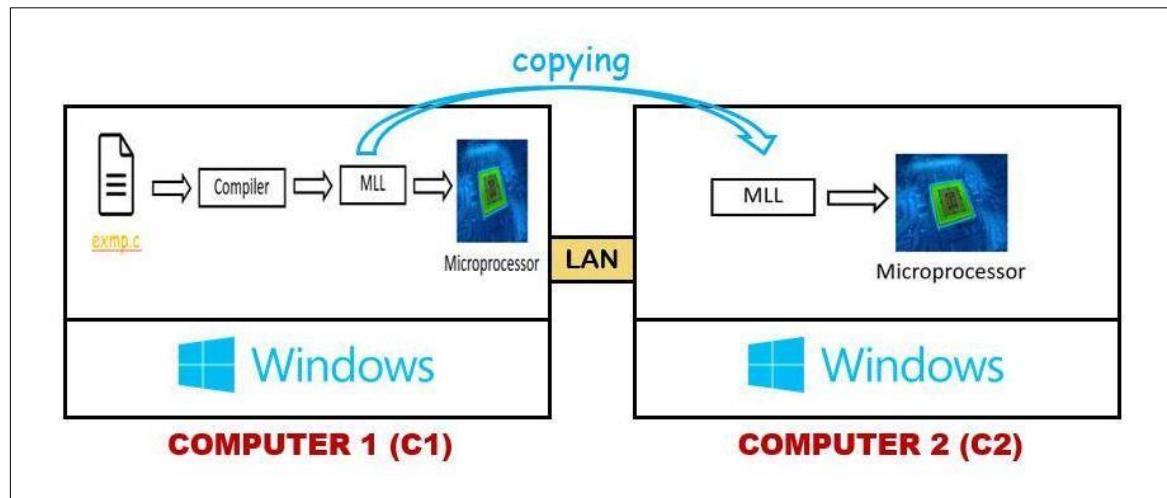
For example: C language & C++ language

The process from typing a program to get a output can be divided into 2 Phase i.e

1. Compilation phase
2. Execution phase.

C programming languages are platform dependent because such programming languages which expects that the platform of compilation and platform of execution must be same.

CASE 2

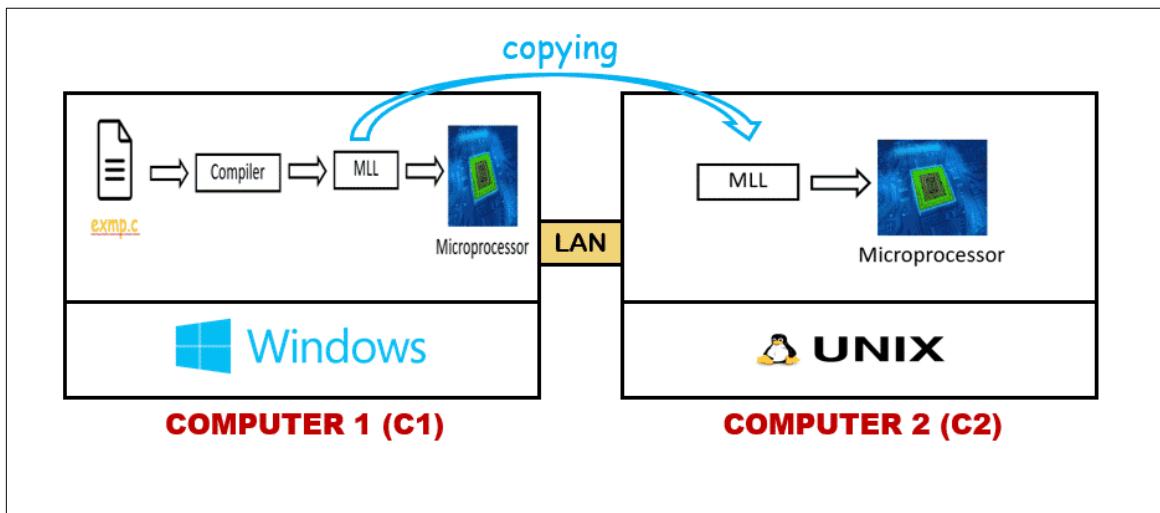


Here we have 2 computers i.e., Computer 1(C1) and Computer 2(C2) are connected to each other using LAN connection [Local Area Network] because data can be transferred from C1 to C2.

C1 will execute and give output, what if copy of machine level code is given to C2. Will we get the output?

If the platform is same, we will get the output and MLL will be executed on C2.

CASE 3

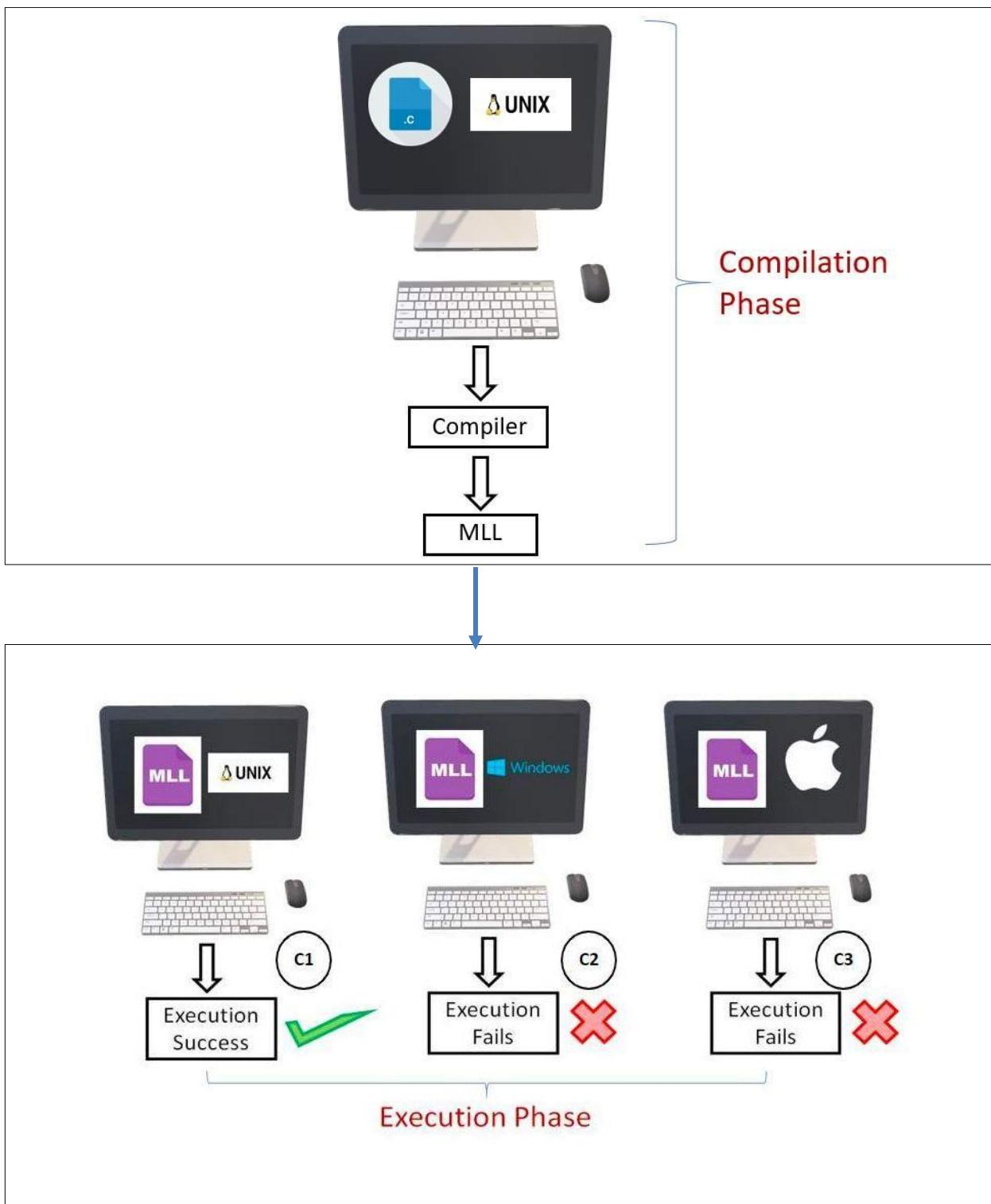


Here Computer 1 has windows operating system and Computer 2 has Linux operating system. The platform of two computers is different.

So, machine level language code in Computer 2 (C2) will not be executed.

Hence, platform dependency means the operating system of compilation and operating system of execution should be same.

Failure of platform dependent programming language



In C2 and C3 platform of compilation and platform of execution is different. So, the execution fails because usage of platform dependent programming language.

JAVA

James Gosling invented a platform independent programming language called as JAVA in the year **1994**.

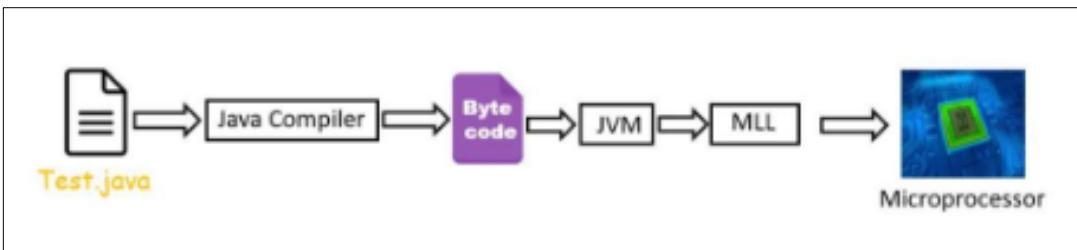
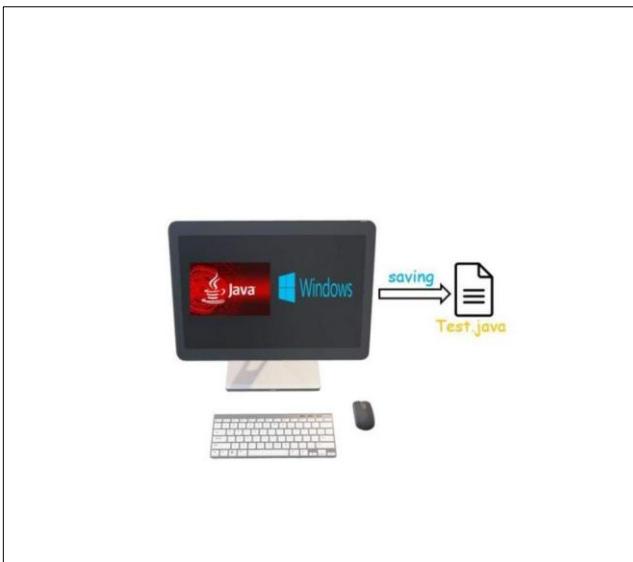
At first Java was called as **OAK**.

Java was originally designed for interactive TV, but the digital cable industry wasn't ready for the technology.

Java achieving platform independence

PLATFORM INDEPENDENCE

JAVA designed its own compiler called as Java Compiler which converts HLL into byte codes. This byte code is given to Java Virtual Machine (JVM) which converts it to MLL.



WORKING:

Let us assume you are writing code using java in your computer which has windows OS. User should save the file with the extension ".java" so that it is consider to be a java file.

For example, consider **Test.java** is a file which consist of HLL code. Since Machine understands **Machine Level code [MLL]** not your **high-level code [HLL]**, conversion must happen.

Let us see how exactly conversion happens in java.

Initially your HLL code is given as input to compiler but java compiler will not give MLL code as output like C and C++ compiler rather it takes **HLL** as input and gives a special type of code as output called as "**BYTE CODE**" which is platform independent.

Byte code is neither **HLL** code nor **MLL** code, hence it is also referred to as *intermediate code*.

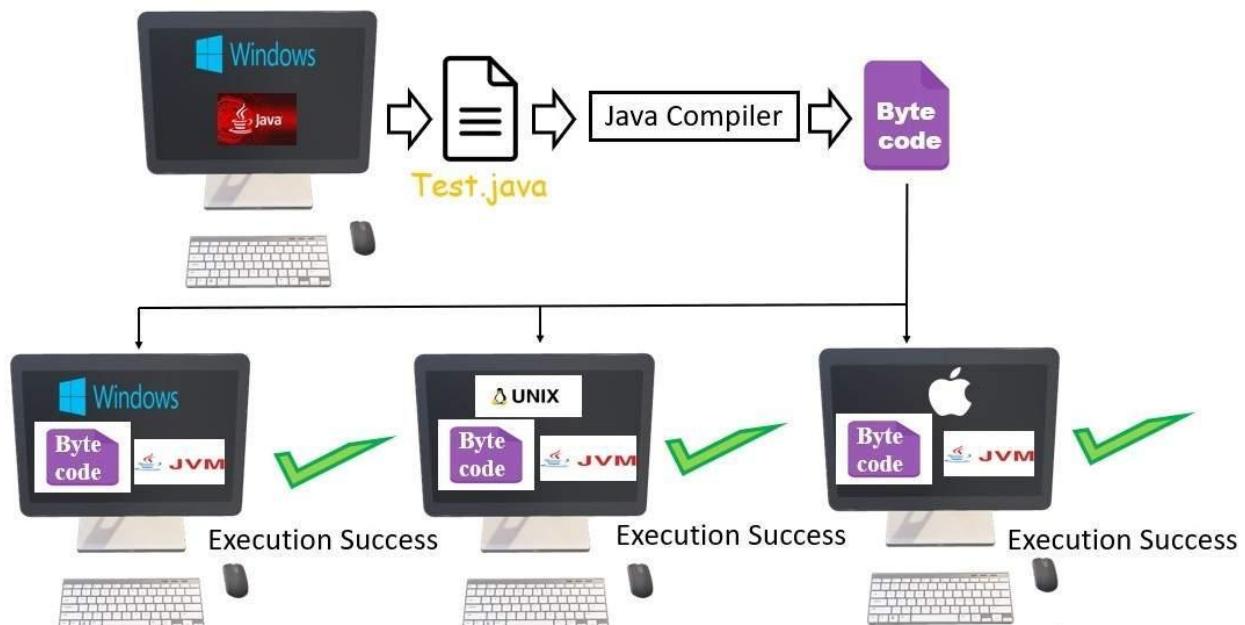
If you can recollect machine understands only **MLL** code but **java compiler gave you byte code**.

To resolve this, **James Gosling** provided a software called as "**Java virtual Machine**"(**JVM**) which was platform dependent that is different OS have different **JVM**.

Since you are writing code on windows OS, you will have to download windows *compatible JVM*.

JVM will now convert byte code to machine level code which machine can easily understand.

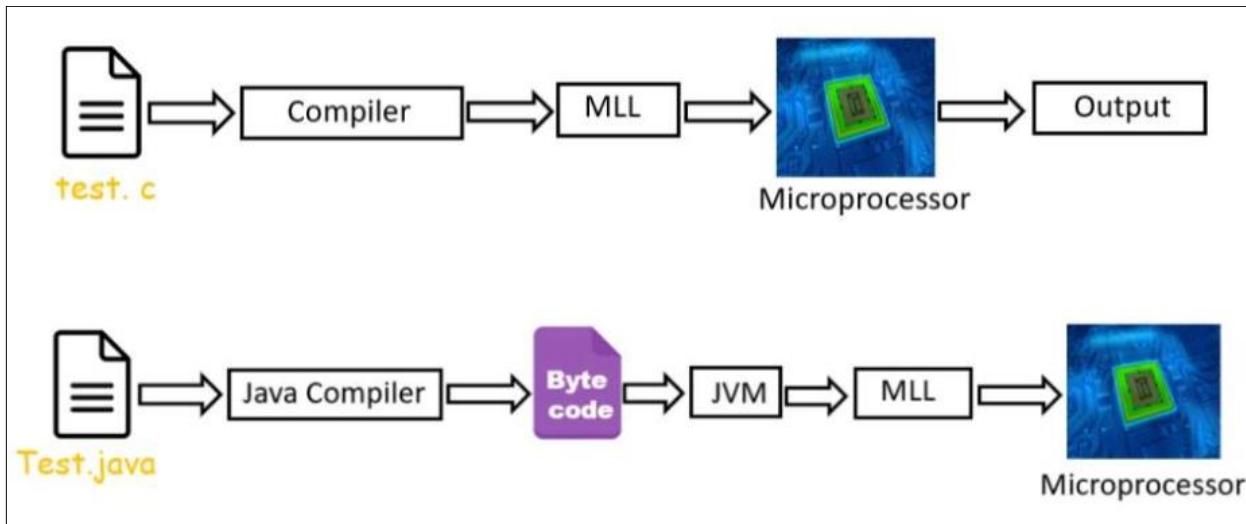
In this way, java achieved platform independence using a special type of code which is byte code.



Object Orientation

Disadvantage of Java

- In java, extra step is involved in conversion from HLL to java code to MLL code java program are relatively slower in execution when compared to C/C++ programs.
- Bullet
- Time consumption is more in java.
- bullet
- Speed of execution is slower.



Flowchart 1 is execution steps in C and Flowchart 2 is execution steps in Java

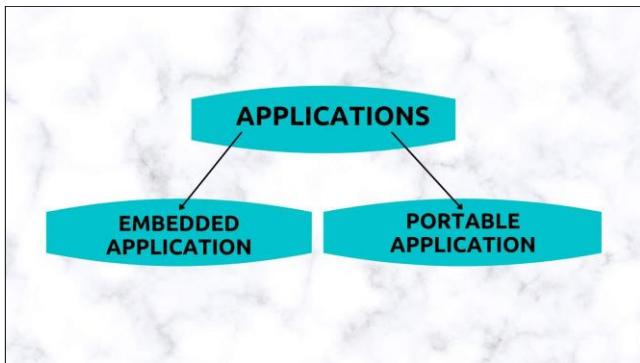
Introduction to Application

Applications

There are two types of applications

Embedded application - Speed of execution is at most important.

Portable application - Platform independence is at most important.



Embedded application

An embedded application is software that is placed permanently inside some kind of device to perform a very specific set of functions.

The program instructions for embedded systems are called firmware, or embedded software, and are stored in read-only memory, or flash memory chips.

For embedded applications Speed is very important factor.



Portable application

A portable application (portable app), sometimes also called standalone application, is a program designed to read and write its configuration settings into an accessible folder in the computer, usually in the folder where the portable application can be found.

This makes it easier to transfer the program with the user's preferences and data between different computers.

A program that doesn't have any configuration options can also be a portable application.

Portable applications can be stored on any data storage device, or in other words these applications are platform independent.

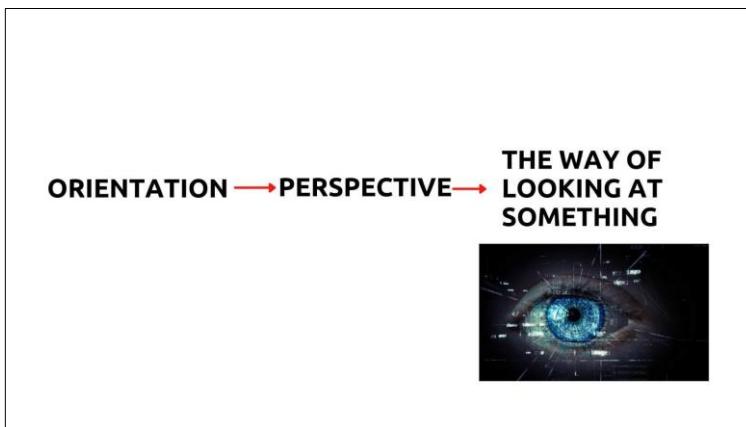


Object Orientation

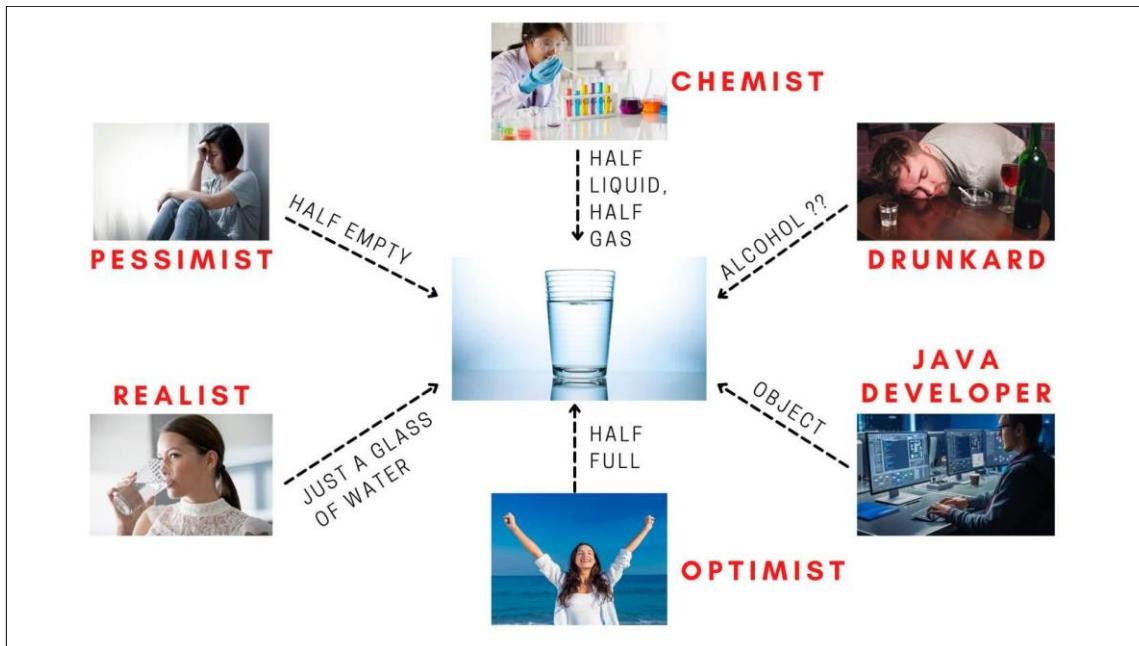
Object Orientation is the way of looking at this world as a collection of objects.

In this world no object is completely useless. All objects are in constant interaction with each other.

No object exists in isolation.



Let us take an example of glass of water and try to understand in better way:



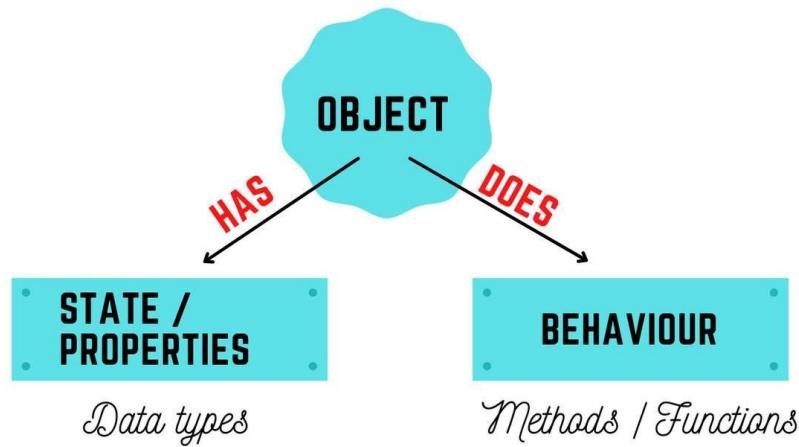
Java became famous because of two main reasons: -

- Platform Independence.
- Object Orientation.

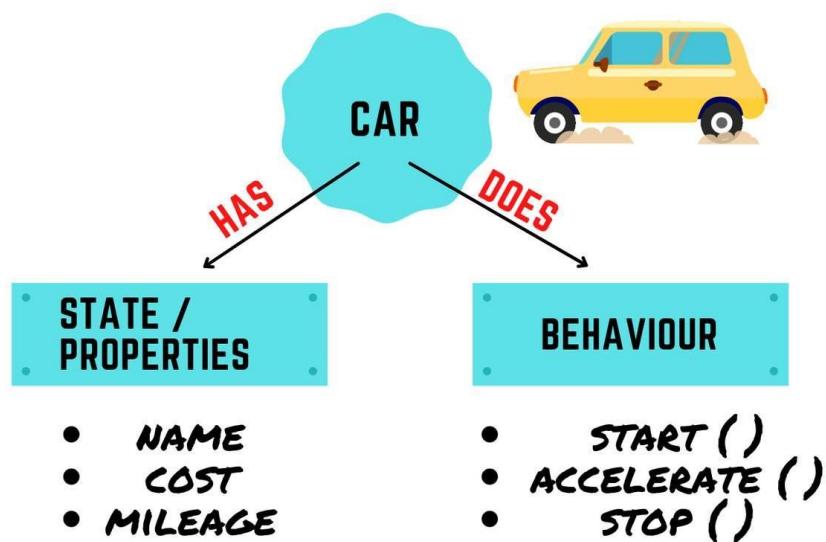
RULES OF OBJECT ORIENTATION

- The world is nothing but collection of objects.
- Every object in the world belongs to type.
- In Java, every object has 2 parts i.e.,
 - State / properties means what an object has.
 - Behavior means what an object does.
- To take care of state we should use **datatypes**.
- To take care of behavior we should use **Methods / Functions**.

Examples for Object Orientation



EXAMPLE 1: CAR OBJECT



CODE

```
class Car
{
    //has part or state
    String name;
    int cost;
    float mileage;

    //does part or behavior
    void start()
    {
        //body of the method;
    }
    void accelerate()
    {
        //body of the method;
    }
    void stop()
    {
        //body of the method;
    }
}
```

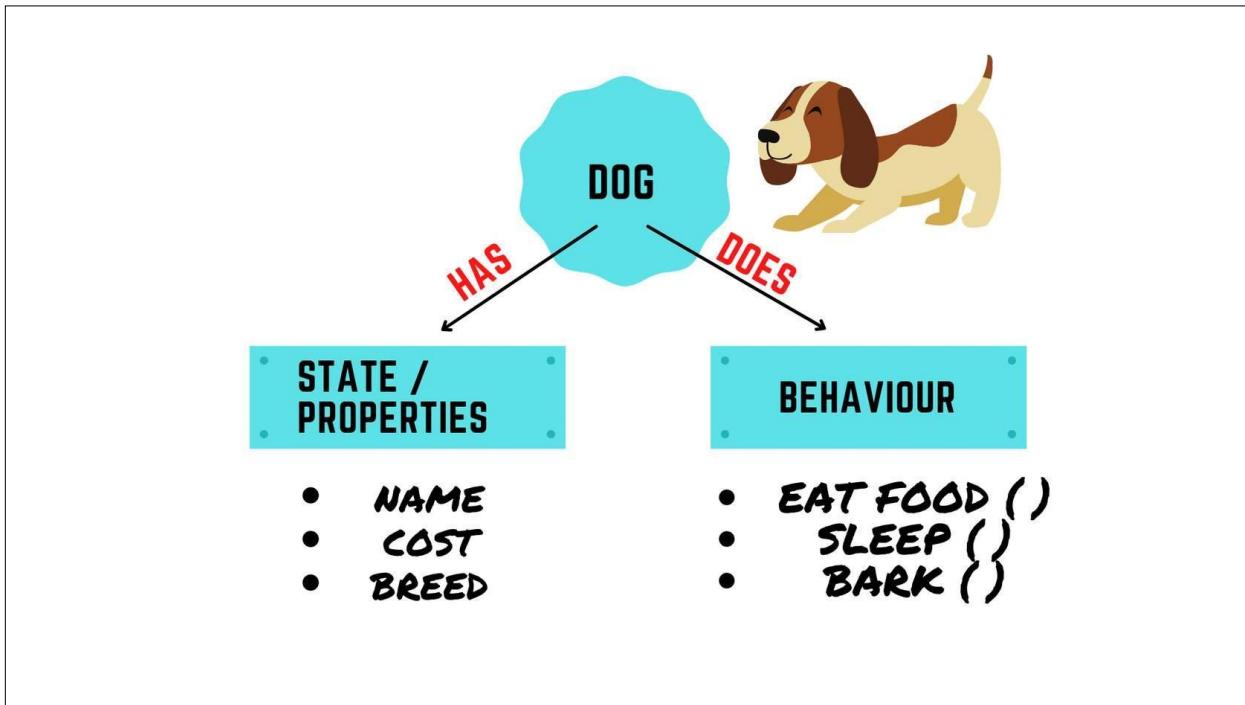
Let's see how to create an object in java:

```
Car c1 = new Car(); // object 1 creation
Car c2 = new Car(); // object 2 creation
```

```
c1.start(); //object 1 calling start ()
c1.accelerate(); //object 1 calling accelerate ()
c1.stop(); //object 1 calling stop ()
```

```
c2.start(); //object 2 calling start ()
c2.accelerate(); //object 2 calling accelerate ()
c2.stop(); //object 2 calling stop ()
```

EXAMPLE 2: DOG OBJECT



CODE

```
class Dog
{
    //has part or state
    String name;
    int cost;
    float breed;

    //does part or behavior
    void eatFood()
    {
        //body of the method;
    }
    void sleep()
    {
        //body of the method;
    }
    void bark()
    {
```

```
//body of the method;  
}  
}
```

Let's see how to create an object in java:

```
Dog d1 = new Dog(); // object 1 creation  
Dog d2 = new Dog(); // object 2 creation
```

```
d1.eatFood(); //object 1 calling eatFood ()  
d1.sleep(); //object 1 calling sleep ()  
d1.bark(); //object 1 calling bark ()
```

```
d2.eatFood(); //object 2 calling eatFood ()  
d2.sleep(); //object 2 calling sleep ()  
d2.bark(); //object 2 calling bark ()
```

Main Method

Object creation in java

How to create an object in java?

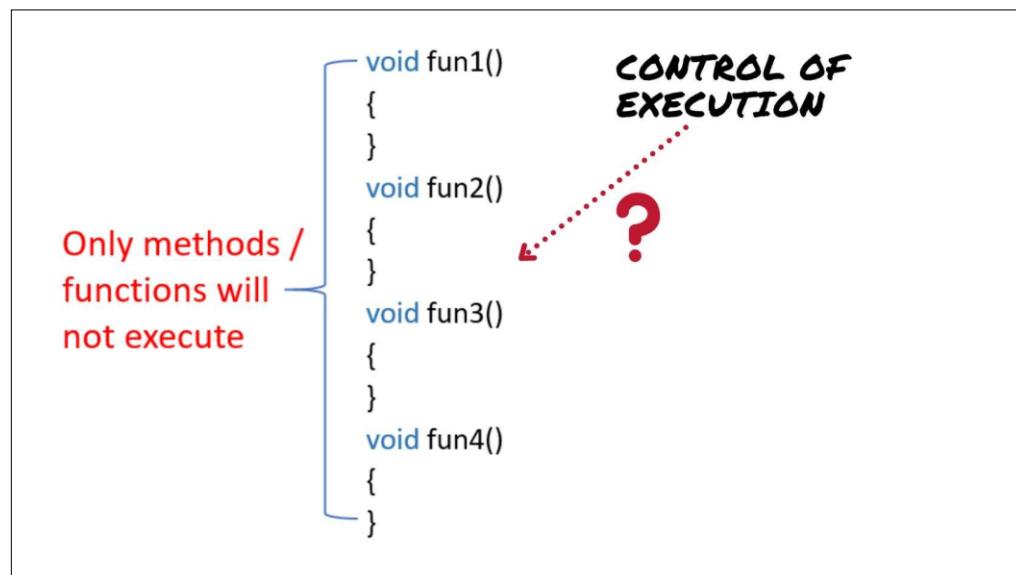
A class provides the blueprints for objects. So basically, an object is created from a class.

In Java, the "new" keyword is used to create new objects. There are three steps when creating an object from a class –

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

Syntax: class_name object_name = new class_name();

For example: Car c1 = new Car();



Even there are 4 functions or method present in this program will not get executed because it requires the control of execution to begin.

The control of execution is nothing but "main method / function" in C language.

Only methods /
functions will
be execute

```
void main() {  
    void fun1()  
    {  
    }  
    void fun2()  
    {  
    }  
    void fun3()  
    {  
    }  
    void fun4()  
    {  
    }  
}
```

**CONTROL OF
EXECUTION**

main ()

Main method in Java

C Language

```
#include <stdio.h>
void main()
{
    printf("HELLO WORLD");
}
```



Main Method in JAVA

Step 1

```
class Demo
{
    public void main()
    {
        System.out.print("HELLO WORLD");
    }
}
```



In Java, basics rule of java is that every single method / function compulsory present within a Class.

Will the above program execute?

Control cannot see the main method because we have enclosed by class Demo.

To make the main method visible to the control though it is enclosed within class.

How can we make it Visible?

By attaching **PUBLIC**

Step 2

```
class Demo
{
    public static void main()
    {
        System.out.print("HELLO WORLD");
    }
}
```



Will the above program execute?

Control can see the main method because we have declared it has public unfortunately it is not accessible by the control.

To make the main method visible to the control though it is enclosed within class.

How can we make it Accessible?

By attaching **STATIC**

Step 3

```
class Demo
{
    public static void main(String [] args)
    {
        System.out.print("HELLO WORLD");
    }
}
```



Will the above program execute now?

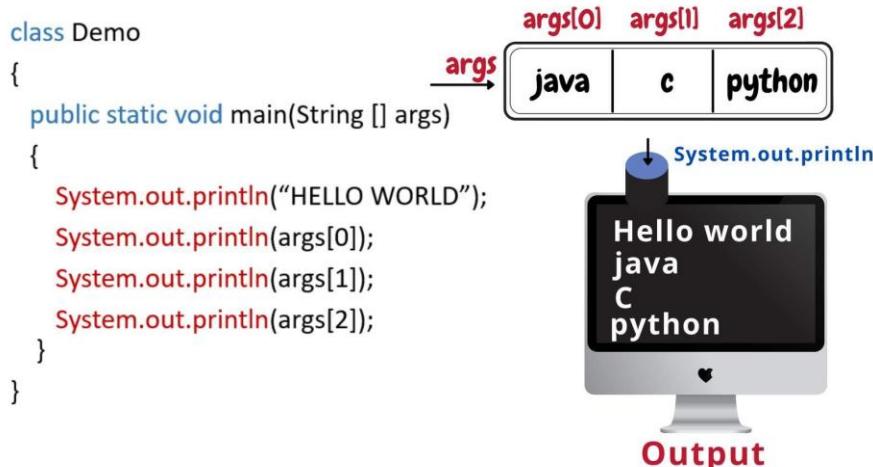
Unfortunately, it will not show the output because JVM will start looking for Identifier as a starting point.

What is this Identifier?

It is nothing but String [] args

It is the command line argument which stores data and it is nothing but an array.

Step 4



String [] args

What is args?

Arguments means data passed on the command line.

Args is technically called as "Dynamic Array" using which the command line arguments are collected.

In java, along with the control of execution we can also give inputs/ data to the main method.

To collect the input / data there is args. Initially args is an empty basket.

Different way to declare Syntax / Signatures

- public static void main (String [] args)
- static public void main (String [] args)
- public static void main (String args [])
- public static void main (String...args)

Key points

- Always save the source file same as the class name in which main() is present along with the .java extension if it is written in notepad.
- When javac source_file_name.java is run in command prompt then a .class file creates which is in byte code format.
- To convert the byte code file to machine level all you have to do is run the following line in command prompt: java source_file_name
- If the main method is not made as public then during execution an error is popped saying main method not found.
- If the main method is not static then during execution it shows error as main method not static.

**MAIN
METHOD**

1. Should a main() method be compulsorily declared in all java programs?

Yes.

2. What is the return type of the main() method?

void.

3. Why is the main() method declared static?

So that the main method becomes accessible to OS even without object creation

4. What is the argument of main() method?

String args[] which is a variable sized array(dynamic array) which is used to collect command line arguments.

5. Does the order of public and static declaration matter in main() method?

No

6. What is meant by orientation?

Orientation refers to Perspective (or) Point of view (or) the way of we look at the things

7. Why should we enclose main() method within a class?

It is a rule in java that every method should be enclosed within a class. The JVM has to start the application somewhere. As Java does not have a concept of “things outside of a class”, the method has to be inside a class.

8. What happens if we do not declare main() as public?

An error message would be displayed as “Main method not found error”.

9. What happens if we do not declare main() as static?

An error message would be displayed as “Main method is not static error”

10.Which command in java is used to invoke the java compiler?

java

11.Which command in java is used to invoke JVM?

java

12.Why should main() method be present in all java applications?

Because OS can handover the control to main method and from the main method execution will start.

13.What are command line arguments?

The command line argument is an argument passed to a program at the time when we run it. To access the command-line argument inside a java program is easy because they are stored in dynamic array String args[]

14.What is the use of command line arguments?

Input can be given to main method through the command line arguments.

15.How does java collect command line arguments?

In the form of String arguments.

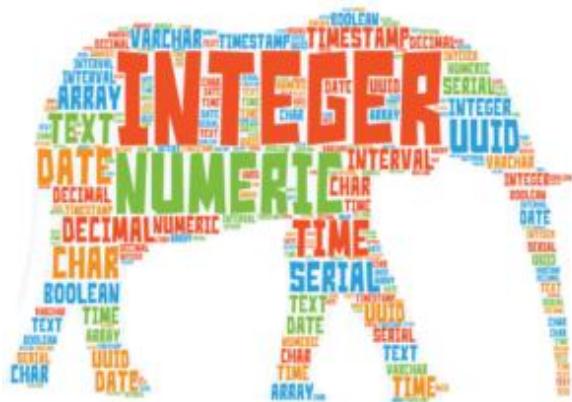
16. How many command line arguments can be passed to a java program?

Any number of Command line arguments can be passed.

17. What happens if a java program expects command line arguments but is not sent to it?

An exception would occur and message would be displayed as “Array

index out of bounds Exception”.



DATA TYPES

Before going ahead with what is data types why do we need it. We should first know **how a data is stored in a system.**

In every electronic system we have **RAM which stores the data temporarily.** All the data that we enter

is in high level but it is always stored in low level inside the system so that the system/machine can understand.

So **RAM** consists of **several bytes** which consists of 8bits, each bit has two **transistors** which stores **high and low value** (1's and 0's).



What does *Data Type* mean?

The data type of a value (or variable in some contexts) is an attribute that tells what kind of data that value can have. Most often the term is used in connection with static typing of variables in programming languages like C/C++, Java and C# etc, where the type of a variable is known at compile time. Data types include the storage classifications like integers, floating point values, strings, characters etc.



Data types define particular characteristics of data used in software programs and inform the compilers about predefined attributes required by specific variables or associated data objects.

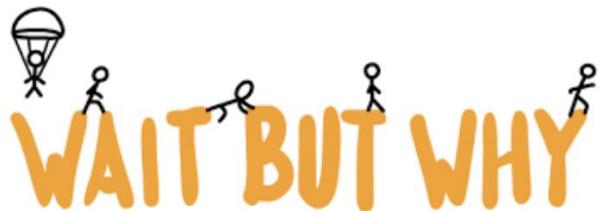
Why Data Types Are Important

Data types are especially important in Java because it is a strongly typed language. This means that all operations are type-checked by the compiler for

type compatibility. Illegal operations will not be compiled. Thus, strong type checking helps prevent errors and enhances reliability. To enable strong type checking, all variables, expressions, and values have a type. There is no concept of a “type-less” variable, for example. Furthermore, the type of a value determines what operations are allowed on it. An operation allowed on one type might not be allowed on another.

Why Data types are required?

Data types are required so that you know what "kind" of data your variable holds, or can hold. If you (and compiler, and runtime and what not) know this information in advance, you can save a lot of runtime issues at compile time only.

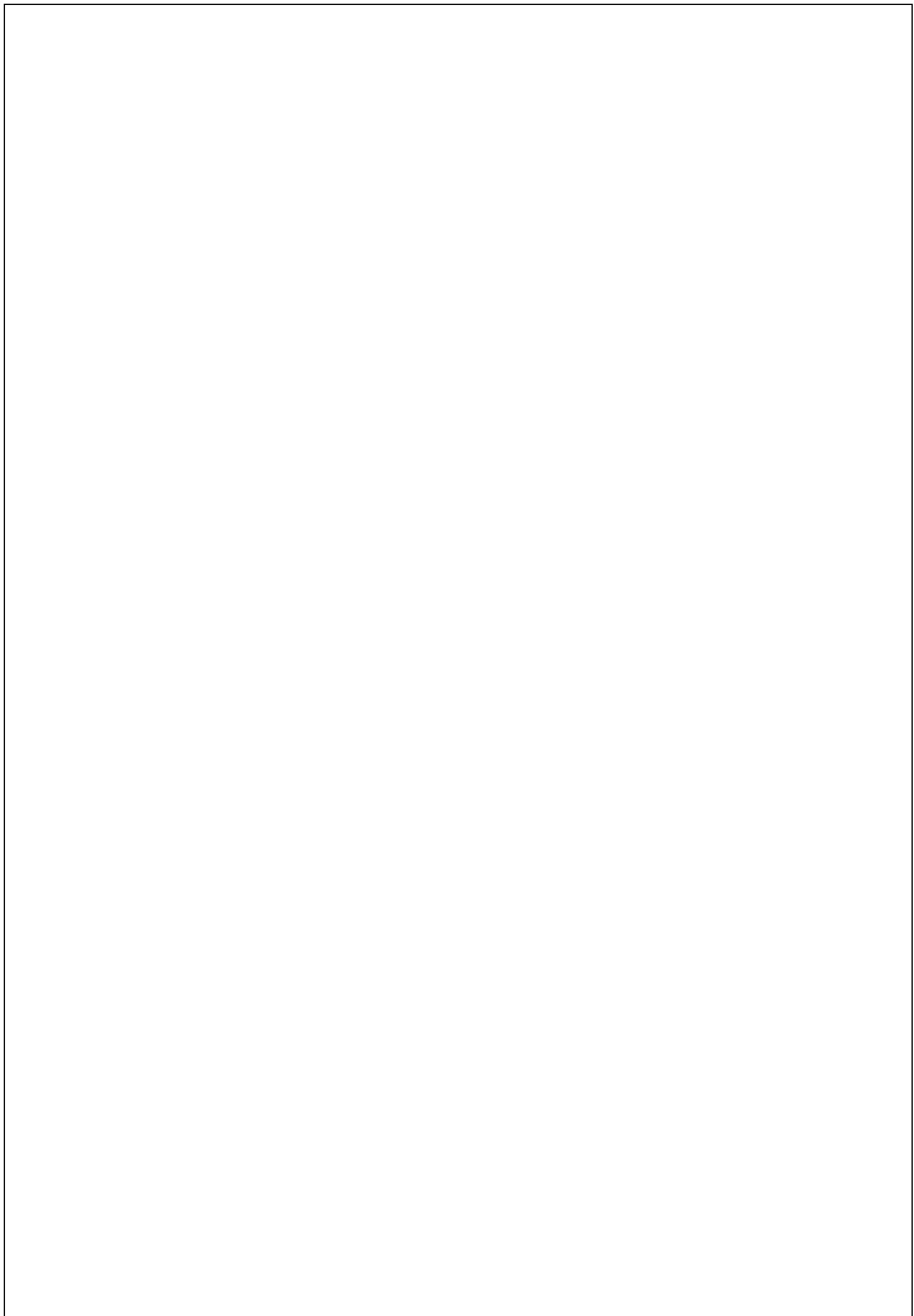


For example, in Java, when you declare an int variable, compiler and JVM knows that this variable is going to hold only integer values, nothing else. But when you declare a variable in Javascript using var, you are just declaring a name. This variable can hold any value, be it a primitive type or a complex object. Here, you can use a variable that has string value in an arithmetic expression, and you won't be able to notice until you run and test it.

There are many more such reasons but first, **reasonable flexibility**. Let's imagine the opposite: a language with a single type. Let's make that *integer*; you can only have operations on integer numbers. Naturally, this language is very limited in its usefulness: it doesn't have *strings* to ask people for names or addresses. It doesn't have *booleans* with which to evaluate the truthfulness of a logical proposition over those numbers and therefore is incapable of decision-making. And finally, as you can only have one type of number, you're missing on the possibilities of calculating anything that may fall outside that numeric set. It'd be less useful than a pocket calculator.

As you can see, a greater variety data types allow you tackle a great number of problems, and any programming language worth learning should even give you tools to create *your own types*.



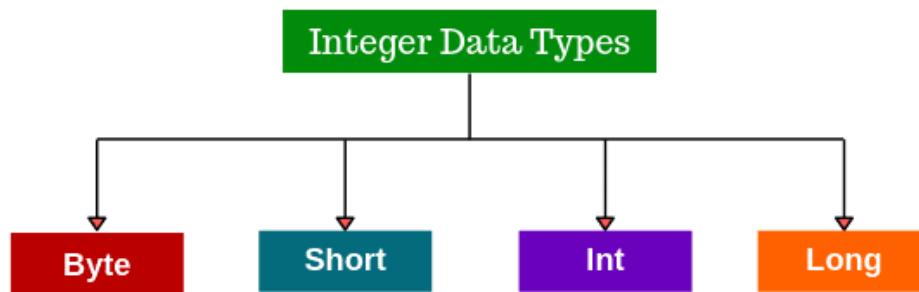


Integer type data:

Integer types of data represent integer number without any fractional parts or decimal points.

For example: age of a person, number of students in a class, distance between planets, distance between galaxy, etc.,

In Java, to store Integer type data we have four data types.



Now the question arises that to store Integer type data why do we require four data types??

Before that let's have a look on a formula

It is important for one to understand that, given a certain amount of memory how much data can be stored inside it. There is a certain maximum value and minimum value that can be stored and the formula to find it is:

For n bits: The minimum value that can be stored is -2^{n-1}

The maximum value that can be stored is $+2^{n-1} - 1$

1) byte is the smallest integer data type which has the least memory size allocated.

Assume, if we create a byte type variable as **byte a;**

It means in the memory **one byte** is allocated and it is referred as **a**.



To convert byte to bits we have, **1 byte = 8 bits**.

And therefore, the minimum value is $-2^{8-1} = -2^7 = -128$

The maximum value is $+2^{8-1} - 1 = +2^7 - 1 = +127$

Now let's see few examples to understand the above topic.

Example-1:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = 127;
        System.out.println(a);
    }
}
```

Output:

127

Example-2:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = 128;
        System.out.println(a);
    }
}
```

Output:

Compilation error (because this is out of range).

Example-3:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = -128;
        System.out.println(a);
    }
}
```

Output:

-128

Example-4:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = -129;
        System.out.println(a);
    }
}
```

Output:

Compilation error (because this is out of range).

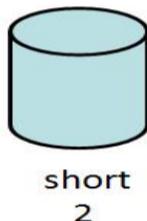
Which means, one value more than +127 or one value less than -128 cannot be stored but any value between the range of -128 to +127 can be stored using a byte type variable.

Therefore, the age of a person, the current month of the year etc., can be stored using byte data type.

2) short is a signed 16-bit byte.

Assume, if we create a short type variable as **short a;**

It means in the memory **two bytes** is allocated and it is referred as **a**.



Converting byte to bits: **2 bytes = 16 bits**.

And therefore, the minimum value is $-2^{16-1} = -2^{15} = -32768$

The maximum value is $+2^{16-1} - 1 = +2^{15} - 1 = +32767$

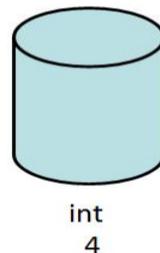
Which means, one value more than +32767 or one value less than -32768 cannot be stored but any value between the range of -32768 to +32767 can be stored using a short type variable.

Therefore, salary of a fresher, height of Mount Everest etc., can be stored using short data type.

3) **int** is a signed 32-bit byte. It is the most commonly used integer type. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.

Assume, if we create an int type variable as **int a;**

It means in the memory **four bytes** is allocated and it is referred as **a**.



Converting byte to bits, **4 bytes = 32 bits**.

And therefore, the minimum value is $-2^{32-1} = -2^{31} = \textcolor{orange}{-2,147,483,648}$

The maximum value is $+2^{32-1} - 1 = +2^{31} - 1 = \textcolor{orange}{+2,147,483,647}$

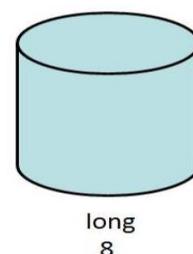
Which means, one value more than +2,147,483,647 or one value less than -2,147,483,648 cannot be stored but any value between the range of -2,147,483,648 to +2,147,483,647 can be stored using an int type variable.

Therefore, distance between planets etc., can be stored using int data type.

4) **long** is a signed 64-bit byte and is useful for those occasions where an int type is not large enough to hold the desired value. The range of long is quite large. This is useful, when big whole numbers are needed.

Assume, if we create an long type variable as **long a;**

It means in the memory **eight bytes** is allocated and it is referred as **a**.



Converting byte to bits, **8 bytes = 64 bits**.

And therefore, the minimum value is $-2^{64-1} = -2^{63} = \textcolor{orange}{-9,223,372,036,854,775,808}$

The maximum value is $+2^{64-1} - 1 = +2^{63} - 1 = \textcolor{orange}{+9,223,372,036,854,775,807}$

Which means, any value between the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 can be stored using a **long type** variable.

Therefore, distance between galaxy etc., can be stored using long data type.

Now let's look an example on long data type.

```
class Demo
{
    public static void main(String[] args)
    {
        long a;
        a = 9,223,372,036,854,775,807;
        System.out.println(a);
    }
}
```

Output:

Compilation error.

In Java, there is rule that whenever we want a value to be stored as long, at the end of the value a suffix of 'L' must be given.

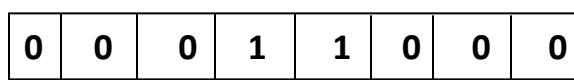
Hope now we have got the answer for our question of having four data types.

Integer values exists in different quantities, to handle different quantities of integer data different data types has been provided.

How Integer data is stored in the memory?

→The below is an example of how positive number is stored.

byte a = 24;



MSB = 0 → positive
1 → negative

2	24
2	12 - 0
2	6 - 0
2	3 - 0
1	- 1

This type of conversion is called base-2 format.

→The below is an example of how negative number is stored.

byte a = -24;

For this, we require **base-2 format + 2's compliment** of a number

We know the base-2 value of 24 is **0001100**.

Now, let's find the 2's compliment of the number.

0 0 0 1 1 0 0 0

Step-1: Find 1's compliment of the number

by converting 0's to 1's and vice versa.

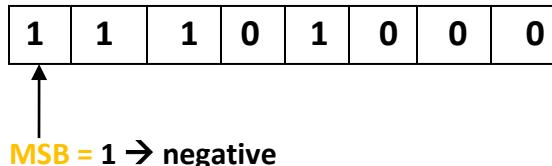
1 1 1 0 0 1 1 1

Step-2: Find 2's compliment by adding 1 to
the number.

+ 1

This is the 2's compliment of the number →

1 1 1 0 1 0 0 0



Real number type data:

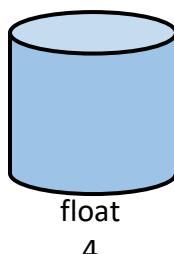
In the real world we are rarely surrounded by Integer type data but often come across decimal numbers. For example, weight: 56.4kg, Temperature: 28.5 degrees, distance: Rajajinagar HO to metro station 0.5km, etc.

In Java to store such real number data types we have float, double.

- 1) **float** The float data type is a single-precision **32-bit** IEEE 754 floating-point. Float data type in Java **stores a decimal value with 6-7 total digits of precision**. So for example 12.12345 can be saved as a float, but 12.123456789 cannot be saved as the float. When representing a float data type in Java we should **append the letter f** to the end of the data type, otherwise, it will save as double.

Assume, if we create an float type variable as **float a;**

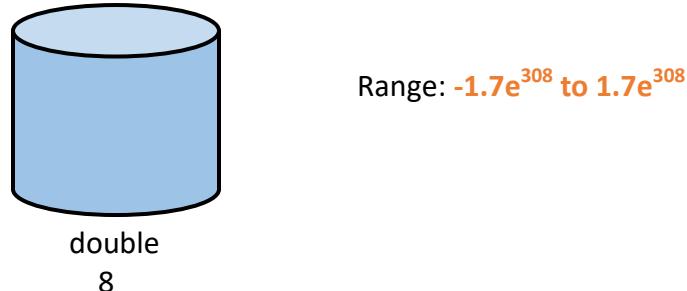
It means in the memory **four bytes** is allocated and it is referred as a.



Range: **-34e³⁸ to 34e³⁸**

- 2) **double** The double data type is a double-precision **64-bit** IEEE 754 floating-point. The double data type is normally the **default choice for decimal values**. The data type should never be used for precise values, such as currency. Double data type stores decimal values with **15-16 digits of precision**. The default value is 0.0d, this means that if you do not append f or d to the end of the decimal, the value will be stored as a double in Java. Assume, if we create an double type variable as **double a;**

It means in the memory **eight bytes** is allocated and it is referred as **a**.



Now let's see few examples to understand the above topic.

Example-1:

```
class Demo
{
    public static void main(String[] args)
    {
        double a=45.5;
        System.out.println(a);
    }
}
```

Output:

45.5

Example-2:

```
class Demo
{
    public static void main(String[] args)
    {
        float a=45.5;
        System.out.println(a);
    }
}
```

Output:

Compilation error.

In Java, there is rule that whenever we want a value to be stored as float, at the end of the value a suffix 'f' must be given, otherwise it is considered to be a double number.

Solution:

Method 1

```
class Demo
{
    public static void main(String[] args)
    {
        double a=(float)45.5;
        System.out.println(a);
    }
}
```

Method 2

```
class Demo
{
    public static void main(String[] args)
    {
        double a=45.5f;
        System.out.println(a);
    }
}
```

Literals: In java values are called as literals.

Now let's see few examples to understand literals.

1) int a=45; ←Decimal literal

S.O.P (a); // Output is 45.

2) int a=045; ←Octal literal

S.O.P (a); // Output is 37.

Note: Here 0 acts as the prefix to the literal and converts it from decimal to octal literal.

0 4 5
 ↓ ↓
 100 101

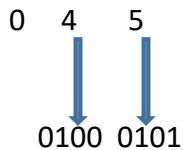
Now we have 100101 as our number, wherever 1 is present we take 2^{index} and add all.

$$2^0 + 2^2 + 2^5 = 1 + 4 + 32 = 37$$

3) int a=0x45;

S.O.P (a); //Output is 69.

Note: Here 0x acts as the prefix to the literal and converts it from decimal to hexadecimal literal.



Now we have 01000101 as our number, wherever 1 is present we take 2^{index} and add all.

We get $2^0+2^2+2^6 = 1+4+64 = 69$

4) int a= 0b100101;

S.O.P (a); // Output is 37.

Note: Here 0b is a prefix for binary literals.

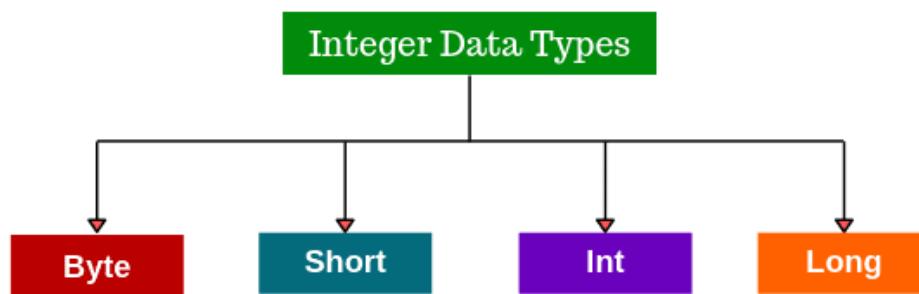
Wherever 1 is present we take 2^{index} and add all. As seen previously we'll get 37.

Integer type data:

Integer types of data represent integer number without any fractional parts or decimal points.

For example: age of a person, number of students in a class, distance between planets, distance between galaxy, etc.,

In Java, to store Integer type data we have four data types.



Now the question arises that to store Integer type data why do we require four data types??

Before that let's have a look on a formula

It is important for one to understand that, given a certain amount of memory how much data can be stored inside it. There is a certain maximum value and minimum value that can be stored and the formula to find it is:

For n bits: The minimum value that can be stored is -2^{n-1}

The maximum value that can be stored is $+2^{n-1} - 1$

1) byte is the smallest integer data type which has the least memory size allocated.

Assume, if we create a byte type variable as **byte a;**

It means in the memory **one byte** is allocated and it is referred as **a**.



To convert byte to bits we have, **1 byte = 8 bits**.

And therefore, the minimum value is $-2^{8-1} = -2^7 = -128$

The maximum value is $+2^{8-1} - 1 = +2^7 - 1 = +127$

Now let's see few examples to understand the above topic.

Example-1:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = 127;
        System.out.println(a);
    }
}
```

Output:

127

Example-2:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = 128;
        System.out.println(a);
    }
}
```

Output:

Compilation error (because this is out of range).

Example-3:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = -128;
        System.out.println(a);
    }
}
```

Output:

-128

Example-4:

```
class Demo
{
    public static void main(String[] args)
    {
        byte a;
        a = -129;
        System.out.println(a);
    }
}
```

Output:

Compilation error (because this is out of range).

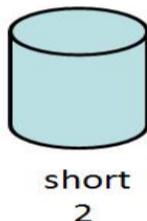
Which means, one value more than +127 or one value less than -128 cannot be stored but any value between the range of -128 to +127 can be stored using a byte type variable.

Therefore, the age of a person, the current month of the year etc., can be stored using byte data type.

2) short is a signed 16-bit byte.

Assume, if we create a short type variable as **short a;**

It means in the memory **two bytes** is allocated and it is referred as **a**.



Converting byte to bits: **2 bytes = 16 bits.**

And therefore, the minimum value is $-2^{16-1} = -2^{15} = -32768$

The maximum value is $+2^{16-1} - 1 = +2^{15} - 1 = +32767$

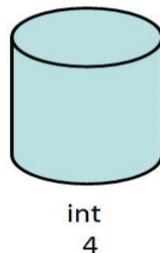
Which means, one value more than +32767 or one value less than -32768 cannot be stored but any value between the range of -32768 to +32767 can be stored using a short type variable.

Therefore, salary of a fresher, height of Mount Everest etc., can be stored using short data type.

3) **int** is a signed 32-bit byte. It is the most commonly used integer type. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.

Assume, if we create an int type variable as **int a;**

It means in the memory **four bytes** is allocated and it is referred as **a**.



Converting byte to bits, **4 bytes = 32 bits**.

And therefore, the minimum value is $-2^{32-1} = -2^{31} = \textcolor{orange}{-2,147,483,648}$

The maximum value is $+2^{32-1} - 1 = +2^{31} - 1 = \textcolor{orange}{+2,147,483,647}$

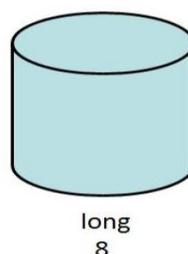
Which means, one value more than +2,147,483,647 or one value less than -2,147,483,648 cannot be stored but any value between the range of -2,147,483,648 to +2,147,483,647 can be stored using an int type variable.

Therefore, distance between planets etc., can be stored using int data type.

4) **long** is a signed 64-bit byte and is useful for those occasions where an int type is not large enough to hold the desired value. The range of long is quite large. This is useful, when big whole numbers are needed.

Assume, if we create an long type variable as **long a;**

It means in the memory **eight bytes** is allocated and it is referred as **a**.



Converting byte to bits, **8 bytes = 64 bits**.

And therefore, the minimum value is $-2^{64-1} = -2^{63} = \textcolor{orange}{-9,223,372,036,854,775,808}$

The maximum value is $+2^{64-1} - 1 = +2^{63} - 1 = \textcolor{orange}{+9,223,372,036,854,775,807}$

Which means, any value between the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 can be stored using a **long type** variable.

Therefore, distance between galaxy etc., can be stored using long data type.

Now let's look an example on long data type.

```
class Demo
{
    public static void main(String[] args)
    {
        long a;
        a = 9,223,372,036,854,775,807;
        System.out.println(a);
    }
}
```

Output:

Compilation error.

In Java, there is rule that whenever we want a value to be stored as long, at the end of the value a suffix of 'L' must be given.

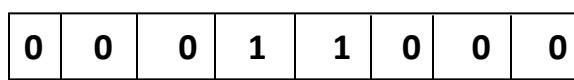
Hope now we have got the answer for our question of having four data types.

Integer values exists in different quantities, to handle different quantities of integer data different data types has been provided.

How Integer data is stored in the memory?

→The below is an example of how positive number is stored.

byte a = 24;



MSB = 0 → positive
1 → negative

2	24
2	12 - 0
2	6 - 0
2	3 - 0
1	- 1

This type of conversion is called base-2 format.

→The below is an example of how negative number is stored.

byte a = -24;

For this, we require **base-2 format + 2's compliment** of a number

We know the base-2 value of 24 is **0001100**.

Now, let's find the 2's compliment of the number.

0 0 0 1 1 0 0 0

Step-1: Find 1's compliment of the number

by converting 0's to 1's and vice versa.

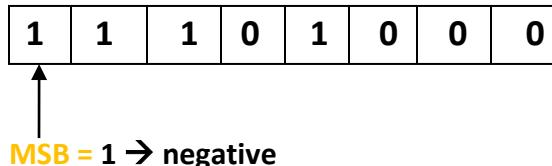
1 1 1 0 0 1 1 1

Step-2: Find 2's compliment by adding 1 to
the number.

+ 1

This is the 2's compliment of the number →

1 1 1 0 1 0 0 0



Real number type data:

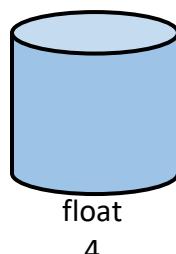
In the real world we are rarely surrounded by Integer type data but often come across decimal numbers. For example, weight: 56.4kg, Temperature: 28.5 degrees, distance: Rajajinagar HO to metro station 0.5km, etc.

In Java to store such real number data types we have float, double.

- 1) **float** The float data type is a single-precision **32-bit** IEEE 754 floating-point. Float data type in Java **stores a decimal value with 6-7 total digits of precision**. So for example 12.12345 can be saved as a float, but 12.123456789 cannot be saved as the float. When representing a float data type in Java we should **append the letter f** to the end of the data type, otherwise, it will save as double.

Assume, if we create an float type variable as **float a;**

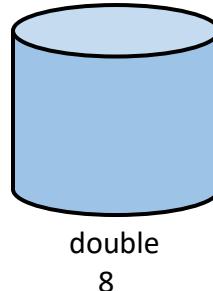
It means in the memory **four bytes** is allocated and it is referred as a.



Range: **-34e³⁸ to 34e³⁸**

- 2) **double** The double data type is a double-precision **64-bit** IEEE 754 floating-point. The double data type is normally the **default choice for decimal values**. The data type should never be used for precise values, such as currency. Double data type stores decimal values with **15-16 digits of precision**. The default value is 0.0d, this means that if you do not append f or d to the end of the decimal, the value will be stored as a double in Java. Assume, if we create an double type variable as **double a;**

It means in the memory **eight bytes** is allocated and it is referred as **a**.



Range: **$-1.7e^{308}$ to $1.7e^{308}$**

Now let's see few examples to understand the above topic.

Example-1:

```
class Demo
{
    public static void main(String[] args)
    {
        double a=45.5;
        System.out.println(a);
    }
}
```

Output:

45.5

Example-2:

```
class Demo
{
    public static void main(String[] args)
    {
        float a=45.5;
        System.out.println(a);
    }
}
```

Output:

Compilation error.

In Java, there is rule that whenever we want a value to be stored as float, at the end of the value a suffix 'f' must be given, otherwise it is considered to be a double number.

Solution:

Method 1

```
class Demo
{
    public static void main(String[] args)
    {
        double a=(float)45.5;
        System.out.println(a);
    }
}
```

Method 2

```
class Demo
{
    public static void main(String[] args)
    {
        double a=45.5f;
        System.out.println(a);
    }
}
```

Literals: In java values are called as literals.

Now let's see few examples to understand literals.

1) int a=45; ←Decimal literal

S.O.P (a); // Output is 45.

2) int a=045; ←Octal literal

S.O.P (a); // Output is 37.

Note: Here 0 acts as the prefix to the literal and converts it from decimal to octal literal.

0 4 5
 ↓ ↓
 100 101

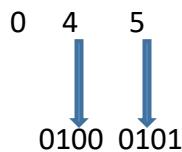
Now we have 100101 as our number, wherever 1 is present we take 2^{index} and add all.

$$2^0 + 2^2 + 2^5 = 1 + 4 + 32 = 37$$

3) int a=0x45;

S.O.P (a); //Output is 69.

Note: Here 0x acts as the prefix to the literal and converts it from decimal to hexadecimal literal.



Now we have 01000101 as our number, wherever 1 is present we take 2^{index} and add all.

We get $2^0+2^2+2^6 = 1+4+64 = 69$

4) int a= 0b100101;

S.O.P (a); // Output is 37.

Note: Here 0b is a prefix for binary literals.

Wherever 1 is present we take 2^{index} and add all. As seen previously we'll get 37.

Let us move ahead and learn about the next data type.

Character type data:



I x

Had a look on the above characters?



Well they all fall under character type data but, computer doesn't understand **A@#*14**. It only understands 0's and 1's and hence conversion should happen. Before understanding that let us first know the different characters.

Let us consider four symbols **A B C D** but, none of these symbols can your computer understand because all it understands is binary numbers 0's and 1's. So, let us attach a binary code to each of these symbols like this



SYMBOLS	CODE
A	00
B	01
C	10
D	11

Let us consider if there were 8 symbols and look at its binary code.

SYMBOLS	CODE
A	000
B	001
C	010
D	011
E	100
F	101
G	110
H	111

As you can see, as the number of **symbols** increase their **code size** also increases.

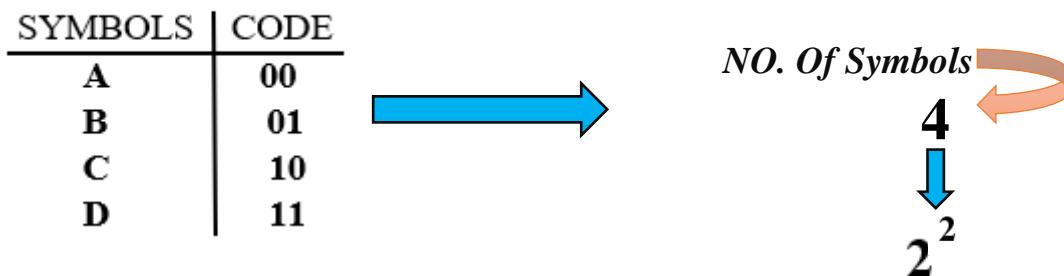
Can you guess the code size for 16 characters?



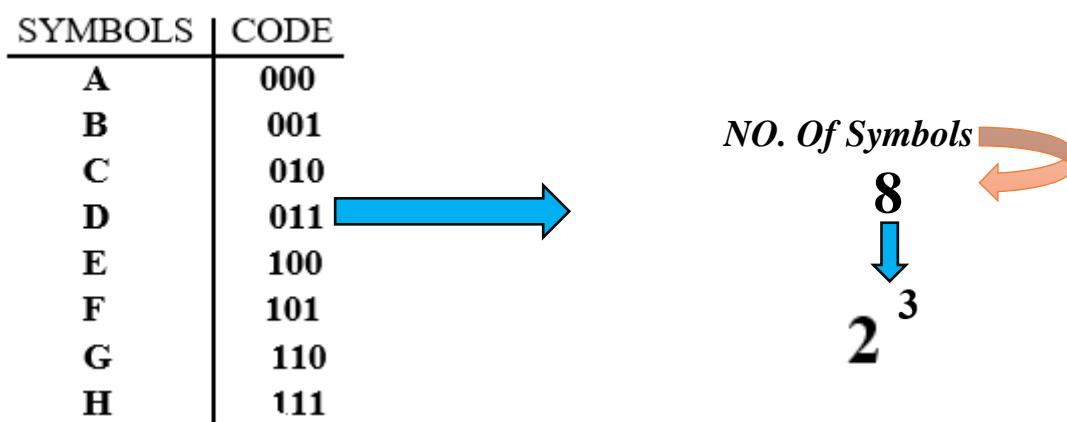
SYMBOLS	CODE	SYMBOLS	CODE
A	0000	I	1000
B	0001	J	1001
C	0010	K	1010
D	0011	L	1011
E	0100	M	1100
F	0101	N	1101
G	0110	O	1110
H	0111	P	1111

If you are focusing then you might have noticed there is a mathematical relation between **no. of symbols** and **code length**.

Let us consider four symbols  **A B C D**



Similarly let us consider eight symbols  **A B C D E F G H**



If you are focusing on the **power of 2**, you can see it is only the **code length**.

Let us consider one more case of 16 symbols to understand this.

16 Symbols



A B C D E F G H I J K L M N O P

SYMBOLS	CODE	SYMBOLS	CODE	NO. Of Symbols
A	0000	I	1000	
B	0001	J	1001	
C	0010	K	1010	
D	0011	L	1011	
E	0100	M	1100	
F	0101	N	1101	
G	0110	O	1110	
H	0111	P	1111	

16
 2^4

Code Length - 4

Now you understood the relation between code length and number of symbols. Like this Americans found **128 symbols** and gave the name as **ASCII**. But Java does not follow ASCII as it only consist of English symbols. Have a look at the ASCII table below.

Hex	Value																
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	'	70	p		
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q		
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r		
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s		
04	EOT	14	DC4	24	\$	34	4	44	D	54	T	64	d	74	t		
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u		
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v		
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w		
08	BS	18	CAN	28	(38	8	48	H	58	X	68	h	78	x		
09	HT	19	EM	29)	39	9	49	I	59	Y	69	i	79	y		
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z		
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[6B	k	7B	{		
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C			
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D]	6D	m	7D	}		
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~		
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL		

ASCII stands for **American standard code for information interchange**.

It is a **7-bit** binary representation for 128 symbols.

Java does not follow ASCII as it is an English biased language and does not support symbols of other languages.

Hence java follows **UNICODE** which provides binary representation for **65,536** symbols of commonly spoken languages across the world.

It is a **16-bit** code and hence a char variable in java takes **2 bytes** of memory.

Let us write a simple code to print character type data

Class Demo

```
{  
    Public static void main(String[] args)  
    {  
        Char ch = 'a';  
        System.out.println(ch);  
    }  
}
```

Output: a



Let us now look at the last data type that is **boolean** data type.

Boolean data type:

To store yes/no type data or true/false type data, java provides boolean data type. Size of this data type is decided by JVM and we have already learned JVM is platform dependent, hence the size of this data type will differ depending on the type of operating system.

The remaining types of data that is audio, video and still pictures are handled using built in libraries.

Type casting in java:

Now you wonder what is type casting?



Well let me tell you, **type casting is a process of converting one type of data to another**.

In Java, there are two types of casting:

Implicit casting (automatically) - converting a smaller type to a larger type size
byte -> short -> char -> int -> long -> float -> double.

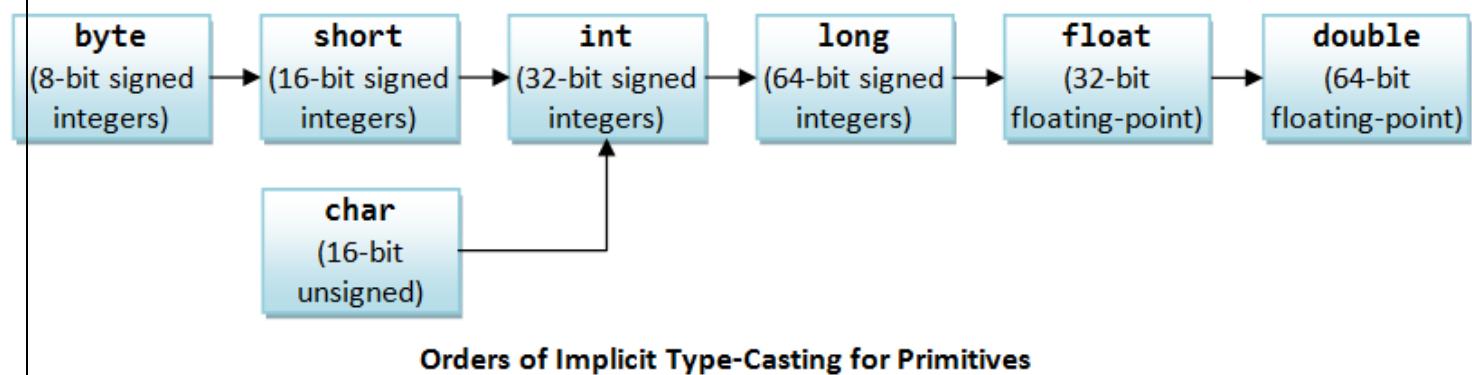
Explicit casting (manually) - converting a larger type to a smaller size type.

Implicit type casting:

When a smaller data type is converted to a larger data type, the conversion is automatically performed by **the java compiler** and is referred to as implicit type casting.

Advantage: No loss of precision.

Consider the **Implicit type casting chart** given below to understand this:

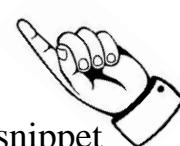
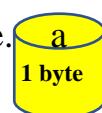


Let us consider a code snippet to understand this:

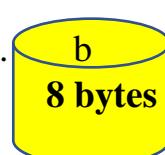
```
byte a = 45;  
double b;  
b = a;
```

let us understand implicit type casting using the above code snippet

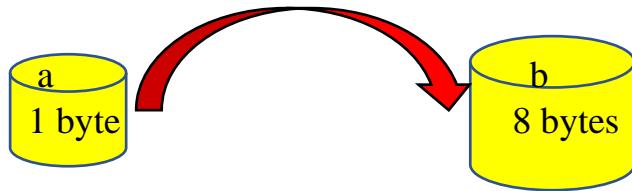
a is a variable of type byte whose size is 1 byte.



b is a variable of type double whose size is 8 bytes.



`b = a;` we are now trying to store the data present in **a** into **b**.
a is of type byte and can store 1 byte. b is of type double and can store 8 bytes.
we are trying to store data of smaller size into larger size.



This conversion is implicitly done without user interaction and hence it is referred to as implicit type casting.

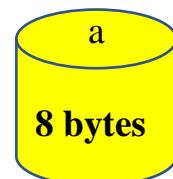
Explicit type casting:

When a larger data type is converted to a smaller data type, the conversion **not automatically performed by the java complier** and must be done by **programmer explicitly** and hence it is referred to as implicit type casting.

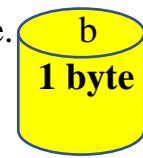
Let us consider a simple code snippet to understand this, the way we understood implicit type casting.

```
double a = 45.5;  
byte b;  
b = a; → error
```

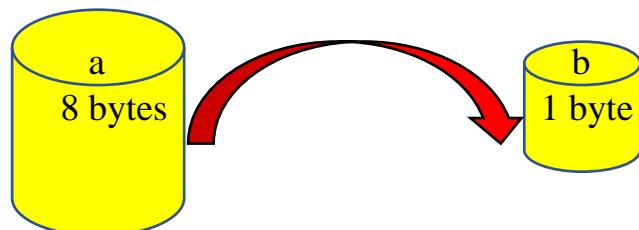
a is a variable of type double whose size is 8 bytes.



b is a variable of type byte whose size is 1 byte.

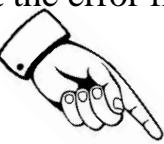


`b = a;` will give you **error** as you are trying to store a larger type of data into smaller type.



The above conversion will result in error as **loss of precision** occurs.

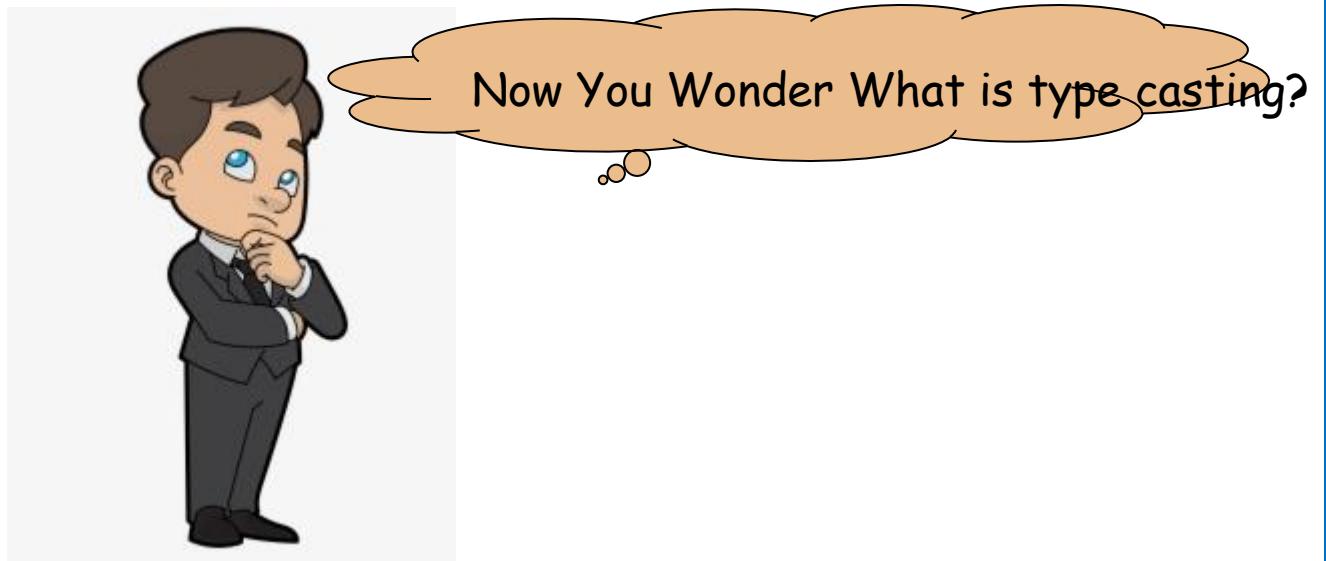
To get the error free output, we have to explicitly convert the data as shown below.



```
double a = 45.5;  
byte b;  
b = byte(a);
```

b is of type byte and it will only store 45 and 0.5 is lost during the conversion which is the disadvantage of explicit type casting.

Type casting in java:



Type casting is a process of converting one type of data to another

In Java, there are two types of casting:

Implicit casting (automatically) - converting a smaller type to a larger type size
byte → short → char → int → long → float → double

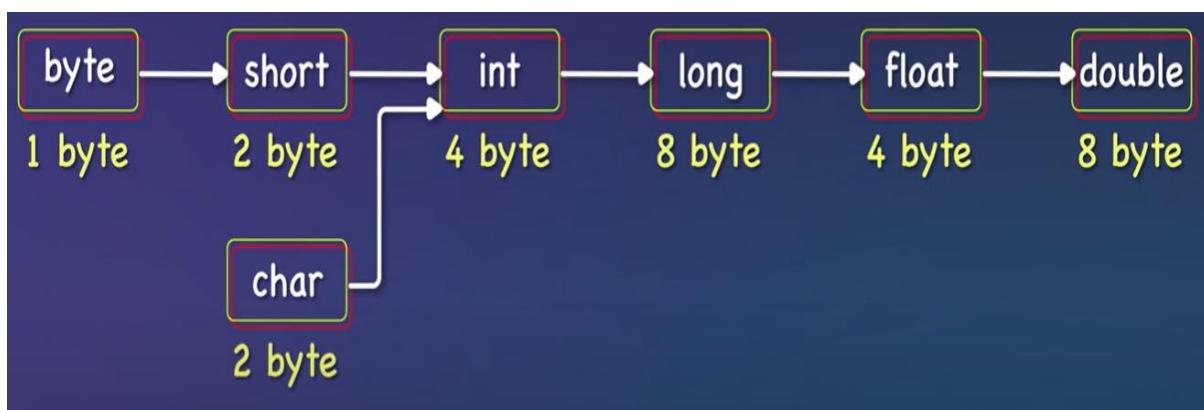
Explicit casting (manually) - converting a larger type to a smaller size type

Implicit type Casting:

When a smaller data type is converted to a larger data type, the conversion is automatically performed by the java compiler and is referred to as implicit type casting.

Advantage: No loss of precision

Consider the **Implicit type casting** chart given below to understand this:



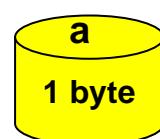
Orders of Implicit Type-Casting for Primitives

Let us consider a code snippet to understand this:

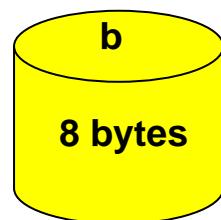
```
byte a = 45;
double b;
b = a;
```

Let us understand implicit type casting using the above code snippet

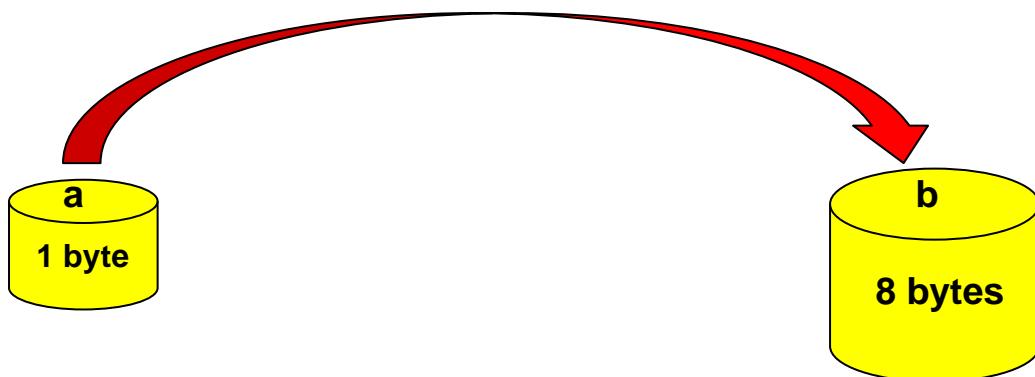
a is a variable of type byte whose size is 1 byte



b is a variable of type byte whose size is 8 byte



b = a; we are now trying to store the data present in a into b;
 a is of type byte and can store 1 byte. b is of type double and can store 8 bytes. We are trying to store data of smaller size into larger size.



This conversion is implicitly done without user interaction and hence it is referred to as implicit type casting

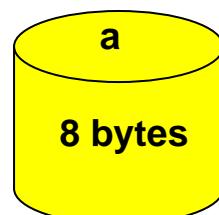
Explicit type Casting:

When a larger data type is converted to a smaller data type, the **conversion is not automatically performed by the java compiler** and must be done by **programmer explicitly** and hence it is referred to as explicit type casting

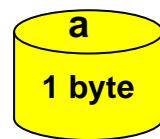
Let us consider a simple code snippet to understand this, the way we understood explicit type casting

```
double a = 45.5;
byte b;
b = a;
```

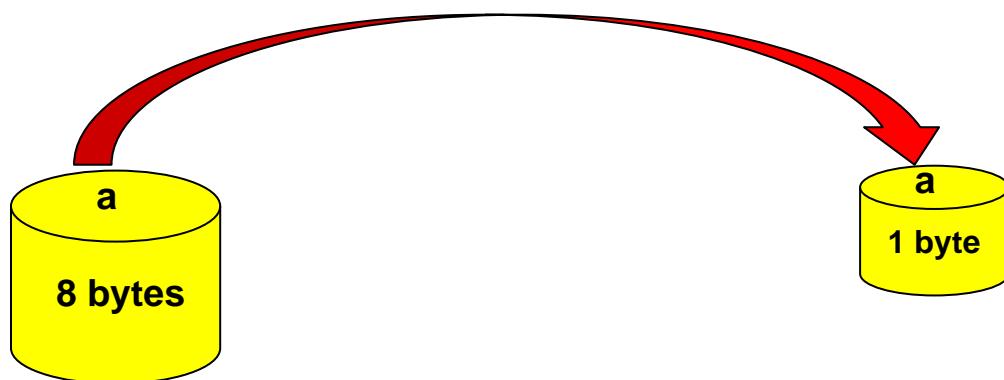
a is a variable of type double whose size is 8 bytes.



b is a variable of type byte whose size is 1 byte.



`b=a;` will give you **error** as you are trying to store a larger type of data into a smaller type.



The above conversion will result in error as loss of precision occurs. To get the error free output, we have to explicitly convert the data as shown below

```
double a = 45.5;  
byte b;  
b = (byte)a;
```

b is of type byte and it will only store 45 and 0.5 is lost during the conversion which is the disadvantage of explicit type casting.

Data types

1. In what format is Integer data stored in memory unit?

base-2 format

2. In what format is real number data stored in memory unit?

IEEE format

3. Which data types in Java are used to handle Integer data?

byte, short, int and long

4. In what format is still picture data stored in memory unit?

.jpeg format and .gif format

5. What are the different types of data in real world?

Character type data, integer type data, real number type data, yes/no type data, still picture type data, audio and video type data

6. In what format is video data stored in memory unit

.mp4 format and .avi format

7. What is the real number data by default treated in Java?

Double

8. What is the range of data that can be stored in long data type?

-9223372036854775808L to +9223372036854775807L

9. How many bytes are allocated for short data type in Java?

2 Bytes

10. Does Java follow ASCII or UNICODE ? Why?

UNICODE. Because UNICODE would be having binary representations for all the symbols of all the languages which are currently used across the globe)

11. How many bytes are allocated for long data type in Java?

8 bytes

12. How many bytes are allocated for float data type in Java?

4 bytes

13. What is the range of data that can be stored in byte data type?

-128 to +127

14. In what format is audio data stored in memory unit?

.mp3 format

15. What is the range of data that can be stored in short data type?

-32768 to +32767

16. How many bytes are allocated for char data type in Java? Why?

2 bytes, because java follows UNICODE format

17. Is zero a positive number or negative number in programming? Why?

Zero is a positive number, because the most significant bit in the positive number is zero

18. Who initiates the process of executing a program?

OS

19. Why does Java provide primitive data types in spite of the fact that it makes it only 99% OOP?

Because creation of variables using primitive data types is faster than creating an object using wrapper classes)

20. What is the range of double?

-1.7e-308 to +1.7e+308

21. Why does Java provide four data types to manage Integer type data?

Because in real world integer data exists in varying magnitudes.

22. Why does Java provide two data types to manage real number type data?

For less precision and higher precision i.e. double type provides more accuracy than float type)

23. Why should data be stored in form of 0s and 1s in the memory?

Because every memory device can store only 0's and 1's

24. How is audio type data handled in Java?

Using in built classes

25. How is video type data handled in Java?

Using in built classes

26. How is still picture type data handled in Java?

Using in built classes

27. How many bytes are allocated for boolean data type in Java?

It is OS dependent or JVM dependent.

28. What is the range of float?

-3.4e-038 to +3.4e+038

29. What are the different data types in Java?

byte, short, int, long, float, double, boolean, char

30. How is data stored in the memory unit?

In the binary form

31. Which data types in Java are used to handle real number data?

float and double

32. In what format is character data stored in memory unit?

UTF-16 format or UTF-32 format

33. In what format is yes/no data stored in memory unit?

It is JVM dependent

34. Why did UNICODE come into existence?

Because ASCII does not have binary representation for all the symbols of all the languages which are currently used across the globe.

35. What is the range of data that can be stored in int data type?

-2147483648 to 2147483647

36. How can you convert double data type to float data type in Java?

- i) By explicit typecasting

Eg. float a = (float) 24.17;

- ii) By adding suffix 'f'

Eg. float a = 24.17f;

37. How many bytes are allocated for byte data type in Java?

1 byte

38. How many bytes are allocated for double data type in Java?

8 bytes

39. How many bytes are allocated for int data type in Java?

4 bytes

40. Which special characters may be used as the first character of an identifier?

_ and \$

41. Which characters may be used as the second character of an identifier, but not as the first character of an identifier?

Digits cannot be used as the first character.

Eg. t6emp=25; and temp6=35; are valid whereas,

6temp=25; is invalid.

42. How many bit format is ASCII exactly?

7 bit format

43. Why is ASCII format forcefully stored as 8-bit format?

Because minimum memory that can be allocated is 1 Byte i.e. 8bits

44. What is UTF?

UTF stands for Universal Translational Format

45. What is UTF-8? When is it used normally?

UTF-8 is used whenever binary representations for only English and its associated symbols are required in the project.

46. What is UTF-16? When is it used normally?

UTF-16 is used whenever the binary representations for all the symbols of all the languages which are currently used across the world are required in the project.

47. What is UTF-32? When is it used normally?

UTF-32 is used whenever along with current languages symbols, even ancient languages symbols' binary representations are required in the project.

48. What is meant by rounding towards zero in integer division?

Truncation, i.e. fractional portion is truncated and only the integer portion is retained.

Eg. (refer class notes)

49. What is meant by truncation?

Truncation is the process of eliminating the fractional part and retaining only the integer portion. It is also called as rounding towards zero.

50. Are true and false keywords?

true and false are reserved words.

51. What is numeric promotion?

When data of a smaller magnitude is placed within a memory location of a larger magnitude, it is called as numeric promotion. Implicit typecasting is also called as numeric promotion.

52. What is the difference between the prefix and postfix forms of the ++ operator?

Pre increment: increment first and then assign, post increment: first assign and then increment (Refer class notes for more examples)

53. What are the rules associated with the usage of underscore in a literal?

With respect to literal creation only one special character is permitted which is ‘_’(underscore). It can only be used in between the literal any number of times and nowhere else i.e.

- 1) Underscore cannot be used before or after literal.

Eg.

```
int temp=9_9;//valid  
int temp=9____9;//valid  
int temp=_99;//invalid  
int temp=99_;//invalid
```

- 2) It cannot be used before prefix or in between the prefix or soon after the prefix

Eg.

```
int temp=_0x45; //invalid  
int temp=0_x45; //invalid  
int temp=0x_45; //invalid  
int temp=0x4_5; //valid
```

- 3) It cannot be used before or after suffix

Eg.

```
float temp=45.5f_;//invalid  
float temp=45.5_f; //invalid  
float temp=4_5.5f; //valid
```

4) It cannot be used before or after decimal points

Eg.

```
float temp=45_.5f;//invalid  
float temp=45._5f;//invalid  
float temp=4_5.5f;//valid
```

54. What is meant by “Java is a strongly typed language”?

Every variable in java must have an associated data type and also a value which is compatible with the data type.

55. Give the implicit upcasting chart or numeric promotion chart?

(Refer class notes)

56. Which Java operator is right to left associative?

Assignment operator (=)

57. Can a double value be cast to a byte?

Yes.

58. Express double a = 123.45 in scientific notation?

double a = 1.2345E+2

59. What is the difference between char literal and string literal?

Character literal is data which is enclosed within single quotes whereas String literal is data which is enclosed within double quotes.

60. Can we use underscore in a literal?

Yes.

61. What is the difference between declaring a variable and defining a variable?

```
int a; // declaring,  
int a=100; // defining
```

62. Can we create binary literals in Java?

Yes. Using a prefix 0b

63. How do we make a project coded in Java a pure OOP project?

Using wrapper classes.

64. What is the role of wrapper classes in Java?

Using wrapper classes, creation of primitive variables can be avoided and hence a pure object oriented project can be developed.

65. What happens if a larger magnitude data is assigned to a value of a data type which cannot handle it?

Overflow or loss of precision occurs.

66. Should type casting be performed explicitly?

Depends upon whether implicit or explicit typecasting is performed.

67. Does type casting reduce the precision of the data?

Depends upon whether implicit or explicit typecasting is performed. Implicit typecasting does not reduce precision whereas explicit typecasting reduces precision.

68. Should numeric promotion be performed explicitly?

No. Numeric promotion also known as implicit casting or java automatic conversions where conversion of a smaller numeric type to a larger numeric type takes place.

69. Does numeric promotion reduce the precision of the data?

No

70. What is the role of formats in data types?

It is used to convert real world data in its original form into 0s and 1s so that it can be stored in the memory unit.

71. What is a variable?

It is a reserved memory location into which a value can be stored

72. What are the types of variables available in Java?

Local variables, instance variables, reference variables and static variables.

73. How is a negative number stored in Byte data type?

Using 2's compliment base-2 format.

74. How is a negative number stored in short data type?

Using 2's compliment base-2 format.

75. How is a negative number stored in int data type?

Using 2's compliment base-2 format.

76. How is a negative number stored in long data type?

Using 2's compliment base-2 format.

77. What is the range of char?

0 to 65535

78. What is a literal?

Literal is a fixed value which is assigned to a variable.

79. What does a prefix 0 indicate in a literal?

It indicates that the number is an Octal.

80. What does a prefix 0x indicate in a literal?

It indicates that the number is Hexadecimal.

81. How do we display a \ in Java?

s.o.p('\\');

82. How do we display “ in Java?

s.o.p('\"');

83. How do we display ‘ in Java?

s.o.p('\''');

84. Do we have unsigned integer format in Java?

Not up to java 1.7. However, from java 1.8 it is supported.

85. Can we use the float data type to hold the precise values such as currency?

No. Rather, inbuilt class Currency is used.

86. Can we use the double data type to hold the precise values such as currency?

No. Rather, inbuilt class Currency is used.

87. What is the default value of long?

0L

88. What is the default value of double?

0.0

89. What is the default value of char?

'\u0000' i.e. blank character.

90. How is a binary literal created in Java?

Using the prefix 0b

91. Identify valid and invalid literals?

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
float pi1 = 3_.1415F;
float pi2 = 3._1415F;
long socialSecurityNumber1 = 999_99_9999_L;
int x1 = 5_2;
int x2 = 52_;
int x3 = 5_____2;
int x4 = 0_x52;
int x5 = 0x_52;
int x6 = 0x5_2;
int x7 = 0x52_;
```


Operators

Increment

Increment and Decrement in Java programming let you easily **add 1** or **subtract 1** from variable.

To achieve this, we have two different types of operators.

INCREMENT

In Java, the increment unary operator increases the value of the variable by one.

" **++** " is the operator used to increment.

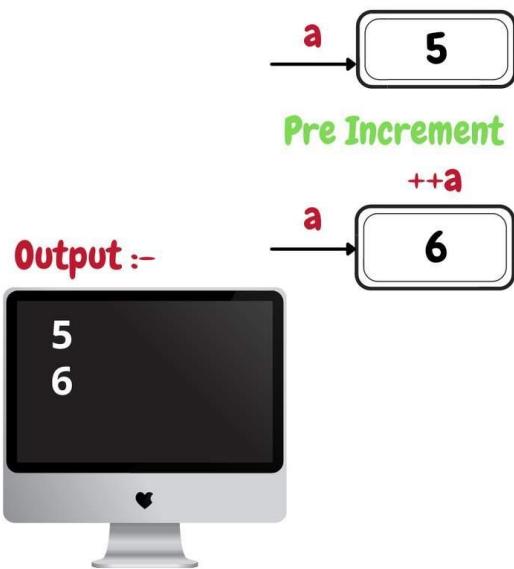
There are two types of Increment

- Pre-Increment.
- Post-Increment.

Pre-Increment-

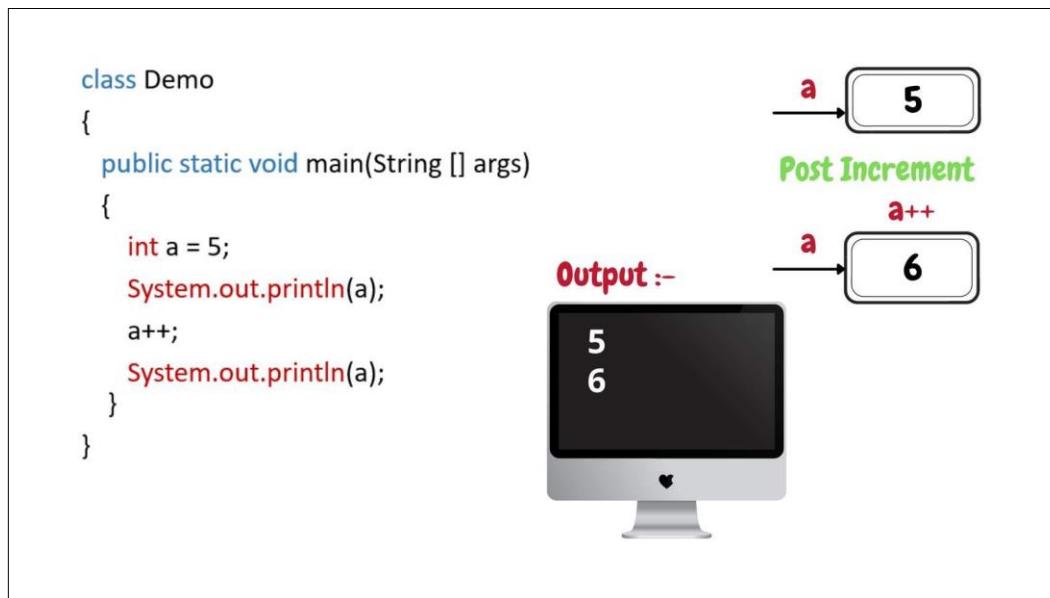
- "**++**" is written before Variable name.
- Value will be Incremented First and then incremented value is used in expression.

```
class Demo
{
    public static void main(String [] args)
    {
        int a = 5;
        System.out.println(a);
        ++a;
        System.out.println(a);
    }
}
```



Post-Increment-

- “++” is written after Variable name.
- Value is used in expression first and then gets incremented.



Decrement

In Java, the decrement unary operator decreases the value of the variable by one.

" -- " is the operator used to decrement.

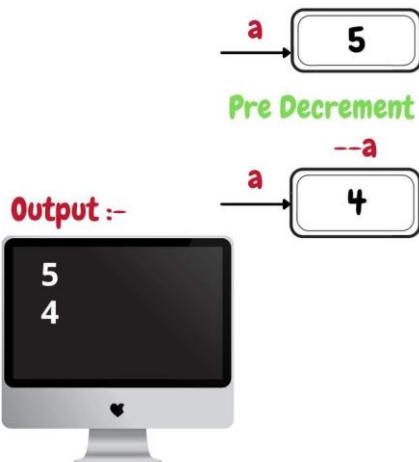
There are two types of Increment

- Post-Decrement.
- Pre-Decrement.

Pre-Decrement-

- “--” is written before Variable name.
- Value is decremented First and then decremented value is used in expression

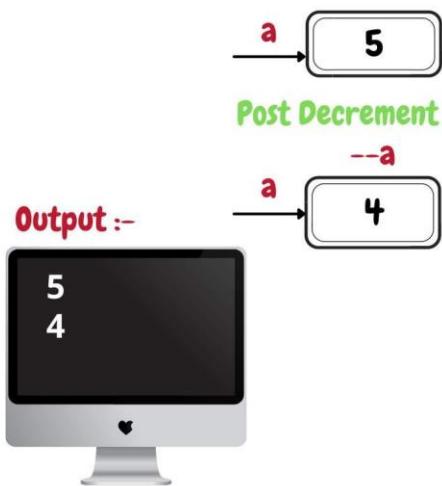
```
class Demo
{
    public static void main(String [] args)
    {
        int a = 5;
        System.out.println(a);
        --a;
        System.out.println(a);
    }
}
```



Post-Decrement-

- “**--**” is written after Variable name.
- Value is used in expression first and then gets decremented.

```
class Demo
{
    public static void main(String [] args)
    {
        int a = 5;
        System.out.println(a);
        a--;
        System.out.println(a);
    }
}
```



Problems on Increment and Decrement.

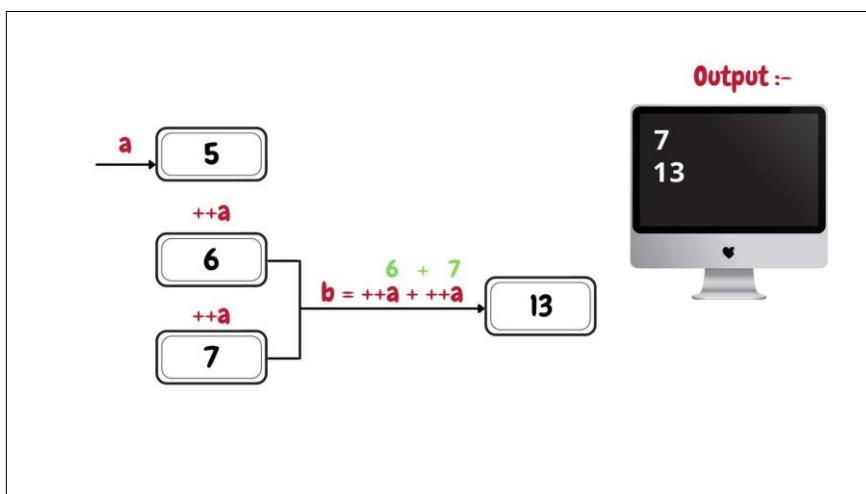
Question 01

Consider $a = 5$, print a and b

$b = ++a + ++a;$

Code:

```
class Demo
{
    public static void main (String [] args)
    {
        int a = 5 ;
        int b;
        b = ++a + ++a;
        System.out.println(a);
        System.out.println(b);
    }
}
```



Question 02

Consider $a = 5$, print a and b

$b = a++ + a++;$

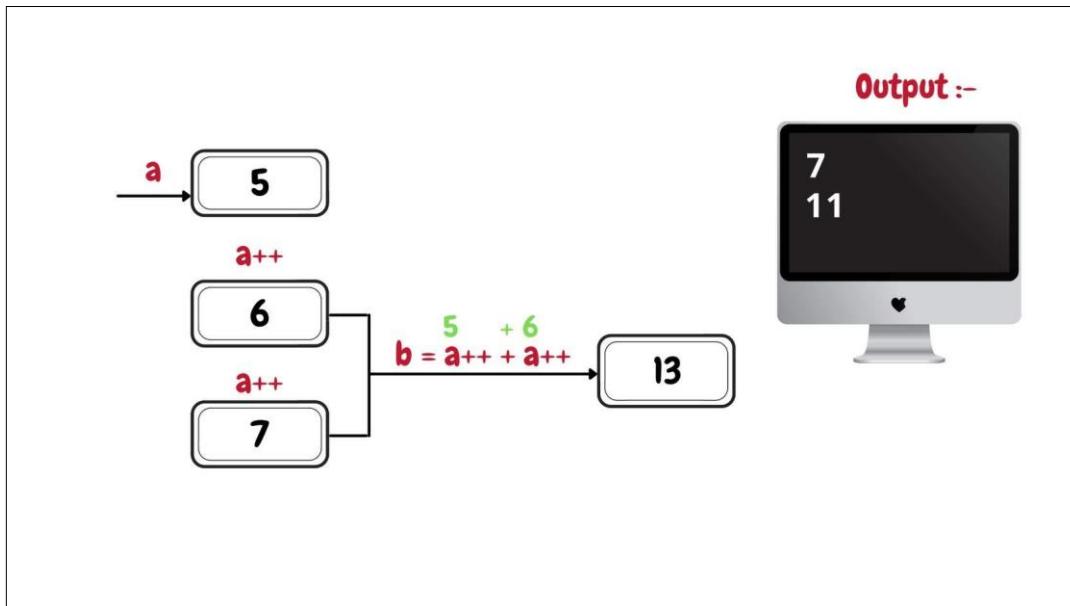
Code:

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

```

{
    int a = 5 ;
    int b;
    b = a++ + a++;
    System.out.println(a);
    System.out.println(b);
}

```



Question 03

Consider $a = 5$, print a and b

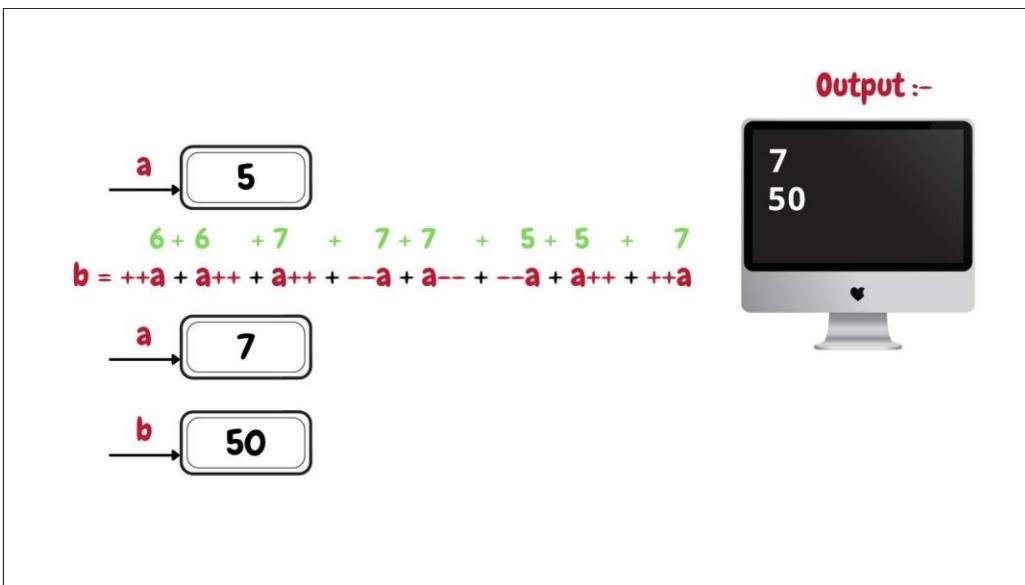
$b = ++a + a++ + a++ + --a + a-- + --a + a++ + ++a;$

Code:

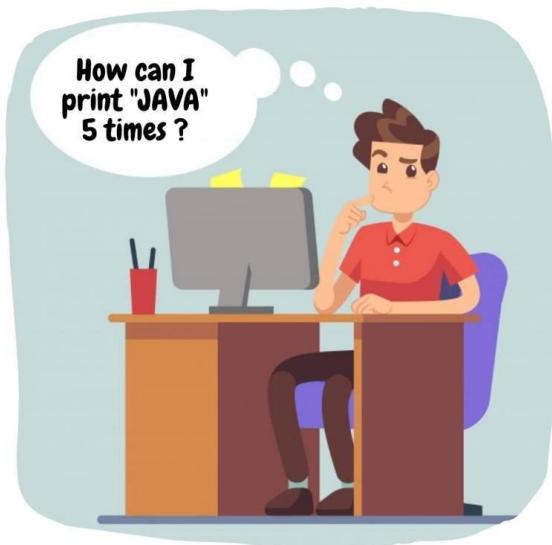
```

class Demo
{
    public static void main (String [] args)
    {
        int a = 5 ;
        int b;
        b = ++a + a++ + a++ + --a + a-- + --a + a++ + ++a;
        System.out.println(a);
        System.out.println(b);
    }
}

```



Loops



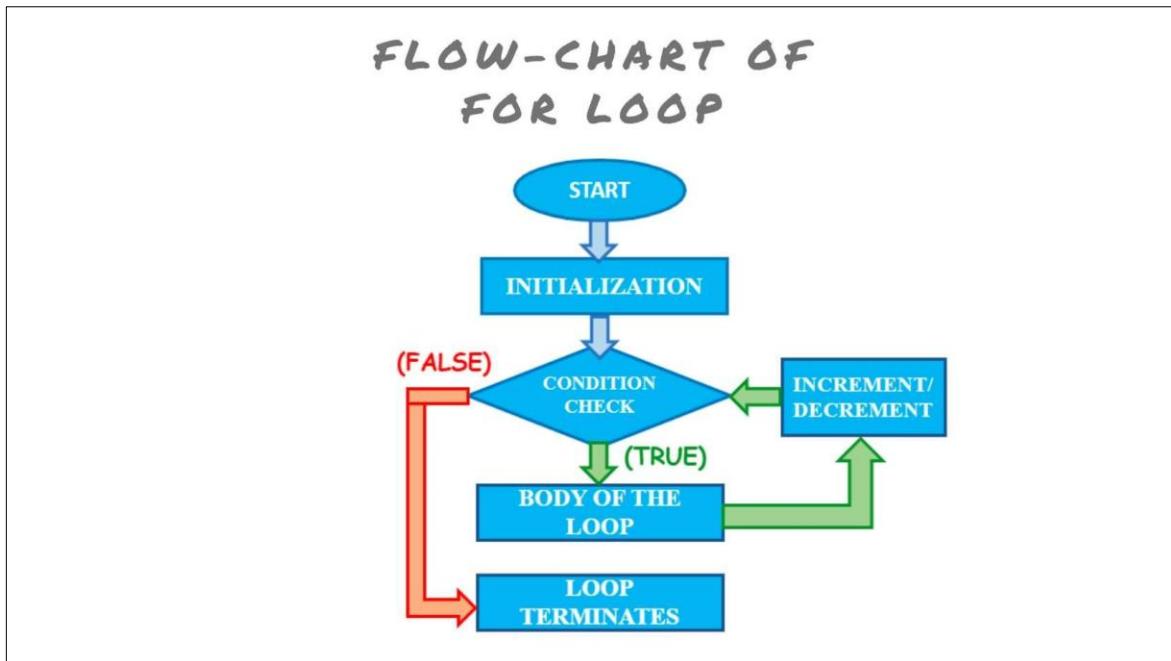
```
class Demo
{
    public static void main(String [] args)
    {
        System.out.println("JAVA");
        System.out.println("JAVA");
        System.out.println("JAVA");
        System.out.println("JAVA");
        System.out.println("JAVA");
    }
}
```

IS THIS EFFICIENT
WAY TO WRITE A
PROGRAM ?
?????

Imagine you have to print "JAVA" 100 times?

Are you going to type the statement 100 times?

Here comes the loop to rescue the programmer and make the job easy. Most fundamental and basic loop is known as FOR loop.



Syntax:

for (initialization; condition; increment/ decrement)

Code:

```
class Demo
{
    public static void main (String [] args)
    {
        int i;
        for ( i=1; i<=5; i++)
        {
            System.out.println("JAVA");
        }
    }
}
```

Output

Operators

Increment

Increment and Decrement in Java programming let you easily **add 1** or **subtract 1** from variable.

To achieve this, we have two different types of operators.

INCREMENT

In Java, the increment unary operator increases the value of the variable by one.

" **++** " is the operator used to increment.

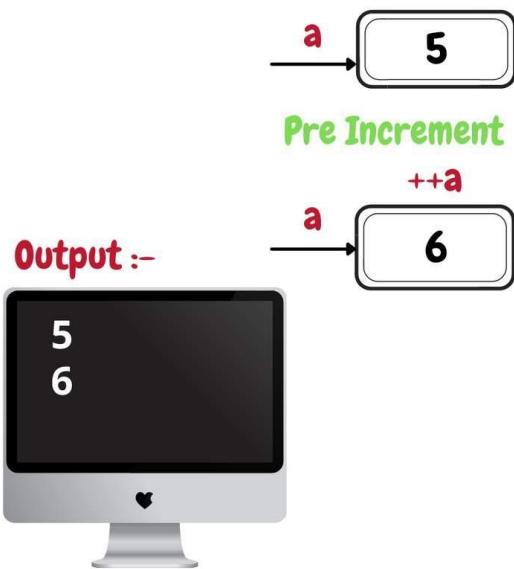
There are two types of Increment

- Pre-Increment.
- Post-Increment.

Pre-Increment-

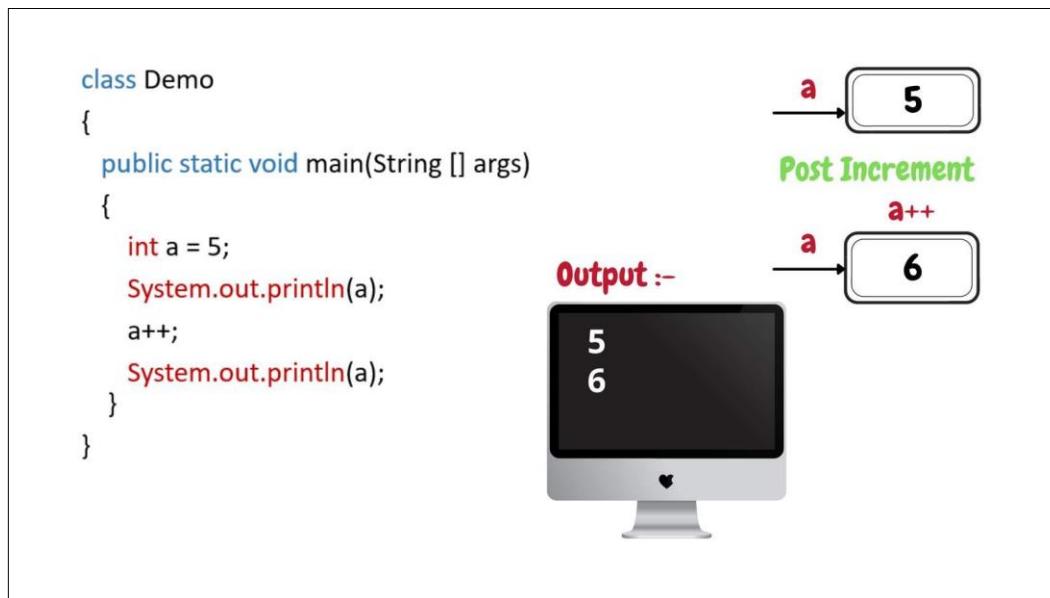
- "**++**" is written before Variable name.
- Value will be Incremented First and then incremented value is used in expression.

```
class Demo
{
    public static void main(String [] args)
    {
        int a = 5;
        System.out.println(a);
        ++a;
        System.out.println(a);
    }
}
```



Post-Increment-

- “++” is written after Variable name.
- Value is used in expression first and then gets incremented.



Decrement

In Java, the decrement unary operator decreases the value of the variable by one.

" -- " is the operator used to decrement.

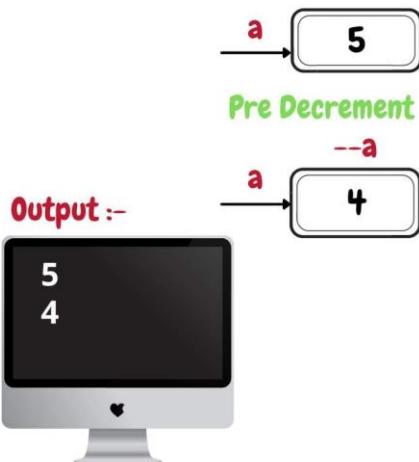
There are two types of Increment

- Post-Decrement.
- Pre-Decrement.

Pre-Decrement-

- “--” is written before Variable name.
- Value is decremented First and then decremented value is used in expression

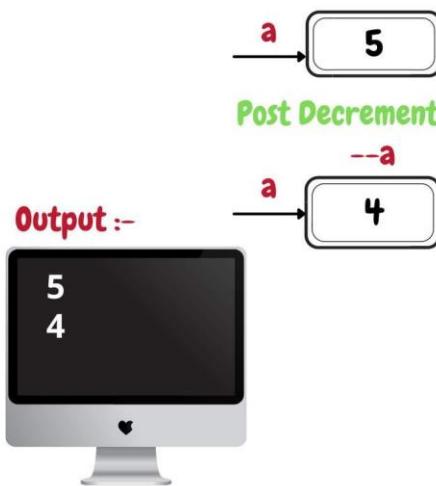
```
class Demo
{
    public static void main(String [] args)
    {
        int a = 5;
        System.out.println(a);
        --a;
        System.out.println(a);
    }
}
```



Post-Decrement-

- “**--**” is written after Variable name.
- Value is used in expression first and then gets decremented.

```
class Demo
{
    public static void main(String [] args)
    {
        int a = 5;
        System.out.println(a);
        a--;
        System.out.println(a);
    }
}
```



Problems on Increment and Decrement.

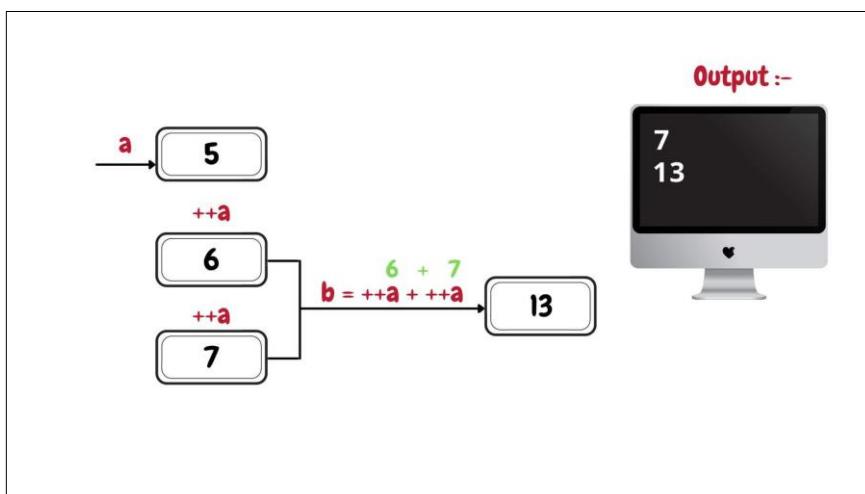
Question 01

Consider $a = 5$, print a and b

$b = ++a + ++a;$

Code:

```
class Demo
{
    public static void main (String [] args)
    {
        int a = 5 ;
        int b;
        b = ++a + ++a;
        System.out.println(a);
        System.out.println(b);
    }
}
```



Question 02

Consider $a = 5$, print a and b

$b = a++ + a++;$

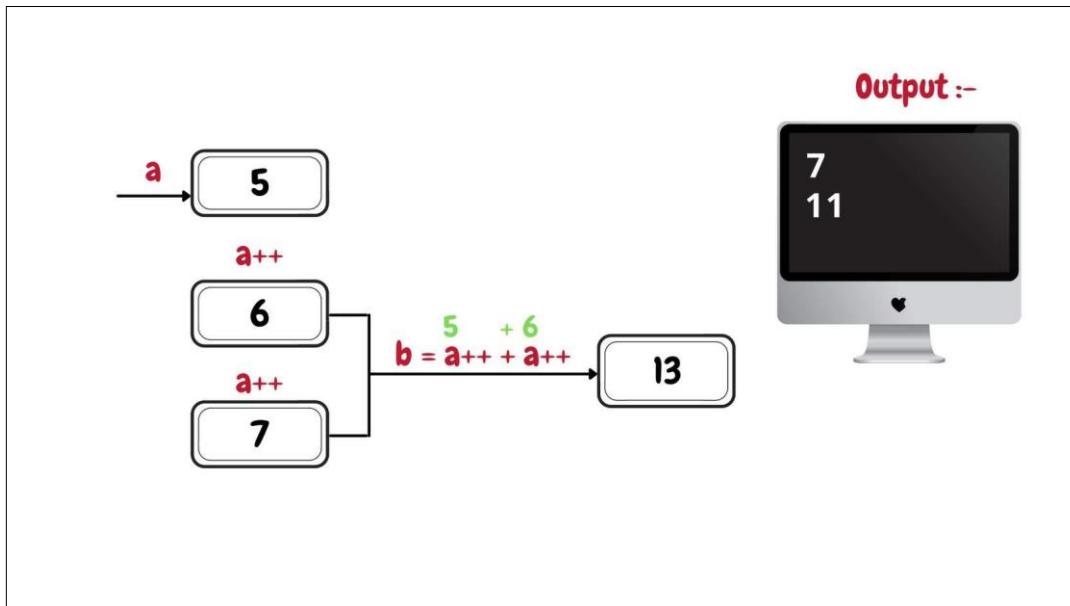
Code:

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

```

{
    int a = 5 ;
    int b;
    b = a++ + a++;
    System.out.println(a);
    System.out.println(b);
}

```



Question 03

Consider $a = 5$, print a and b

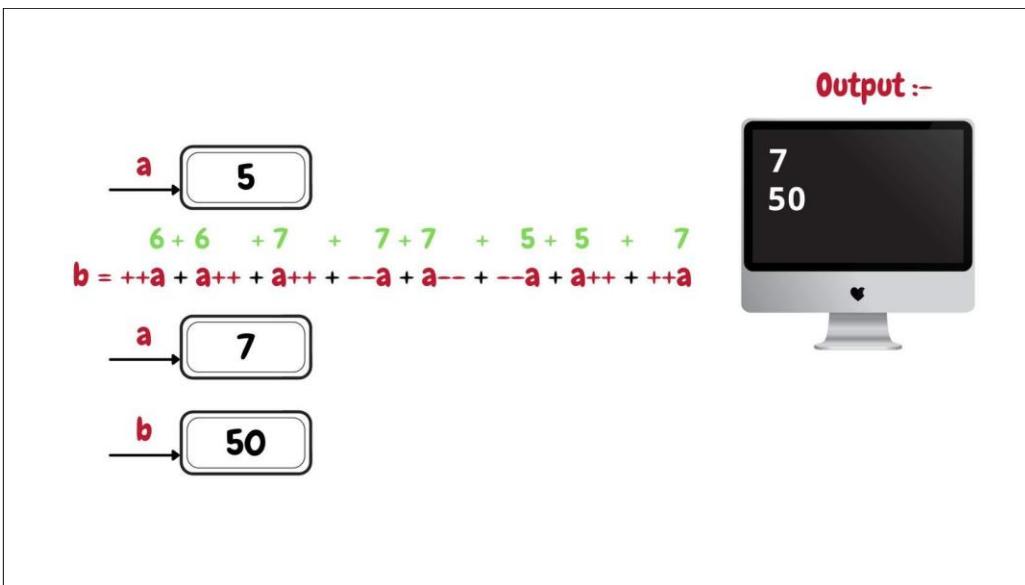
$b = ++a + a++ + a++ + --a + a-- + --a + a++ + ++a;$

Code:

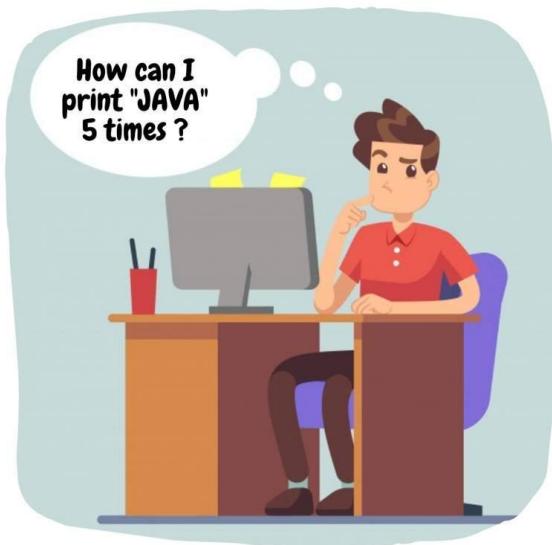
```

class Demo
{
    public static void main (String [] args)
    {
        int a = 5 ;
        int b;
        b = ++a + a++ + a++ + --a + a-- + --a + a++ + ++a;
        System.out.println(a);
        System.out.println(b);
    }
}

```



Loops



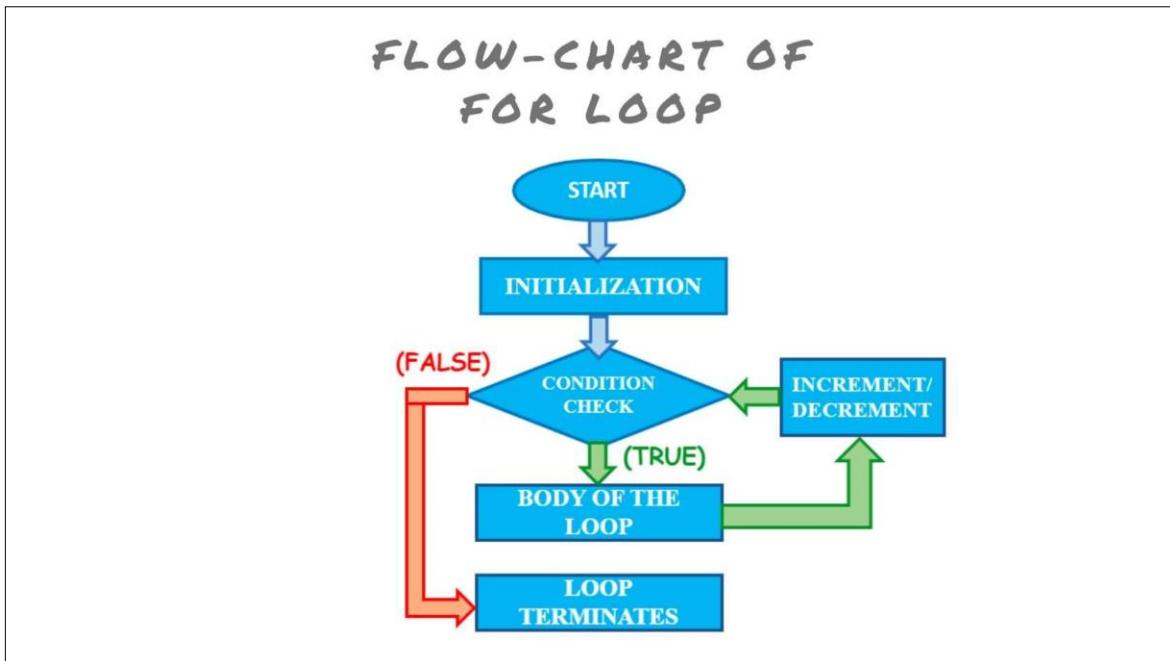
```
class Demo
{
    public static void main(String [] args)
    {
        System.out.println("JAVA");
        System.out.println("JAVA");
        System.out.println("JAVA");
        System.out.println("JAVA");
        System.out.println("JAVA");
    }
}
```

IS THIS EFFICIENT
WAY TO WRITE A
PROGRAM ?
?????

Imagine you have to print "JAVA" 100 times?

Are you going to type the statement 100 times?

Here comes the loop to rescue the programmer and make the job easy. Most fundamental and basic loop is known as FOR loop.



Syntax:

for (initialization; condition; increment/ decrement)

Code:

```
class Demo
{
    public static void main (String [] args)
    {
        int i;
        for ( i=1; i<=5; i++)
        {
            System.out.println("JAVA");
        }
    }
}
```

Output

1. Print below pattern

*

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

2. Print below pattern

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        for(int i = 1; i <= 5; i++)  
        {  
            System.out.print("*");  
        }  
  
    }  
  
}
```

3. Print below pattern

*

*

*

*

*

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        for(int i = 1; i <= 5; i++)  
        {  
            System.out.println("*");  
        }  
  
    }  
}
```

4. Print below pattern


```
public class Demo {  
  
    public static void main(String[] args) {  
  
        for(int i = 1; i <= 5; i++)  
        {  
            for(int j = 1; j<= 5; j++)  
            {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

5.Print below pattern

```
1 1 1 1 1  
2 2 2 2 2  
3 3 3 3 3  
4 4 4 4 4  
5 5 5 5 5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        for(int i = 1; i <= 5; i++)  
        {  
            for(int j = 1; j<= 5; j++)  
            {  
                System.out.print(i + " ");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

6.Print below pattern

```
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        for(int i = 1; i <= 5; i++)  
        {  
            for(int j = 1; j<= 5; j++)  
            {  
                System.out.print(j + " ");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

7. Print below pattern

```
* * * * *  
  
* *  
  
* *  
  
* *  
  
* * * * *
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j < n; j++)  
            {  
                if(i == 0 || i == n-1 || j == 0 || j == n-1)  
                {  
                    System.out.print("*");  
                }  
            }  
        }  
    }  
}
```

```
        else
        {
            System.out.print(" ");
        }
    System.out.println();
}

}
```

8.Print below pattern

```
01 02 03 04 05
06 07 08 09 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

```
public class Demo {

    public static void main(String[] args) {

        int n = 5;
        int count = 1;
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < n; j++)
            {
                if(count < 10)
                {
                    System.out.print("0");
                }
                System.out.print(count + " ");
                count++;
            }
            System.out.println();
        }
    }
}
```

```
    }

}

}
```

9. Print below pattern

```
01 02 03 04 05  
02 04 06 08 10  
03 06 09 12 15  
04 08 12 16 20  
05 10 15 20 25
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 1; i <= n; i++)  
        {  
            for(int j = 1; j <= n; j++)  
            {  
                if( i*j < 10)  
                {  
                    System.out.print("0");  
                }  
                System.out.print(i*j + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

10. Print below pattern

```
1 2 3 4 5  
2 3 4 5 6  
3 4 5 6 7  
4 5 6 7 8  
5 6 7 8 9
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j < n; j++)  
            {  
                System.out.print(i+j+1 + " ");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

11. Print below pattern

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

12. Print below pattern

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print(j+1 + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

```
    }

}

}
```

13. Print below pattern

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

```
public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j <= i; j++)
            {
                System.out.print(i+1 + " ");
            }
            System.out.println();
        }
    }
}
```

14. Print below pattern

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int k = 0; k < n-1-i; k++)  
            {  
                System.out.print(" ");  
            }  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

15. Print below pattern

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int k = 0; k < n-1-i; k++)  
            {  
                System.out.print(" ");  
            }  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print("* " + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

16. Print below pattern

```
*  
* *  
* * *  
* * * * *
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int k = 0; k < n-1-i; k++)  
            {  
                System.out.print(" ");  
            }  
            for(int j = 0; j <= i; j++)  
            {  
                if( j == 0 || j == i || i == n-1)  
                {  
                    System.out.print("*" + " ");  
                }  
                else  
                {  
                    System.out.print(" ");  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

17. Print below pattern

```
1  
1 2  
1   3  
1       4  
1 2 3 4 5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int k = 0; k < n-1-i; k++)  
            {  
                System.out.print(" ");  
            }  
            for(int j = 0; j <= i; j++)  
            {  
                if( j == 0 || j == i || i == n-1)  
                {  
                    System.out.print(j+1 + " ");  
                }  
                else  
                {  
                    System.out.print(" ");  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

18. Print below pattern

```
1  
1 2  
1   3  
1     4  
1 2 3 4 5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j <= i; j++)  
            {  
                if( j == 0 || j == i || i == n-1)  
                {  
                    System.out.print(j+1 + " ");  
                }  
                else  
                {  
                    System.out.print(" ");  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

19. Print below pattern

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print(j+1 + " ");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

20. Print below pattern

```
1 2 3 4 5  
1 2 3 4  
1 2 3  
1 2  
1
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j < n-i; j++)  
            {  
                System.out.print(j+1 + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

21. Print below pattern

```
1 2 3 4 5  
2 3 4 5  
3 4 5  
4 5  
5
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j < n-i; j++)  
            {  
                System.out.print(i+j+1 + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

```
    }

}

}
```

22. Print below pattern

```
1 2 3 4 5
2     5
3   5
4 5
5
```

```
public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < n-i; j++)
            {
                if(i == 0 || i == n-1 || j == 0 || j == n-i-1)
                {
                    System.out.print(i+j+1 + " ");
                }
                else
                {
                    System.out.print("  ");
                }
            }
            System.out.println();
        }
    }
}
```

23. Print below pattern

```
*  
* * *  
* * * * *  
* * * * * * * *  
* * * * * * * * *
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 1; i <= n; i++)  
        {  
            for(int j = 1; j <= (2*i)-1; j++)  
            {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
  
    }  
}
```

24. Print below pattern

```
1  
1 2 3  
1 2 3 4 5  
1 2 3 4 5 6 7  
1 2 3 4 5 6 7 8 9
```

```

public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 1; i <= n; i++)
        {
            for(int j = 1; j <= (2*i)-1; j++)
            {
                System.out.print(j);
            }
            System.out.println();
        }

    }
}

```

25. Print below pattern

```

*
* *
* * *
* * * *
* * * * *
* * * * * *

```

```

public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 1; i <= n; i++)
        {
            for(int k = 1; k <= n-i; k++)
            {
                System.out.print(" ");
            }
            for(int j = 1; j <= (2*i)-1; j++)

```

```

        {
            System.out.print("* ");
        }
        System.out.println();
    }

}

```

26. Print below pattern

```

1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1

```

```

public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 1; i <= n; i++)
        {
            int count = 1;
            for(int j = 1; j <= (2*i)-1; j++)
            {
                if(j < i)
                {
                    System.out.print(count++ + " ");
                }
                else
                {
                    System.out.print(count-- + " ");
                }
            }
            System.out.println();
        }
    }
}

```

```
    }

}

}
```

27. Print below pattern

```
1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
```

```
public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 1; i <= n; i++)
        {
            int count = 1;
            for(int k = 1; k <= n-i; k++)
            {
                System.out.print(" ");
            }
            for(int j = 1; j <= (2*i)-1; j++)
            {
                if(j < i)
                {
                    System.out.print(count++ + " ");
                }
                else
                {
                    System.out.print(count-- + " ");
                }
            }
            System.out.println();
        }
    }
}
```

```
    }

}

}
```

28. Print below pattern

```
1 1 1 1 1 2  
2 2 2 2 2 3  
3 3 3 3 3 4  
5 4 4 4 4 4  
5 5 5 5 5 6
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 1; i <= n; i++)  
        {  
            if(i%2 == 0)  
            {  
                System.out.print(i+1);  
            }  
            for(int j = 1; j <= n; j++)  
            {  
                System.out.print(i);  
            }  
            if(i%2 != 0)  
            {  
                System.out.print(i+1);  
            }  
            System.out.println();  
        }  
    }  
}
```

29. Print below pattern

```
1  
2 2  
3 3 3  
4 4 4 4  
5 5 5 5 5  
4 4 4 4  
3 3 3  
2 2  
1
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print(i+1 + " ");  
            }  
  
            System.out.println();  
        }  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j < n-i; j++)  
            {  
                System.out.print(n-i + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

30. Print below pattern

```
1  
2 3  
4 5 6  
7 8 9 10  
11 12 13 14 15
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        int count = 1;  
        for(int i = 0; i < n; i++)  
        {  
            for(int j = 0; j <= i; j++)  
            {  
                System.out.print(count++ + " ");  
            }  
  
            System.out.println();  
        }  
  
    }  
}
```

31. Print below pattern

```
1  
3 * 2  
6 * 5 * 4  
10 * 9 * 8 * 7  
15 * 14 * 13 * 12 * 11
```

```

public class Demo {

    public static void main(String[] args) {

        int n = 5;
        for(int i = 1; i <= n; i++)
        {
            int count = (i*(i+1))/2;
            for(int j = 1; j <= i; j++)
            {
                System.out.print(count-- + " ");
                if(j < i)
                {
                    System.out.print("* ");
                }
            }
            System.out.println();
        }

    }
}

```

32. Print below pattern

A
 B B
 C C C
 D D D D
 E E E E E

```

public class Demo {

    public static void main(String[] args) {

        int n = 5;
        char ch = 'A';
        for(int i = 1; i <= n; i++)

```

```
{  
    for(int j = 1; j <= i; j++)  
    {  
        System.out.print(ch + " ");  
    }  
    System.out.println();  
    ch++;  
}  
}  
}
```

33. Print below pattern

A
A B
A B C
A B C D
A B C D E

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        for(int i = 1; i <= n; i++)  
        {  
            char ch = 'A';  
            for(int j = 1; j <= i; j++)  
            {  
                System.out.print(ch + " ");  
                ch++;  
            }  
            System.out.println();  
        }  
    }  
}
```

34. Print below pattern

A
B C
D E F
G H I J
K L M N O

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        char ch = 'A';  
        for(int i = 1; i <= n; i++)  
        {  
            for(int j = 1; j <= i; j++)  
            {  
                System.out.print(ch + " ");  
                ch++;  
            }  
            System.out.println();  
        }  
    }  
}
```

35. Print below pattern

A
C B
D E F
J I H G
K L M N O

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        int n = 5;  
        char ch = 'A';  
        for(int i = 1; i <= n; i++)  
        {  
            if(i%2 == 0)  
            {  
                char chRev = (char)(ch + i - 1);  
                for(int j = 1; j <= i; j++)  
                {  
                    System.out.print(chRev-- + " ");  
                    ch++;  
                }  
                System.out.println();  
            }  
            else  
            {  
                for(int j = 1; j <= i; j++)  
                {  
                    System.out.print(ch + " ");  
                    ch++;  
                }  
                System.out.println();  
            }  
        }  
    }  
}
```

Assignment 1:

1. Pattern1

1 1 1 1 1
1 2 1 1 1
1 1 3 1 1
1 1 1 4 1
1 1 1 1 5

2. Pattern2

1 1 1 1 1
1 2 1 1 1
1 2 3 1 1
1 2 3 4 1
1 2 3 4 5

3. Patten 3

1 1 1 1 1
2 2 1 1 1
3 3 3 1 1
4 4 4 4 1
5 5 5 5 5

4. Pattern4

5 5 5 5 5
4 4 4 4 4
3 3 3 3 3
2 2 2 2 2
1 1 1 1 1

5. Pattern5

5 4 3 2 1
5 4 3 2 1
5 4 3 2 1

5 4 3 2 1
5 4 3 2 1

6. Pattern6

101010
010101
101010
010101
101010

7. Pattern7

11111
01111
00111
00011
00001

8. Pattern8

10000
02000
00300
00040
00005

9. Pattern9

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

10. Pattern10

5
4 5
3 4 5
2 3 4 5
1 2 3 4 5

11. Pattern11

ABCDE
BCDE
CDE
DE
E

12. Pattern 12

EDCBA
EDCB
EDC
ED
E

13. Pattern 13

A
B B
C C C
D D D D
E E E E E

14.Pattern 14

A
CBA
EDCBA
GFEDCBA
IHGFEDCBA

15.Pattern 15

E
D D
C C
B B
A A
B B
C C
D D
E

16.Pattern 16

E
DE
CDE
BCDE
ABCDE

17.Pattern 17

1

3 2

4 5 6

10 9 8 7

11 12 13 14 15

18.Pattern 18

EEEEEE

DDDD

CCC

BB

A

19.Pattern 19

ABCDE

ABCDE

ABCDE

ABCDE

ABCDE

20.Pattern 20

ABCDE

BCDE

CDE

DE

E

VARIABLES

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location.

There are three types of variables in java:

- local
- Instance
- Static

Now let's understand about Local and Instance Variables

Instance Variables:

The variables which will get created inside the class outside the methods are called instance variables

Code segment

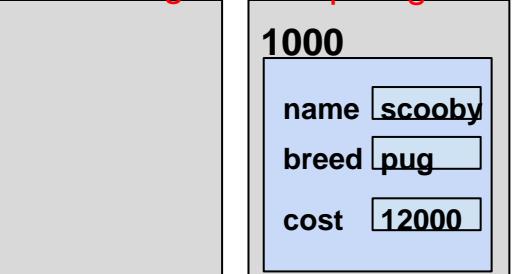
```
class Dog {
    String name = "scooby";
    String breed = "pug";
    int cost = 12000;

    public static void main(String args)
        Dog d = new Dog();
        System.out.println(d.name);
        System.out.println(d.breed);
        System.out.println(d.cost);
    }
}
```

Static segment



Stack Segment



Here control of execution will start from the main method. In the main method when a new keyword is called an object is created in the heap segment and memory for instance variables name, breed and cost is allocated the default values for the

variables is allocated then the values that is assigned will get allocated i.e name="scooby", breed = "pug", cost = 12000.

Local Variables:

The variables which will get created inside the class inside the methods are called local variables

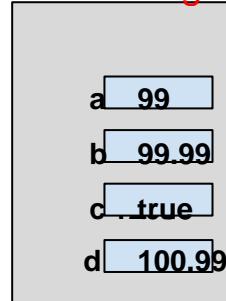
Code segment

```
class Test {  
    public static void main(String args)  
        int a =99;  
        float b = 99.99f;  
        double d =100.99;  
        boolean c = true;  
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(c);  
        System.out.println(d);  
    }  
}
```

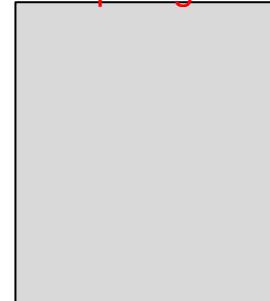
Static segment



Stack Segment



Heap Segment



The control of execution will start from the main method and the memory for variables a,b,c,d is allocated in the stack segment and values will be assigned. Default values for local variables will not be assigned by jvm.

Methods

There are four different types of methods

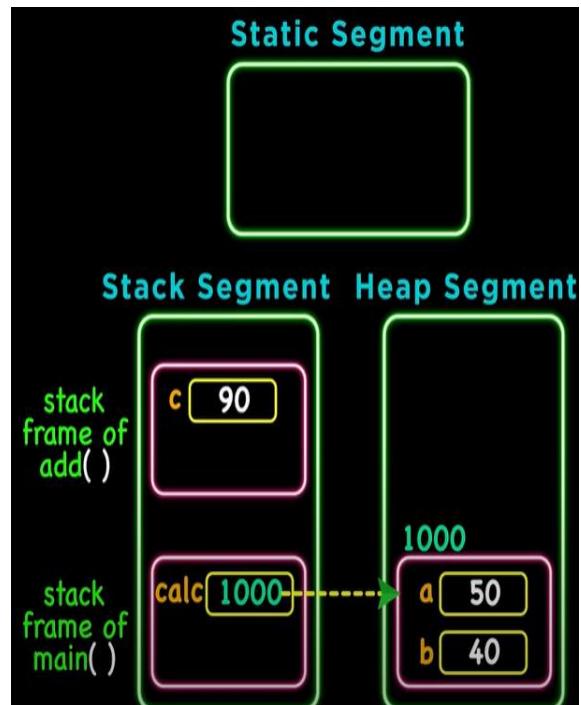
- Without Input and without Output
- With Input and without output
- Without Input and with output
- With input and with output

Case-1: Without input and without output

Code Segment

```
class Calculator {
    int a = 50;
    int b = 40;

    void add()
    {
        int c = a + b;
        System.out.println(c);
    }
    public static void main(String args[])
    {
        Calculator calc = new Calculator();
        calc.add();
    }
}
```



Output:

30

Explanation:

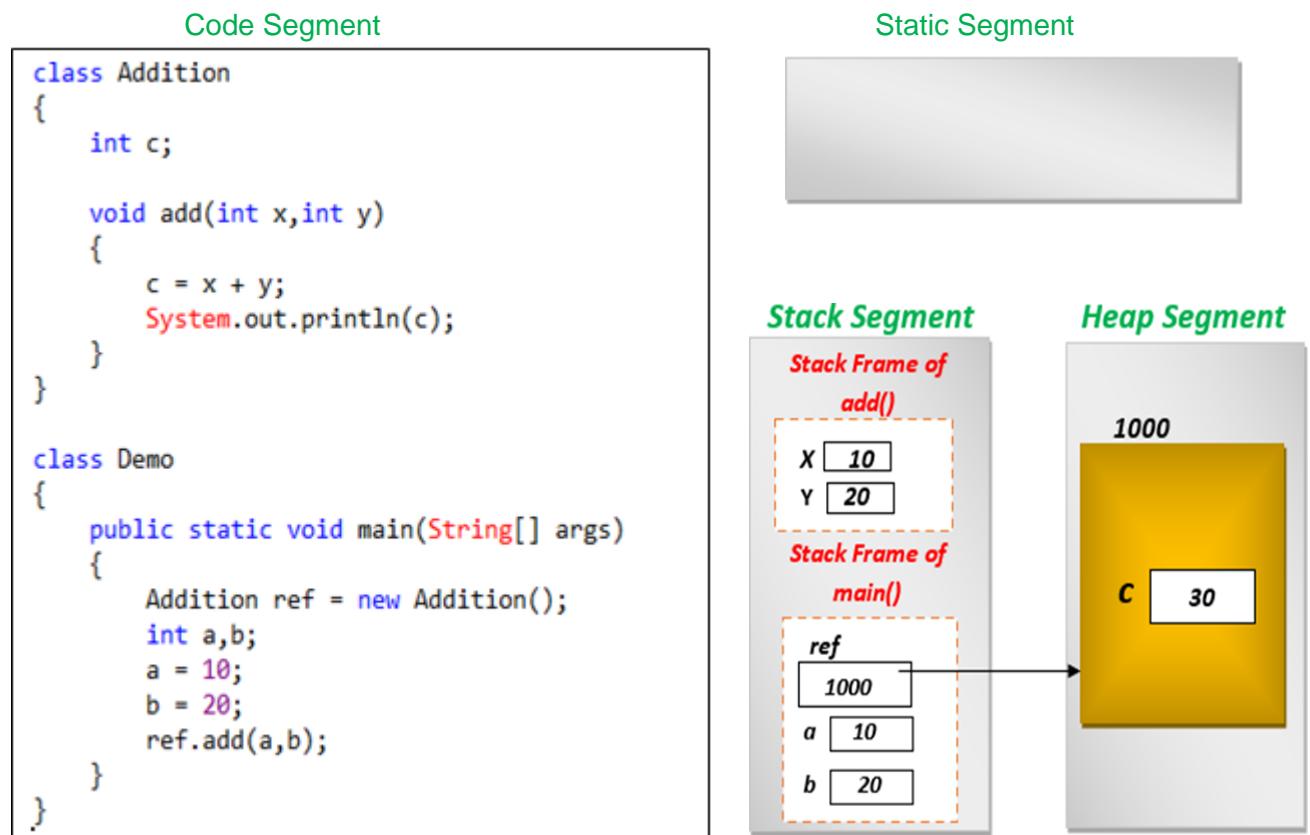
Here, the execution starts from the `main method()` which is called by the Operating System. Whenever, a method is called a region is created in the stack segment called **Stack frame**. And therefore, the `stack frame of main()` gets created on the stack segment.

By using a “`new`” keyword an object is created and memory for it is allocated in the heap segment. The instance variable **a** and **b** is allocated memory in the heap segment, local variable **c** allocated is memory on stack segment in stack frame and default values are given

to instance variables by the JVM. A reference variable is created with name **c** and is created in Stack segment.

Now, **add()** is a method that is executing. This method **add()** is called and the stack frame of **add()** is created in the Stack segment. Then, the body of **add()** is executed, now this method adds two values and is collected in **c** value and control goes back to the caller of the method.

Case-2: With input and without output



Output:

30

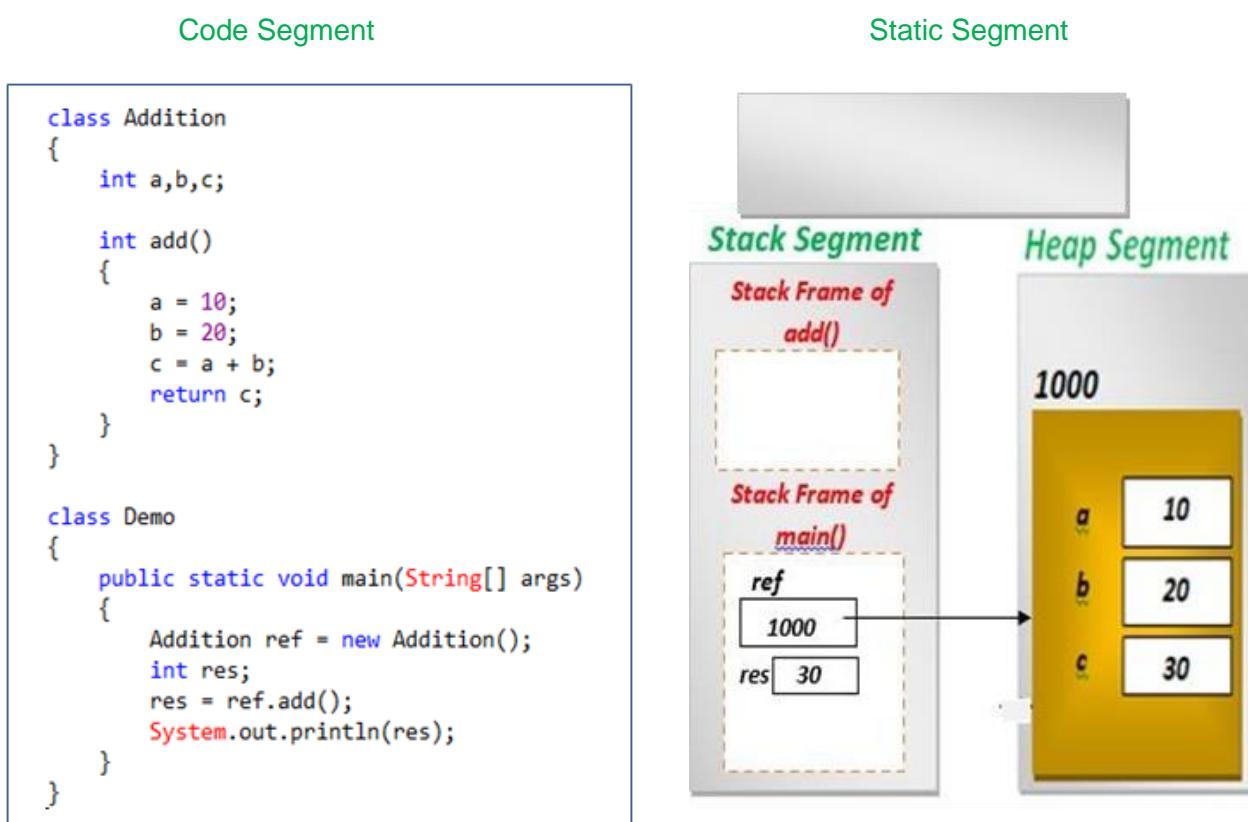
Explanation:

Here, the execution starts from the **main** method() which is called by the Operating System. Whenever, a method is called a region is created in the stack segment called **Stackframe**. And therefore, the stack frame of **main()** gets created on the stack segment.

By using a “**new**” keyword an object is created and memory for it is allocated in the heap segment. The instance variable **c** is allocated memory in the heap segment, local variables **x,y,a,b** are allocated memory on the stack segment on their respective stack frames and default values are given to instance variables by the JVM. A reference variable is created with name **ref** and is created in Stack segment.

Now, **add()** is a method which accepts two parameters as inputs. This method **add()** is called and the stack frame of **add()** is created in the Stack segment. Then, the body of **add()** is executed, now this doesn’t return any value hence, control goes back to the caller of the method.

Case-3: Without input and with output



Output:

30

Explanation:

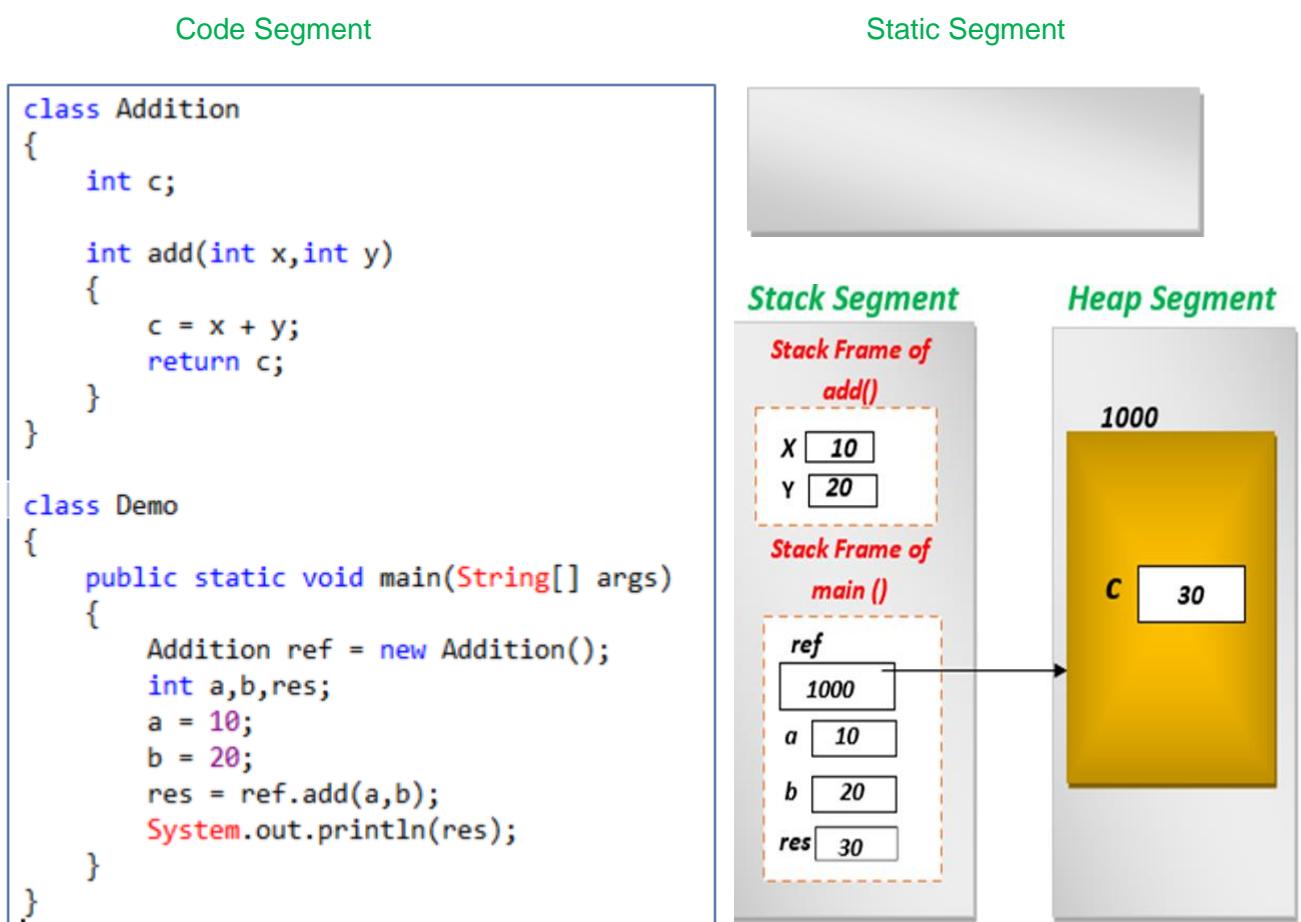
Here, the execution starts from **main method()** which is called by the Operating System. Whenever, a method is called a region is created in the stack segment called **Stack frame**. And therefore, stack frame of **main()** gets created on the stack segment.

By using a “**new**” keyword an object is created and memory for it is allocated in the heap segment. The instance variable **a,b,c** are allocated memory in the heap segment, local variable **res** allocated is memory on stack segment in stack frame of **main()** and default

values are given to instance variables by the JVM. A reference variable is created with name **ref** and is created in Stack segment.

Now, **add()** is a method which accepts 0 parameters as inputs. This method **add()** is called and stack frame of **add()** is created in the Stack segment. Then, the body of **add()** is executed, now this method return **c** value which is collected in **res** in **main()** and control goes back to the caller of the method.

Case-4: With input and with output



Output:

30

Explanation:

Here, the execution starts from the **main** method() which is called by the Operating System. Whenever, a method is called a region is created in the stack segment called **Stackframe**. And therefore, the stack frame of **main()** gets created on the stack segment.

By using a “**new**” keyword an object is created and memory for it is allocated in the heap segment. The instance variable **c** is allocated memory in the heap segment, local variable **a,b,x,y,res** allocated is memory on stack segment in stack frame and default values are given to intsnace variables by the JVM. A reference variable is created with name **ref** and is created in Stack segment.

Now, **add()** is a method which accepts 2 parameters as inputs. This method **add()** is called and stack frame of **add()** is created in the Stack segment. Then, the body of **add()** is executed, now this method return **c** value which is collected in **res** in **main()** and control goes back to the caller of the method.

Value type assignment

```
class Test2
{
    public static void main(String[] args)
    {
        int x = 100; // Assigning the value to the variable x
        int y; // Declaring the variable y

        y = x; // assigning the value present in x to y

        System.out.println(x); // print the value of x
        System.out.println(y); // print the value of y

        x = 200; // modifying the value of x

        System.out.println(x); // print the value of x
        System.out.println(y); // print the value of y
    }
}
```

Output:

```
100
100
200
100
```

If you observe from the above output, if you change the value present in one variable it will not affect other variable

Reference type assignment

- Let's create two instance variable name and cost and assign values to those variables

```
class Car
{
    String name; // instance variable name is created of type string
    int cost; // instance variable cost is created of type int
}

class Test2
{
    public static void main(String[] args)
    {
        Car x = new Car(); // car class object is created
        x.name = "maruthi"; // assigning value to the instance variable
        name using object reference
        x.cost = 200000; // assigning value to the instance variable cost
        using object reference

        System.out.println(x.name); // print the value present in instance
        variable
        System.out.println(x.cost); // print the value present in instance
        variable

    }
}
```

Output:

```
maruthi
200000
```

- Let's create one more object reference y of type car and assign the reference present in x to y.

```
class Car
{
    String name; // instance variable name is created of type string
    int cost;    // instance variable cost is created of type int
}

class Test2
{
    public static void main(String[] args)
    {
        Car x = new Car(); // car class object is created
        x.name = "maruthi"; // assigning value to the instance variable
        name using object reference
        x.cost = 200000; // assigning value to the instance variable cost
        using object reference

        System.out.println(x.name); // print the value present in instance
        variable name
        System.out.println(x.cost); // print the value present in instance
        variable cost

        Car y; // create an object reference of type y
        y = x; // assign the reference present inside x to y

        System.out.println(y.name); // print the value present in instance
        variable name using object reference y
        System.out.println(y.cost); // print the value present in instance
```

```
variable cost using object reference y
```

```
}
```

Output:

```
maruthi
200000
maruthi
200000
```

If you observe from the above output when you print the values using object reference x and y it is giving you same result it is because both are pointing to same object

- **Now both the object reference x and y are pointing to the same object. Let's try to modify the value of instance variable using one reference**

```
class Car
{
    String name;
    int cost;
}

class Test2
{
    public static void main(String[] args)
    {
        Car x = new Car();
        x.name = "maruthi";
        x.cost = 200000;

        System.out.println(x.name);
    }
}
```

```

System.out.println(x.cost);

Car y;
y = x;

System.out.println(y.name);
System.out.println(y.cost);

y.name = "BMW"; // now let's modify the instance variable name
using object reference y
y.cost = 500000; // now let's modify the instance variable cost
using object reference y

System.out.println(x.name); // print the value of name using
object reference x
System.out.println(x.cost); // print the value of cost using object
reference x
}
}

```

Output:

```

maruthi
200000
maruthi
200000
BMW
500000

```

If you observe from the above output if you modified the value using one object reference it will effect all other references pointing to the same object

Pass by value

- Create a method which accepts two parameters and return the result

```
class Calculator
{
    int c; // create an instance variable c of type int

        // define the method add which accepts two
parameters perform addition operation and return the
result
    int add(int a, int b)
    {
        c = a + b;
        return c;
    }

    public static void main(String[] args)
    {
        Calculator calc = new Calculator(); // create
an object of class calculator
        int num1 = 50; // assign value 50 to the
variable num1
        int num2 = 40; //assign the value 40 to the
variable num2
        int res = calc.add(num1, num2); // call add
method and store the returned value store it in res
        System.out.println(res); //print res
    }
}
```

Output:

90

```
class Calculator
{
    int c;
    int add(int a, int b)
    {
        a = 500; // here local variable value is
assigned with value 500
        b = 400; // here local variable value is
assigned with value 400
        c = a + b; //Addition of two variable a and b
is done and store the result in c
        return c; // result the value present in c
    }

    public static void main(String[] args)
    {
        Calculator calc = new Calculator(); // create
an object of class
        int num1 = 50; // declare local variable num1
and assign the value 10
        int num2 = 40; // declare local variable num1
and assign the value 10
        int res = calc.add(num1, num2); //call the
method by passing the value present in num1 and num2
        System.out.println(res); //print res that is
returned by the method

    }
}
```

Output:

900

- Now let's try to create the method which accept the reference of type car and modify the value of instance variable inside that method

```
class Car
{
    String name;
    int cost;

    void modifyCar(Car y)
    {
        y.name = "BMW";
        y.cost = 500000;
    }
}

class Test2
{
    public static void main(String[] args)
    {
        Car x = new Car();
        x.name = "maruthi";
        x.cost = 200000;

        System.out.println(x.name);
        System.out.println(x.cost);

        x.modifyCar(x);

        System.out.println(x.name);
        System.out.println(x.cost);
    }
}
```

Output:

```
maruthi  
200000  
BMW  
500000
```

Here if you observe from the above output the you have updated the value by using the reference y now if you try to access it using the reference x you are getting updated value it is because of pass by reference

Variables

1. What is an object also called as?

Object is also called as Instance.

2. What is the default name of the object?

No default name is associated with an object.

3. How do we access an object?

Through a reference or handle.

4. Why do we require a reference variable?

To access the has part and does part of an object.

5. What are the contents of an object also called as?

Instance variables.

6. What is a reference variable also called as?

Handle.

7. Who allocates memory for the object?

JVM.

8. Where is memory for the object allocated? Why?

Memory for an object would be allocated on Heap segment so that garbage collector can de-allocate memory of an object.

9. When is memory for an object allocated?

When new keyword is executed object would be allocated memory on the Heap segment.

10. When is memory for an object de-allocated?

Whenever there is no reference referring to an object then an object becomes garbage and garbage collector would de-allocate memory.

11. What are the different types of segments on the RAM?

There are four segments namely Code segment, Stack segment, Static segment and Heap segment.

12. Who allocates memory for the reference variables?

JVM

13. Where is memory for reference variable allocated? Why?

Memory for the reference variables would be created in the stack segment inside the activation record. Whenever an object is created, it's always stored in the heap segment and stack memory contains the reference to it.

14. When is memory for reference variable allocated?

When the control enters into the method.

15. When is memory for a reference variable deallocated?

When the control leaves the method.

16. Who allocates memory for the local variables?

JVM

17. Where is memory for local variable allocated? Why?

In the stack segment, inside the activation record.

18. When is memory for local variable allocated?

When the control enters into the method.

19. When is memory for a local variable deallocated?

When the control leaves the method.

20. What does 4000 represent? Value or address?

It depends on the type of variable which is holding it.

21. What happens when assignment operator is applied on two value type variables?

Value present in one variable gets assigned to another variable.

22. What happens when assignment operator is applied on two reference type variables?

Both references start pointing to the same object.

23. What is the difference between local variables and instance variables?

Local variables: These are visible only in the method or block they are declared.

Instance variables: They can be seen by all methods in the class as they are declared inside a class but outside a method.

Local variables are not initialised by default, whereas **instance variables** are initialised by default.

24. Why do we not create a reference to a local variable?

Because local variables are already inside the stack and no reference is required to access them.

25. What are the default values associated with instance variables?

It depends upon data types. int-0, double-0.0, String-null, char-\u0000, Boolean-false.

26. What are the default values associated with local variables?

Local variables do not associate with any default values. Rather they must manually be initialized.

27. Can instance variables be used without initialization?

Yes.

28. Can local variables be used without initialization?

No.

29. Can a Student object be cast to a double value using type casting?

No. Type casting can take place between primitive types, type casting can also take place between the classes. However between a primitive type and an object, casting cannot take place.

30. What is the role of instanceof operator?

instanceof operator enables a programmer to verify the class to which an object belongs to.

31. What is a field variable?

Instance variable is also called as field variable.

32. What does the code segment contain?

It contains program code.

33. What does the stack segment contain?

It contains local variables, reference variables, activation records

34. What does the static segment contain?

It contains static variables, static blocks and static methods.

35. What does the heap segment contain?

It contains instances or objects

36. Is memory for a class allocated? Explain

Yes. Memory for a class would be allocated on the code segment but not on the heap segment. Only for object, memory would be allocated on the heap segment

37. Which are the variable sized segments on the Ram?

Stack segment and Heap segment are the variable sized segments.

38. Which are the fixed sized segments on the Ram?

Code segment and Static segment are the fixed sized segments.

39. Why the variable sized segments are called so?

Because their size would increase and decrease during program execution.

40. Why the fixed sized segments are called so?

Because their size would not increase or decrease during program execution.

41. Why is the class called a “blueprint”?

Because JVM creates an object by referring to the class.

42. Can an object have zero reference?

Yes. An anonymous object has zero reference.

43. Can an object have one reference?

Yes.

44. Can an object have more than one reference?

Yes.

45. **Can a local variable have a reference?**

No

46. **Can we have a local variable with zero name?**

No

47. **Can we have a local variable one name?**

Yes.

48. **Can we have a local variable with many names?**

No.

49. **When does an active object become “garbage”?**

When there is no reference referring to an object, then active object becomes garbage. Such garbage object would be collected by the garbage collector.

Introduction to Wrapper class:



Let us first understand why wrapper class?

Java is an impure object-oriented programming language because it supports **primitive data type**.

You may be wondering; does it really matter if java is pure or impure object oriented.



To understand this, you must know an important fact about primitive data types which is **Primitive data types are not treated as objects in java.** ~~object~~

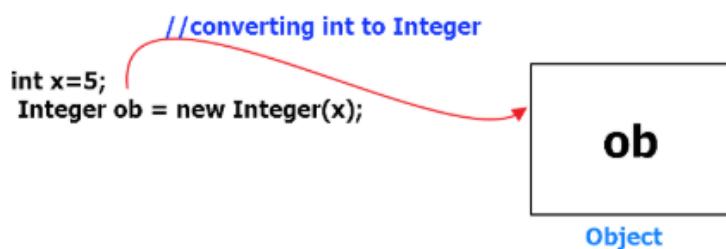
However, 100% pure object-oriented programs can be written in java using **Wrapper Class**.

Now you understood why wrapper class let us understand what is Wrapper class.

What is Wrapper class?

The **wrapper class in Java is a class** provides the mechanism **to convert primitive data types into object and object into primitive data type**.

Well how if you wonder, consider the below example:



In the above example **int** is a **primitive data** type and **Integer** is a **wrapper class**. Let us have a look at one more example of character data type.

Character

if you are using a single character value, you will use the primitive char type

```
char ch = 'a';
```

- ✓ There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected.
- ✓ The Java programming language provides a wrapper class that "wraps" the char in a Character object for this purpose. An object of type Character contains a single field, whose type is char. This Character class also offers a number of useful class (i.e., static) methods for manipulating characters.
- ✓ You can create a Character object with the Character constructor:

```
Character characterObj= new Character('a');
```

Advantage of Wrapper class: The program becomes pure object oriented.

Disadvantage of wrapper class: Execution speed decreases.

Advantage of Primitive data type: Faster in execution.

Disadvantage of Primitive data type: The program becomes impure object oriented.

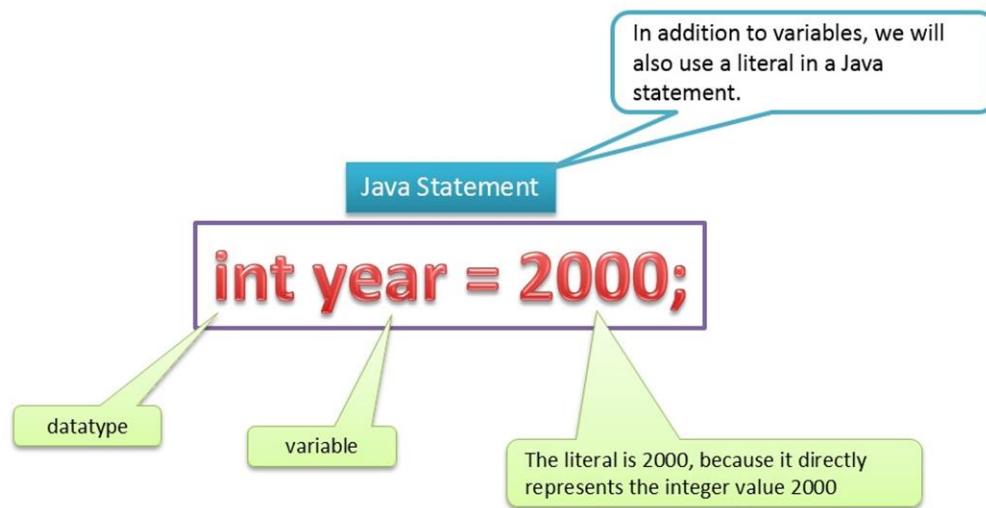
Different wrapper classes for different primitive data types is given below:

Wrapper Classes for Primitive Data Types

Primitive Data Types	Wrapper Classes
int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean

Literals in java:

Any constant value which can be assigned to the variable is called as literal.
Consider the example shown below:



Let us have a look at some valid and invalid syntaxes of variable names and literals.

Variable Names:

The symbols that are allowed in variable names are → _ and \$

Valid Syntax:

```
int rooman_ = 99;  
int _rooman_ = 99;  
int _r_ooman_ = 99;  
int roman$ = 99;  
int $rooman_ = 99;  
int r_o$$oman_ = 99;
```

Invalid Syntax:

```
int &rooman_ = 99;  
int _roo%man$ = 99;  
int roo man = 99;  
int roo “ man = 99;
```

Literals:

The only symbol that is allowed in literal is \Rightarrow “_”(underscore)

Valid Syntax:

```
int rooman_ = 9_9;  
int _rooman_ = 9__9;  
int _r_ooman_ = 9____9;
```

Invalid Syntax:

```
int rooman_ = 99_;  
int rooman = _99;  
float rooman = 99._9f;  
float rooman = 99.9_9f;  
float rooman = 99.99_f;
```

Introduction to Arrays:

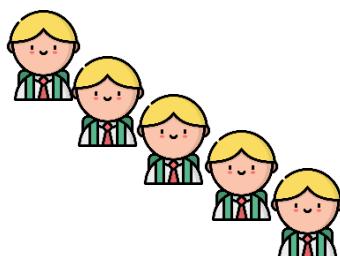
*In Java array is an **object** which contains elements of a similar data type.*

After getting to know the definition of array, let us now understand why array-based approach was introduced.

To understand this, let us first understand the limitations of variable approach by considering three different cases:

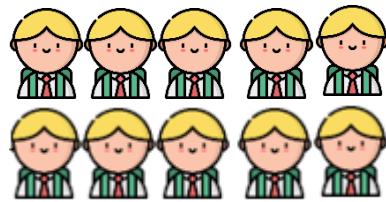
Case i) Write code to store the ages of 5 students.

```
int a;  
int b;  
int c;  
int d;  
int e;
```



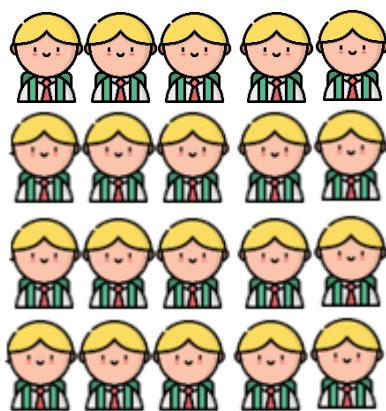
Caseii) Write code to store the ages of 10 students.

```
int a;  
int b;  
int c;  
int d;  
int e;  
int f;  
int g;  
int h;  
int i;  
int j;
```



Caseiii) Write code to store the ages of 20 students.

```
int a;  
int b;  
int c;  
int d;  
int e;  
int f;  
int g;  
int h;  
int i;  
int j;  
int k;  
int l;  
int m;  
int n;  
int o;  
int p;  
int q;  
int r;  
int s;  
int t;
```



After considering three different cases, if I consider case iii and ask you which variable stores the marks of 18th student, you'll now have to check each line to answer this question which is one of the limitations of variable approach.

The variable approach has two limitations:

- 1) ***Creation is difficult.***
- 2) ***Remembering multiple names and accessing them is difficult.***

Due to these limitations, array-based approach was invented.

Arrays are of two types:

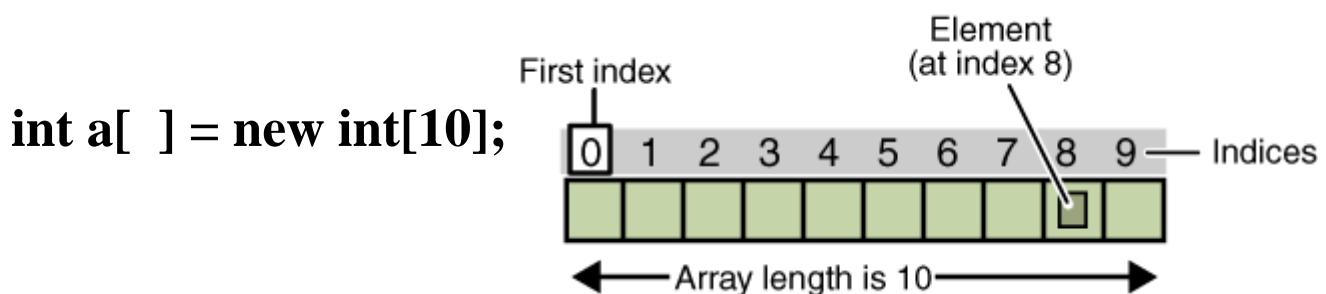
- 1)**Regular array.**
- 2)**Jagged array.**

1)Regular array: Regular array is further divided into 1-Dimensional, 2-Dimensional, 3-Dimensional array and so on.

1-Dimensional regular array/Rectangular array.

Let us learn the creation of 1-D array using an example.

Ex: Create an array to store the ages of 10 students:



2-Dimensional regular array/Rectangular array.

Let us learn the creation of 2-D array using an example.

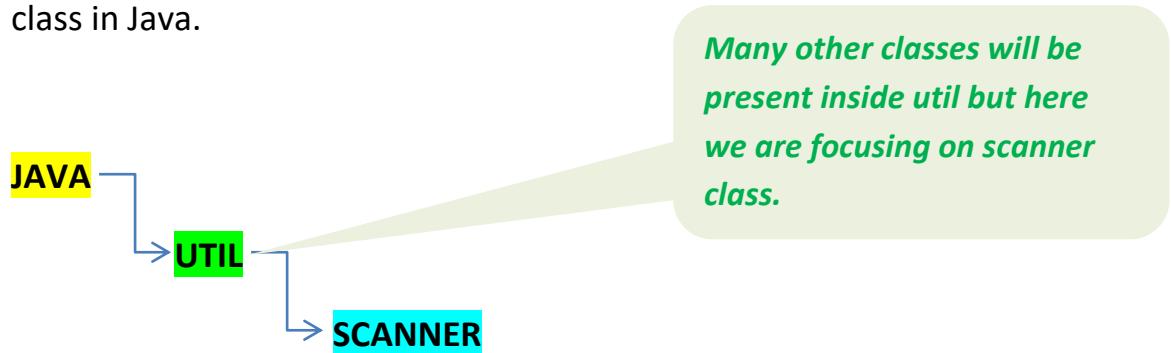
Ex: Create an array to store the ages of students belonging to 2 classrooms with 5 students each.

`int a[][] = new int[2][5];`
`a[1][0] = 5;`

	0	1	2	3	4
0	0	0	0	0	0
1	5	0	0	0	0
← Array length is 10 →					
2					
Rows					
Columns					

How to take input from users???

After knowing arrays in java, let's move towards how take inputs from scanner class in Java.



To import scanner class all you have to do is:

- 1) **Import java.util.Scanner;** //Add this line at top of the code.
- 2) Create object of Scanner class by saying :
Scanner scan = new Scanner(System.in);
- 3) To read different types of input, look at the table below:

Method	Description
nextBoolean()	Reads a Boolean value from the user.
nextByte()	Reads a Byte value from the user.
nextDouble()	Reads a Double value from the user.
nextFloat()	Reads a Float value from the user.
nextInt()	Reads a Integer value from the user.
nextLine()	Reads a Line value from the user.
nextLong()	Reads a Long value from the user.
nextShort()	Reads a Short value from the user.

Continuation with arrays....

Now that you know why we go for array approach and what are different types of arrays, let's start with how to make use of it.

Example 1: Create an array to store the ages of 5 students

Solution:

```
import java.util.Scanner;

class Demo

{
    public static void main(String[] args)
    {
        int a[] = new int[5];

        Scanner scan = new Scanner(System.in);

        System.out.println("Enter the age:");
        a[0] = scan.nextInt();

        System.out.println("Enter the age:");
        a[1] = scan.nextInt();

        System.out.println("Enter the age:");
        a[2] = scan.nextInt();

        System.out.println("Enter the age:");
        a[3] = scan.nextInt();

        System.out.println("Enter the age:");
        a[4] = scan.nextInt();

        System.out.println("Enter the age:");
        a[5] = scan.nextInt();
    }
}
```



Whenever a set of instructions is repeating it's wise to use loops.

Let's see how to make the code efficient with the help of for loop.

```
import java.util.Scanner;

class Demo

{
    public static void main(String[] args)

    {
        int a[] = new int[5];

        Scanner scan = new Scanner(System.in);

        for(int i =0; i<=4; i++)

        {
            System.out.println("Enter the age:");

            a[i] = scan.nextInt();

        }
    }
}
```



But instead of specifying array size in the for loop condition part we can make use of a built-in property called as length. Which will give the length of the array by saying **a.length-1** (-1 because the array indexing starts from 0).

Let's see how length works for 2D, 3D array

Consider we have a 2D array,

```
int a[][] = new int[2][5];
```

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0

a.length → Number of rows

a[0].length → Number of elements in a[0]

a[1].length → Number of elements in a[1]

a[i].length → Number of columns

Let's now consider 3D array,

```
int a[][][] = new int [2][3][5];
```

a.length → Number of blocks

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0

a[i].length → Number of rows

a[i][j].length → Number of Columns

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0

Let us now try to code a 2D array

```
import java.util.Scanner;

class Demo

{
    public static void main(String[] args)

    {
        int a[][] = new int[2][5];

        Scanner scan = new Scanner(System.in);

        for(int i =0; i<=a.length-1; i++)

        {
            for(int j=0; j<=a[i].length-1; j++)

            {
                System.out.println("Enter the age of class "+ i
                    +"student"+ j );

                a[i][j] = scan.nextInt();

            }
        }
    }
}
```



Continuation of arrays...

Let us try to write the code for few other examples.

Example 1): Create an array to store the ages of students belonging to 2 classrooms with 5 students each.

Solution:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        int a[][] = new int[2][5];
        Scanner scan = new Scanner(System.in);

        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                System.out.println("Enter the ages of school"+i+"class"+j+"student: ");
                a[i][j] = scan.nextInt();
            }
        }
        System.out.println("The ages are: ");
        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                System.out.println(a[i][j]);
            }
        }
    }
}
```

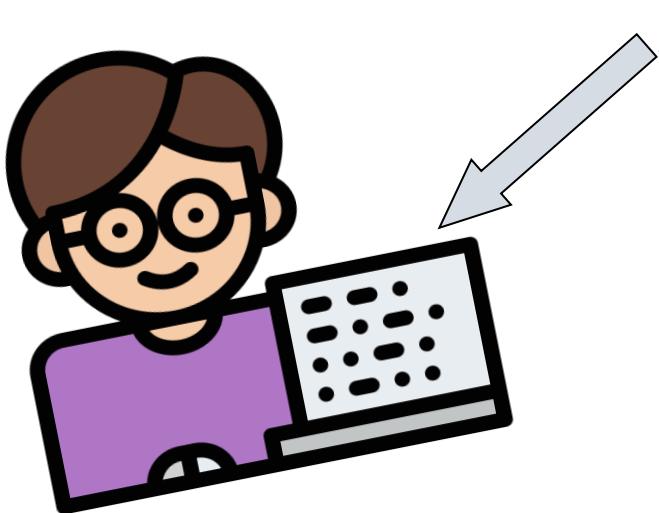


Example 2): Create an array to store the ages of students belonging to 2 schools having 3 classrooms with 5 students each.

Solution:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        int a[][][] = new int[2][3][5];
        Scanner scan = new Scanner(System.in);

        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                for(int k=0;k<=a[i][j].length-1;k++)
                {
                    System.out.println("Enter the ages of school "+i+"class "+j+"student"+k);
                    a[i][j][k]= scan.nextInt();
                }
            }
        }
        System.out.println("The ages are: ");
        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                for(int k=0;k<=a[i][j].length-1;k++)
                {
                    System.out.println(a[i][j][k]);
                }
            }
        }
    }
}
```



HE GOT THE OUTPUT. DID YOU?

Let us look at an example of two-dimensional jagged array.

Example 3): Create an array to store the ages of students belonging to 2 classrooms where the first classroom has 3 students and second classroom has 5 students.

Solution:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        int a[][] = new int[2][];
        a[0] = new int[3];
        a[1] = new int[5];
        Scanner scan = new Scanner(System.in);

        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                System.out.println("Enter the ages of class "+i+"student"+j);
                a[i][j]= scan.nextInt();
            }
        }
        System.out.println("The ages are: ");
        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                System.out.println(a[i][j]);
            }
        }
    }
}
```

DID YOU KNOW?

Bill Gates began programming
Computers at the age of 13.



Similarly let us try coding for 3D jagged array.

Example 3): Create an array to store the data given below:

School Classrooms Students

0	0	0-1
	1	0-2
	2	0-2
1	0	0-1
	1	0-2

Solution:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        int a[][][] = new int[2][][][];
        a[0] = new int[3][];
        a[1] = new int[2][];
        a[0][0] = new int[2];
        a[0][1] = new int[3];
        a[0][2] = new int[3];
        a[1][0] = new int[2];
        a[1][1] = new int[3];
        Scanner scan = new Scanner(System.in);

        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                for(int k=0;k<=a[i][j].length-1;k++)
                {
                    System.out.println("Enter the ages of school "+i+"class "+j+"student"+k);
                    a[i][j][k]= scan.nextInt();
                }
            }
        }
        System.out.println("The ages are: ");
        for(int i=0;i<=a.length-1;i++)
        {
            for(int j=0;j<=a[i].length-1;j++)
            {
                for(int k=0;k<=a[i][j].length-1;k++)
                {
                    System.out.println(a[i][j][k]);
                }
            }
        }
    }
}
```

Disadvantages of Array

It is important for one to understand when to use an array and when not to use an array.

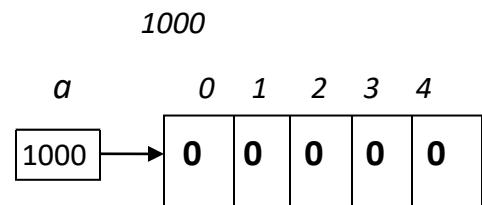
And to understand that, let us have a look at few Cases on Array.

Case-1:

Let us consider a one-dimensional array,

```
int a[ ] = new int [5];
```

which will be represented in the memory as,



In the above statement, we tell JVM to create an array of size 5 which can store only integer type values. Once this information is given to the JVM, what type of data it has to store, then only those types of data are allowed to store within the array.

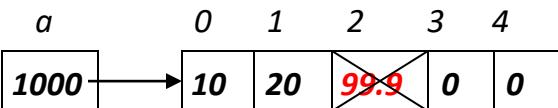
Arrays once created can only store data of same type.

Look at the below example:

```
int a[ ] = new int [5];
```

1000

```
a[0] = 10;
```



```
a[1] = 20;
```

```
a[2] = 99.9; ← Error
```

Here, in the example in $a[2] = 99.9$; we are trying to store float type of data which is not possible.

Arrays can only store homogeneous data.

Case-2:

Let us consider a one-dimensional array,

```
int a[ ] = new int [5];
```

```
a[0] = 10;
```

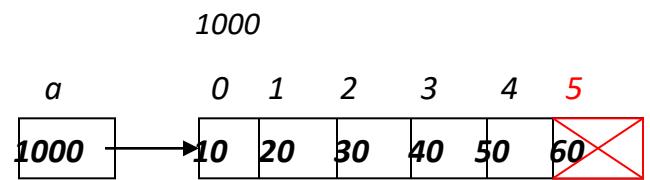
```
a[1] = 20;
```

```
a[2] = 30;
```

```
a[3] = 40;
```

```
a[4] = 50;
```

```
a[5] = 60;
```



In the above example, we have created an array of size 5. But by assigning `a[5] = 60;`

We are trying to store value 60 at the index 5, which means adding an extra size to the array. But this is not possible because the size of array is fixed, it cannot grow or shrink in size.

Arrays cannot grow or shrink in size.

Case-3:

We know that RAM is the main memory and it is collection of bytes, if an array is created even that makes use of RAM.

Let's assume we create an array of size 5.

```
int a[ ] = new int [5];
```

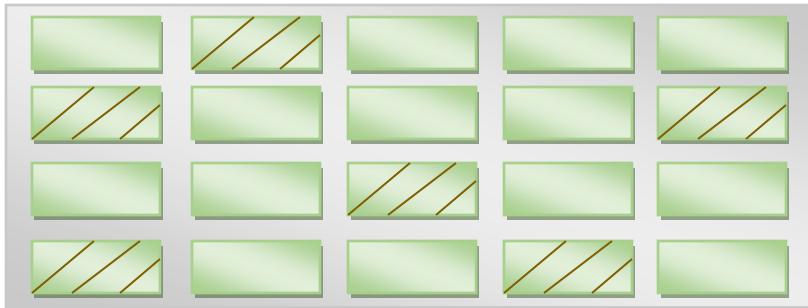
Arrays have an expectation that the size we mention to an array in terms of memory, those 5 memory locations must and should be **contiguous** (which means one next to the other) as shown below.



Contiguous memory

But in real life, in our computer multiple software's would be working and all make use of RAM. Therefore, free memory locations in reality are rarely available in RAM in continuous way, most of time it is dispersed.

Arrays demand continuous memory allocations and cannot make use of dispersed/scattered memory allocations.



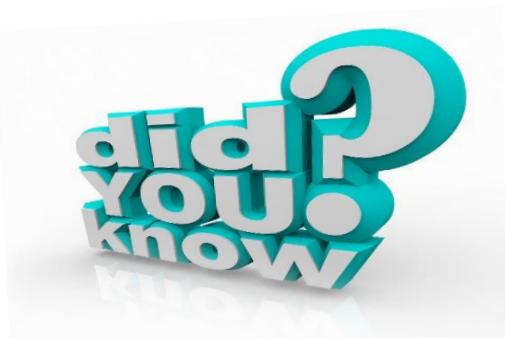
Dispersed memory

Arrays require contiguous memory allocation.

Points to remember:

- Arrays are objects
- They can even hold the reference variables of other objects
- They are created during runtime
- They are dynamic, created on the heap
- The Array length is fixed
- Array requires contiguous memory allocation
- They can only store homogeneous data.

**The original name for Java was
Oak.**



Arrays

1. When should an array data structure be used?

- a. Array data structure should be used when numerous data of the same type has to be stored.

2. What is an advantage of an array data structure?

Creation of an array is simple.

Inserting data into an array by making use of loops is simple. Also,

Extracting data from the array is simple by making use of loops.

3. What is the disadvantage of an array data structure?

- i) Arrays can store only homogeneous data.
Heterogeneous data cannot be stored
- ii) The size of an array is fixed throughout the program execution. Size of an array
- b. is neither increased nor decreased during program execution.
- c. iii) Arrays demand contiguous memory locations on the RAM. Arrays cannot utilize
- d. dispersed vacant memory locations.

4. Who creates arrays in Java and where?

- a. JVM creates array on the heap segment.

5. What are the different types of arrays in Java?

- a. 1-D, 2-D, 3-D and Multi-Dimensional Regular and jagged array.

6. Design an array data structure to collect marks of 100 students of a class?

- a. int a[] = new int[100];

7. Design an array data structure to collect marks of students in 5 classes each with 100 students?

- a. int a[][] = new int[5][100];

8. Design an array data structure to collect marks of students in 3 schools each with 5 classes each with 100 students?

- a. int a[][][] = new int[3][5][100];

9. Design an array data structure to collect marks of students in 3 schools with first school having 5 classes second school having 4 classes third school having 3 classes. Number of students in each classes is given below

First School	
Class	Number of students
1	3
2	4
3	2

4	1
5	5

Second School	
Class	Number of students
1	1
2	3
3	4
4	2

Third School	
Class	Number of students
1	2
2	3
3	4

10. Design an array data structure to collect data of 3 Banks with following number of branches and customers.

First Bank	
Branch	Number of Customer
1	2
2	1
3	3
4	5
5	4

Second Bank	
Branch	Number of Customer
1	4

2	1
3	2
4	3

Third Bank	
Branch	Number of Customer
1	3
2	2
3	4

11. What is the advantage of jagged array data structure?

- a. In real life, the data is not regular. Rather in most of the time, data is irregular or jagged. Hence, to provide a solution for the jagged data, java supports jagged array data structure.

12. Create an array of 5 person object?

Person p[] = new Person[5];

13. What are the exceptions associated with arrays?

ArrayIndexOutOfBoundsException, NegativeArraySizeException and ArrayStoreException.

14. Can array be used without initialization?

- a. Yes.

15. Are arrays objects in Java?

- a. Yes.

16. What are the default values associated with an array?

- a. The default value of an array depends upon the data type of an array.

Eg: int-0, float-0.0, char-blank character, boolean -false, String-null.

17. Who de-allocates memory of an array?

- a. A thread called as Garbage collector thread would de-allocate memory of an array.

18. Are arrays primitive data types?

- a. No. Arrays are objects in java.

19. Can an array grow in size or shrink in size after they are created?

- a. No.

20. What is the default value present in char array?

- a. '\u0000' i.e. blank character.

21. What is the default value present in objects array?

- a. Null

22. What is meant by buffer overrun?

- a. Buffer overrun refers to the process of attempting to access an index which is outside the boundary of an array. In java, if this is attempted, `ArrayIndexOutOfBoundsException` will occur.

23. What are most famous internet worms that were designed using buffer overrun?

- a. Morris worm, code red worm, scammer worm.

24. Does buffer overrun occur in java? Why?

- a. No, because java arrays are bound checked.

25. Can 1-dimensional arrays be printed directly?

No, because arrays are treated as objects in java. Contents of the objects cannot be printed directly except string. It can be printed using loops. If we have to print directly then we have to convert it into string using `toString()`. Example:`System.out.println(Arrays.toString(a));`

26. Can 2-dimensional arrays be printed directly?

- a. No.

27. What facility is available in java to directly print 1-dimensional arrays?

- a. Loops

28. What facility is available in java to directly print 2-dimensional arrays?

- a. Nested loops

29. What facility is available in Java to directly print other user defined objects? Ans: `toString()`

1. Discount Calculator

The iron and steel industry manufactures various products using iron, steel, tungsten, and nickel. Customers are the ones who place bulk orders to the industry. The industry gives two discount cards for its customers. One is based on the type of metal purchased and another one is based on the total purchase cost. At any point in time, customer can use one of the two discount cards.

The goal is to calculate the discount using the two discount cards and recommend the customer to use the one which gives the highest discount.

For example, the industry has the below two discount cards

Type of metal purchased	Discount% on purchase amount
Iron	7%
Steel	3%
Tungsten	2%
Nickel	1%

Total purchase cost in Rs	Discount% on the purchase cost
Till 25000	Nil
25001 to 50000	5%
50001 to 100000	7%
>100000	10%

If the customer purchases Rs.15000 worth of Iron, Rs.10000 worth of steel, Rs.2000 worth of Tungsten and Rs.1500 worth of Nickel, then as per the two discounts rate cards, then the discount can be calculated as follows,

As per discount rate card 1, discount rate = $(15000*7/100) + (10000*3/100) + (2000*2/100) + (1500*1/100)$ = Rs.1405.00

As per discount rate card 2, total purchase amount = $15000+10000+2000+1500= 28500$, hence discount rate = $28500*5/100 = \text{Rs.}1425.00$

As the discount is more using the second discount rate, the customer can use the second discount rate card and second discount amount.

Function Description

Complete the function calculateDiscount in the editor and its prints the discount rate.

calculateDiscount has the following parameter([S](#)):

Purchase cost of Iron: integer

Purchase cost of Steel: integer

Purchase cost of Tungsten: integer

Purchase cost of Neckel: integer

Constraints

- All the four input values are integers
- Final output should be printed in a float with two precisions

Input Format For Custom Testing

The first line contains an integer, n1, denoting the purchase cost of iron

The second line contains an integer, n2, denoting the purchase cost of steel

The third line contains an integer, n3, denoting the purchase cost of tungsten

The forth line contains an integer, n4, denoting the purchase cost of nickel

Sample Input

15000

10000

2000

1500

Sample Output

1425.00

Explanation

As per discount rate card 1, discount rate = $(15000*7/100) + (10000*3/100) + (2000*2/100) + (1500*1/100)$ = Rs.1405.00

As per discount rate card 2, total purchase amount = $15000+10000+2000+1500= 28500$, hence discount rate = $28500*5/100$ = Rs.1425.00

Higher discount is Rs.1425.00

Solution:

```
import java.util.Scanner;

public class DiscountCalculator {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);
        int iron = scan.nextInt();
        int steel = scan.nextInt();
        int tun = scan.nextInt();
        int nic = scan.nextInt();

        calculateDiscount(iron, steel, tun, nic);

    }

    static void calculateDiscount(int i, int s, int t, int n) {
```

```
float d1 = 0, d2 = 0;

d1 = i * (7.0f / 100) + s * (3.0f / 100) + t * (2.0f / 100) + n * (1.0f / 100);
float total = i + s + t + n;

if (total >= 25001 && total <= 50000) {
    d2 = total * (5.0f / 100);
} else if (total >= 50001 && total <= 100000) {
    d2 = total * (7.0f / 100);
} else if (total > 100000) {
    d2 = total * (10.0f / 100);
}

if (d1 > d2) {
    System.out.println(d1);
} else {
    System.out.println(d2);
}

}
```

2. Triangle Game

The Westland Game Fair is the premier event of its kind for kids interested in some intellectual and cognitive brain games. Exciting games were organized for kids between age group of 8 and 10. One such game was called the "Triangle game", where different number boards in the range 1 to 180 are available. Each kid needs to select three number boards, where the numbers on the boards correspond to the angles of a triangle.

If the angles selected by a kid forms a triangle, he/she would receive Prize 1. If the angles selected by a kid forms a right triangle, he/she would receive Prize 2. If the angles selected by the kids form an equilateral triangle, he/she would receive Prize 3. If the angles selected by a kid do not form even a triangle, then he/she will not receive any prizes. Write a program for the organizers to fetch the result based on the number boards selected by the kids.

Input Format:

There are 3 lines in the input, each of which corresponds to the numbers on the boards that the kids select.

Output Format:

The output should display "Prize 1" or "Prize 2" or "Prize 3" or "No Prize" based on the conditions given.

Refer sample input and output for formatting specifications.

Sample Input 1:

60

50

70

Sample Output 1:

Prize 1

Sample Input 2:

60

60

70

Sample Output 2:

No Prize

Solution:

```
import java.util.Scanner;

public class TriangleGame {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);
        int a = scan.nextInt();
        int b = scan.nextInt();
        int c = scan.nextInt();

        if (a + b + c == 180) {
            if (a == 60 && b == 60 && c == 60) {
                System.out.println("Prize 3");
            } else if (a == 90 || b == 90 || c == 90) {
                System.out.println("Prize 2");
            }
        }
    }
}
```

```
        System.out.println("Prize 1");
    }
} else {
    System.out.println("No Prize");
}

}
```

3. Count Divisors

You have been given 3 integers - l , r , and k . Find how many numbers between l and r (both inclusive) are divisible by k . You do not need to print these numbers, you just have to find their count.

Input Format

The first and only line of input contains 3 space-separated integers l , r , and k .

Output Format

Print the required answer on a single line.

Sample Input:

1 10 1

Sample Output:

10

Solution:

```
import java.util.Scanner;

public class CountDivisors {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);
        int l = scan.nextInt();
        int r = scan.nextInt();
        int k = scan.nextInt();

        int count = 0;
        for (int i = l; i <= r; i++) {
            if (i % k == 0) {
                count++;
            }
        }

        System.out.println(count);
    }
}
```

}

}

Angry Professor:

<https://www.hackerrank.com/challenges/angry-professor/problem>

```
import java.util.Scanner;

public class Demo {
    static String angryProfessor(int k, int[] a) {
        int count = 0;
        for (int i = 0; i < a.length; i++) {
            if (a[i] <= 0) {
                count++;
            }
        }
        if (count >= k) {
            return "NO";
        } else {
            return "YES";
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int k = scan.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < a.length; i++) {
            a[i] = scan.nextInt();
        }
        System.out.println(angryProfessor(k, a));
    }
}
```

Migratory Birds:

Link: <https://www.hackerrank.com/challenges/migratory-birds/problem>

```
import java.util.Scanner;

public class Demo {
    static int migratoryBirds(int[] a) {
        int[] birds = new int[6];
        for (int i = 0; i < a.length; i++) {
            birds[a[i]]++;
        }
        int maxIndex = 0, max = -1;
        for (int i = 1; i < birds.length; i++) {
            if (birds[i] > max) {
                max = birds[i];
                maxIndex = i;
            }
        }
        return maxIndex;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int[] a = { 1, 1, 2, 2, 3, 4 };
        System.out.println(migratoryBirds(a));
    }
}
```

Walk In a Row:

Vanya and his friends are walking along the fence of height h and they do not want the guard to notice them. In order to achieve this the height of each of the friends should not exceed h . If the height of some person is greater than h he can bend down and then he surely won't be noticed by the guard. The height of the i -th person is equal to a_i .

Consider the width of the person walking as usual to be equal to 1, while the width of the bent person is equal to 2. Friends want to talk to each other while walking, so they would like to walk in a single row. What is the minimum width of the road, such that friends can walk in a row and remain unattended by the guard?

Input Format

The first line of the input contains two integers n and k ($1 \leq n \leq 1000$, $1 \leq k \leq 1000$) — the number of friends and the height of the fence, respectively.

The second line contains n integers a_i ($1 \leq a_i \leq 2h$), the i -th of them is equal to the height of the i -th person.

Output Format

Print a single integer — the minimum possible valid width of the road.

Input:

```
5 6
4 5 14 7 5
```

Output:

```
7
```

Solution:

```
import java.util.Scanner;
public class Demo {
    static int walkWidth(int[] a, int k) {
        int width = 0;
        for (int i = 0; i < a.length; i++) {
            if (a[i] <= k) {
                width = width + 1;
            } else {
                width = width + 2;
            }
        }
        return width;
    }

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

```
Scanner scan = new Scanner(System.in);
int n = scan.nextInt();
int k = scan.nextInt();
int[] a = new int[n];
for (int i = 0; i < a.length; i++) {
    a[i] = scan.nextInt();
}
System.out.println(walkWidth(a, k));
}
```

Circular Array Rotation

Link:

<https://www.hackerrank.com/challenges/circular-array-rotation/problem>

Solution:

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Scanner;
public class Test {
    static int[] circularArrayRotation(int[] a, int k, int[] q) {
        int[] res = new int[q.length];
        int[] b = new int[a.length];
        for (int i = 0; i < a.length; i++) {
            b[(i + k) % a.length] = a[i];
        }
        for (int i = 0; i < q.length; i++) {
            res[i] = b[q[i]];
        }
        return res;
    }
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int k = scan.nextInt();
        int q = scan.nextInt();
        int a[] = new int[n];
        int queries[] = new int[q];
        for (int i = 0; i < a.length; i++) {
            a[i] = scan.nextInt();
        }
        for (int i = 0; i < queries.length; i++) {
            queries[i] = scan.nextInt();
        }
        int[] res = circularArrayRotation(a, k, queries);
        for (int i = 0; i < res.length; i++) {
            System.out.println(res[i]);
        }
    }
}
```

Merge Sorted Array:

```
import java.util.Scanner;

public class Test {
    static int[] mergeSortedArray(int[] ar1, int[] ar2) {
        int i = 0, j = 0, k = 0;
        int[] res = new int[ar1.length + ar2.length];
        while (i < ar1.length && j < ar2.length) {
            if (ar1[i] < ar2[j]) {
                res[k] = ar1[i];
                i++;
                k++;
            } else {
                res[k] = ar2[j];
                j++;
                k++;
            }
        }
        while (i < ar1.length) {
            res[k] = ar1[i];
            i++;
            k++;
        }
        while (j < ar2.length) {
            res[k] = ar2[j];
            j++;
            k++;
        }
        return res;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int[] ar1 = { 3, 5, 9, 12, 15 };
        int[] ar2 = { 1, 6, 8 };
        int[] res = mergeSortedArray(ar1, ar2);
        for (int i = 0; i < res.length; i++) {
            System.out.print(res[i] + " ");
        }
    }
}
```



AES NUMBER:

Write a program to print the number of aes numbers in the given range.

NOTE: AES number is the number which contains exactly 4 divisors then it is called as aes number

```
package mock;

import java.util.Scanner;
class Demo{

    static boolean isAesNumber(int n) {
        int count = 0;
        for (int i = 1; i <=n; i++) {
            if(n%i==0) {
                count++;
            }
        }

        if(count==4) {
            return true;
        }
        else {
            return false;
        }
    }

    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        int l = scan.nextInt();
        int r = scan.nextInt();
        int aes = 0;

        for(int i=l; i<=r; i++)
        {
            if(isAesNumber(i) == true) {
                aes++;
            }
        }
    }
}
```

```
        System.out.println(aes);
    }
}
```

Devu's friendship test

Devu has **n** weird friends. Its his birthday today, so they thought that this is the best occasion for testing their friendship with him. They put up conditions before Devu that they will break the friendship unless he gives them a grand party on their chosen day. Formally, **i**th friend will break his friendship if he does not receive a grand party on **d***i*th day.

Devu despite being as rich as Gatsby, is quite frugal and can give at most one grand party daily. Also, he wants to invite only one person in a party. So he just wonders what is the maximum number of friendships he can save. Please help Devu in this tough task !!

Input

- The first line of the input contains an integer **T** denoting the number of test cases.
The description of **T** test cases follows.
- First line will contain a single integer denoting **n**.
- Second line will contain **n** space separated integers where **i**th integer corresponds to the day **d***i*th as given in the problem.

Output

Print a single line corresponding to the answer of the problem.

Sample Input:

```
2
2
3 2
2
1 1
```

Sample Output:

```
2
1
```

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

```
class Sorting{
    static void selectionSort(int[] a) {
        for (int i = 0; i < a.length - 1; i++) {
            int min_i = i;
            for (int j = i + 1; j < a.length; j++) {
                if (a[j] > a[min_i]) {
                    min_i = j;
                }
            }
            int temp = a[i];
            a[i] = a[min_i];
            a[min_i] = temp;
        }
    }
}

class Demo{

    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int[] d = new int[n];

        for (int i = 0; i < d.length; i++) {
            d[i] = scan.nextInt();
        }

        Sorting.selectionSort(d);

        int count = 0;
        for (int i = 0; i < d.length; i++) {
            if(d[i] != d[i+1]) {
                count++;
            }
        }

        System.out.println(count+1);
    }
}
```

Linear search

```
class Searching{
    public static int linearSearch(int[] a, int key) {
        for (int i = 0; i < a.length; i++) {
            if(a[i] == key) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args)
    {
        int[] a = {50,10,33,40,26};
        int key = 99;
        System.out.println(linearSearch(a, key));
    }
}
```

Binary search

```
import java.util.Arrays;

class tr {
    public static int binarySearch(int[] a, int key) {
        int l = 0, h = a.length-1, mid = 0;

        while(l <= h) {
            mid = (l+h)/2;
            if(key == a[mid]) {
                return mid;
            }
            else if(key < a[mid]) {
                h = mid-1;
            }
            else {
                l = mid+1;
            }
        }
        return -1;
    }

    public static void main(String[] args)
    {
        int[] a = {3,5,6,8,12,15,16,19,21};
        Arrays.sort(a);
        int key = 15;
        System.out.println(binarySearch(a, key));
    }
}
```

Bubble Sort

```
public class Sorting {  
    static void bubbleSort(int[] a) {  
        for (int i = 0; i < a.length - 1; i++) {  
            for (int j = 0; j < a.length - i - 1; j++) {  
                if (a[j] < a[j + 1]) {  
                    int temp = a[j];  
                    a[j] = a[j + 1];  
                    a[j + 1] = temp;  
                }  
            }  
        }  
  
        public static void main(String[] args) {  
            int[] a = { 3, 2, 7, 5, 9 };  
            for (int i = 0; i < a.length; i++) {  
                System.out.print(a[i] + " ");  
            }  
            System.out.println();  
            bubbleSort(a);  
            for (int i = 0; i < a.length; i++) {  
                System.out.print(a[i] + " ");  
            }  
        }  
    }  
}
```

Selection Sort

```
public class Sorting {  
    static void selectionSort(int[] a) {  
        for (int i = 0; i < a.length - 1; i++) {  
            int min_i = i;  
            for (int j = i + 1; j < a.length; j++) {  
                if (a[j] > a[min_i]) {  
                    min_i = j;  
                }  
            }  
            int temp = a[i];  
            a[i] = a[min_i];  
            a[min_i] = temp;  
        }  
    }  
  
    public static void main(String[] args) {  
        int[] a = { 3, 2, 7, 5, 9 };  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
        System.out.println();  
        selectionSort(a);  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
    }  
}
```

Insertion sort

```
public class Sorting {  
    static void insertionSort(int[] a) {  
        int j = 0;  
        for (int i = 1; i < a.length; i++) {  
            j = i - 1;  
            int key = a[i];  
            while (j >= 0 && a[j] < key) {  
                a[j + 1] = a[j];  
                j--;  
            }  
            a[j + 1] = key;  
        }  
    }  
  
    public static void main(String[] args) {  
  
        int[] a = { 3, 2, 7, 5, 9 };  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
        System.out.println();  
        insertionSort(a);  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
    }  
}
```

Introduction to Wrapper Class:

Let us first understand why wrapper class?

Java is an impure object-oriented programming language because it supports **primitive data types**.

You may be wondering; does it really matter if java is pure or impure object oriented

To understand this, you must know an important fact about primitive data types which is **Primitive data types are not treated as object in java**

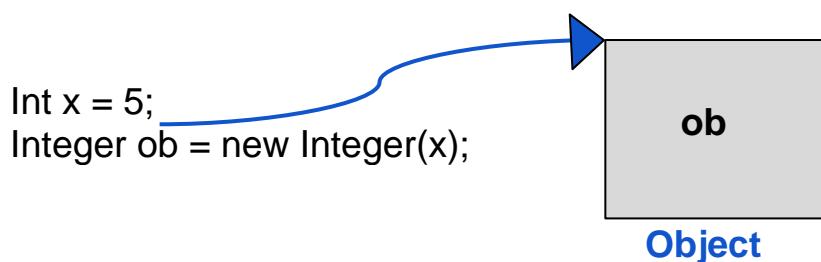
However, 100% pure object-oriented programs can be written in java using **wrapper class**

Now you understood why wrapper class lets us understand what is wrapper class.

What is Wrapper class?

The **Wrapper class in java** is a class that provides the mechanism to convert primitive data types into objects and objects into primitive data type.

Well how if you wonder, consider the below example.



In the above example **int** is a **primitive data type** and **Integer** is a **wrapper class**. Let us have a look at one more example of character data type.

If you are using a single character value, you will use the primitive char type

```
char ch = 'a'
```

There are times, however, when you need to use a char as an object - for example, as a method argument where an object is expected.

The java programming language provides a wrapper class that "wraps" the char in a character object for this purpose. An object of type character contains a single field, whose type is char. This character class also offers a number of useful class (i.e. static) methods for manipulating characters.

Character characterObj = new Character('a');

You can create a character object with character constructor.

Advantage of wrapper class: The program becomes pure object oriented

Disadvantage of wrapper class: Execution speed decreases.

Advantage of primitive data type: Faster in execution

Disadvantage of primitive data type: The program becomes impure object oriented.

STRINGS

Unlike other programming languages, Strings in java are different. Well how you wonder? Let us first understand what is a string.

C



is this a single character or multiple character?

If you can recollect, it is a single character. Now let me ask you the same question by considering one more example as shown below:

CODE

→ *is this a single character or multiple character?*

Now you will answer, **it is a sequence of characters or collection of character or series of characters.**

The different words you have used to answer the second example is only referred to as **STRINGS** in java.

You are able to differentiate between a **character** and a **string**. But computer can't tell the difference between character and string.

To resolve this **problem**, java came up with a **solution**.

Well what you under?



The solution is:



Put the characters in single quotes → ‘C’

Put the string in double quotes → “CODE”

When the computer encounters single quotes, it treats it as character. Similarly, when the computer encounters double quotes, it treats it as strings.

What is a Character?

A Character is a value enclosed within single quotes.

What is a String?

A String is a series of characters enclosed within double quotes.

Strings are treated as **objects** in java.

In java, Strings are further divided into two types:

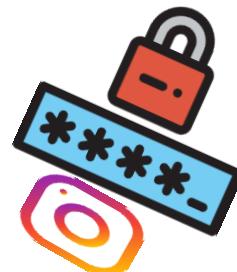
- 1) Mutable Strings**
- 2) Immutable Strings.**

As the name itself indicates, **mutable strings** are such strings which are **changeable** and **immutable strings** are such strings which are **non-changeable**.

Let us look at few real time examples to understand this:



Months in a year



Instagram Password

MUTABLE STRINGS



Gmail id



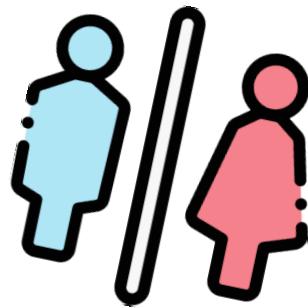
Passport Details.

If you look at the examples given above, you may notice they are all changeable and hence they fall under the category of Mutable Strings.

Let us now have a look at non-changeable strings:



Earths colour



Gender

IMMUTABLE STRINGS



Date-of-birth



Flags colour Combination

Similarly, if you look at the examples given above, you may notice they are all non-changeable and hence they fall under the category of Immutable Strings.

*Let us now learn **Immutable strings** in detail.*



Definition: *Immutable strings are such strings that are unmodifiable or unchangeable. They are Created using **String Class**.*

Different ways of creating Immutable strings:

- 1) `char name[] = { 'j' , 'a' , 'v' , 'a' }`
`String s = new String(name);`
- 2) `String s = "java";`
- 3) `String s = new String ("java");`

After getting to know different ways of creating strings, one must now learn different ways of comparing strings.

Different ways of comparing strings:

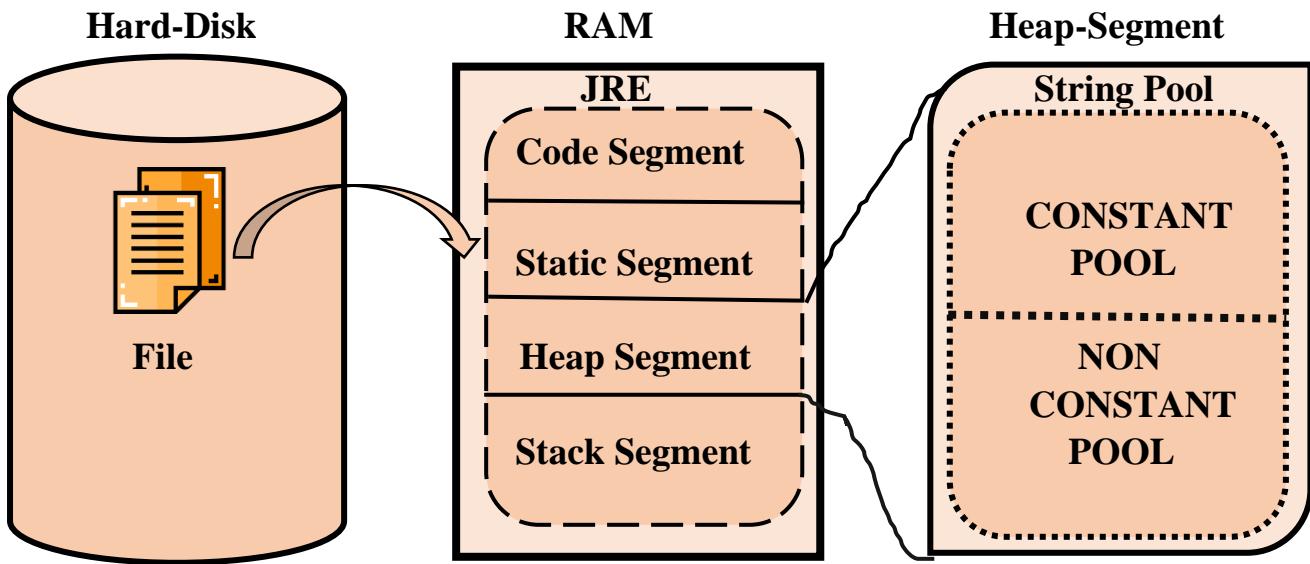
- 1) `==` → String references are compared.
- 2) `equals()` → String values/literals are compared.
- 3) `CompareTo()` → Strings are compared character by character.
- 4) `equalsIgnoreCase()` → Strings are compared by ignoring the case.

Before we start writing our first code on strings, let us understand few other concepts which one should be aware of while coding strings.

We have already discussed hard-disk, ram, and microprocessor. Now you may wonder why am I introducing them here and how is this related to strings.



Consider the diagram shown below:



We know microprocessor is the most important device on our computer and there are two other devices which play a crucial role when it comes to storing the data which are hard-disk and ram. The file is stored on hard-disk but if it has to execute it should be placed on ram. Also, we have learned, when the file is placed on ram, the entire region is not allocated for its execution instead, one region of ram is allocated for this program file to execute which is only referred to as **JRE**.

JRE is further divided into four segments:

- 1) **Code Segment**
- 2) **Static Segment**
- 3) **Heap Segment**
- 4) **Stack Segment.**

Out of all those segments, **Strings** are allocated memory on **Heap Segment**.

Heap Segment further consists of two pools:

- 1)**Constant pool**
- 2)**Non-Constant Pool.**

To understand strings from memory perspective, one must know the features of these pools which is shown below:

CONSTANT POOL

Strings that are created **without using new keyword** are allocated memory on this pool.

Duplicates are not allowed.

Non-CONSTANT POOL

Strings that are created **using new keyword** are allocated memory on this pool.

Duplicates are allowed.

The wait is over, you now have sufficient knowledge to start coding strings.



Example 1)

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "JAVA";
        String s2 = "JAVA";
        if(s1 == s2)
        {
            System.out.println("References are equal");
        }
        else
        {
            System.out.println("References are not equal");
        }
    }
}
```

Output: References are equal.

Explanation: In the above code we made use of “`==`” operator to compare two strings. We know “`==`” operator compares two strings based on the references. Both the strings are created without using new keyword and if you recollect, they will now be allocated memory on constant pool where duplicates are not allowed. Hence only one copy of string literal is created and both the references will point to the same string which means both `s1` and `s2` will have same address as only one string object is created. Thus, the output references are equal.

Example 2)

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "JAVA";
        String s2 = "JAVA";
        if(s1.equals(s2) == true)
        {
            System.out.println("String values are equal");
        }
        else
        {
            System.out.println("String values are not equal");
        }
    }
}
```

Output: String values are equal.

Explanation: In the above code we made use of “equals ()” to compare two strings. We know equals() compares two strings based on the values. Both the strings are created without using new keyword and if you recollect, they will now be allocated memory on constant pool where duplicates are not allowed. Hence only one copy of string literal is created. Since both the strings are same, we get the output as String values are equal.

Example 3)

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "JAVA";
        String s2 = new String("JAVA");
        if(s1 == s2)
        {
            System.out.println("References are equal");
        }
        else
        {
            System.out.println("References are not equal");
        }
    }
}
```

Output: References are not equal.

Explanation: In the above code we made use of “==” operator to compare two strings. We know “==” operator compares two strings based on the references. String s1 is created without using new keyword so it gets allocated memory on constant pool and String s2 is created using new keyword so it gets allocated memory on non-constant pool. So now, two different string objects are created with two different addresses. S1 and s2 are the references pointing to two string objects hence they will now consist of different addresses and thus we get the output as references are not equal.



HE UNDERSTOOD. DID YOU?

String Comparison in Java

We can compare string in java on the basis of **content** and **reference**.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

- By == operator
- By equals() method
- By compareTo() method

Let's start with understanding == operator:

The == operator compares **references not values**.

Example1

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = new String("JAVA");
        String s2 = new String("JAVA");

        if(s1==s2)
        {
            System.out.println("String references are equal.");
        }
        else
        {
            System.out.println("String references are not equal.");
        }
    }
}
```



Output: String references are not equal.

Whenever **new** keyword is used to create string, it gets created in non-constant pool. And when == operator is used to compare two strings its references are compared and not the values. And as s1 has different reference than s2, we get the output as references are not equal.

Now let's see comparison by equals() method:

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

Example2

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = new String("JAVA");
        String s2 = new String("JAVA");

        if(s1.equals(s2)==true)
        {
            System.out.println("String references are equal.");
        }
        else
        {
            System.out.println("String references are not equal.");
        }
    }
}
```

Output: String values are equal.

Example3

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "java";
        String s2 = new String("JAVA");

        if(s1.equals(s2)==true)
        {
            System.out.println("String values are equal.");
        }
        else
        {
            System.out.println("String values are not equal.");
        }
    }
}
```

Output: String values are not equal.



Example4

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "java";
        String s2 = new String("JAVA");

        if(s1.equalsIgnoreCase(s2)==true)
        {
            System.out.println("String values are equal.");
        }
        else
        {
            System.out.println("String values are not equal.");
        }
    }
}
```



Output: String values are equal.

Example5

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "JAVA";
        String s2;
        s2 = s1;
        if(s1==s2)
        {
            System.out.println("references are equal.");
        }
        else
        {
            System.out.println("references are not equal.");
        }
    }
}
```

Output: references are equal.

Here as new keyword is not used s1 gets created in constant pool. And s2 is reference which gets created in constant pool. So s1 reference is assigned to s2, which means both are referring to same value.

String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

- By + (string concatenation) operator
- By concat() method

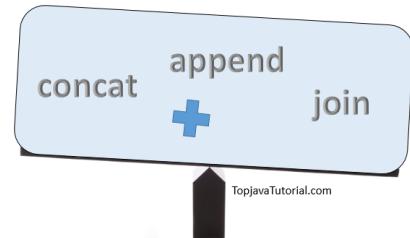
String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings.

Example6

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "JAVA";
        String s2 = "PYTHON";
        String s3 = "JAVA"+ "PYTHON";
        String s4 = "JAVA" + "PYTHON";

        if(s3==s4)
        {
            System.out.println("references are equal.");
        }
        else
        {
            System.out.println("references are not equal.");
        }
    }
}
```



Output: references are equal.

As here values are getting added and not the references. And duplication cannot happen in constant pool, therefore references are same.

Example7

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "JAVA";
        String s2 = "PYTHON";
        String s3 = s1+s2;
        String s4 = s1+s2;

        if(s3==s4)
        {
            System.out.println("references are equal.");
        }
        else
        {
            System.out.println("references are not equal.");
        }
    }
}
```

Output: references are not equal.

As here values are not getting added but references are, because s1 and s2 are not literals. And duplication is possible in non-constant pool, therefore references are not the same.

String Concatenation by concat()

The String concat() method concatenates the specified string to the end of current string.

Syntax: **public** String concat(String another)

Let's see the example of String concat() method.

Example9

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "java";
        String s2 = "python";
        String s3 = s1.concat(s2);
        System.out.println(s3);

    }
}
```

Output: javapython

Although it is important to note that Strings are immutable. Now lets see what does this mean.

Immutable String in java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

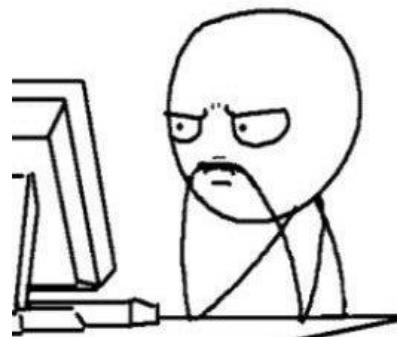
Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

Example10

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "java";
        String s2 = "python";
        System.out.println(s1);
        System.out.println(s2);
        s1.concat(s2);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output: java
python
java
python



In java whenever an immutable string is attempted to be modified then that string will not be modified, instead another string is created in non-constant pool with new reference. And as of now no object is assigned to store the new reference, which is why we cannot see the concatenated string.

So now let's see how to get the concatenated string...

```
class Demo
{
    public static void main(String[] args)
    {
        String s1 = "java";
        String s2 = "python";
        System.out.println(s1);
        System.out.println(s2);
        s1 = s1.concat(s2);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output:

```
java
python
javapython
python
```

Previously we had a reference for the concatenated string but we did not assign it to anything, whereas here we are assigning it back to s1 so that now s1 is pointing to the new concatenated string.

Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refer to one object "java". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

But 'Why'?



Going further.. If you think we forgot about the third way of comparing strings that is with the help of `compareTo()` then certainly we haven't....So let's see how to do it.

The String `compareTo()` method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- `s1 == s2` :0
- `s1 > s2` :positive value
- `s1 < s2` :negative value



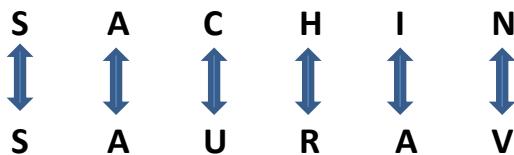
Confused??? Let's understand with example.

Case1:

S1 = "SACHIN"

S2 = "SAURAV"

Comparison happens as shown below



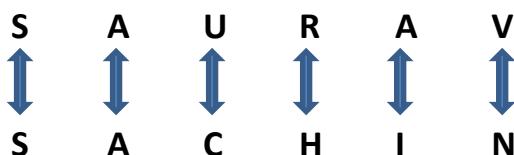
Each character is compared in **lexical order**. In the above example S,A are same in both string but when C and U are compared we know that **C comes before U** therefore **s1 is considered lesser than s2**. Which will give **negative number** when printed on screen.

Case2:

S1= "SAURAV"

S2= "SACHIN"

Comparison happens as shown below



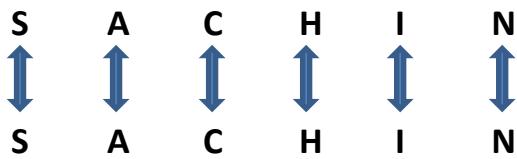
In the above example S,A are same in both string but when U and C are compared we know that **U comes after C** therefore **s1 is considered greater than s2**. Which will give **positive number** when printed on screen.

Case3:

S1 = "SACHIN"

S2 = "SACHIN"

Comparison happens as shown below



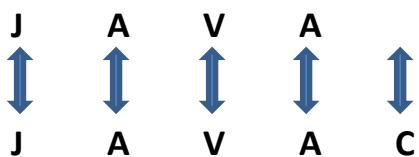
Above we see that all the **characters are same** in both the strings. Therefore when compared character by character there is **no character greater or smaller** than the other which means it will give the result as 0 when printed on the screen.

Case4:

S1= “JAVA”

S2= “JAVAC”

Let's now see what happens when the size of the string is different.



We can see that both the strings contain “JAVA” but **the second string has extra character C**. Therefore **second string is considered to be greater than the first**. And when printed on screen we will get **negative number**.

There are many such string methods, let's have a look at them...

contains() :



The **java string contains()** method searches the sequence of characters in this string. It returns *true* if sequence of char values are found in this string otherwise returns *false*.

Syntax: `string_name.contains("sequence");`

Returns: *true* if sequence of char value exists, otherwise *false*.

Throws a NullPointerException if the sequence is null.

endsWith() :

The **java string endsWith()** method checks if this string ends with given suffix. It returns true if this string ends with given suffix else returns false.

Syntax: `string_name.endsWith("sequence or character");`

Returns: true or false.

indexOf() :

The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

There are 4 types of indexOf method in java. The signature of indexOf methods are given below:

No.	Method	Description
1	<code>int indexOf(int ch)</code>	Returns index position for the given char value
2	<code>int indexOf(int ch,int fromIndex)</code>	Returns index position for the given char value and from index
3	<code>int indexOf(String substring)</code>	Returns index position for the given substring
4	<code>int indexOf(String substring, int fromIndex)</code>	Returns index position for the given substring and from index

Parameters:

- **ch:** char value i.e. a single character e.g. 'a'
- **fromIndex:** index position from where index of the char value or substring is returned
- **substring:** substring to be searched in this string

Returns: index of the string

charAt() :

The **java string charAt()** method returns a *char value at the given index number.*

The index number starts from 0 and goes to n-1, where n is length of the string. It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.

Returns : char value

startsWith() :

The **java string startsWith()** method checks if this string starts with given prefix. It returns true if this string starts with given prefix else returns false.

Syntax :

```
public boolean startsWith(String prefix)  
public boolean startsWith(String prefix, int offset)
```

parameters : prefix : Sequence of character

Returns: true or false

substring() :

The **java string substring()** method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.

There are two types of substring methods in java string.

Syntax:

1. **public String substring(int startIndex)**
2. **public String substring(int startIndex, int endIndex)**

parameters:

- **startIndex** : starting index is inclusive
- **endIndex** : ending index is exclusive

Returns: specified string

Throws: StringIndexOutOfBoundsException if start index is negative value or end index is lower than starting index.

toLowerCase() :

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

Syntax:

- **public String toLowerCase()**
- **public String toLowerCase(Locale locale)**

The second method variant of toLowerCase(), converts all the characters into lowercase using the rules of given Locale.

Returns: string in lowercase

toUpperCase() :

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

Syntax:

- **public String toUpperCase()**
- **public String toUpperCase(Locale locale)**

The second method variant of toUpperCase(), converts all the characters into uppercase using the rules of given Locale.

Returns: string in uppercase

There are definitely many other methods which you will be exploring once you start practicing.



Mutable String

Therefore mutable strings are those strings whose content can be changed without creating a new object. **StringBuffer** and **StringBuilder** are mutable versions of **String** in java.

StringBuffer Constructors:

- **StringBuffer():** This creates an empty StringBuffer with a default capacity of 16 characters.
- **StringBuffer(int capacity):** This creates an empty StringBuffer with a specified capacity.
- **StringBuffer(CharSequence charseq):** This creates a StringBuffer containing the same characters as in the specified character sequence.
- **StringBuffer(String str):** Creates a StringBuffer corresponding to specific string.

```
public class Demo{  
  
    public static void main(String[] args) {  
        StringBuffer st = new StringBuffer(); // Creating  
object of StringBuffer  
        System.out.println(st.capacity()); // Initial default  
capacity is 16  
    }  
}
```

Output:

16

In the above code StringBuffer object is created by default 16 memory is allocated.

Let's see how to append strings to StringBuffer.

```
public class Demo{  
  
    public static void main(String[] args) {  
        StringBuffer st = new StringBuffer();  
        System.out.println(st.capacity());  
        st.append("Java"); // Java is added to string Buffer  
        st.append("JavaScript"); // JavaScript is added to  
        string Buffer  
        System.out.println(st);  
    }  
}
```

Output:

```
16  
JavaJavaScript
```

Let's see if you try to append the string more than the default capacity

```
public class Demo{  
  
    public static void main(String[] args) {  
        StringBuffer st = new StringBuffer();  
        System.out.println(st.capacity());  
        st.append("Java");  
        st.append("JavaScript");  
        System.out.println(st);  
        st.append("James Gosling");  
        System.out.println(st.capacity()); //once the capacity  
        is full size will be increase by (capacity*2)+2  
    }  
}
```

Output:

16
JavaJavaScript
34

If the size is larger than the size of characters then you can reduce the storage using `trimToSize()`.

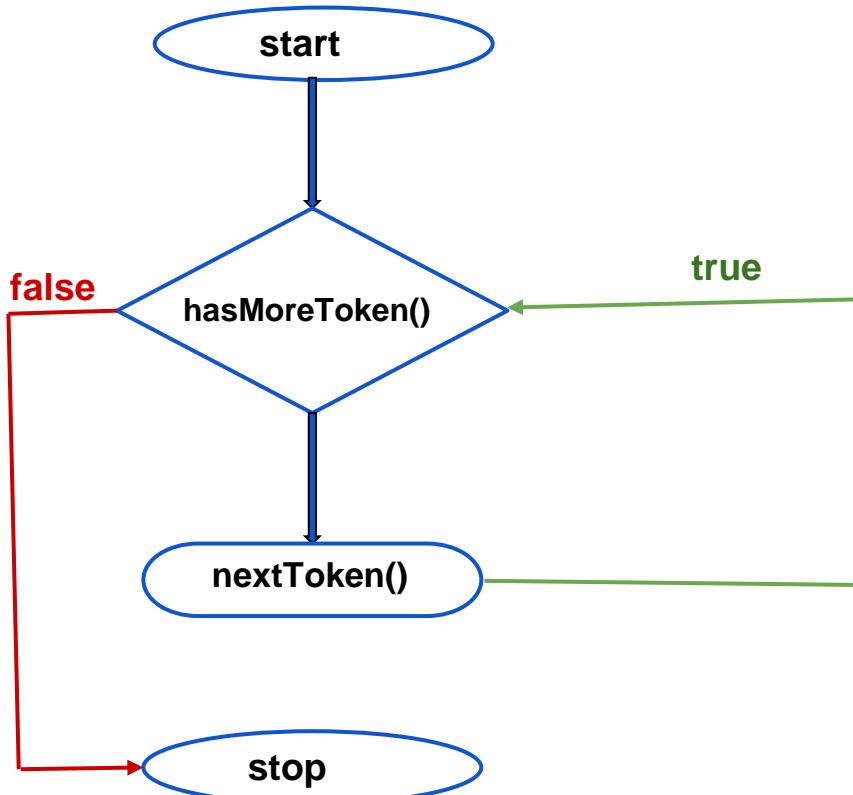
```
public class Demo{  
  
    public static void main(String[] args) {  
        StringBuffer st = new StringBuffer();  
        System.out.println(st.capacity());  
        st.append("Java");  
        st.append("JavaScript");  
        System.out.println(st);  
        st.append("James Gosling");  
        System.out.println(st.capacity());  
        st.trimToSize();  
        System.out.println(st.capacity());  
  
    }  
}
```

Output:

16
JavaJavaScript
34
27

String Tokenizer:

StringTokenizer class in Java is used to break a string into tokens.



```

import java.util.StringTokenizer;

public class Demo{
    public static void main(String[] args) {
        String s = "JAVA PYTHON SQL AI";
        StringTokenizer st = new StringTokenizer(s);

        while(st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
  
```

Output:

JAVA
PYTHON
SQL
AI

Strings

1. What is a String?

String is a series of characters, which is enclosed within double quotes and is treated as an object.

2. Is String an object in Java?

Yes

3. Can we access individual character in String?

NO

4. Does a Java String end with a null character?

No

5. What are the types of String in Java?

Mutable, immutable

6. Who de-allocates memory of a string?

Garbage-collector

7. What are the two ways of creating immutable strings?

String s="Rama";

String s=new String("Rama");

8. Why are immutable strings required?

In real world immutable String data exists: dob, gender, name, father name ,mother name .

9. How are mutable strings created in Java?

StringBuffer and StringBuilder

10. Why are mutable strings required in Java?

Real life has mutable string data : Address, qualification, password

11. Where are immutable strings created in Java?

heap Segment

12. What is a constant pool?

Is a region of the heap segment on the ram where memory is allocated for such strings that are created without using the new operator. Duplicates are not permitted in this region.

13. What is non constant pool?

Is a region of the heap segment where memory is allocated for such strings which are created using the new operator or using expressions. Duplicates are permitted in this region.

14. What is the role of equals() in Java?

It compares two strings

15. What is the role of == operator in Java?

It compares two references

16. What is the role = operator in Java?

Assignment operators are used in Java to assign values to variables. For example, int age; age = 5; The **assignment operator** assigns the value on its right to the variable on its left, and hence it is right to left associative.

17. Name some of the commonly used built methods of String class?

startsWith(), endsWith(), toUpperCase(), toLowerCase(), contains(), etc.

18. Mention the difference between C Strings and Java Strings.

C-String	Java-String
null terminated,	not null terminated
not treated as an object	treated as an object
Memory is not allocated on the heap seg	is allocated in the heap segment
Memory is not deallocated by the garbage collector	it is deallocated by the garbage collector

individual characters in the string can be accessed	individual characters in the string cannot be accessed by the programmer's
not categorization as immutable and mutable	Categorized as immutable and mutable.

19. How do you compare two Strings in Java?

`equals()`, `compareTo()`, `==`, `equalsIgnoreCase()`

20. What is a String pool in Java?

Where memory is allocated for strings, it is divided into 2 regions namely **constant pool** and **non-constant pool**.

21. Can we use String in switch case?

Upto jdk 1.7, only integers were allowed. From jdk 1.8 onwards even Strings are permitted in switch case.(Ex.Refer class notes.)

22. Why is String class considered immutable?

Strings which are created using “String” class cannot be altered.Hence String class is considered as immutable.

23. How is it possible for two string objects with identical values not to be equal under == operator?

- i) If one String is in constant pool and the other String is in non-constant pool
- ii) If both Strings are present in non constant pool.

24. Explain the difference between the following statements?

`String s="Welcome to ABC";`

The above string would be created in constant pool region.

`String s= new String("Welcome to ABC");`

The above String would be created in the non-constant pool region.

25. How to concatenate two different Strings?

Strings can be concatenated in following three ways

- i) Using `concat()`
- ii) Using `append()`
- iii) “+” operator.

26. What is the output of the following program?

```

class Test
{
    public static void main(String[] args)
    {
        String s="Hello";
        String s1=new String("Hello");
        System.out.println("s==s1 "+(s==s1));
        System.out.println("s.equals(s1)"+(s.equals(s1)));
    }
}
Ans: s==s1 false
      s.equals(s1) true
  
```

27. Is null a keyword in java?

No. Null is not a keyword. Rather it is a literal with special meaning. Null is the value of a reference variable. However we cannot use the word “null” to create a variable.

Eg. String s = null; valid statement.
 String null; invalid statement.

28. What happens when you add a char value to a String?

The char type data would be converted to a String and would be concatenated with the existing String.

Eg:

```

System.out.println("Hello world!" +A);
System.out.println("Hello world!" +999);
System.out.println("Hello world!" +999.9);
System.out.println("Hello world!" +999.9f);
System.out.println("Hello world!" +true);
  
```

Output:

Hello world! A

Hello world! 999

Hello world! 999.9

Hello world! 999.9

Hello world! true

Note:

System.out.println can only print String. If we try to concatenate character integer, float, boolean etc., then System.out.println automatically converts those values to strings using `toString()` and `valueOf()` methods.

```
System.out.println("Hello world!" +999);
```

System.out.println("Hello world!"
+999.toString());

would be carried out

internally.

```
System.out.println("Hello world!" +999.valueOf());
```

```
System.out.println("Hello world!" +"999.9");
```

```
System.out.println("Hello world! 999");
```

Output:

Hello world! 999.

Note:

System.out.println cannot print any other type of data. If some other type of data is given to System.out.println, then immediately `toString()` method would be called automatically. This inturn calls `valueOf()` method.

The `valueOf()` method gives string equivalent of the new data. It is this string equivalent that gets printed.

Eg:

```
int a =999;
```

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

```
System.out.println(a.toString());
```

```
System.out.println(a.valueOf());
```

```
System.out.println("999");
```

Output:

999(which is actually a string).

The char, int, double, float, Boolean type data would be converted with the existing string and would be printed.

29. What happens when you add a int value to a String?

Refer Q.28.

30. What happens when you add a double value to a String?

Refer to Q.28.

31. What happens when you add a float value to a String?

Refer to Q.28.

32. What happens when you add a Boolean value to a String?

Refer to Q.28.

33. Can we directly print objects?

No. Instead of the object value getting printed , Object address gets printed as shown below

```
class Dog
{
    String name;
    String breed;
    int cost;
}
```

```
class Launch
{
    public static void main(String args[])
    {
        Dog d=new Dog();
        d.name="Labo";
        d.breed="BullDog";
        d.cost=9000;
        System.out.println(d);//Directly printing the object
    }
}
output:Dog@ 15db9742
```

34. Can we directly print primitive data types?

Yes.

Eg: int a=99;

```
System.out.println(a);
aOutput:99
```

35. Does ArrayIndexOutOfBoundsException occur in case of immutable Strings? Why?

No. This problem does not occur because individual characters cannot be accessed in Java Strings.

36. Does ArrayIndexOutOfBoundsException occur in case of mutable Strings? Why?

No. This problem does not occur because individual characters cannot be accessed in Java Strings.

37. Name a few Immutable classes?

Immutable class means that once the String **object** is created its value cannot be altered. In Java, all the wrapper classes such as Byte, Short, Integer, Long, Char, Float, Double, Boolean and String class are immutable.

38. Is the String class mutable or immutable?

Immutable

39. In which package is the String class present?

java.lang package

40. What is a String literal?

Any value which is enclosed within double quotes is called as string literal.

41. What are the four ways of creating a String object?

- 1) String s="rama";
- 2) String s=new String("rama");
- 3) char name[]={'r','a','m','a'};
String s=new String(name);
- 4) char[] name={'r','a','m','a'};
String s= new String(name);

42. What are the two ways in which Strings can be concatenated?

Strings can be concatenated in following three ways

- i) Using concat()
- ii) Using append()
- iii) "+" operator.

43. In how many ways can String comparison be performed?

There are four ways to compare two Strings in java.

- i) Using equals() which compares two Strings.
- ii) Using equalsIgnoreCase() which compares two strings by ignoring the case sensitivity.
- iii) Using compareTo() which compares two strings character by character.
- iv) Using == which compares references of two strings.

44. Give the memory map for String s1="Rama";

Stack segment

Heap Segment

2000

CP

2000

R A M A

S1

NCP

45. What is the role of charAt()?

It returns the character at specific index position.

46. What is the role of equals()?

It compares two strings by considering case sensitivity.

47. What is the role of equalsIgnoreCase()?

It compares two strings by ignoring case sensitivity.

48. What is the role of length()?

It is used to return the length of the string.

49. What is the role of substring()?

substring() is used for getting a substring of a particular String.
(example: refer class notes)

50. Explain the two versions of substring()?

String s = “RajaRamMohanRoy”;

There are two variants of substring() method:

1) String substring(int beginIndex):

s.substring(7);

The above statement would display a substring from 7th character onwards till the end of the String. (MohanRoy)

2) String substring(int beginIndex, int endIndex):

s.substring(7,12);

The above statement would display a substring from 7th character to 11th character. (Mohan)

51. What is the role of toLowerCase()?

It returns a string by converting it into lower case.

52. What is the role of toUpperCase()?

It returns a string by converting it into upper case.

53. What is the role of valueOf()?

valueOf() method converts different types of values into String. Using valueOf() method we can convert int to string, char to string, float to string, Boolean to string etc., It returns string representation of the given value.

54. How does valueOf() behave on objects?

It returns the string representation of an object.

55. What is the role of toString()?

toString() is used to convert a non-string object to string object.

56. What is the role of contains()?

It is used to verify that if a string contains a particular substring or not.
Ex: refer class notes.

57. What is the role of concat()?

The concat() method concatenates is used to concatenate or combine

two strings.

58. What is the role of isEmpty()?

It is used to check whether the string is empty or not.

59. What is the role of indexOf()?

It returns the index of a specified character within a string

60. Which classes are used to create mutable Strings in java?

StringBuffer and StringBuilder builtin classes would be used to create mutable strings in java.

61. What is the difference between StringBuffer and StringBuilder class?

(Refer class notes)

62. What are the three constructors of StringBuffer class?

StringBuffer s=new StringBuffer("RAMA");

StringBuffer s=new StringBuffer();

StringBuffer s=new StringBuffer(10);

63. What is the default buffer size of the StringBuffer class?

The initial capacity of the **StringBuffer class** is 16.

64. What is the role of append()?

It is used to concatenate or combine two mutable strings.

65. What is the role of capacity()?

It returns the capacity of StringBuffer or StringBuilder.

66. What is the role of ensureCapacity()?

It is used to change the capacity of StringBuffer or StringBuilder.

67. What are the three constructors of StringBuilder?

StringBuilder s=new StringBuilder("RAMA");

StringBuilder s=new StringBuilder();

StringBuilder s=new StringBuilder(10);

68. In which class is the toString() present?

Object class

69. What is the role of intern()?

It is used to create a copy of the string present in non-constant pool into the constant pool region.

70. What is the role of StringTokenizer class?

It is used to chop a large string at a specified character in order to provide tokens.

Eg:

```
import java.util.*;
class Abc
{
    public static void main(String args[])
    {
        StringTokenizer st=new StringTokenizer("ABC for technology
training");
        while(st.hasMoreTokens()==true)
        {
            System.out.println(st.nextToken());
        }
    }
}
```

Output: ABC

for
technology
training

71. In which package is the StringTokenizer class present?

java.util

72. When was the StringBuilder class introduced in Java?

jdk 1.5

73. Can we explicitly put a string object which is in the non-constant pool into the constant pool?

Yes, by using intern(). But for the reverse operation there is no method.

74. What is an immutable object?

An object whose state can never be changed is called an immutable object.

75. What would happen if an immutable object is attempted for modification?

A copy of the object would be taken and would be modified whereas the original object would still remain immutable.

76. Which exception is generated if we try to mutate an immutable object?

IllegalObjectStateException

77. Where does the String pool exist?

String pool exists on the perm space on the heap segment.

78. Is String class final in Java?

Yes.

79. Why is char array preferred to store password in Java than a String?

Passwords are normally frequently modified. Hence it must be created either using **StringBuffer** or **StringBuilder** class or **char** array can be used.

80. Why is the String class final in Java?

Because String class is immutable, it has been made as final class so that inheritance of the String class can be prevented.

81. Is the String class thread safe?

Yes

82. Is StringBuffer class thread safe?

Yes

83. Is StringBuilder class thread safe?

No

84. What is the difference between declaring a String and defining a String?

Declaration

String s;

Defining

String s = "Rama";

toUpperCase()

```
import java.util.Scanner;
public class Demo{

    static String toUpperCase(String s) {
        String t = "";

        for(int i=0; i < s.length(); i++) {
            char c = s.charAt(i);
            if(c>='a' && c<='z') {
                t = t + (char)(c-32);
            }else {
                t = t + c;
            }
        }
        return t;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();

        System.out.println(toUpperCase(s));
    }
}
```

toLowerCase()

```
import java.util.Scanner;
public class Demo{

    static String toLowerCase(String s) {
        String t = "";

        for(int i=0; i < s.length(); i++) {
            char c = s.charAt(i);
            if(c>='A' && c<='Z') {
                t = t + (char)(c+32);
            }else {
                t = t + c;
            }
        }
        return t;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();

        System.out.println(toLowerCase(s));
    }
}
```

indexOf()

```
import java.util.Scanner;
public class Demo{

    static int indexOf(String s, String c) {
        char key = c.charAt(0);

        for(int i=0; i<s.length(); i++) {
            if(s.charAt(i) == key) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        String c = scan.nextLine();

        System.out.println(indexOf(s,c));
    }
}
```

lastIndexOf()

```
import java.util.Scanner;
public class Demo{

    static int lastIndexOf(String s, String c) {
        char key = c.charAt(0);

        for(int i=s.length()-1; i>=0; i--) {
            if(s.charAt(i) == key) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        String c = scan.nextLine();

        System.out.println(lastIndexOf(s,c));
    }
}
```

trim()

```
import java.util.Scanner;
public class Demo{

    static String trim(String s) {
        String t = "";
        int si=0,ei=0;

        for (int i = 0; i < s.length(); i++) {
            if(s.charAt(i) != ' ') {
                si=i;
                break;
            }
        }

        for (int i = s.length()-1; i>=0; i--) {
            if(s.charAt(i) != ' ') {
                ei=i;
                break;
            }
        }

        for(int i=si; i<=ei; i++) {
            t=t+s.charAt(i);
        }

        return t;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        String c = scan.nextLine();

        System.out.println(trim(s));
    }
}
```

toCharArray()

```
import java.util.Scanner;
public class Demo{

    static char[] toCharArray(String s) {
        char[] c = new char[s.length()];

        for(int i=0; i<s.length(); i++) {
            c[i] = s.charAt(i);
        }

        return c;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        System.out.println(toCharArray(s));

    }
}
```

equals()

```
import java.util.Scanner;
public class Demo{

    static boolean equals(String s1, String s2) {
        for(int i=0; i<s1.length();i++) {
            if(s1.charAt(i) != s2.charAt(i)) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s1 = scan.nextLine();
        String s2 = scan.nextLine();
        System.out.println>equals(s1, s2));
    }
}
```

startsWith()

```
import java.util.Scanner;
public class Demo{

    static boolean startsWith(String s, String t) {
        int count = 0;
        for(int i=0; i<t.length();i++) {
            if(s.charAt(i) == t.charAt(i)) {
                count++;
            }
        }
        if(t.length() == count) {
            return true;
        }else {
            return false;
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        String t = scan.nextLine();
        System.out.println(startsWith(s, t));

    }
}
```

endsWith()

```
import java.util.Scanner;
public class Demo{

    static boolean endsWith(String s, String t) {
        int count = 0;
        for(int i=s.length()-t.length(); i<t.length();i++) {
            if(s.charAt(i) == t.charAt(count)) {
                count++;
            }
        }
        if(t.length() == count) {
            return true;
        }else {
            return false;
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        String t = scan.nextLine();
        System.out.println(endsWith(s, t));
    }
}
```

contains()

```
import java.util.Scanner;
public class Demo{

    static boolean endsWith(String s, String t) {
        int count = 0;
        for(int i=0; i<s.length() && count < t.length();i++) {
            if(s.charAt(i) == t.charAt(count)) {
                count++;
            }
            else {
                count = 0;
            }
        }
        if(t.length() == count) {
            return true;
        }else {
            return false;
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        String t = scan.nextLine();
        System.out.println(endsWith(s, t));

    }
}
```

replace()

```
import java.util.Scanner;
public class Demo{

    static char[] toCharArray(String s) {
        char[] c = new char[s.length()];

        for(int i=0; i<s.length(); i++) {
            c[i] = s.charAt(i);
        }

        return c;
    }

    static String replace(String str, char old_char, char new_char) {
        char[] s = toCharArray(str);
        for(int i=0; i<s.length; i++) {
            if(s[i] == old_char) {
                s[i] = new_char;
            }
        }

        return new String(s);
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        char old_char = scan.next().charAt(0);
        char new_char = scan.next().charAt(0);
        System.out.println(replace(s, old_char, new_char));

    }
}
```

subString(int startIndex)

```
import java.util.Scanner;
public class Demo{

    static String subString(String s, int startIndex) {
        String t = "";
        for(int i=startIndex; i <s.length(); i++) {
            t = t+s.charAt(i);
        }
        return t;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();
        int n = scan.nextInt();
        System.out.println(subString(s, n));

    }
}
```

subString(int startIndex, int endIndex)

```
import java.util.Scanner;
public class Demo{

    static String subString(String s, int startIndex, int endIndex) {
        String t = "";
        for(int i=startIndex; i < endIndex; i++) {
            t = t+s.charAt(i);
        }
        return t;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String st = scan.nextLine();
        int s = scan.nextInt();
        int e = scan.nextInt();
        System.out.println(subString(st, s, e));

    }
}
```

Count the number of vowels in a given string

```
import java.util.Scanner;

public class Demo {
    static int countVowels(String s) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c
== 'u' || c == 'A' || c == 'E' || c == 'O' || c == 'I'
                || c == 'U') {
                count++;
            }
        }
        return count;
    }

    public static void main(String[] args) {
        String s = "abcbea";
        System.out.println(countVowels(s));
    }
}
```

Character count

```
public class Demo {  
    static int characterCount(String s, String t) {  
        int count = 0;  
        char key = t.charAt(0);  
        for (int i = 0; i < s.length(); i++) {  
            if (s.charAt(i) == key)  
                count++;  
        }  
        return count;  
    }  
  
    public static void main(String[] args) {  
        String s = "How are you";  
        String t ="o";  
        System.out.println(characterCount(s,t));  
    }  
}
```

String Reverse

```
import java.util.Scanner;

public class Demo2 {
    static String reverse(String s) {
        String t = "";
        for (int i = s.length() - 1; i >= 0; i--) {
            t = t + s.charAt(i);
        }
        return t;
    }

    public static void main(String[] args) {
        String s = "TAPACADEMY";
        System.out.println(reverse(s));
    }
}
```

Palindrome

```
import java.util.Scanner;

public class Demo2 {
    static boolean isPalindrome(String s) {
        int i = 0;
        int j = s.length() - 1;
        while (i <= j) {
            if (s.charAt(i) != s.charAt(j)) {
                return false;
            }
            i++;
            j--;
        }
        return true;
    }

    public static void main(String[] args) {
        String s = "level";
        System.out.println(isPalindrome(s));
    }
}
```

Space count

```
public class Demo4 {  
    static int spaceCount(String s) {  
        int count = 0;  
        for (int i = 0; i < s.length(); i++) {  
            if (s.charAt(i) == ' ') {  
                count++;  
            }  
        }  
        return count;  
    }  
  
    public static void main(String[] args) {  
        String s = "How are you";  
        spaceCount(s);  
    }  
}
```

Word Count

```
public class Demo4 {  
    static int wordCount(String s) {  
        int count = 0;  
        for (int i = 0; i < s.length(); i++) {  
            if (s.charAt(i) == ' ') {  
                count++;  
            }  
        }  
        return count + 1;  
    }  
  
    public static void main(String[] args) {  
        String s = "How are you";  
        wordCount(s);  
    }  
}
```

Print All Substring

```
import java.util.Arrays;

public class Demo4 {

    static void printAllSubstring(String st) {
        int n = st.length();
        for (int len = 1; len < n; len++) {
            for (int s = 0; s <= (n - len); s++) {
                for (int e = s; e <= (s + len) - 1; e++) {
                    System.out.println(st.charAt(e));
                }
                System.out.println();
            }
        }
    }

    public static void main(String[] args) {
        String s = "tapacademy";
        printAllSubstring(s);
    }
}
```

Print substring of length 3

```
import java.util.Arrays;

public class Demo4 {

    static void printAllSubstring(String st, int k) {
        int n = st.length();
        for (int s = 0; s <= (n - k); s++) {
            for (int e = s; e <= (s + k) - 1; e++) {
                System.out.println(st.charAt(e));
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        String s = "tapacademy";
        int k = 3;
        printAllSubstring(s,k);
    }
}
```

Print Longest non repeating substring

```
import java.util.Arrays;

public class Demo4 {
    static boolean isPalindrome(String s) {
        int i = 0;
        int j = s.length() - 1;
        while (i <= j) {
            if (s.charAt(i) != s.charAt(j)) {
                return false;
            }
            i++;
            j--;
        }
        return true;
    }

    static String printLongestPalinSubstring(String st) {
        int n = st.length();
        String res = "";
        for (int len = 1; len < n; len++) {
            for (int s = 0; s <= (n - len); s++) {
                String temp = "";
                for (int e = s; e <= (s + len) - 1; e++) {
                    temp = temp + st.charAt(e);
                }
                if (isPalindrome(temp)) {
                    if (temp.length() > res.length())
                        res = temp;
                }
            }
        }
        return res;
    }

    public static void main(String[] args) {
        String s = "tapacademy";
        System.out.println(printLongestPalinSubstring(s));
    }
}
```

Repeated Strings

<https://www.hackerrank.com/challenges/repeated-string/problem>

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

class Result {
    public static long repeatedString(String s, long n) {
        int countA = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == 'a') {
                countA++;
            }
        }
        long count1 = n / s.length() * countA;
        long count2 = 0;
        for (int i = 0; i < n % s.length(); i++) {
            if (s.charAt(i) == 'a') {
                count2++;
            }
        }
        return count1 + count2;
    }
}

public class Solution {
    public static void main(String[] args) throws IOException {
        BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
        BufferedWriter bufferedWriter = new BufferedWriter(new
FileWriter(System.getenv("OUTPUT_PATH")));
        String s = bufferedReader.readLine();
        long n =
    }
}
```

```
Long.parseLong(bufferedReader.readLine().trim());
    long result = Result.repeatedString(s, n);
    bufferedWriter.write(String.valueOf(result));
    bufferedWriter.newLine();
    bufferedReader.close();
    bufferedWriter.close();
}
}
```

Game of thrones

<https://www.hackerrank.com/challenges/game-of-thrones/problem>

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

class Result {
    public static String gameOfThrones(String s) {
        char[] ar = s.toCharArray();
        Arrays.sort(ar);
        int i = 0;
        int errorCount = 0;
        while (i < ar.length) {
            if (i < ar.length - 1 && ar[i] == ar[i + 1]) {
                i = i + 2;
            } else {
                i++;
                errorCount++;
            }
        }
        if (errorCount <= 1) {
            return "YES";
        } else {
            return "NO";
        }
    }
}

public class Solution {
    public static void main(String[] args) throws IOException {
        BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
        BufferedWriter bufferedWriter = new BufferedWriter(new
FileWriter(System.getenv("OUTPUT_PATH")));
        String s = bufferedReader.readLine();
        String result = Result.gameOfThrones(s);
        bufferedWriter.write(result);
        bufferedWriter.newLine();
    }
}
```

```
        bufferedReader.close();
        bufferedWriter.close();
    }
}
```

Anagram

```
import java.util.Arrays;

public class Demo4 {
    static boolean isAnagram(String s1, String s2) {
        String temp = "";
        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != ' ') {
                temp = temp + s1.charAt(i);
            }
        }
        s1 = temp;
        temp = "";
        for (int i = 0; i < s2.length(); i++) {
            if (s2.charAt(i) != ' ') {
                temp = temp + s2.charAt(i);
            }
        }
        s2 = temp;
        s1 = s1.toLowerCase();
        s2 = s2.toLowerCase();
        char[] arr1 = s1.toCharArray();
        char[] arr2 = s2.toCharArray();
        Arrays.sort(arr1);
        Arrays.sort(arr2);
        s1 = new String(arr1);
        s2 = new String(arr2);
        return s1.equals(s2);
    }

    public static void main(String[] args) {
        String s = "tapacademy";
        System.out.println(isAnagram(s));
    }
}
```

Pangram

```
import java.util.Arrays;

public class Demo4 {
    static boolean isPangram(String s) {
        String t = "abcdefghijklmnopqrstuvwxyz";
        s = s.toLowerCase();
        int count = 0;
        for (int i = 0; i < t.length(); i++) {
            if (s.indexOf(t.charAt(i)) >= 0)
                count++;
            else
                break;
        }
        if (count == 26)
            return true;
        else
            return false;
    }

    public static void main(String[] args) {
        String s = "tapacademy";
        System.out.println(isPangram(s));
    }
}
```

Invalid Bracket

```
import java.util.Scanner;

public class Demo4 {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.nextLine();

        int bracket=0, count=0;

        for(int i=0; i<s.length(); i++) {
            char c = s.charAt(i);
            if(c == '(') {
                bracket++;
            }
            else {
                if(bracket <= 0) {
                    count++;
                }else {
                    bracket--;
                }
            }
        }

        System.out.println(count + bracket);
    }
}
```

Mars Exploration:

```
import java.util.Scanner;

public class Demo4 {

    static int marsExploration(String s) {
        int count = 0;
        for(int i=0; i<s.length(); i=i+3) {
            if(s.charAt(i) != 'S') {
                count++;
            }
            if(s.charAt(i) != 'O') {
                count++;
            }
            if(s.charAt(i) != 'S') {
                count++;
            }
        }
        return count;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.next();

        System.out.println(marsExploration(s));
    }
}
```

Camel Case

```
import java.util.Scanner;

public class Demo4 {

    static int camelCase(String s) {
        int count = 1;
        for(int i=0; i<s.length(); i=i+3) {
            if(s.charAt(i) >= 'A'
                && s.charAt(i) <= 'Z') {
                count++;
            }
        }
        return count;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.next();

        System.out.println(camelCase(s));
    }
}
```

Strong Password

```
import java.util.Scanner;

public class Demo4 {

    static int minimumNumber(int n, String s) {
        int digit=1, lc=1, uc=1, sc=1;

        for(int i=0; i<s.length(); i++) {
            char c = s.charAt(i);
            if(c>='0' && c<='9') {
                digit = 0;
            }
            else if(c>='a' && c<='z') {
                lc = 0;
            }
            else if(c>='A' && c<='Z') {
                uc = 0;
            }else {
                sc=0;
            }
        }

        if((digit+lc+uc+sc) > 6-n) {
            return (digit+lc+uc+sc);
        }else {
            return 6-n;
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        String s = scan.next();

        System.out.println(minimumNumber(n, s));
    }
}
```

Method overloading

Method Overloading is a feature that allows a class to have **more than one method having the same name**, if their **argument lists are different**. When I say argument list it means the parameters that a method has: For example the argument list of a method **add(int a, int b)** having two parameters is different from the argument list of the method **add(int a, int b, int c)** having three parameters.

There are three ways to overload a method:

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

- area(int, int)
- area(int, int, int)



2. Data type of parameters.

For example:

- area(int, float)
- area(int, int)

3. Sequence of Data type of parameters.

For example:

- area(int, float)
- area(float, int)

Invalid case of method overloading:

When I say argument list, I am not talking about **return type of the method**, for example if two methods have **same name, same parameters** and have **different return type**, then this is **not a valid method overloading** example. This will throw **compilation error**.

For example: **int area(int, int)**

float area(int, int)

Method overloading is an example of **Static Polymorphism**. We will look into polymorphism in deep in further classes.

- Static Polymorphism is also known as **compile time binding or early binding**.
- Static binding happens at **compile time**. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Now let's see through code..

We are trying to find the area of rectangle with all possible inputs. And just to demonstrate the different number of parameters rule, 3rd parameter is included.



www.clipartall.com - 1246277

```
import java.util.Scanner;

class Printer{
    void print(int a) {
        System.out.println(a);
    }
    void print(float a) {
        System.out.println(a);
    }
    void print(char a) {
        System.out.println(a);
    }
    void print(boolean a) {
        System.out.println(a);
    }
    void print(short a) {
        System.out.println(a);
    }
    void print(int a, int b) {
        System.out.println(a + " " + b);
    }
}
```

```
void print(float a, float b) {  
    System.out.println(a + " " + b);  
}  
void print(double a, double b) {  
    System.out.println(a + " " + b);  
}  
void print(int a, float b) {  
    System.out.println(a + " " + b);  
}  
void print(int a, int b, int c) {  
    System.out.println(a + " " + b + " " + c);  
}  
void print(int a, char b, float c) {  
    System.out.println(a + " " + b + " " + c);  
}  
}  
public class Demo{  
  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
        int c = 30;  
        char ch = 'a';  
        float f = 45.5f;  
        double d = 55.5;  
        Printer p = new Printer();  
        p.print(ch);  
        p.print(a,a);  
        p.print(a,a,a);  
  
    }  
}
```

Output:

```
a  
10 10  
10 10 10
```

Method overloading and Type Promotion

When a **data type of smaller size** is promoted to the **data type of bigger size** than this is called **type promotion**, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

What does it has to do with method overloading?

Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.

Let's take an example to see what I am talking here:



```
class Rectangle
{
    float area(int l,float b) //closest match
    {
        return l*b;
    }
    double area(double l,double b)
    {
        return l*b;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Rectangle ref = new Rectangle();
        int a=10,b=20;
        System.out.println(ref.area(a,b));
    }
}
```

Output: 200.0

Type Promotion table:

The **data type** on the **left side** can be **promoted** to any of the data types present on the **right side** of it.

```
byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double
```



```
import java.util.Scanner;
class Printer{
    void print(int a, float b) {
        System.out.println(a + " " + b);
    }
    void print(float a, int b) {
        System.out.println(a + " " + b);
    }
}
public class Demo{
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        Printer p = new Printer();
        p.print(a,b);
    }
}
```

Here `print()` is called by passing two integer values. There are two methods `print()` which accept integer and floating point numbers, one more method with floating point and integer numbers. Now java is confused which method to execute as in both methods one value is matching one have type promotion so **ambiguous error** will occur

Setters and Getters in Java

In Java, getter and setter are two **conventional methods** that are used for **retrieving and updating value of a variable**.

As we **cannot access** the **private members** of the class **directly outside** the class. But we can give **protected access** by providing setters and getters.

So, a **setter** is a method that **updates value of a variable**. And a **getter** is a method that **reads value of a variable**.

Getter and setter are also known as **accessor** and **mutator** in Java.

Why setter and getter?

So far, setter and getter methods protect a variable's value from unexpected changes by outside world - the caller code.

When a variable is hidden

by **private** modifier and can be accessed only through getter and setter, it is **encapsulated**. Encapsulation is one of the **fundamental principles** in object-oriented programming (OOP), thus implementing **getter and setter** is **one of ways to enforce encapsulation** in program's code.

But 'Why?'



Some frameworks such as Hibernate, Spring, Struts... can inspect information or inject their utility code through getter and setter. So providing getter and setter is necessary when integrating your code with such frameworks.

Naming Convention for getter and setter

The naming scheme of setter and getter should follow **Java bean naming convention** as follows:

getXXX() and setXXX()

where XXX is **name of the variable**. For example with the following variable name:

```
private String name;
```

Then the appropriate setter and getter will be:

```
public void setName(String name) { }
```

```
public String getName() { }
```

If the variable is of type **boolean**, then the getter's name can be either **isXXX()** or **getXXX()**, but the former naming is preferred.

For example:

```
private Boolean single;
```

```
public String isSingle( ) { }
```



Common mistakes while implementing getters and setters

People often make mistakes, so do developers. This section describes the most common mistakes when implementing setter and getter in Java, and workarounds.

Mistake #1: Have setter and getter, but the variable is declared in less restricted scope.

If the variable is declared as public, then it can be accessed using dot (.) operator directly, making the setter and getter useless. Workaround for this case is using more restricted access modifier such as protected and private.

Mistake #2: Assign object reference directly in setter

Considering the following setter method:

```
private int[] scores;  
  
public void setScores(int[] scr) {  
    this.scores = scr;  
}
```

And following is code that demonstrates the problem:

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

setScores(myScores);
displayScores();
myScores[1] = 1;
displayScores();
```

An array of integer numbers `myScores` is initialized with 6 values (line 1) and the array is passed to the `setScores()` method (line 2). The method `displayScores()` simply prints out all scores from the array:

```
public void displayScores() {
    for (int i = 0; i < this.scores.length; i++) {
        System.out.print(this.scores[i] + " ");
    }
    System.out.println();
}
```

Line 3 will produce the following output:

5 5 4 3 2 4

That are exactly all elements of the `myScores` array.

Now at line 4, we can modify the value of the 2nd element in the `myScores` array as follow:

`myScores[1] = 1;`

What will happen if we call the method `displayScores()` again at line 5? Well, it will produce the following output:

5 1 4 3 2 4

You can realize that the value of 2nd element is changed from 5 to 1, as a result of the assignment in line 4. Why does it matter? Well, that means the data can be modified outside scope of the setter method which breaks encapsulation purpose of the setter. And why that happens? Let's look at the `setScores()` method again:



VectorStock®

VectorStock.com/25155084

```
public void setScores(int[] scr) {  
    this.scores = scr;  
}
```

The member variable scores is assigned to the method's parameter variable scr directly. That means both the variables are referring the same object in memory - the myScores array object. So changes made to either scores variable or myScores variable are actually made on the same object.

Workaround for this situation is to copy elements from scr array to scores array, one by one. The modified version of the setter would be like this:

```
public void setScores(int[] scr) {  
    this.scores = new int[scr.length];  
    System.arraycopy(scr, 0, this.scores, 0, scr.length);  
}
```

What's difference? Well, the member variable scores is no longer referring to the object referred by scr variable. Instead, the array scores is initialized to a new one with size equals to the size of the array scr. Then we copy all elements from the array scr to the array scores, using System.arraycopy() method.

Run the example again and it will give the following output:

```
5 5 4 3 2 4  
5 5 4 3 2 4
```

Now the two invocation of displayScores() produce the same output. That means the array scores is independent and different than the array scr passed into the setter, thus the assignment:

```
myScores[1] = 1;
```

does not affect the array scores.

So, the rule of thumb is, if you pass an object reference into a setter method, then don't copy that reference into the internal variable directly. Instead, you should find some ways to copy values of the passed object into the internal object, like we have copied elements from one array to another using System.arraycopy() method.

Mistake #3: Return object reference directly in getter

Consider the following getter method:

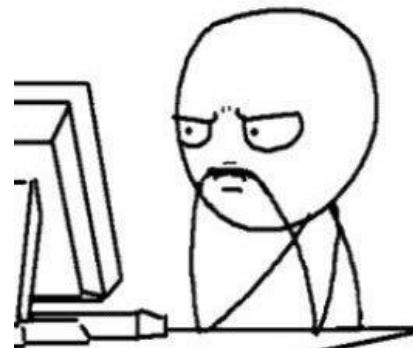
```
private int[] scores;

public int[] getScores() {
    return this.scores;
}
```

And the following code snippet:

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

setScores(myScores);
displayScores();
int[] copyScores = getScores();
copyScores[1] = 1;
displayScores();
```



it will produce the following output:

```
5 5 4 3 2 4
5 1 4 3 2 4
```

As you notice, the 2nd element of the array scores is modified outside the setter, at line 5. Because the getter method returns reference of the internal variable scores directly, so the outside code can obtain this reference and makes change to the internal object.

Workaround for this case is that, instead of returning the reference directly in the getter, we should return a copy of the object, so the outside code can obtain only a copy, not the internal object. Therefore we modify the above getter as follows:

```
public int[] getScores() {  
    int[] copy = new int[this.scores.length];  
    System.arraycopy(this.scores, 0, copy, 0, copy.length);  
    return copy;  
}
```

So the rule of thumb is: do not return reference of the original object in getter method. Instead, it should return a copy of the original object.

Variable Shadowing

If the **instance variable and local variable** have **same name** whenever you print (access) it in the method. The **value of the local variable will be printed** (shadowing the instance variable).

If you still, need to access the **values of instance variables** in a method you need to access them using **this keyword** (or object).



Now let's see the special type of setter which are called as Constructors

Constructors are special type of setters which follow the following rules:

- Method name will be same as that of the class name.
- No return type.
- Arguments are passed during the object creation.

Note: Whenever the programmer **does not provide a single constructor**, then during the **object creation** when we call the constructor, we are basically calling the **default constructor**.

Is there any difference between constructor and method???

Well definitely there is a difference between method and constructor. So let us see what the difference is...



Sl. No.	Constructor	Method
1	Constructor name should be same as class name.	Method name may or may not be same as class name.
2	Constructor never returns any value so it has no return type.	Method must have a return type in Java and returns only a single value.
3	There are 2 types of constructor available in Java.	Java supports 6 types of method.
4	Constructor only can be called by new keyword in Java	Method can be called by class name, object name or directly.
5	If there is no constructor designed by user then the Java compiler automatically provides the default constructor.	Javac never provides any method by default.
6	Constructor cannot be overridden in Java.	Method can be overridden.
7	Constructor cannot be declared as static.	Method can be declared as Static.

Constructor Overloading

Like methods, **constructors** can also be overloaded.

Let's understand what is constructor overloading and Why do we do it by considering an example shown below:



Example:

```
class Car
{
    private String name;
    private int mileage;
    private int cost;
    public Car() → Zero parameterized constructor
    {
        name = "BMW";
        mileage = 10;
        cost = 7000000;
    }
    public Car(String name,int mileage,int cost)
    {
        this.name = name;
        this.mileage = mileage;
        this.cost = cost;
    }
    public String getName()
    {
        return name;
    }
    public int getMileage()
    {
        return mileage;
    }
    public int getCost()
    {
        return cost;
    }
}
```

Zero parameterized constructor

Parameterized constructor

```
class Demo
{
    public static void main(String[] args)
    {
        Car c1 = new Car();
        System.out.println(c1.getName());
        System.out.println(c1.getMileage());
        System.out.println(c1.getCost());

        Car c2 = new Car("Ferrari",5,9000000);
        System.out.println(c2.getName());
        System.out.println(c2.getMileage());
        System.out.println(c2.getCost());
    }
}
```

Output:

Ferrari
5
9000000

What is constructor overloading?

Having Multiple constructors within a class is referred to as constructor overloading.

Why do we do it?

Constructor overloading is required so that different objects can be initialized differently.

In one year, Java gets downloaded one billion times.



Local Chaining

What is local chaining?

Local Chaining is the process of a constructor of a class calling another constructor of the same class.

Why do we do it?

Local chaining allows you to maintain your initialization from a single location, while providing multiple constructors to the user.

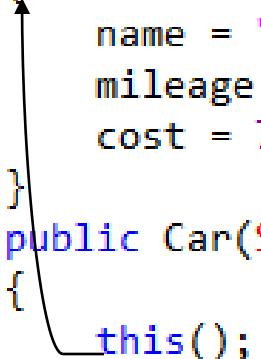
How to achieve local chaining?

*Local chaining can be achieved using `this()` function call.
`this()` should compulsorily be the first line in the constructor.*

Let's understand what is local chaining by considering the example shown below:

Example:

```
class Car
{
    private String name;
    private int mileage;
    private int cost;
    public Car()
    {
        name = "BMW";
        mileage = 10;
        cost = 7000000;
    }
    public Car(String name,int mileage,int cost)
    {
        this();
    }
}
```



```

public String getName()
{
    return name;
}
public int getMileage()
{
    return mileage;
}
public int getCost()
{
    return cost;
}
}

class Demo
{
    public static void main(String[] args)
    {
        Car c1 = new Car("Ferrari",5,9000000);
        System.out.println(c1.getName());
        System.out.println(c1.getMileage());
        System.out.println(c1.getCost());
    }
}

```

In the above example, `this.name = name;`
`this.mileage = mileage`
`this.cost = cost;`



Replaced by `this()`
 In parametrized constructor

In the above code, during object creation we are calling 3 parameterized constructor inside which first line is `this()` which will now give control to zero parameterized constructor and this way we achieve local chaining.

Let us understand this() in detail by considering different examples.

Example - 1

```
class Car
{
    private String name;
    private int mileage;
    private int cost;
    public Car()
    {
        name = "BMW";
        mileage = 10;
        cost = 7000000;
    }
    public Car(String name,int mileage,int cost)
    {
        this();
        this.name = name;
        this.mileage = mileage;
        this.cost = cost;
    }
    public String getName()
    {
        return name;
    }
    public int getMileage()
    {
        return mileage;
    }
    public int getCost()
    {
        return cost;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Car c1 = new Car("Ferrari",5,9000000);
        System.out.println(c1.getName());
        System.out.println(c1.getMileage());
        System.out.println(c1.getCost());
    }
}
```



In the above code, during object creation we are calling 3 parameterized constructor inside which the first line is this() which will now control to zero parameterized constructor. Once the body of zero parameterized constructor execute, control comes back to where it came from which is parameterized constructor.

Example – 2

```
class Car
{
    private String name;
    private int mileage;
    private int cost;
    public Car()
    {
        name = "BMW";
        mileage = 10;
        cost = 7000000;
    }
    public Car(String name,int mileage,int cost)
    {
        this(name);
    }
    public Car(String name)
    {
        this();
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
    public int getMileage()
    {
        return mileage;
    }
    public int getCost()
    {
        return cost;
    }
}
```

```
class Demo
{
    public static void main(String[] args)
    {
        Car c1 = new Car("Ferrari",5,9000000);
        System.out.println(c1.getName());
        System.out.println(c1.getMileage());
        System.out.println(c1.getCost());
    }
}
```

In the above code, during object creation we are calling 3 parameterized constructor. Inside 3 parameterized constructor, the first line is this(name) which will now give control to one parameterized constructor inside which first line is this() which in turn will give control to 0 parameterized constructor. In this way, chaining between different constructor takes place using this().

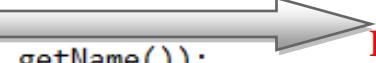


He got it. Did you?

Try tracing the codes given below:

Example – 3

```
class Car
{
    private String name;
    private int mileage;
    private int cost;
    public Car(String name,int mileage,int cost)
    {
        this.name = name;
        this.mileage = mileage;
        this.cost = cost;
    }
    public String getName()
    {
        return name;
    }
    public int getMileage()
    {
        return mileage;
    }
    public int getCost()
    {
        return cost;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Car c1 = new Car("Ferrari",5,9000000);
        System.out.println(c1.getName());
        System.out.println(c1.getMileage());
        System.out.println(c1.getCost());

        Car c2 = new Car();  Error
        System.out.println(c2.getName());
        System.out.println(c2.getMileage());
        System.out.println(c2.getCost());
    }
}
```

Output: **Error (there is no zero parameterized constructor in your code)**

The error in the above code is removed by inserting a zero parameterized constructor as shown below.

Example - 4

```
class Car
{
    private String name;
    private int mileage;
    private int cost;
    public Car(String name,int mileage,int cost)
    {
        this.name = name;
        this.mileage = mileage;
        this.cost = cost;
    }
    public Car()
    {

    }
    public String getName()
    {
        return name;
    }
    public int getMileage()
    {
        return mileage;
    }
    public int getCost()
    {
        return cost;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Car c1 = new Car("Ferrari",5,9000000);
        System.out.println(c1.getName());
        System.out.println(c1.getMileage());
        System.out.println(c1.getCost());

        Car c2 = new Car();
        System.out.println(c2.getName());
        System.out.println(c2.getMileage());
        System.out.println(c2.getCost());
    }
}
```

Static

Before understanding the static keyword in java, one must be aware of the different members a class can have.

A class in java consist of the following members

Class Demo {

Static variables

Static blocks

Static methods

Instance variables

Instance block

Instance method

Constructors

}



Static members of a class

**Non static or
Instance members of a class**

After getting to know different members of a class one must also know Static members belong to **class** and Non-static members belong to **object**.

Class

*Static variables
Static blocks
Static methods*

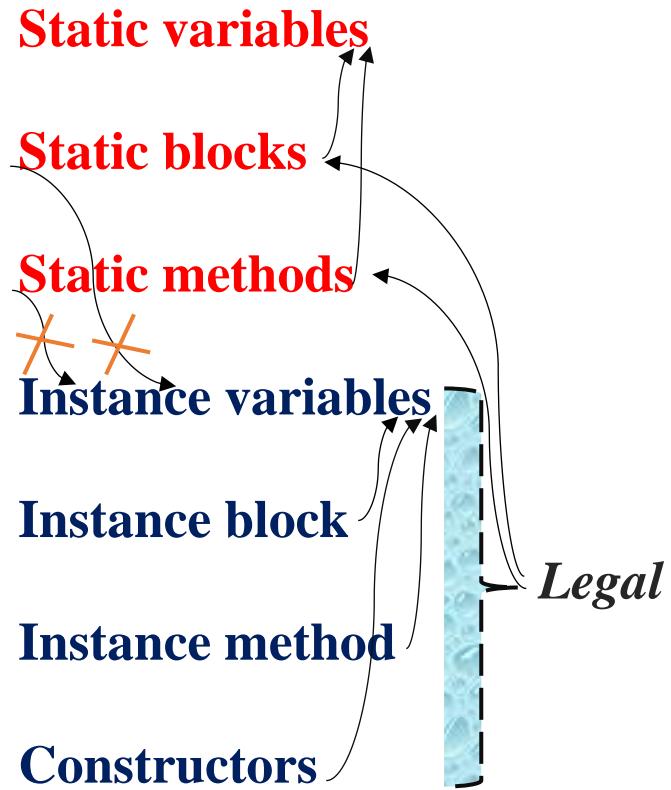
Object

**Instance variables
Instance blocks
Instance methods
Constructors.**

You are now aware of what belongs to a class and what belongs to an object but this knowledge is not sufficient to understand when to use static and when not to use. To understand that one must learn few rules of static.



RULES OF STATIC:



Static Variables can be accessed by static as well as instance variables.

Instance variables can be accessed by only instance members.

Static members cannot access instance variables.

In the Java programming language, **the keyword *static* indicates that the particular member belongs to a type itself, rather than to an instance of that type.**

This means that only one instance of that static member is created which is shared across all instances of the class.

The keyword can be applied **to variables, methods, blocks and nested class.**

Static variables: Static variable in Java is a variable which belongs to the class, not to object and initialized only once at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.

- A single copy to be shared by all instances of the class
- A static variable can be accessed directly by the class name and doesn't need any object

Syntax: <*class-name*>.<*variable-name*>

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

- A static method can call only other static methods and cannot call a non-static method from it.
- A static method can be accessed directly by the class name and doesn't need any object.
- A static method cannot refer to "this" or "super" keywords in anyway.

Syntax: <*class-name*>.<*method-name*>

Static block: The static block is a block of statement inside a Java class that will be executed after object creation and before call to the constructor. A **static block helps to initialize the static data members**, just like constructors help to initialize instance members.

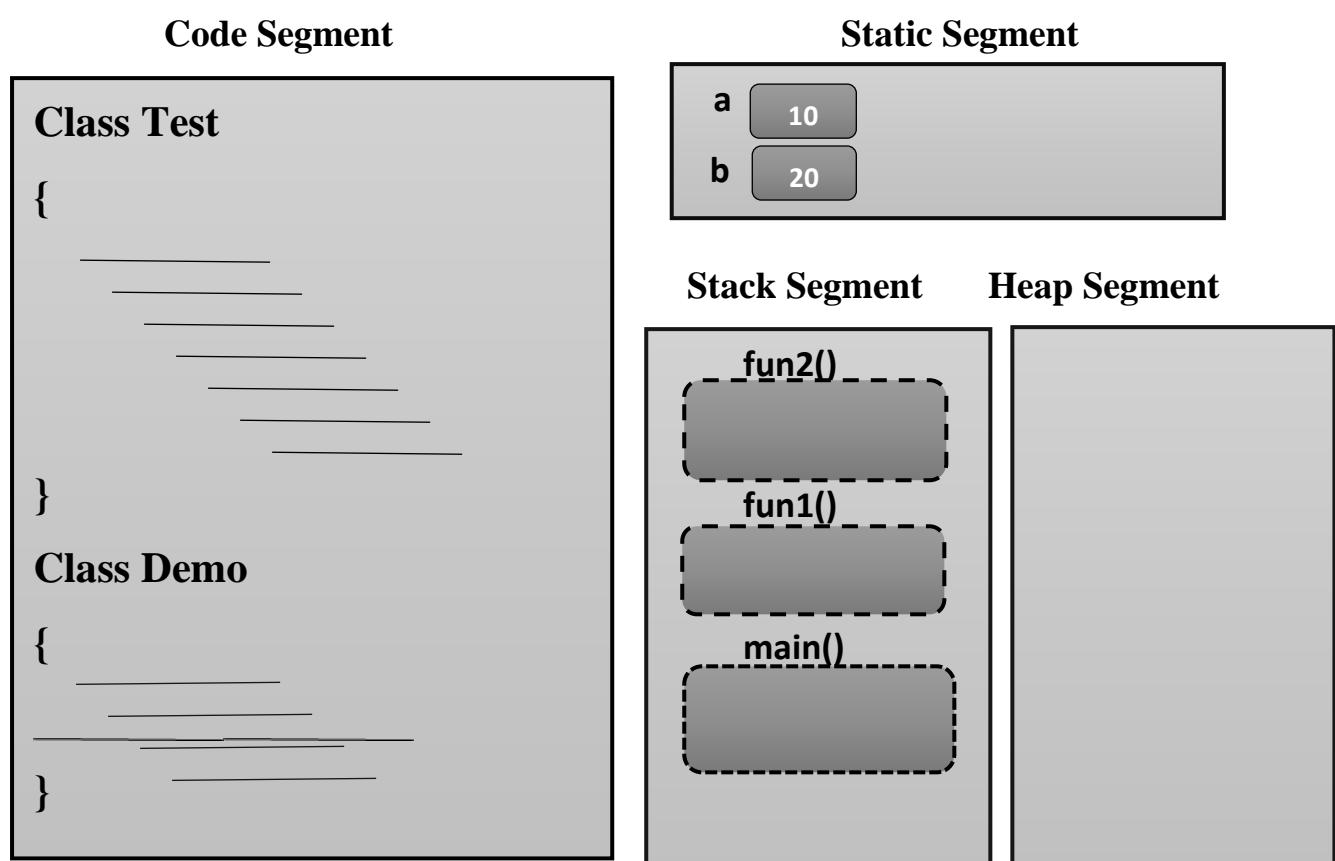
Let us explore static in detail with the help of codes.

Have a look at the code given below which consists Of all the members.



```
class Test
{
    static int a,b;
    static
    {
        System.out.println("Inside static block");
        a = 10;
        b = 20;
    }
    static void fun1()
    {
        System.out.println("Inside static method");
    }
    int x,y;
    {
        System.out.println("Inside instance block");
    }
    void fun2()
    {
        System.out.println("Inside instance method");
    }
    Test()
    {
        System.out.println("Inside constructor");
        x = 30;
        y = 40;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Test.fun1();
        Test t = new Test();
        t.fun2();
    }
}
```

Let us now trace the code and understand it from the memory perspective.



Explanation:

When the first class which consists of `main()` is loaded onto the code segment, the first check that is made JVM is **static variables**. Since there are no static variables inside class Demo, It will now check for **static blocks** and again there is no static block inside class Demo. Now the JVM will search for `main()`. Now `main()` gets called and its **stack frame** gets created on stack segment. Now `main()` starts executing and inside `main()` the very first line is **call to fun1()**. But `fun1()` is present inside class Test which is not yet loaded inside code segment. JVM will now ask **class loader** to load Test class. Once the class Test is loaded, JVM will now trace the class and look for static variables. Class Test consists of **2 static variables a,b** which are allocated memory **on static segment** and JVM gives default values. Now JVM checks for static block. Since the Test class consists of static block, it gets executed. Now control goes back to the `main()`. The first line is call to `fun1()`. Control now goes to Test class and **stack frame of fun1()** is created on stack segment. Once this `fun1()` gets executed, control leaves the `fun1()` and its stack frame gets **deleted**. Control comes back to `main()` and the next line is object creation. Assignment means start from RHS and the moment JVM encounters new keyword, it allocates a block of memory. It now goes to class Test and allocates memory for instance variables **x,y**. It also gives default values to **x,y**.

After object creation and before call to the constructor, **instance block gets executed**. After this, constructor gets executed. Now the reference "**t**" is assigned to the object. The last line inside `main()` is call to `fun2()`. Once a method gets called, its stack frame gets created and then it gets executed. Once there are no more lines left inside `fun2()` control leaves the `fun2()` and its stack frame gets deleted. Control now goes back to where it came from that is to `main()`. Inside `main()` there are no more lines left so its stack frame also gets deleted.

In this way, a complete java program executes.

Static block

Static block is used for **initializing the static variables**. Although *static* variables can be initialized directly during declaration, there are situations when we're required to do the multiline processing.

This block gets executed when the **class is loaded in the memory**. A class can have multiple Static blocks, which will execute in the **same sequence** in which they have been written into the program.

Java static variables

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

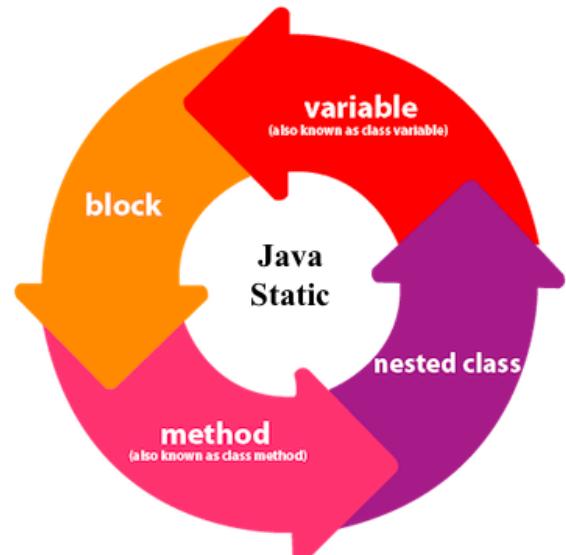
Few Important Points:

- Static variables are also known as Class Variables/static fields.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

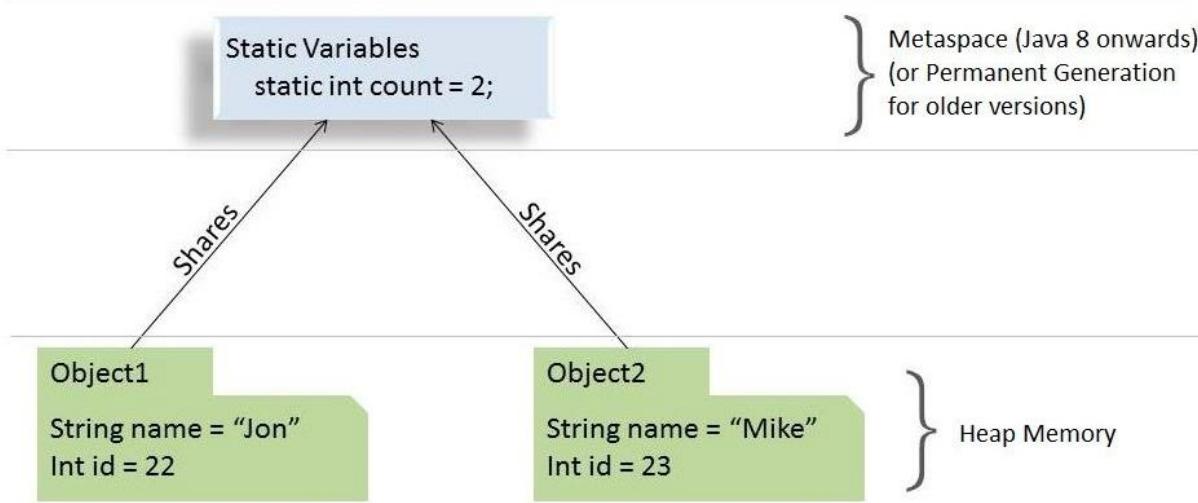


Let's see what static variables can do...

1. Static variables can be accessed directly in **Static method**.
2. Static variables are **shared among all the instances of class**. Whereas **non-static variables cannot be accessed by static but can only be accessed by non-static**. That is because **non-static/instance variable would have not got allocated in the memory**



In Java, if a field is declared ***static***, then exactly a single copy of that field is created and shared among all instances of that class. It doesn't matter how many times we initialize a class; there will always be only one copy of ***static*** field belonging to it. The value of this ***static*** field will be shared across all object of either same or any different class.



From the memory perspective, **static variables go in a particular pool in JVM memory called Metaspace** (before Java 8, this pool was called Permanent Generation or PermGen, which was completely removed and replaced with Metaspace).

Where to use static variables????

- When the value of variable is independent of objects.
- The static variable can be used to refer to the **common property of all objects** (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- If ***static*** variables require additional, multi-statement logic while initialization, then a ***static*** block can be used.



If you are thinking is there any **advantage** of using **static variables** then yes there is: It makes your program **memory efficient** (i.e., it saves memory).

How?? Well this is how...

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Key points to remember..

- Since *static* variables belong to a class, they can be **accessed directly using class name** and **don't need any object reference**.
- *static* variables can only be **declared at the class level**.
- *static* fields can be **accessed without object initialization**.
- Although we can access *static* fields or the class variables using an object reference, we should refrain from using it as in this case it becomes difficult to figure whether it's an instance variable or a class variable; instead, we should always refer to *static* variables using class.
- If initialization of *static* variables requires some additional logic except the assignment
- If the initialization of static variables is error-prone and requires exception handling

Static Methods

Similar to *static* variables, **static methods also belong to a class instead of the object**, and so they can be **called without creating the object of the class** in which they reside. They're meant to be used without creating objects of the class.

If you apply *static* keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

static methods are generally used to perform an operation that is **not dependent upon instance creation**.

If there is a code that is supposed to be **shared across all instances of that class**, then write that code in a *static* method.

A static method can **access only static variables of class and invoke only static methods of the class**



The **main()** method that is the entry point of a java program itself is a **static method**. Wonder why? It is **because the object is not required to call a static method**. If it were a non-static method, **JVM creates an object first then call main() method that will lead the problem of extra memory allocation**.

static methods are also widely used to **create utility or helper classes** so that they can be obtained without creating a new object of these classes.

Why to use static methods???

- To access/manipulate static variables and other static methods that don't depend upon objects.
- *static* methods are widely used in utility and helper classes.

Points to be remembered...

- *static* methods in Java are resolved at compile time. Since method overriding is part of Runtime Polymorphism, **so static methods can't be overridden.**
- abstract methods can't be static. //will be covered in future classes
- *static* methods **cannot use *this* or *super* keywords.** //will be covered in future classes
- The static method **cannot use non static data member** or call non-static method directly.
- The following combinations of the instance, class methods and variables are valid:
 1. Instance methods can directly access both instance methods and instance variables
 2. Instance methods can also access *static* variables and *static* methods directly
 3. *static* methods can access all *static* variables and other *static* methods
 4. **static methods cannot access instance variables and instance methods directly;** they need some object reference to do so

Now let's start with new concept which is one of the important features of OOP which is INHERITANCE

Inheritance in Java is a mechanism in which **one object acquires all the properties and behaviours of a parent object.** It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can **create new classes that are built upon existing classes.** When you inherit from an existing class, **you can reuse methods and fields of the parent class.** Moreover, **you can add new methods and fields in your current class also.**

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Child Class

The class that **extends the features of another class** is known as child class, sub class or derived class.



Parent Class

The **class whose properties and functionalities are used (inherited) by another class** is known as parent class, super class or Base class.



- One of the key benefits of inheritance is to **minimize the amount of duplicate code in an application by sharing common code amongst several subclasses**. Where equivalent code exists

in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.

- **Inheritance can also make application code more flexible** to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass
- **Reusability** - facility to use public methods of base class without rewriting the same.
- **Extensibility** - extending the base class logic as per business logic of the derived class.
- **Data hiding** - base class can decide to keep some data private so that it cannot be altered by the derived class
- **Overriding** -With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

The syntax of Java Inheritance

```
class subclass_name extends superclass_name
{
    //body of the class
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Let's understand with code

```
class CreditCard // parent class
{
    int cardNo = 12345;
    int pin = 8888;
}
class Hacker extends CreditCard //child class
{
    void viewDetails()
    {
        System.out.println(cardNo); //inherited variables
        System.out.println(pin);
    }
    void changeDetails()
    {
        cardNo = 6789; // Overriding variables
        pin = 9999;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Hacker h = new Hacker();
        h.viewDetails();
        h.changeDetails();
        h.viewDetails();
    }
}
```



Output | 12345
| 8888
| 6789
| 9999

There are certain rules in inheritance in java

- **Rule1:** Private members do not participate in inheritance, this is so that encapsulation does not get affected. However public, protected and default members participate in inheritance.
(You'll learn about private, public, default and protected access specifiers in future classes)

// Other rules will be covered in future classes.



Download from
Dreamstime.com

88073983
Chris Dorney | Dreamstime.com

Rules of inheritance:

Rule2: Multi-level Inheritance is permitted in java.

- When multiple classes are involved and their parent-child relation is formed in a chained way then such formation is known as multi-level inheritance.
- In multilevel inheritance, a parent class has a maximum of one direct child class only.
- In multi-level inheritance, the inheritance linkage is formed in a linear way and minimum 3 classes are involved.

```
class Demo1
{
    void fun1()
    {
    }

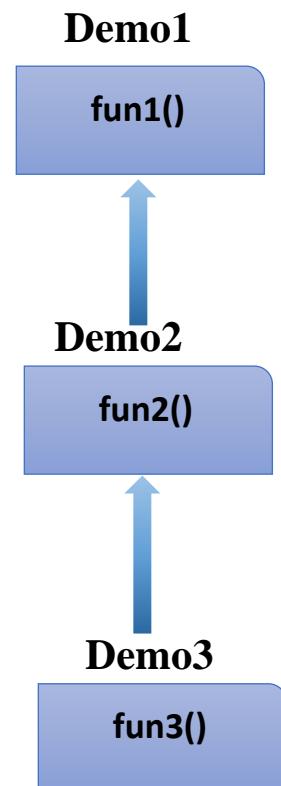
}

class Demo2 extends Demo1
{
    void fun2()
    {
    }

}

class Demo3 extends Demo2
{
    void fun3()
    {
    }

}
```



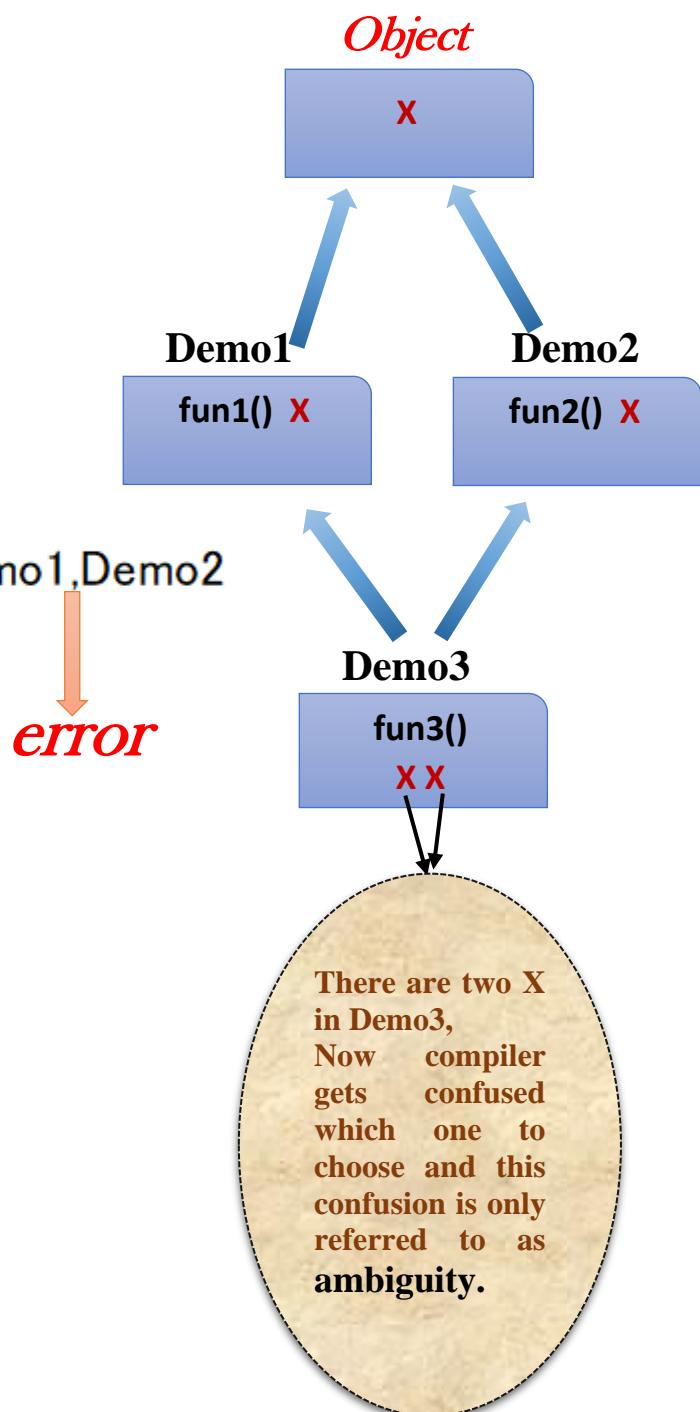
Like any other successful tech entrepreneurs,
Bill gates was a college dropout. **DID YOU
KNOW?**



Rule3: Multiple Inheritance is not permitted in java as it leads to diamond shaped problem which results in ambiguity.

Multiple Inheritance is a feature of object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

```
class Demo1 extends Object
{
    void fun1()
    {
    }
}
class Demo2 extends Object
{
    void fun2()
    {
    }
}
class Demo3 extends Demo1,Demo2
{
    void fun3()
    {
    }
}
```

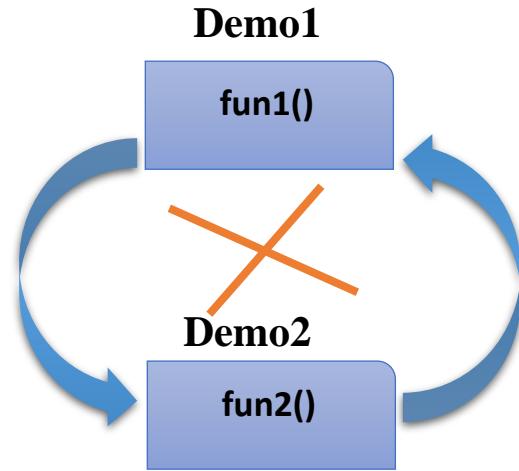


Rule4: Cyclic Inheritance is not permitted in.

It is a type of inheritance in which a class extends itself and form a loop itself. Now think if a class extends itself or in any way, if it forms cycle within the user-defined classes, then is there any chance of extending the Object class.

```
class Demo1 extends Demo2
{
    void fun1()
    {

    }
}
class Demo2 extends Demo1
{
    void fun2()
    {
    }
}
```



Rule5: Constructors do not participate in inheritance.

In Java, constructor of base class with no argument gets automatically called in derived class constructor by a process called as **constructor chaining**.

Constructor chaining is a process of child class constructor calling its parent class constructor using **super() call**.

The first line is automatically super() call.

Irrespective of whether it is parameterized constructor or zero parameterized constructor of child class, the super() call will automatically take the control to **zero parameterized constructor** of parent class.

Let us understand constructor chaining in detail by considering few examples as shown below:



```
Class Object
{
    Object()
    {

    }
}

Class Test1 extends Object
{
    int x,y;
    Test1()
    {
        super();
        x=100;
        y=200;
    }

    Test1(int x, int y)
    {
        this.x=x;
        this.y=y;
    }
}

Class Test2 extends Test1
{
    int a,b;
    Test2()
    {
        super();
        a=300;
        b=400;
    }

    Test2(int a, int b)
    {
        this.a=a;
        this.b=b;
    }
}
```

```
void disp()
{
    System.out.println(x);
    System.out.println(y);
    System.out.println(a);
    System.out.println(b);
}
}

Class Demo
{
    public static void main(String[ ] args)
    {
        Test2 t2 = new Test2();
        t2.disp();
    }
}
```

Output:

100
200
300
400

Calling parameterized constructor.

In the previous example we had seen constructor chaining of zero parameterized constructor using super() call.

Calling parameterized constructor in constructor chaining is similar to Calling parameterized constructor in local chaining.

In local chaining, to call parameterized constructor we had passed parameters to **this()**, in the same way to call parameterized constructor in constructor chaining all we have to do is **pass parameters to super()**.



*Let us try to understand this with the help
Of code:*

```
class Object
{
    Object()
    {

    }
}

class Test1 extends Object
{
    int x,y;
    Test1() //zero parameterized constructor
    {
        x=100;
        y=200;
    }
    Test1(int x, int y) //parameterized constructor
    {
        super();
        this.x = x;
        this.y = y;
    }
}
```

```

class Test2 extends Test1
{
    int a,b;
    Test2() //zero parameterized constructor
    {
        a=300;
        b=400;
    }
    Test2(int a, int b) //parameterized constructor
    {
        super(a,b);          | Call to parameterized
        this.a = a;           | constructor
        this.b = b;
    }
    void disp()
    {
        System.out.println(x);
        System.out.println(y);
        System.out.println(a);
        System.out.println(b);
    }
}
class Demo
{
    public static void main(String[ ] args)
    {
        Test2 t2 = new Test2(9,99);
        t2.disp();
    }
}

```

OUTPUT

9
99
9
99

After learning local chaining and constructor chaining in detail, try to predict the output of the code given below which consists of both local and constructor chaining.

Exercise:

```
class Object
```

```
{
```

```
    Object()
```

```
{
```

```
}
```

```
}
```

```
class Test1 extends Object
```

```
{
```

```
    int x,y;
```

```
    Test1() //zero parameterized constructor
```

```
{
```

```
        super(); //call to object class constructor(constructor chaining)
```

```
        x=100;
```

```
        y=200;
```

```
}
```

```
    Test1(int x, int y) //parameterized constructor
```

```
{
```

```
        this.x = x;
```

```
        this.y = y;
```

```
}
```

```
}
```

```
class Test2 extends Test1
```

```
{
```

```
    int a,b;
```

```
    Test2() //zero parameterized constructor
```

```
{
```

```
        this(9,99); //call to parameterized constructor in same class(local chaining)
```

```
}
```

```
    Test2(int a, int b) //parameterized constructor
```

```
{
```

```
        super(); //call to parent class constructor(constructor chaining);
```

```
        this.a = a;
```

```
        this.b = b;
```

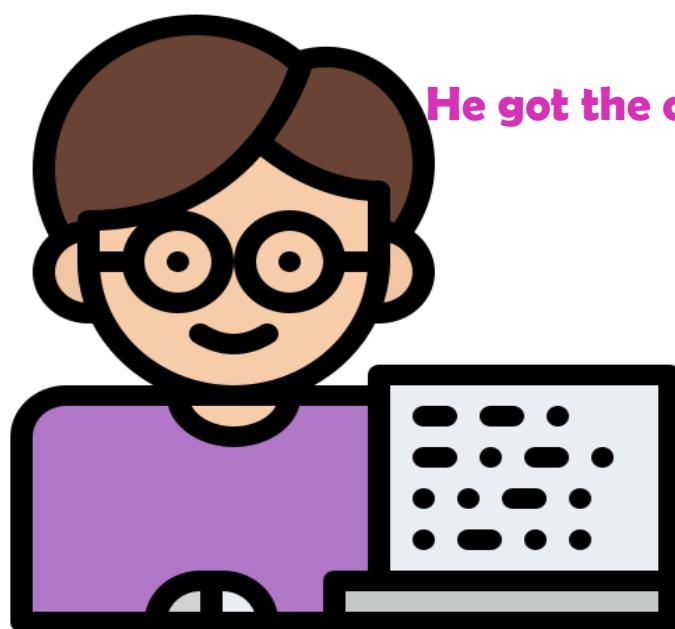
```
}
```



```
void disp()
{
    System.out.println(x);
    System.out.println(y);
    System.out.println(a);
    System.out.println(b);
}
class Demo
{
    public static void main(String[ ] args)
    {
        Test2 t2 = new Test2();
        t2.disp();
    }
}
```

OUTPUT

100
200
9
99



He got the correct output. Did you?

Key points:

- The first line of a constructor can be either the super() call or this() call.
- Super() call would perform **constructor chaining**, whereas this() call would perform **local chaining**.
- Both this() and super() must compulsorily be the **first line of constructor**, hence the first line of constructor can only be either this() or super() but not both.

Types of methods in inheritance

Inheritance has three different types of methods



A derived class has the ability to inherit, redefine/override an inherited method, replacing the inherited method by one that is specifically designed for the derived class and to have a method which is not present in base class.

Definitions:

Inherited Methods are such methods which are inherited from parent class and used without any modification in child class.

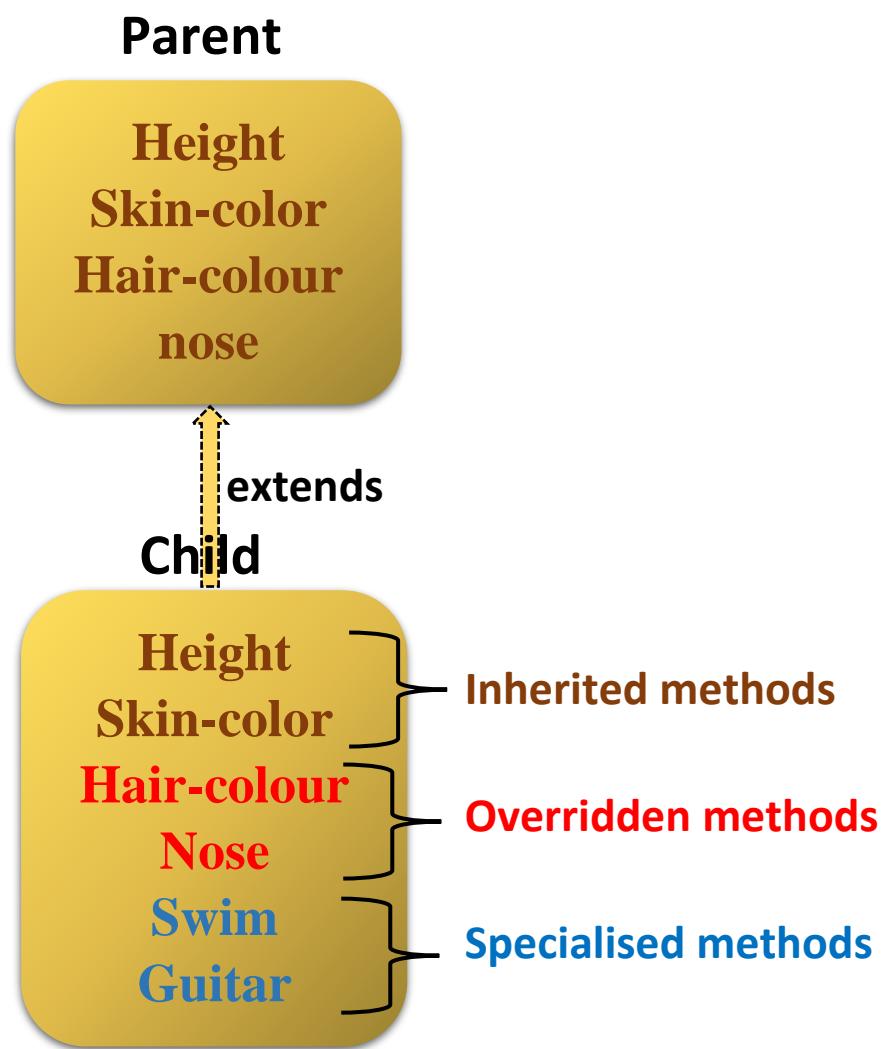
Overridden methods are such methods which are inherited from parent class and are modified and used in child class.

Specialised methods are such methods which are only present in child class but not in parent class.

Let us try to understand these methods in detail with the help of Parent-Child example.



Consider the UML diagram shown below:



In the above diagram, height and colour are inherited from the parent and used without modification, hence they are inherited methods.

Hair-colour is inherited and modified by the child; hence it is now an overridden method. Similarly, nose is modified and used by the child and hence it is also an overridden method.

Swim and Guitar are not present in parent class but in child class, hence they are specialised methods.

Let us start coding to understand these methods.



```
class Plane
{
    void takeoff()
    {
        System.out.println("Plane is taking off");
    }
    void fly()
    {
        System.out.println("Plane is flying");
    }
    void land()
    {
        System.out.println("Plane is landing");
    }
}
class CargoPlane extends Plane
{
    void fly()
    {
        System.out.println("Plane is flying at low heights");
    }
}
class PassengerPlane extends Plane
{
    void fly()
    {
        System.out.println("Plane is flying at medium heights");
    }
}
class FighterPlane extends Plane
{
    void fly()
    {
        System.out.println("Plane is flying at higher heights");
    }
}
```

```
class Demo
{
    public static void main(String[ ] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        cp.takeoff();
        cp.fly();
        cp.land();

        pp.takeoff();
        pp.fly();
        pp.land();

        fp.takeoff();
        fp.fly();
        fp.land();
    }
}
```

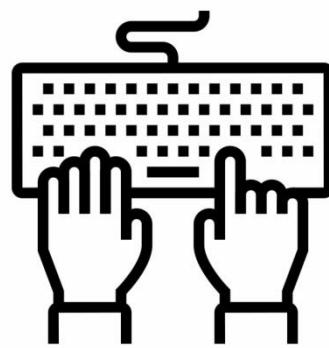
OUTPUT:

Plane is taking off
Plane is flying at low heights
Plane is landing
Plane is taking off
Plane is flying at medium heights
Plane is landing
Plane is taking off
Plane is flying at higher heights
Plane is landing

If you carefully look at the above output, it's the overridden methods that gets executed not the inherited methods.

Specialized methods in java

Let us understand with the help of code...



```
class Plane //parent class
{
    void takeOff()
    {
        System.out.println("Plane is taking off");
    }
    void fly()
    {
        System.out.println("Plane is flying");
    }
    void land()
    {
        System.out.println("Plane is landing");
    }
}
class CargoPlane extends Plane //child class
{
    void fly() // Overridden method
    {
        System.out.println("CargoPlane is flying at low heights");
    }
    void carryCargo() //specialized method
    {
        System.out.println("CargoPlane carries Cargo");
    }
}
class PassengerPlane extends Plane // child class
{
    void fly() // Overridden method
    {
        System.out.println("PassengerPlane is flying at medium heights");
    }
    void carryPassengers() //specialized method
    {
        System.out.println("PassengerPlane carries passengers");
    }
}
class FighterPlane extends Plane // child class
{
    void fly() // Overridden method
    {
        System.out.println("FighterPlane is flying at great heights");
    }
    void carryWeapons() //specialized method
    {
        System.out.println("FighterPlane is carrying weapons");
    }
}
```

```
class Demo
{
    public static void main(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        cp.takeOff(); //Inherited method
        cp.fly(); //Overridden method
        cp.land(); //Inherited Method
        cp.carryCargo(); //Specialized method

        pp.takeOff();
        pp.fly();
        pp.land();
        pp.carryPassengers();

        fp.takeOff();
        fp.fly();
        fp.land();
        fp.carryWeapons();
    }
}
```

Output:

```
Plane is taking off
CargoPlane is flying at low heights
Plane is landing
CargoPlane carries Cargo
Plane is taking off
PassengerPlane is flying at medium heights
Plane is landing
PassengerPlane carries passengers
Plane is taking off
FighterPlane is flying at great heights
Plane is landing
FighterPlane is carrying weapons
```

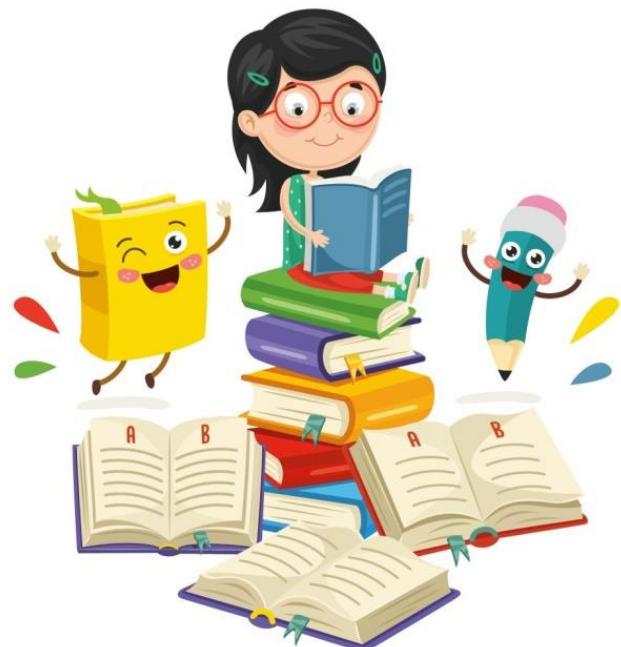
What did we see in the above example??

1. Inherited methods are such methods which are inherited from the parent class and used without any modification within the child class.
2. Overridden methods are such methods which are inherited from parent class and are modified as per the requirement of child class and used within the child class.
3. Specialized methods are such methods which are present only within the child class and not within the parent class.
4. Advantage of inherited methods is code reusability.
5. Advantage of overridden method is that the child class can modify the parent class method to satisfy its needs.
6. Advantage of specialized methods is that it increases the features of the child class when compared to parent class.



Amongst these four pillars of OOP, we have seen Encapsulation and Inheritance and virtual polymorphism. Now let's see how to achieve true polymorphism using inheritance.

1. **Abstraction** : Abstraction is the process of **showing only essential/necessary features** of an entity/object to the outside world and hide the other irrelevant information. (Will be discussed in detail in further classes.)
2. **Encapsulation** : Encapsulation means **wrapping up data** and member function (Method) together into a single unit i.e. class. Encapsulation automatically achieve the concept of **data hiding providing security to data** by making the **variable as private** and expose the property to access the private data which would be public.
3. **Inheritance** : The ability of **creating a new class from an existing class**. Inheritance is **when an object acquires the property of another object**. Inheritance allows a class (subclass) to acquire the properties and behaviour of another class (super-class). It helps to **reuse, customize and enhance the existing code**. So it helps to write a code accurately and reduce the development time.
4. **Polymorphism**: Polymorphism is derived from 2 Greek words: poly and morphs. The word "**poly**" means **many** and "**morphs**" means **forms**. So polymorphism means "**many forms**". A subclass can define its own unique behaviour and still share the same functionalities or behaviour of its parent/base class. A subclass can have their own behaviour and share some of its behaviour from its parent class not the other way around. A parent class cannot have the behaviour of its subclass.



In previous example if we change the main method to the following let's see what happens...



```
class Demo
{
    public static void main(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        cp.fly(); //one reference one behaviour
        pp.fly(); //one reference one behaviour
        fp.fly(); //one reference one behaviour
    }
}
```

We have three different references exhibiting 3 different behaviours. So now let's modify the main() and achieve polymorphism.

```
class Demo
{
    public static void main(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        Plane ref; //Parent type reference
        ref=cp; //assigning child type reference to parent type
        ref.fly(); //one reference one behaviour

        ref=pp; //assigning child type reference to parent type
        ref.fly(); //same reference same behaviour

        ref=fp; //assigning child type reference to parent type
        ref.fly(); //same reference same behaviour
    }
}
```

Creating parent type reference to child object is called loose coupling, through which we have achieved polymorphism. Whereas before we had tight coupling, where only child class's reference can access child class's behaviour. For example CargoPlane reference cp can access only CargoPlane behaviours and not of FighterPlane or PassengerPlane.

There is one limitation of parent type reference to child type. That is, using parent type reference we cannot directly access the specialized methods of child class.

With the same example as above here we are trying to access specialized methods through parent type reference ref. Let's see if we get any error, if yes then where is it pointing to...

```
50 class Demo
51 {
52     public static void main(String[] args)
53     {
54         CargoPlane cp = new CargoPlane();
55         PassengerPlane pp = new PassengerPlane();
56         FighterPlane fp = new FighterPlane();
57
58         Plane ref; //Parent type reference
59         ref=cp; //assigning child type reference to parent type
60         ref.fly(); //one reference one behaviour
61         ref.carryCargo(); //accessing specialized method
62
63         ref=pp; //assigning child type reference to parent type
64         ref.fly(); //same reference same behaviour
65         ref.carryPassengers(); //accessing specialized method
66
67         ref=fp; //assigning child type reference to parent type
68         ref.fly(); //same reference same behaviour
69         ref.carryWeapons(); //accessing specialized method
70     }
71 }
```



Output

```
Demo.java:61: error: cannot find symbol
        ref.carryCargo(); //accessing specialized method
                           ^
symbol:   method carryCargo()
location: variable ref of type Plane
Demo.java:65: error: cannot find symbol
        ref.carryPassengers(); //accessing specialized method
                           ^
symbol:   method carryPassengers()
location: variable ref of type Plane
Demo.java:69: error: cannot find symbol
        ref.carryWeapons(); //accessing specialized method
                           ^
symbol:   method carryWeapons()
location: variable ref of type Plane
3 errors
```

We can see that we have got compile time error wherever we tried accessing specialized method using parent type reference.

So now let's see how to overcome this...

```
class Demo
{
    public static void main(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        Plane ref; //Parent type reference
        ref=cp; //assigning child type reference to parent type
        ref.fly(); //one reference one behaviour
        ((CargoPlane)(ref)).carryCargo(); //Downcasting

        ref=pp; //assigning child type reference to parent type
        ref.fly(); //same reference same behaviour
        ((PassengerPlane)(ref)).carryPassengers(); //Downcasting

        ref=fp; //assigning child type reference to parent type
        ref.fly(); //same reference same behaviour
        ((FighterPlane)(ref)).carryWeapons(); //Downcasting
    }
}
```



Output:

```
CargoPlane is flying at low heights  
CargoPlane carries Cargo  
PassengerPlane is flying at medium heights  
PassengerPlane carries passengers  
FighterPlane is flying at great heights  
FighterPlane is carrying weapons
```

We have overcome it by changing the type of ref to respective child type reference explicitly. And this is called as Downcasting.



Polymorphism

- Code reusability
- flexibility
- Reduction in complexity



Let's see how to achieve these advantages

```
class Plane //parent class
{
    void takeOff()
    {
        System.out.println("Plane is taking off");
    }
    void fly()
    {
        System.out.println("Plane is flying");
    }
    void land()
    {
        System.out.println("Plane is landing");
    }
}
class CargoPlane extends Plane //child class
{
    void takeOff() // Overridden method
    {
        System.out.println("CargoPlane is taking off from long size runway");
    }
    void fly() // Overridden method
    {
        System.out.println("CargoPlane is flying at low heights");
    }
    void land() // Overridden method
    {
        System.out.println("CargoPlane is landing at long sized runways");
    }
}
```



www.clipartof.com · 1246277

```

class PassengerPlane extends Plane // child class
{
    void takeOff() // Overridden method
    {
        System.out.println("PassengerPlane is taking off from medium sized runway");
    }
    void fly() // Overridden method
    {
        System.out.println("PassengerPlane is flying at medium heights");
    }
    void land() // Overridden method
    {
        System.out.println("PassengerPlane is landing at medium sized runways");
    }
}
class FighterPlane extends Plane // child class
{
    void takeOff() // Overridden method
    {
        System.out.println("FighterPlane is taking off from short sized runway");
    }
    void fly() // Overridden method
    {
        System.out.println("FighterPlane is flying at great heights");
    }
    void land() // Overridden method
    {
        System.out.println("FighterPlane is landing at short short sized runways");
    }
}

class Demo
{
    public static void main(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        Plane ref; //Parent type reference
        ref=cp; //assigning child type reference to parent type
        ref.takeOff(); //one reference one behaviour
        ref.fly();
        ref.land();

        ref=pp; //assigning child type reference to parent type
        ref.takeOff(); //same reference same behaviour
        ref.fly();
        ref.land();

        ref=fp; //assigning child type reference to parent type
        ref.takeOff(); //same reference same behaviour
        ref.fly();
        ref.land();
    }
}

```

Output:

```
CargoPlane is taking off from long size runway  
CargoPlane is flying at low heights  
CargoPlane is landing at long sized runways  
PassengerPlane is taking off from medium sized runway  
PassengerPlane is flying at medium heights  
PassengerPlane is landing at medium sized runways  
FighterPlane is taking off from short sized runway  
FighterPlane is flying at great heights  
FighterPlane is landing at short short sized runways
```

The above code has achieved polymorphism but it hasn't achieved advantages of polymorphism yet. So next we will see how to achieve code reduction and code flexibility.

```
class Airport  
{  
    void permit(Plane ref) //Code reduction by  
    {  
        //passing parent type  
        ref.takeOff(); //reference as parameter  
        ref.fly(); //and calling the same function  
        ref.land();  
    }  
}  
class Demo  
{  
    public static void main(String[] args)  
    {  
        CargoPlane cp = new CargoPlane();  
        PassengerPlane pp = new PassengerPlane();  
        FighterPlane fp = new FighterPlane();  
  
        Airport a = new Airport();  
        a.permit(cp); //code flexibility is achieved  
        a.permit(pp); //by making use of one function  
        a.permit(fp); //call for all the child type ref  
    }  
}
```



In the above code we have now achieved the advantages of polymorphism.

Access Specifiers

What are access specifiers?

The access specifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.



There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

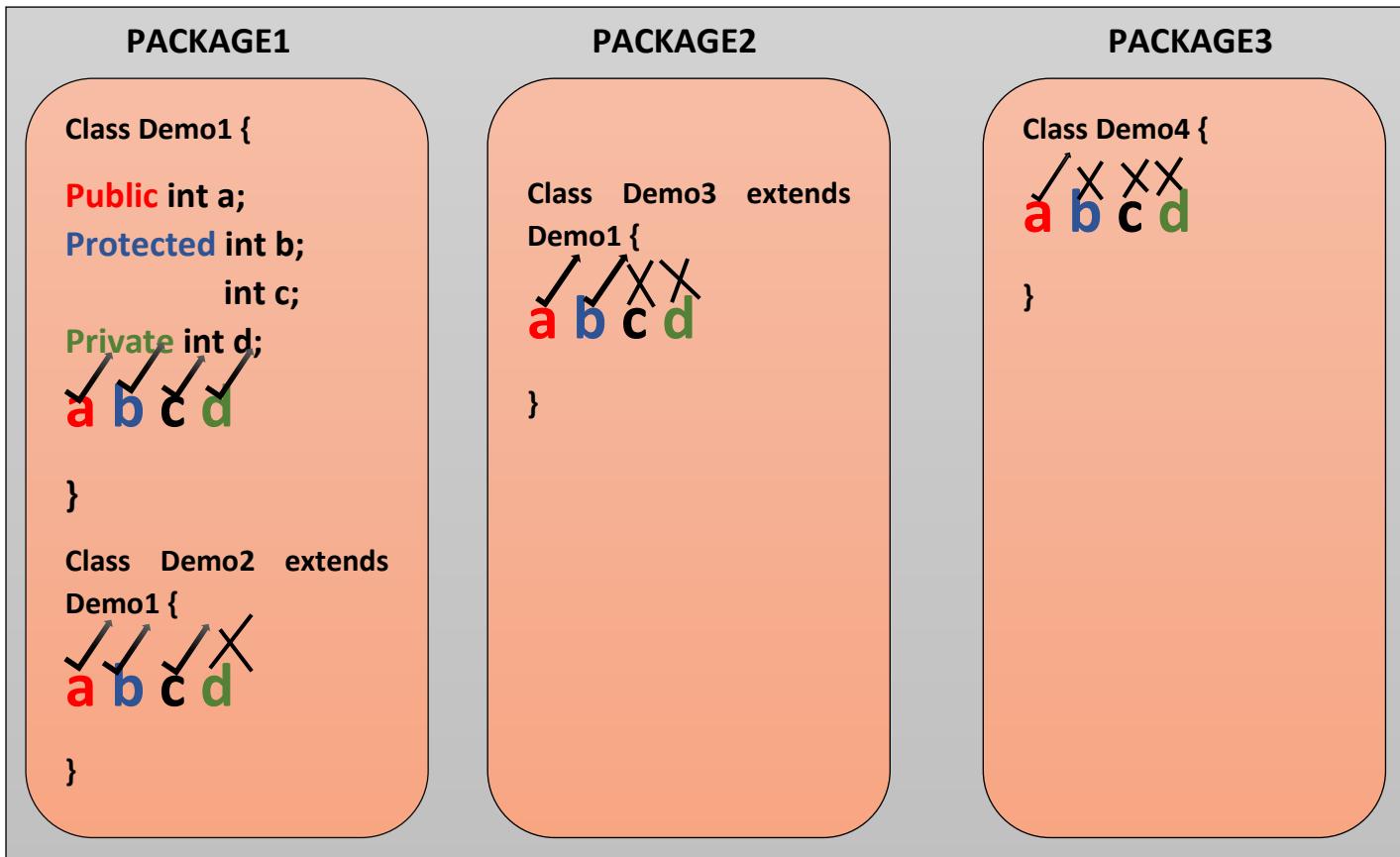
Didn't get it?? CONFUSED????

Let us try to understand these specifiers with the help of packages as shown below:



The 30th of November is known as “Computer Security Day”.

JAVA PROJECT



In the above java project, there are three packages. It consists of four variables:

Variable **a** whose access specifier is **Public**.

Variable **b** whose access specifier is **Protected**.

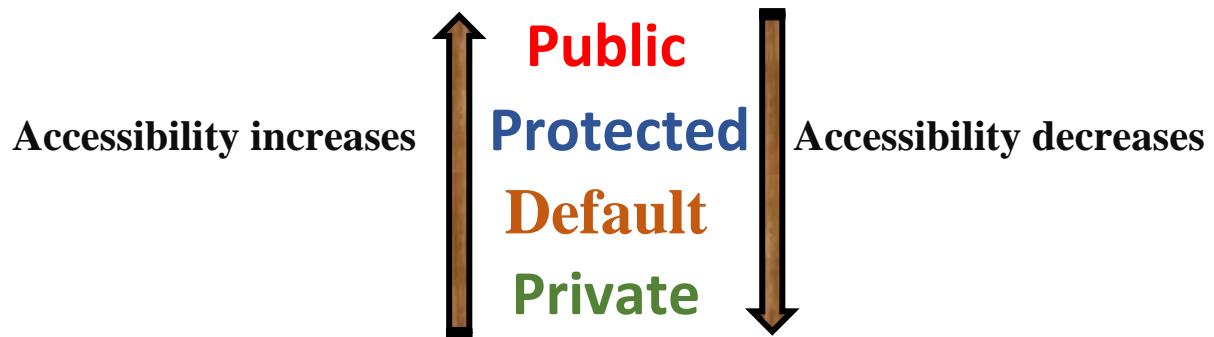
Variable **c** whose access specifier is **Default**.

Variable **d** whose access specifier is **Private**.

Let us understand the access of these variables by a simple table:

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Accessibility:



Rules of method overriding

Rule1: The overridden method of child class must either maintain the same access modifier as the parent class method or provide greater access however, it can never reduce.

```
class Parent
{
    public void fun()
    {
        System.out.println("Parent class method");
    }
}
class Child extends Parent
{
    void fun() // access reduced to default
    {
        System.out.println("Overidden child class method");
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Child c = new Child();
        c.fun();
    }
}
```

Output:

```
Demo.java:10: error: fun() in Child cannot override fun() in Parent
    void fun()
               ^
attempting to assign weaker access privileges; was public
1 error
```



Rule2: The overridden method of child class must maintain the same return types as the parent class method.

```
class Parent
{
    public void fun()
    {
        System.out.println("Parent class method");
    }
}
class Child extends Parent
{
    public int fun() // return type changed to int
    {
        System.out.println("Overridden child class method");
        return 1;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Child c = new Child();
        c.fun();
    }
}
```

Output:

```
Demo.java:10: error: fun() in Child cannot override fun() in Parent  
    public int fun()  
        ^  
        return type int is not compatible with void  
1 error
```



Rule3: In the overridden method of child class the return types can be different provided between the return types, **is-a relationship** exists. Such return types between which is-a relationship exists is called as **co-varient** return types.

```
class Plane  
{  
}  
  
class CargoPlane extends Plane  
{  
}  
  
class Parent  
{  
    public Plane fun()  
    {  
        System.out.println("Parent class method");  
        Plane p = new Plane();  
        return p;  
    }  
}
```

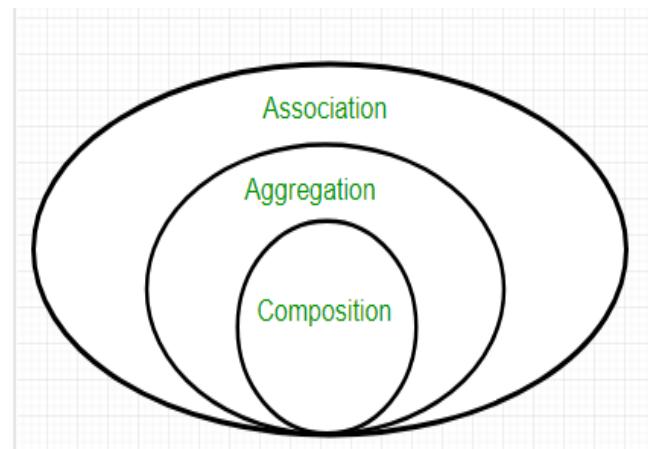
```
class Child extends Parent
{
    public CargoPlane fun()
    {
        System.out.println("Overridden child class method");
        CargoPlane cp = new CargoPlane();
        return cp;
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Child c = new Child();
        c.fun();
    }
}
```

Output:

Overridden child class method.

AGGREGATION AND COMPOSITION

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an **Object** communicates to other **Object** to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.



In the real world not only does **is-a** relationship exists also **has-a** relationship exists.

Is-a relationship is handled using **inheritance** whereas has-a relationship is handled using **aggregation and composition**.

Aggregation is a loose bound has-a relationship.
Composition is tight bound has-a relationship.

Let us now try to understand this in detail using few examples.

Example 1:

In the example shown below, a library contains students And books. Relationship between **library and student** is **aggregation** whereas relationship between **library and books** is **composition**.

A student can exist without library and therefore is aggregation. (**loosely bound**)

A book cannot exist without library and it is composition. (**tightly bound**)



Students



**has-a
aggregation**

Library



**has-a
composition**

Books

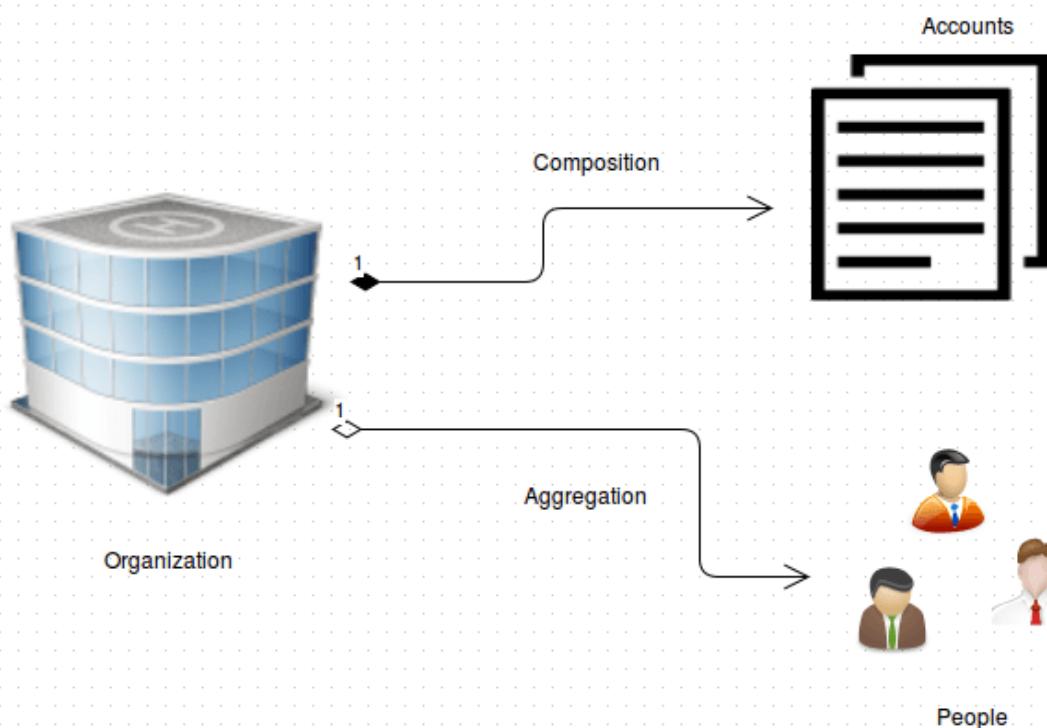


Example 2:

In this example shown below, an organization contains accounts and people. Relationship between **people and organization** is **aggregation** whereas relationship between **people and accounts** is **composition**.

People can exist without **organization** and therefore it is aggregation. (**loosely bound**)

Accounts cannot exist without **organization** and therefore it is composition. (**tightly bound**)



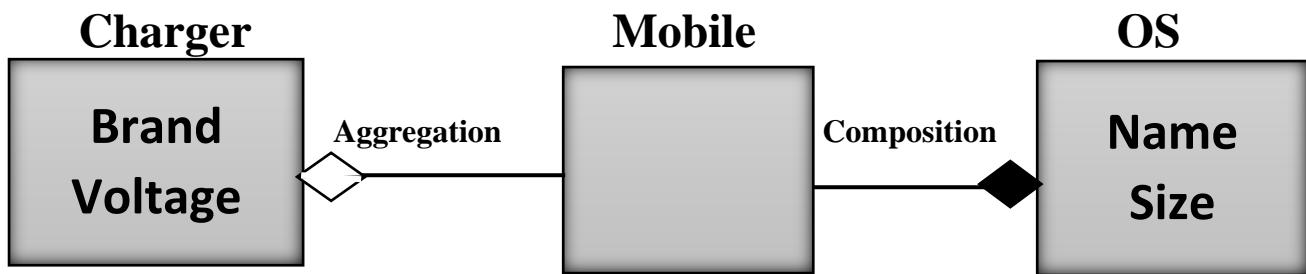
Example 3: In this example shown below, a phone contains an os and charger. Relationship between **phone and charger** is **aggregation** whereas relationship between **phone and os** is **composition**.

Charger can exist without **phone** and therefore it is aggregation. (**loosely bound**)

OS cannot exist without **phone** and therefore it is composition. (**tightly bound**)



Let us represent the third example in the form of UML diagram and start coding.



In the above UML Mobile has tight bound relationship with OS and loose bound relationship with charger. Charger has its own brand and voltage, similarly OS has its own name and size. Based on this information let us now start writing code.

CODE:

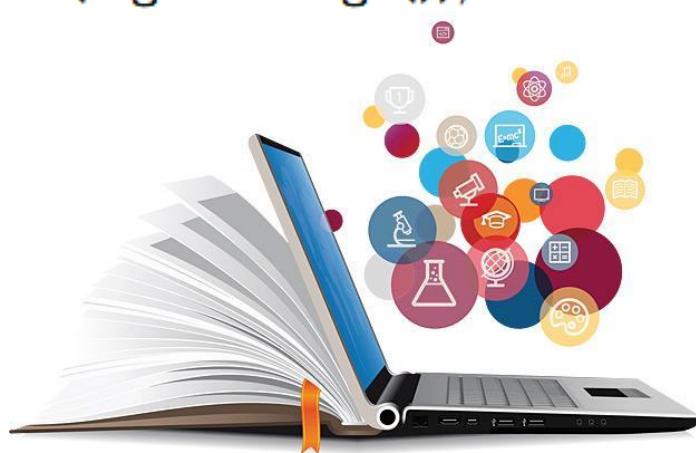
```
class Os
{
    private String name;
    private int size;
    Os(String name, int size)
    {
        this.name = name;
        this.size = size;
    }
    String getName()
    {
        return name;
    }
    int getSize()
    {
        return size;
    }
}
```

```

class Charger
{
    private String brand;
    private int voltage;
    Charger(String brand, int voltage)
    {
        this.brand = brand;
        this.voltage = voltage;
    }
    String getBrand()
    {
        return brand;
    }
    int getVoltage()
    {
        return voltage;
    }
}

class Mobile
{
    Os os = new Os("Android",512);
    void hasA(Charger c)
    {
        System.out.println(c.getBrand());
        System.out.println(c.getVoltage());
    }
}

```



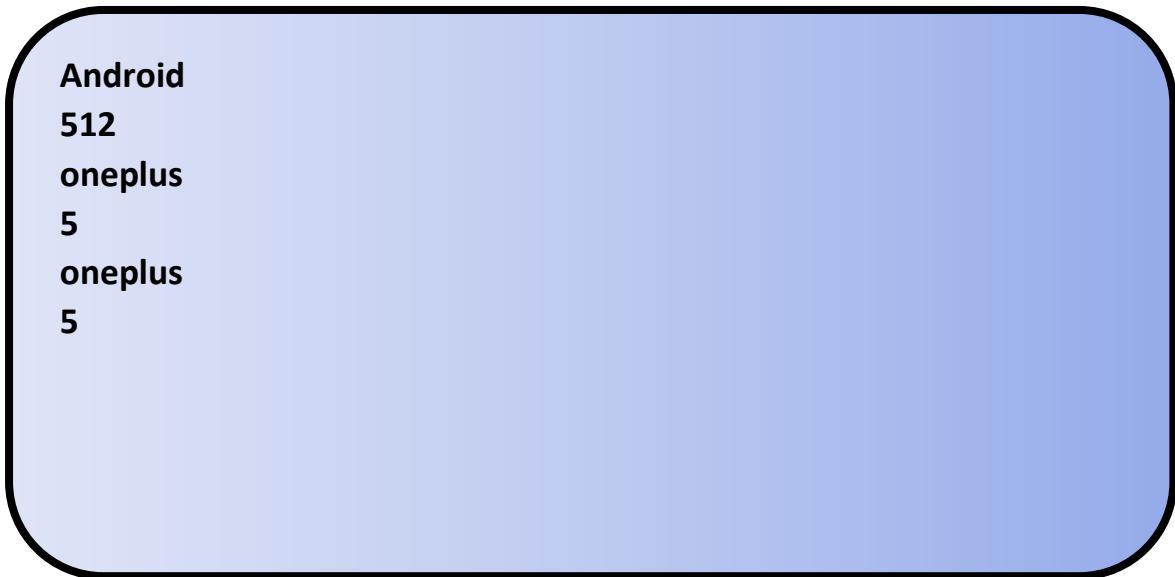
```

class Demo
{
    public static void main(String[] args)
    {
        Charger c = new Charger("oneplus",5);
        Mobile m = new Mobile();
        System.out.println(m.os.getName());
        System.out.println(m.os.getSize());
        m.hasA(c);
        m=null;
        //System.out.println(m.os.getName()); --->error
        //System.out.println(m.os.getSize()); --->error
        //m.hasA(c); --->error
        System.out.println(c.getBrand());
        System.out.println(c.getVoltage());
    }
}

```

In the above code if the **reference** to the **mobile** is made **null**, we **cannot access** **os** as os has **tight bound** relationship with mobile and if attempted we **get error**. But we can **access charger using charger type reference** as charger has **loose bound relationship with mobile**.

OUTPUT

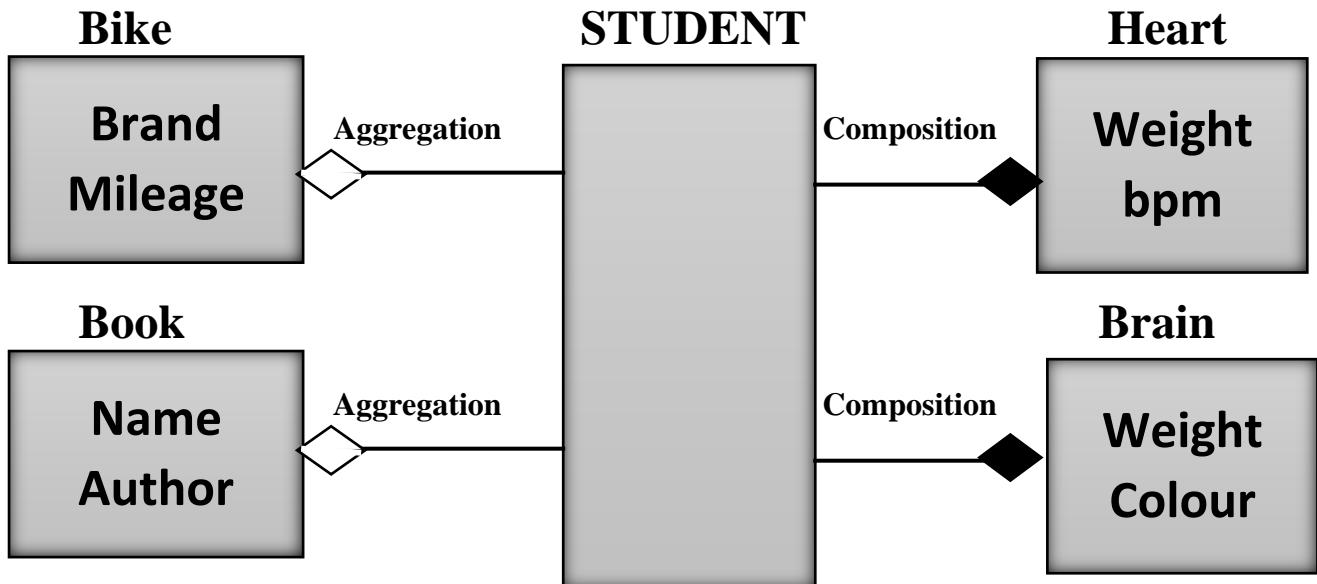


```

Android
512
oneplus
5
oneplus
5

```





In the above UML, student has tight bound relationship with brain and heart, and loose bound relationship with book and bike.

Bike Consists of **brand** and **mileage**.

Book consists of **name** and **author**.

Heart consists of **weight** and **bpm**.

Brain consists of **weight** and **colour**.

Based on the above information let us now start writing code:

CODE:

```

class Heart
{
    private int weight;
    private int bpm;
    Heart(int weight, int bpm)
    {
        this.weight = weight;
        this.bpm = bpm;
    }
    int getWeight()
    {
        return weight;
    }
    int getBpm()
    {
        return bpm;
    }
}
  
```

```
class Brain
{
    private int weight;
    private String colour;
    Brain(int weight, String colour)
    {
        this.weight = weight;
        this.colour = colour;
    }
    int getWeight()
    {
        return weight;
    }
    String getColour()
    {
        return colour;
    }
}

class Book
{
    private String name;
    private String author;
    Book(String name, String author)
    {
        this.name = name;
        this.author = author;
    }
    String getName()
    {
        return name;
    }
    String getAuthor()
    {
        return author;
    }
}
```

```
class Bike
{
    private String brand;
    private int mileage;
    Bike(String brand, int mileage)
    {
        this.brand = brand;
        this.mileage = mileage;
    }
    String getBrand()
    {
        return brand;
    }
    int getMileage()
    {
        return mileage;
    }
}

class Student
{
    Heart h = new Heart(289,72);
    Brain b = new Brain(1400,"gray");
    void hasA(Book book)
    {
        System.out.println(book.getName());
        System.out.println(book.getAuthor());
    }
    void hasA(Bike bike)
    {
        System.out.println(bike.getBrand());
        System.out.println(bike.getMileage());
    }
}
```

```

class Demo
{
    public static void main(String[] args)
    {
        Student s = new Student();
        Bike bike = new Bike("Duke",35);
        Book book = new Book("Java","JG");
        System.out.println(s.h.getWeight());
        System.out.println(s.h.getBpm());
        System.out.println(s.b.getWeight());
        System.out.println(s.b.getColour());
        s.hasA(bike);
        s.hasA(book);
        s=null;

        //System.out.println(s.h.getWeight()); ---->error
        //System.out.println(s.h.getBpm()); ---->error
        //System.out.println(s.b.getWeight()); ---->error
        //System.out.println(s.b.getColour()); ---->error
        //s.hasA(bike); ---->error
        //s.hasA(book); ---->error
    }
}

```

```

System.out.println(bike.getBrand());
System.out.println(bike.getMileage());
System.out.println(book.getName());
System.out.println(book.getAuthor());

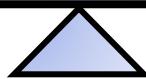
```

Can you guess the output?????



OUTPUT

```
289  
72  
1400  
gray  
Duke  
35  
Java  
JG  
Duke  
35  
Java  
JG
```



Difference between Aggregation and Composition:

Aggregation	Composition
Aggregation is a weak Association.	Composition is a strong Association.
Class can exist independently without owner.	Class can not meaningfully exist without owner.
Have their own Life Time.	Life Time depends on the Owner.
A uses B.	A owns B.
Child is not owned by 1 owner.	Child can have only 1 owner.
Has-A relationship. A has B.	Part-Of relationship. B is part of A.
Denoted by a empty diamond in UML.	Denoted by a filled diamond in UML.
We do not use "final" keyword for Aggregation.	"final" keyword is used to represent Composition.
Examples: - Car has a Driver. - A Human uses Clothes. - A Company is an aggregation of People. - A Text Editor uses a File. - Mobile has a SIM Card.	Examples: - Engine is a part of Car. - A Human owns the Heart. - A Company is a composition of Accounts. - A Text Editor owns a Buffer. - IMEI Number is a part of a Mobile.

Constructors

- 1. Can we have two constructors with the same name, same number of arguments and same type of arguments?**

Yes. Provided the order is different.

- 2 What happens if we don't explicitly provide a constructor in our class?**

Java compiler adds default constructor(zero parameterized constructor) inside which a call to super class constructor is present.

- 3 Who inserts a no argument constructor into our class?**

Java compiler.

- 4 When does the no argument constructor get inserted into our class?**

If the class does not contain any other constructor.

- 5 What is the no argument constructor also called as?**

Default constructor or zero parameterized constructor.

- 6 Whom does the no argument constructor of our class implicitly call?**

Super class constructor(Parent class constructor).

- 7 Can we use access modifier in a constructor declaration?**

Yes. Any access modifiers such as private , public , default and protected can be associated with a constructor.

- 8 What is the difference between parameters and arguments?**

Dog d = new Dog("Lobo", "Bulldog", 1234);

Lobo, Bulldog, 12324 are called as arguments (Actual Parameters).

public Dog(String name, String breed, int cost)

name, breed, cost are called as parameters (Formal Parameters).

9 What relation must hold good between parameters and arguments?

The number of arguments and the number of parameters should be same. Datatypes of arguments and parameters should be the same and also order should match.

10 Can we pass a method into another method as a parameter?

Yes.

11 How can we pass a method into another method as a parameter?

Using lambda expression.

12 When does a parameter shadow the class field?

When parameter names and instance variable's names are same then it would shadow the class field.

13 How can shadowing problem be resolved?

Using "this" keyword.

14 Is it compulsory to have a constructor within a class?

Yes. It is compulsory for a class to have a constructor. If the programmer does not manually provide a constructor, then java compiler automatically adds a default constructor(zero parameterized constructor).

15 Can we overload constructors?

Yes.

16 What is meant by constructor chaining?

The process of child class constructor calling its super class(parent class) constructor using super() method is called as constructor chaining.

17 What is the return type of a constructor?

Constructors do not have return type.

18 What does the constructor return?

It returns the address of an object which would normally be collected in a reference variable.

19 Does the compiler include default constructor if the class already contains a user defined constructor?

No.

20 How can local chaining of constructors be achieved?

Using this() method.

21 Which construct must be used to achieve local chaining? What is its limitation?

Using this() method local chaining can be achieved.

The limitation of this() method is that, in order to use this() method minimum two constructors must be present within a class. If this() method is used when only one constructor is present within a class, then it would result in a recursive call and leads to StackOverflow.

Eg:

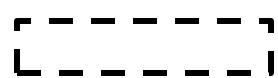
```
class Dog
{
    String name;
    String breed;
    int cost;

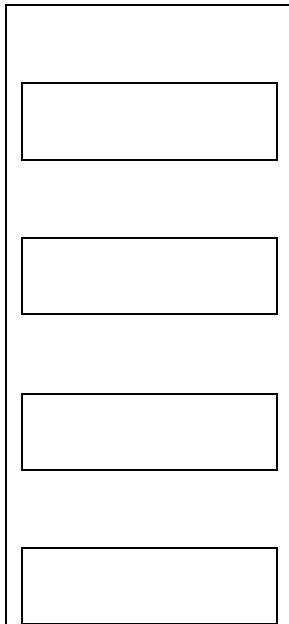
    Dog()
    {
        this(); //error
    }
}

class Launch
{
    public static void main(String args[])
    {
        Dog d = new Dog();
    }
}
```

```
}
```

Output:
Error

 Stack over flow



Dog()
Dog()
Dog()
main()

Stack segment

22 Does every constructor call its superclass constructor?

Yes. If the programmer has not explicitly called this() method within a constructor then automatically super() method would be present and hence super class constructor gets called.

23 When is an implied super() not included in each constructor?

If the programmer manually calls this() method, then super() method would not be included.

24Can super() have parameters in it?

Yes. However if super() method calls object class constructor then it is not possible.

25 If there are multiple constructors in the superclass then which constructor does super() call?

Default constructor of the parent class would be called.

26Why is a constructor called a constructor?

It is the constructor that converts an empty object into a meaningful object by assigning values to the instance variables. JVM only allocates memory for an object. It is the constructor that constructs the object by giving values, so it is called constructor.

27What is a role of default constructor?

The default constructor within its body would have a call to the super class constructor. The super class constructor would initialize the instance variables to default values.

28What is the difference between constructor and ordinary methods?

Constructors	Ordinary methods
Has the same name as that of the class.	Does not have the same name as that of the class.
Has no return type	Has return type
It is called during object creation.	It is called after object creation.

It is involved in construction of an object.	It is involved in exhibiting other behaviours.
Does not participate in inheritance.	Participate in inheritance.
super() and this() methods can be used only inside a constructors.	super() and this() methods cannot be used within an ordinary methods.
Constructors cannot be abstract.	Ordinary methods can be abstract.
Cannot be final	Can be final
Cannot be static	Can be static
Cannot be native	Can be native
Cannot be synchronized	Can be synchronized.

29 What are the operations that a constructor performs?

It normally performs initialization i.e construction of an object.

30 Can constructors be abstract?

No

31 Can constructors be final?

No

32 Can constructors be native?

No

33 Can constructors be static?

No

34 Can constructors be synchronized?

No

35 Can constructors be public?

Yes

36 Can constructors be private?

Yes. If in case it is made private then it would enforce singleton design pattern.

37 Can constructors be protected?

Yes

38 Can constructors be default?

Yes

39 What is the difference in the use of “this” by a constructor and an ordinary method?

“this” in a constructor plays two role.

- i) performs local chaining and
- ii) resolves shadowing problem.

“this” keyword in an ordinary method is used to resolve shadowing problem. However this() cannot be used in an ordinary method.

40 What is the difference in the use of “super” by a constructor and an ordinary method?

In ordinary method “super” keyword is used to access a super class variable which has same name as that of the subclass variables.

However super() cannot be used in an ordinary method.

In case of constructors “super” keyword is used to access a super class variable which has same name as that of the subclass variables and super() would be used for constructor chaining.

41 When we create a class such as public class Example{ } what extra additions would be performed by the compiler automatically?

public class Example extends Object

{

 public Example()

{

```
        Super();  
    }  
}  
public class Object  
{  
    public Object()  
    {  
        ---  
        ---  
    }  
}
```

42Can constructors be inherited?

No

43Can ordinary methods be inherited?

Yes

44How is constructor chaining achieved in java?

Using super() method.

45How are this() and super() used with constructors?

this() method is used for local chaining and super() method is for constructor chaining.

46What is the access modifier of the default constructor?

Default constructor would be having the same access modifier as that of the class to which it belongs to.

47What happens when a constructor is declared as private?

It results in singleton design pattern.

48 What is a constructor in java?

Constructor is a specialized setter which performs initialization of an instance variables that are present within an object.

49 Why do we require overloaded constructors in java?

The objects that are created for a class may demand different types of initialization.

Some objects may have to be initialized to their default values in which case, default constructor is required.

Some objects may have to be initialized to different values as required by the programmer for which parameterized constructors must be used. Therefore a class would normally contain overloaded constructor.

50 Does a class inherit the constructors of its superclass?

No

51 Can a top level class be private or protected?

It can't be both. It should be either public or default.

52 What type of parameter passing does Java support?

pass by value and pass by reference.

53 Primitive data types are passed by reference or pass by value?

pass by value.

54 Objects are passed by value or by reference?

pass by reference.

Inheritance

1. What is the advantage of inheritance?

Polymorphism, code re-usability and hence inheritance enhances profitability.

2. What is a parent class also called as?

Super class or base class.

3. What is a child class also called in java?

Derived class or sub class.

4. Which relationship must be satisfied to achieve inheritance in java?

“is-a” relationship.

5. What is the meaning of extends keyword in java and when is it used?

Meaning of extends keyword is “is-a”. “extends” keyword would be used during inheritance.

6. Can classes be circularly related in java?

No. Cyclic dependency is not permitted in java.

7. How do constructor execute in case of inheritance?

Constructors do not participate in inheritance. However, constructors would get executed through constructor chaining using super() method. This rule is made in order to preserve encapsulation.

8. What is the use of inheritance in java?

Polymorphism, code re-usability and hence inheritance enhances profitability.

9. Does java support multiple inheritance explain?

No. Because diamond shaped problem would be occurred which leads to ambiguity.

(Eg. Refer class notes)

10. Does inheritance promote “is-a” relationship or “has-a” relationship?

Inheritance promotes “is-a” relationship.

11. Why does java not support multiple inheritance? Explain with example?

No. Because diamond shaped problem would be occurred which leads to ambiguity.

(Eg. Refer class notes)

12. Can constructors be inherited in java?

No, constructors cannot be inherited in java. However, the control would go from the derived class to the base class constructor using super() method and constructor of the base class gets executed. Therefore, even though constructors are not inherited, yet they are executed

13. Can we reduce the visibility of the inherited method?

No. Either the visibility of the inherited method must be the same as that of the parent or it should be having higher visibility.

14. Is inheritance aggregation loose bound or tight bound?

Loose bound.

15. Is composition loose bound or tight bound?

Tight bound.

16. Is it compulsory for the method to be inherited for it to be overridden?

Yes.

17. Can we override a private method? Why?

No. Private method would never get inherited and hence can never be overridden.

18. What is a co-variant return type?

Co-variant return types are such return types which have “is-a” relationship between themselves. Java permitted co-variant return types from JDK 1.5 onwards.

(Eg. Refer class notes)

19.What actually happens in inheritance?

The features present in the parent would be inherited by the child. The child would override such features of the parent which it is not interested in and would also adds a few specialized features.

20.Using extends keyword can a subclass inherit all the properties of super class?

No. However, except private members and constructors, all other properties of the super class can be inherited by the sub class.

21. How is instanceof operator used in inheritance?

“instanceof” operator is used to test whether the object is an instance of the specified class or not. It returns true if the object belongs to the specified class.

22. Can a subclass have any number of superclasses?

No. Because multiple inheritance is not supported in java.

23.Can a superclass have any number of subclasses?

Yes.

24.Do both properties and behaviors get inherited in java?

Yes. However, private members and constructors do not get inherited.

25.Which is the superclass of all classes in java?

Object class.

26.Which is the superclass of Object class in java?

Object class is the top most super class. There is no super class above it.

27.Does java support multi-level inheritance?

Yes.

28.In which package is the Object class present?

java.lang package.

29.What does the Object class contain?

Object class contains number of methods which are required for all the sub classes such as **toString()**, **hashCode()**, **getClass()**, **finalize()** etc.

30.In the class hierarchy which classes are more generalized in behavior?

In the hierarchical design, the classes present above in the hierarchy are generalized and as we flow down in the hierarchy they become more and more specific.

31.In the class hierarchy which classes are more specialized in behavior?

In the hierarchical design, the classes present above in the hierarchy are generalized and as we flow down in the hierarchy they become more and more specific.

32.Can we have a method in a subclass with a same name as in the superclass?

Yes. It is called as overriding.

33.Can we have a variable in a subclass with a same name as in the superclass?

Yes.

34. Can we declare new fields in the subclass which are not there in the superclass?

Yes. Such fields are called as specialized fields.

35.Why does not java permit constructor inheritance?

It is not permitted in order to preserve encapsulation.

36.What is the difference between hiding, overriding and shadowing?

Hiding refers to the process of programming the objects such that other objects cannot directly access the most important components of the current objects. Though direct access is prevented, controlled access is permitted through setters and getters. It is also referred to as encapsulation.

Overriding refers to the process of the subclass inheriting a method of the super class and modifying it to suit its specific needs.

Shadowing refers to the name clash that occurs when the local variable names and the instance variable names are the same. It can be overcome using “this” keyword.

37.Can we declare new methods in the subclass which are not there in the superclass?

Yes. Such methods are called specialized methods.

38.Can a subclass access private members of the superclass directly?

No.

39.What are the ways in which private members of a superclass can be accessed?

It can be accessed through public setters and getters.

40. What is not possible in java inheritance?

- i) Private inheritance is not possible.
- ii) Constructor inheritance is not possible.
- iii) Multiple inheritance is not possible.
- iv) Cyclic inheritance is not possible.

41.Comment on the visibility of private?

Visible only within the given class.

42. Comment on the visibility of protected?

Visible in the current package and also visible to the classes of other packages provided the classes in the other packages are sub classes of the current class.

43. Comment on the visibility of default?

Accessible in the current package.

44. Comment on the visibility of public?

Accessible throughout the project.

45. What is an alternative to inheritance?

Delegation model.

46. How is delegation better when compared to inheritance?

Though inheritance has many advantages, yet one of the limitations is that the subclass would be forced to inherit all the methods of the super class irrespective of whether the subclass requires them or not. If the subclass requires only few methods of the super class, then such an arrangement is not possible in inheritance. It can be achieved using delegation model.

However, since delegation model is not a hierarchical model, loose coupling cannot be achieved and hence polymorphism cannot be achieved.

47. Which relationship does inheritance implement?

“is-a” relationship.

48. Who promotes “is-a” relationship?

Inheritance.

49. Who promotes “has-a” relationship?

Aggregation and composition.

50. What is meant by implicit typecasting?

The process of placing a smaller magnitude of data into a larger data type automatically by the compiler without programmer intervention is called as implicit typecasting. It is also called as numeric promotion.

Eg. byte can automatically converted to short type.

51.What is meant by downcasting?

(Refer class notes)

52.What is meant by upcasting?

(Refer class notes)

53.How many levels above the current level would the super keyword enable to access in case of inheritance?

“super” keyword enables to access fields only one level above in the hierarchy.

54. Which constructor of the superclass gets called automatically from the subclass?

Default constructor.

55. Can we call the parameterized constructor of the subclass from the baseclass?

No.

56. Can a subclass somehow access the private members of the superclass?

Yes. Through public setters and getters.

57.What is constructor chaining?

It refers to the process of the subclass constructor calling its super class constructor by using super() method.

58.Can a parent reference point to the child object?

Yes.

59.What are the advantages of parent reference to the child object?

It results in loose coupling and hence enables to achieve polymorphism. Through this flexibility and code reduction also can be achieved.

60.What is meant by loose coupling?

Creating a parent reference to point to child object is called as loose coupling.

61.What is meant by tight coupling?

Creating a child reference to point to child object is called as tight coupling.

62.Is parent reference to child object loose coupling or tight coupling?

Loose coupling.

63. What is the limitation associated with parent reference to child object?

Using such a parent reference only the overridden methods present in the child class can be accessed directly. The specialized methods present in the child cannot be accessed directly.

64. How can the limitation of parent reference to child object be overcome?

By downcasting.

65.How can we prevent the inheritance of a class?

By making the class as final.

66.Why is cyclic dependency not permitted in java?

Because it does not exists in real life.

67.What are the advantages of method overriding?

Overriding enables the subclass to redesign the method to suit its specific requirements.

68.Why are private members not inherited?

To preserve the concept of encapsulation.

69.Why are constructors not inherited?

To preserve the concept of encapsulation.

70. When a derived class object is created, why does the constructor of base class get called?

Due to constructor chaining.

71.What is meant by diamond shape problem?

Multiple inheritance. Any project design must not entertain diamond shape problem since it would lead to ambiguity.

72.Does java support multi-level inheritance? Why?

Yes. Because it is existing in real life.

73. How is has-a relationship implemented in java?

Using composition, aggregation and delegation

74.What is composition?

It refers to tight bound has-a relationship.

75.What is aggregation?

It refers to loose bound has-a relationship.

76. Can we access the composite object while the enclosing object is accessible?

Yes.

77.Can we access the aggregate object while the enclosing object is accessible?

Yes.

78.Can we access the composite object while the enclosing object is not accessible?

No.

79.Can we access the aggregate object while the enclosing object is not accessible?

Yes.

80.Can you identify composite objects in this interview room?

Yes. Candidate has a heart, has a knowledge etc.

81.Can you identify aggregate objects in this interview room?

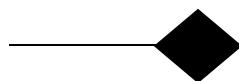
Yes. Candidate has a resume, has a book, has a pen etc.

82.Which classes in the UML diagram must be made final?

The leaf nodes.

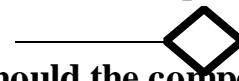
83.Which notation is used to represent composite objects in UML?

Diamond shape with coloring inside (shaded rhombus)



84.Which notation is used to represent aggregate objects in UML?

Diamond shape without coloring.



85.Why should the composite objects get destroyed when the enclosing object is destroyed?

Because that is the way real life objects behave.

86.Why should the aggregate objects not get destroyed when the enclosing object is destroyed?

Because that is the way real life objects behave.

87.Can we represent both “is-a” and “has-a” relationship in a single UML?

Yes.

88.Does the delegation model implement “is-a” relationship?

Not exactly. However, it can be used as an alternative to inheritance.

89.Does the delegation model implement “has-a” relationship?

Yes.

90.What happens if the class is made final?

It cannot be further inherited.

91.What happens if the method is made final?

It cannot be overridden.

92.What happens if the variable is made final?

It behaves like a constant.

93.Can you name an inbuilt class which is final?

String class

94.What is the casting from a general to a more specific type called as?

Downcasting.

95.What is the casting from a specific to a more general type called as?

Upcasting

96.What is the difference between primitive numeric type casting and casting between object references?

In primitive numeric typecasting we have two types of casting namely implicit typecasting and explicit typecasting. On object references, we do not have implicit and explicit typecasting, rather we have upcasting and downcasting concepts.

Static Keyword

1) What are the different types of static elements that can exist in a class?

Static variables, static blocks and static methods.

2) When should we declare a variable as static?

When a common copy of the value is required for all the objects of the class.

3) When should we create static blocks?

To initialise static variables and to execute a set of statements even before the execution of main() method.

4) When should we declare a method as static?

When the method is a utility method or generic method. When the method in real life is callable without any object.

5) When should we declare a variable as non static?

When we require a separate values for each variables of an object, then we should declare non static variables.

6) Why should we create non static blocks?

To initialize non static data and it also enables to count the number of objects of a class.

7) When should we declare a method as non static?

If it is not a utility method or if it is a specific method or if it is such a method which has to be accessed through a reference.

8) Can static method access non static data? Why ?

No. Because memory would not have been allocated for non static variables, no values would be present inside them.

9) Can static block access non static data? Why ?

No. Because memory would not have been allocated for non static variables, no values would be present inside them.

- 10) Can static method access static data? Why?**
Yes
- 11) Can static block access static data? Why?**
Yes
- 12) Can non static method access non static data? Why ?**
Yes
- 13) Can non static block access non static data? Why ?**
Yes
- 14) Can non static method access static data? Why?**
Yes
- 15) Can non static block access static data? Why?**
Yes
- 16) Who initializes static data to default value?**
JVM.
- 17) Who initializes non static data?**
JVM
- 18) When does the static block get executed?**
Soon after the memory allocation of static variables , before the execution of main() method.
- 19) When does the non static block get executed?**
After new operator creates an object and before calling the constructor, non static block gets executed.

20) From where would the static methods be loaded into the stack?

From the static segment.

21) From where would the non static methods be loaded into the stack?

Code segment

22) What is a class loader?

It is the part of JVM which loads all the static elements from code segment onto the static segment. The roles of class loader are,

- i) Loads all the static variables on static segment.
- ii) Loads static block onto the segment and executes it.
- iii) Loads static methods from code segment onto the static segment. However, class loader would not execute static methods. Rather static methods would be executed when the control enters into the method

23) Why should the main() be declared static?

If the main() method is not declared as static then class loader cannot load the main() method into the static segment. JVM after receiving the control from the class loader would automatically searches for the main() method in the static segment. If it is not present in static segment, then JVM cannot handover the control to the main() method and load it onto the stack segment. In other words the application never gets launched.

24) List practical applications of static keyword?

- i. Aadhar card-nationality
- ii. for students-university
- iii. In boys' school-gender
- iv. Institute-fees

25) I want to print "Hello" even before main() is executed. How can I achieve that?

If that message is present within the static block, then it would get executed even before the main() method.

Example:

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

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

Output

Hello
Inside main

26) Can we have a static variable and a non static variable with the same name in a given class?

No

27) Can a class be declared as static?

Outer class cannot be declared as static. However, inner class (nested class) can be declared as static.

28) Can we use this keyword on a static variable?

No

29) Can we use this keyword on a non static variable?

Yes

30) Can we declare a static variable inside a static method?

No, if static variables declared inside the method, then it would become local variable and memory would be allocated only when the control

enters into the method and memory would be deallocated once the control leaves the method. IF the memory for static variable is deallocated in this manner then all the objects of the class cannot share the value of it. Memory for static variable has to be deallocated only after program terminates its execution.

31) Can we declare a static variable inside a non-static method?

No

32) What is the difference between class variables, instance variables and local variables?

For class variables, memory would be allocated only once.in other words only one copy of class variables would be created. All the objects of the class can access the same copy.

For instance variables, memory would be allocated separately for each object i,e. one copy per object.

Local variables would not be initialized automatically.

33) What happens to a static variable that is defined within a method of a class?

It cannot be created. Error would occur.

Example:

```
class Launch
{
    public static void main(String[] args)
    {
        static int a=10; // error
        System.out.println(a);
    }
}
```

34) How many static initializers can you have and in what order do they get executed?

We can have any number of static initializers (static blocks).they would get executed in the same order in which they are created .

35) Can main() method be overloaded?

Yes, provided the signature is different if they are present in the class. however, if they are present in different classes we can have multiple main methods with the same signature.

(Eg. Refer class notes)

36) What is the main role of static keyword?

It helps in Memory management

37) Can we apply static keyword on variables?

Yes

38) Can we apply static keyword on methods?

Yes

39) Can we apply static keyword on blocks?

Yes

40) Can we apply static keyword on class?

No, however, for inner class (nested class) static keyword can be apply.

41) Can we apply static keyword on nested class?

Yes

42) Can we apply static keyword on package?

Yes

43) Can we apply static keyword on constructor?

No

44) What is a static variable also known as?

Class variable

45) What is a static method also known as?

Class method or utility method or Generic method.

46) How can we count the number of objects that are created of a class?

(refer class notes)

47) Can we call a static method without using an object?

Yes, by using class name.

48) Can we call a static method using an object?

Yes

49) Can we call a non static method without using an object?

No

50) Can we call a non static method using an object?

Yes

51) Can we execute a program without main()?

NO

52) What is the difference between static variable and non static variable?

Static variable	Non static variable
Also called as class variable	Also called as instance variable
Memory is allocated in the static segment	Memory is allocated on heap segment
Memory would be deallocated only when the program terminates its execution	Memory would be deallocated when the object does not have any reference referring to it

A single copy would be created and all the objects have to share the single copy	A copy is created for each object of the class.
“this” keyword should not be used	“this” keyword can be used

53) What would happen if a top level class in java is declared as static?

Error would occur.

54) When are static variables initialized

Once class loader loads static variables from load code segment to static segment, static variables would get initialized.

55) When are non static variables initialized?

Once objects are created, during call to constructor, non static variables would be initialized.

56) Can static methods be overloaded?

Yes

57) What is the problem when static keyword is used in multithreaded environment?

Race condition occurs.

58) What are the disadvantages with static methods?

Static methods can not be overridden. They can not be used effectively in multithreaded environment.

59) Which design pattern can be implemented through static method?

Factory design pattern.

60) Name a few static classes in java?

Wrapper classes, java.lang.Math, java.lang.Collections etc.

- 61) Which keyword must be used along with static for creating constants?**

“final” keyword.

- 62) What is the use of nested static class?**

Packaging convenience.

- 63) What are the restrictions imposed on static methods?**

Static methods cannot access non static data and cannot use this.

- 64) How can “one per class” policy enforced in java?**

Using “static” keyword.

- 65) What is the difference between nested classes and inner classes?**

If inner class is declared as static then it becomes nested.

- 66) Can the nested classes have access to the other members of the enclosing class?**

No, but inner classes can access.

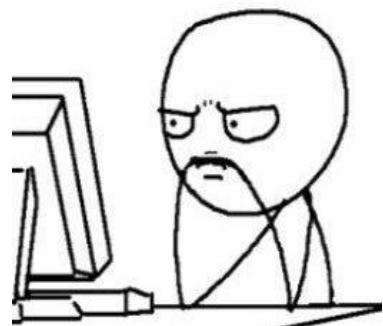
- 67) Can the inner class have access to the other members of the enclosing class?**

Yes

- 68) What are the advantages of nested classes?**

Increases encapsulation, makes packaging more streamlined, makes code readable and maintainable.

Wondering why do we need the body for the methods that have been overridden in subclasses..?? Well let's see how to get rid of it without affecting inheritance or polymorphism.



Abstraction in Java

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are **hidden from the user**. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.



Likewise in Object-oriented programming, **abstraction** is a process of **hiding the implementation details from the user**, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In simpler words, data **abstraction** is the process of **hiding certain details and showing only essential information** to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next couple of classes).

The **abstract** keyword is a **non-access modifier**, used for classes and methods:

- **Abstract class:** is a restricted class that **cannot be used to create objects** (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and **it does not have a body**. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods.

Example

```
abstract class Plane
{
    abstract void takeOff();
    abstract void fly();
    abstract void land();
}
class CargoPlane extends Plane
{
    void takeOff()
    {
        System.out.println
            ("CargoPlane is taking off from a long sized runway.");
    }
    void fly()
    {
        System.out.println
            ("CargoPlane is flying at low heights.");
    }
    void land()
    {
        System.out.println
            ("CargoPlane is landing on long sized runway.");
    }
}

class PassengerPlane extends Plane
{
    void takeOff()
    {
        System.out.println
            ("PassengerPlane is taking off from a medium sized runway.");
    }
    void fly()
    {
        System.out.println
            ("PassengerPlane is flying at medium heights.");
    }
    void land()
    {
        System.out.println
            ("PassengerPlane is landing on medium sized runway.");
    }
}
class FighterPlane extends Plane
{
    void takeOff()
    {
        System.out.println
            ("FighterPlane is taking off from a short sized runway.");
    }
    ..
```



www.clipartof.com · 1246277

```

void fly()
{
    System.out.println
        ("FighterPlane is flying at great heights.");
}
void land()
{
    System.out.println
        ("FighterPlane is landing on short sized runway.");
}
}

class Airport
{
    void permit(Plane ref)
    {
        ref.takeOff();
        ref.fly();
        ref.land();
    }
}
class Demo
{
    public static void main(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();

        Airport a = new Airport();
        a.permit(cp);
        a.permit(pp);
        a.permit(fp);
    }
}

```



Output

CargoPlane is taking off from a long sized runway.
CargoPlane is flying at low heights.
CargoPlane is landing on long sized runway.
PassengerPlane is taking off from a medium sized runway.
PassengerPlane is flying at medium heights.
PassengerPlane is landing on medium sized runway.
FighterPlane is taking off from a short sized runway.
FighterPlane is flying at great heights.
FighterPlane is landing on short sized runway.

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved with Interfaces, which you will learn more about in the next chapter.

But 'Why'?

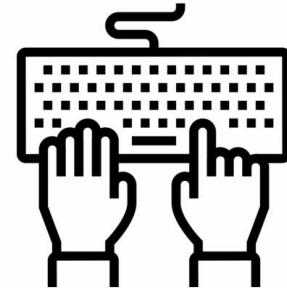


Example

```
import java.util.Scanner;
abstract class Shape
{
    float area;
    abstract void acceptInput();
    abstract void calcArea();
    void dispArea()
    {
        System.out.println(area);
    }
}
class Square extends Shape
{
    float side;

    void acceptInput()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the length of the side:");
        side = scan.nextFloat();
    }
    void calcArea()
    {
        area = side*side;
    }
}
class Rectangle extends Shape
{
    float length;
    float breadth;

    void acceptInput()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the length:");
        length = scan.nextFloat();
        System.out.println("Enter the breadth:");
        breadth = scan.nextFloat();
    }
    void calcArea()
    {
        area = length*breadth;
    }
}
```



```

class Circle extends Shape
{
    float radius;

    void acceptInput()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the radius:");
        radius = scan.nextFloat();
    }
    void calcArea()
    {
        area = 3.141f*radius*radius;
    }
}
class Geometry
{
    void permit(Shape ref)
    {
        ref.acceptInput();
        ref.calcArea();
        ref.dispArea();
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Square s = new Square();
        Rectangle r = new Rectangle();
        Circle c = new Circle();

        Geometry g = new Geometry();
        g.permit(s);
        g.permit(r);
        g.permit(c);
    }
}

```

Output

Enter the length of the side:
5
25.0
Enter the length:
10
Enter the breadth:
20
200.0
Enter the radius:
5
78.525



Hierarchy of abstraction in java

Anything whose context is not clear is said to be abstract.

Let's relate to the real world example, when you were a baby you didn't know anything about how your future, career, schooling anything. But when you are 10 years old you'll have some idea about what you want to be based on the influence of people you are surrounded with. But you still aren't clear with what measures you take to achieve that and why you need to be that is still not clear. So many things are still abstract and not concrete yet. A decade passes by and now you are in 20's and you might have completed your education, but still not clear with where you'll be working, whom you'd marry etc. So still there are not some things which are abstract but the level of abstraction is reduced. And once you are 30, now you know where you are working, your designation and whom you'll be married to. So generally now many things are concrete and abstraction is reduced.



Similarly in java also as you come down the hierarchy all methods and class are more concrete and less abstract.

Let's try to understand with code

```
abstract class Bird
{
    abstract void eat();
    abstract void fly();
}
abstract class Eagle extends Bird
{
    void fly()
    {
        System.out.println("Eagle flies at great heights");
    }
}
class SerpentEagle extends Eagle
{
    void eat()
    {
        System.out.println("Serpent Eagle hunts over mountains and eats"
    }
}
```



www.clipartof.com · 1246277

```

class SerpentEagle extends Eagle
{
    void eat()
    {
        System.out.println("Serpent Eagle hunts over mountains and eats");
    }
}
class GoldenEagle extends Eagle
{
    void eat()
    {
        System.out.println("Golden Eagle hunts over mountains and eats");
    }
}
class Demo
{
    public static void main(String[] args)
    {
        SerpentEagle se = new SerpentEagle();
        GoldenEagle ge = new GoldenEagle();

        se.eat();
        se.fly();
        ge.eat();
        ge.fly();
    }
}

```



Output:

```

Serpent Eagle hunts over mountains and eats
Eagle flies at great heights
Golden Eagle hunts over mountains and eats
Eagle flies at great heights
Press any key to continue . . .

```

Key Points

- An abstract class can contain both abstract methods as well as concrete methods.
- A normal class can inherit from an abstract class.
- An abstract class can inherit from another abstract class.
- An abstract class can inherit from a normal class.
- An abstract class can contain only concrete methods as well.

If a class contain even a single abstract method then the class should be declared as abstract.

Abstract class objects cannot be created/instantiated. If you want to create it then it must be inherited from another class.



Final keyword in java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a **final variable that have no value it is called blank final variable or uninitialized final variable**. It can be **initialized in the constructor only**. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

Final variable

If you make any variable as final, you **cannot change the value** of final variable (It will be constant).

Example:

```
class Test1
{
    final int a=100;
}
class Demo
{
    public static void main(String[] args)
    {
        Test1 t1 = new Test1();
        System.out.println(t1.a);
        t1.a=999; //<---Error
        System.out.println(t1.a);
    }
}
```

Output: Demo.java:11: error: cannot assign a value to final variable a
t1.a=999; //<---Error
^
1 error
Press any key to continue . . .

final class

If you make any method as final, you **cannot override** it.

Example:

```
class Test1
{
    final void fun()
    {
        System.out.println("Inside parent class method");
    }
}

class Test2 extends Test1
{
    void fun()
    {
        System.out.println("Inside child class overridden method");
    }
}

class Demo
{
    public static void main(String[] args)
    {
        Test2 t2 = new Test2();
        t2.fun();
    }
}
```

Output:

```
Demo.java:11: error: fun() in Test2 cannot override fun() in Test1
    void fun()
               ^
  overridden method is final
1 error
Press any key to continue . . .
```

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

Example:



```

final class Test1
{
    void fun()
    {
        System.out.println("Inside parent class method");
    }
}

class Test2 extends Test1
{
    void fun()
    {
        System.out.println("Inside child class overridden method");
    }
}

class Demo
{
    public static void main(String[] args)
    {
        Test2 t2 = new Test2();
        t2.fun();
    }
}

```

Output:

```

Demo.java:9: error: cannot inherit from final Test1
class Test2 extends Test1
^
1 error
Press any key to continue . . .

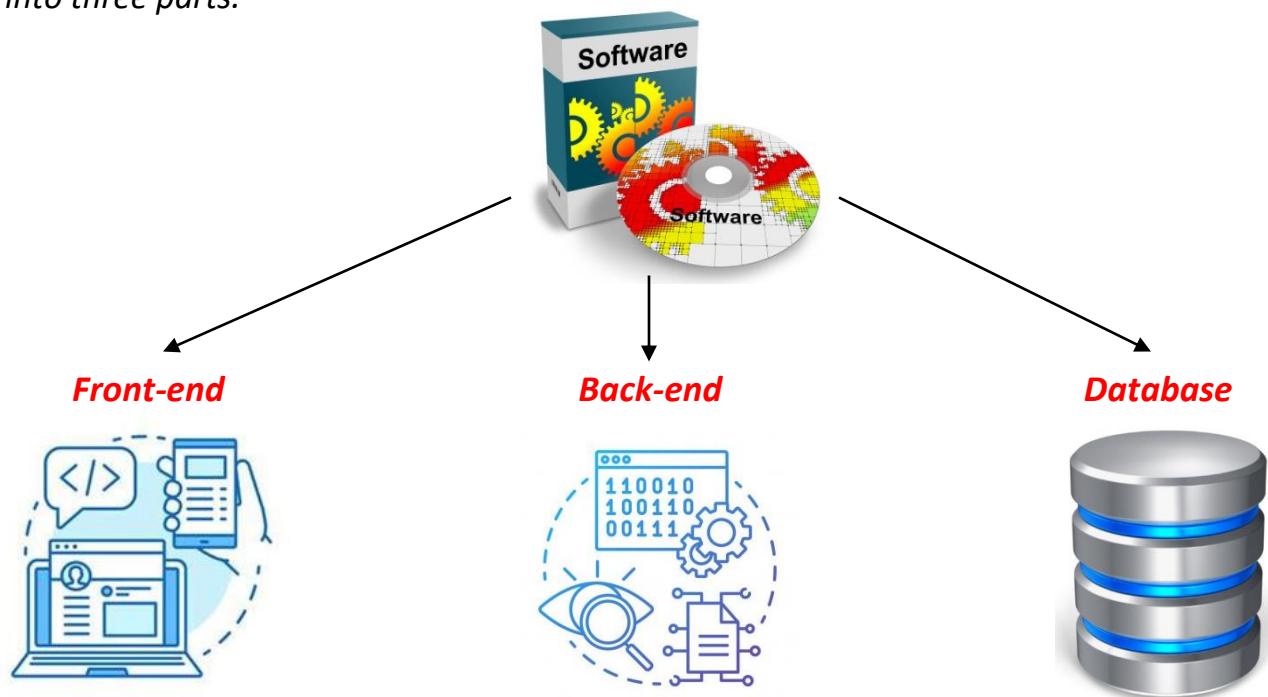
```



INTERFACE

Before getting directly into the definition of Interface, let us understand it through an example.

Let us consider a software, from the software engineer's view it is internally divided into three parts.



Front-end Development is responsible for implementing visual elements that users see and interact with in a web application.

A front-end is like a dummy, it makes software to look good, you can type things but it is not capable of execution.

To create Front-end few technologies used are:



It is in the **Back-end** all the execution and processing happens with the help of programming languages. This is not visible to the user.

Backend development works in tandem with the front end to deliver the final product to the end user.

To create Back-end few technologies used are:



A **Database** is a collection of information that is organized so that it can be easily accessed, managed and updated.

Code written by back end developers is what communicates the database information to the browser. To take care of databases there are database Management software's.

The few Database Management Systems software's are:

ORACLE

Informix



The Front-end interacts with the back-end; the back-end understands the requirement of the front-end and supports it. If there is need of some data then back-end interacts with Database takes the data, processes it and gives back to the front-end. Therefore, software works with the combination of all three.

Now you might be wondering what these have to do with Interface??



Initially Java didn't have the facility to connect to the database. It was in 1997, a feature in Java was introduced using which a Java program can be connected to the Database Management Software. And the technology which gave Java as a programming language the ability to interact/connect to Database Management Software is what called as "Java Database Connectivity".

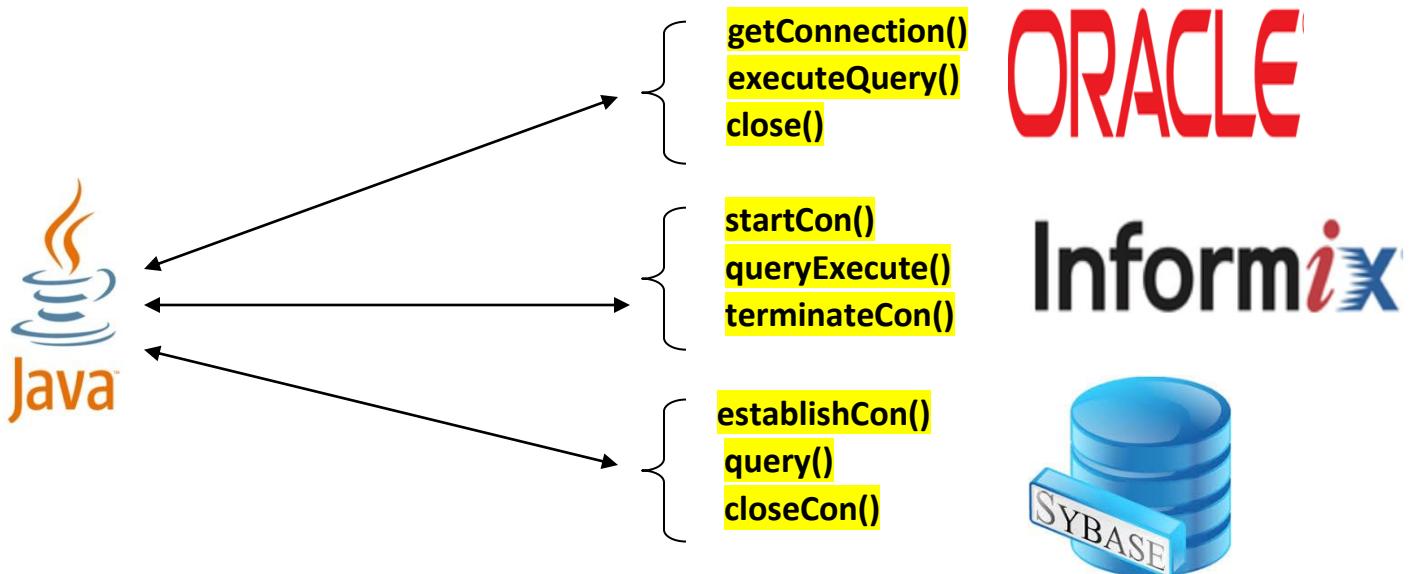
If a programmer through a Java program wants to connect to the database what should be the method name? if he/she wants to send a query to the database what should be method name? Once all the execution is done, to terminate the connection what should be the method name??



Query is a request for data or information from a database.

Databases store data in a structured format, which can be accessed using queries and the special type of language which is used for this purpose is called as **Structured Query Language(SQL)**.

So as a solution for above question each Database Management Software came up with their own method names.



But this is not the good approach of programming.

Well, let me explain you why?

Imagine a Java programmer has got a project which had to be worked in Oracle Database, he remembers the method names for establishing the connection, execute the query, many other method names as per the requirement and finally closes the connection and completes the project.

Now assume, he got another project but now in which he has to work with Informix Software.

Now don't you think the method names for this database software is different and he has to remember all those methods too.



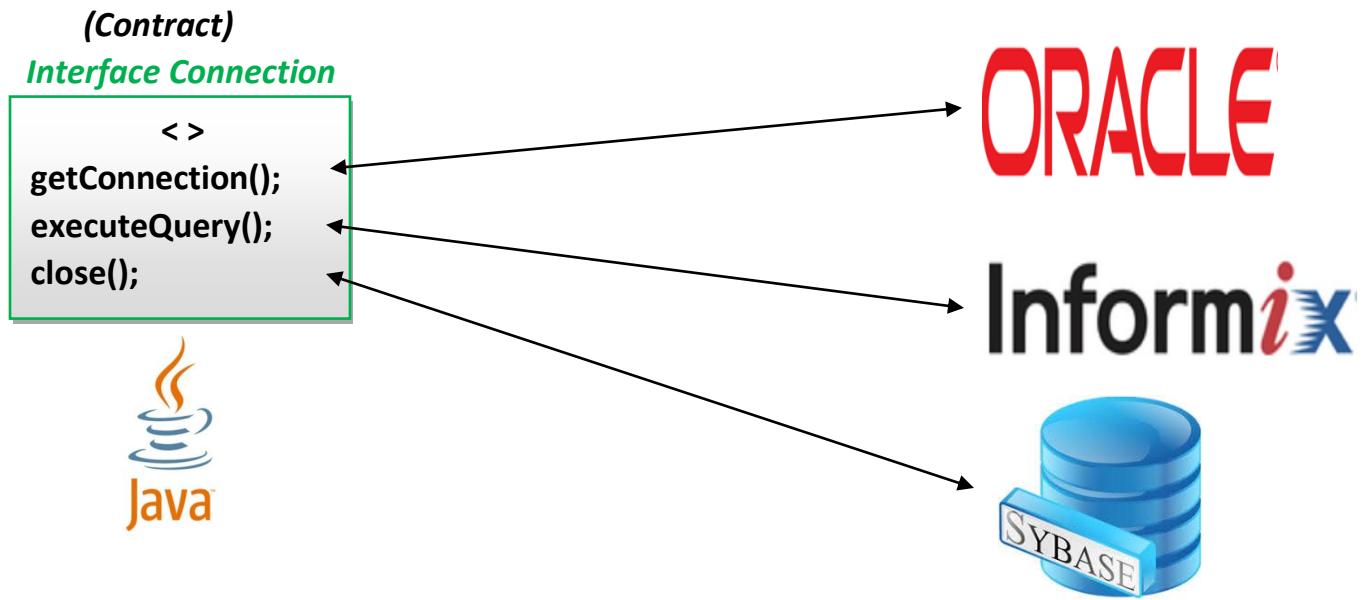
Therefore, Java giving the flexibility to the Database Management Software to decide their own methods the, Java programmer were put into trouble. Because every time the Database software changes the programmer had to remember different method names.

So as a solution Java developed a common methods names irrespective of what the Database software it is.

But how???



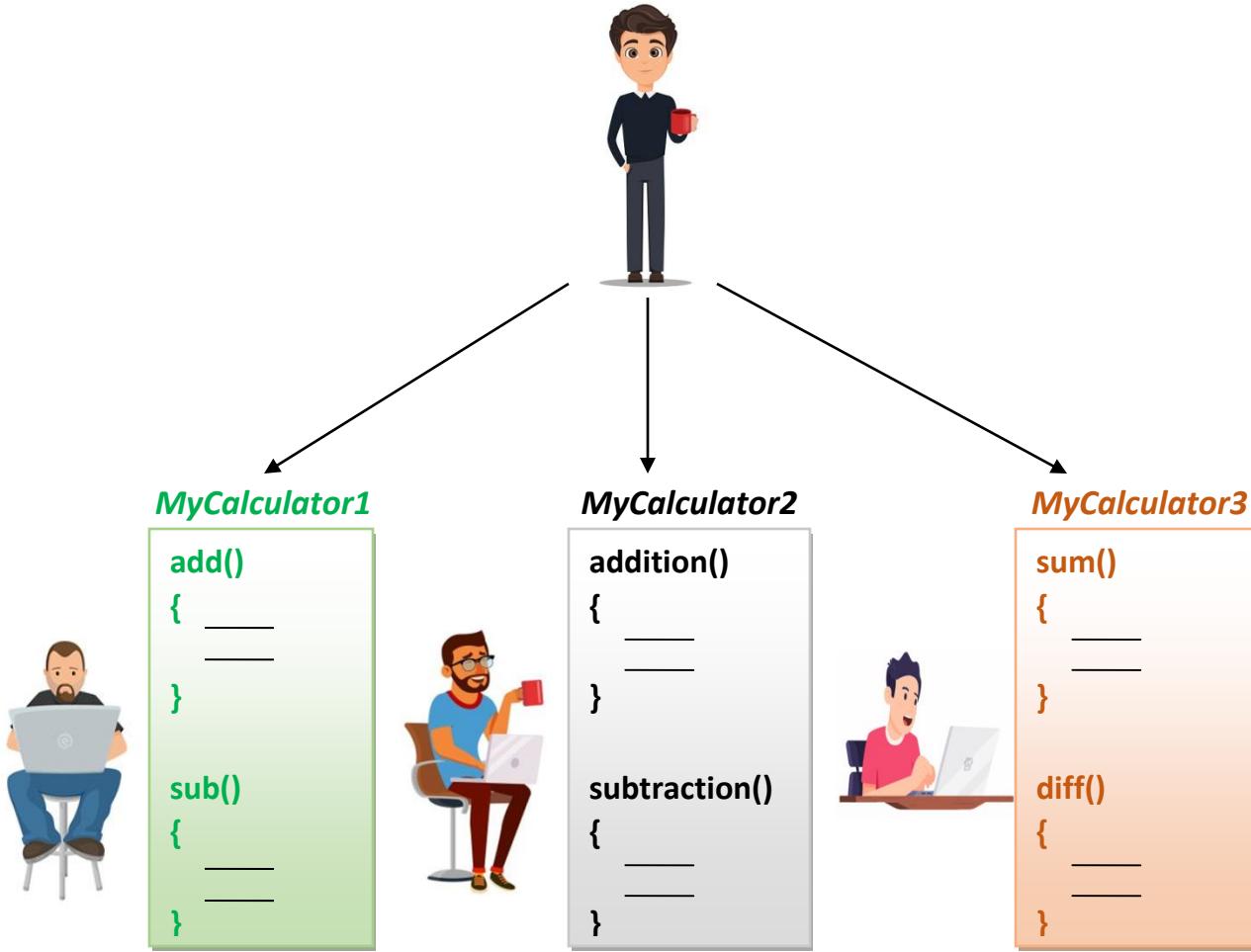
Java came up with a contract in which the method names would be mentioned and the Database software's had to use the same method names as in the contract.



An Interface is like a contract which when implemented helps achieves standardization. Whatever mentioned in the contract must be used as it is without any changes. An interface can contain any number of methods.

Let us see a scenario which explains Interface.

Let us consider that a function of addition and subtraction is to be performed for a user. Therefore, let us imagine that there are three developers who came up with three different classes which has different method names to perform the same function of addition and subtraction, like has shown in the below representation.



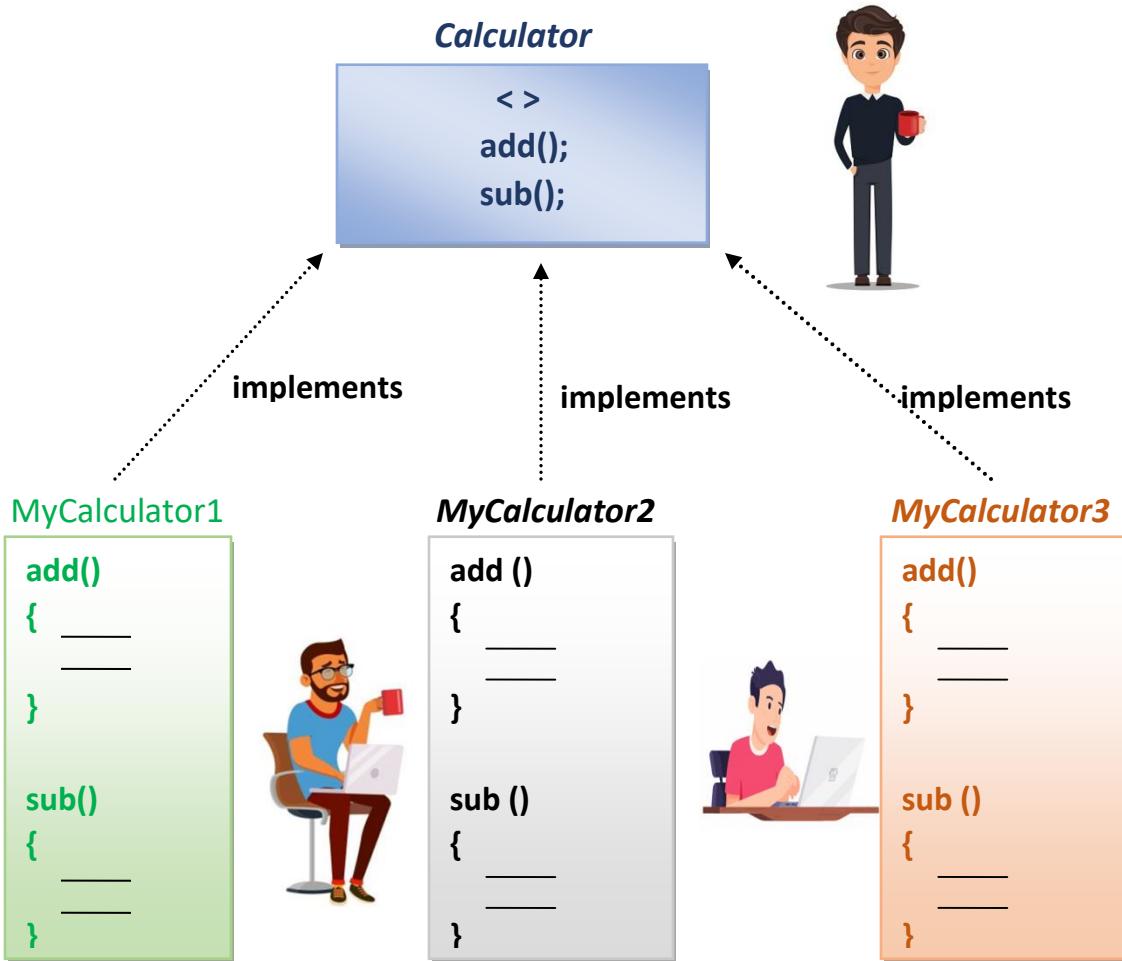
The problem with this approach was, to perform the same function as addition and subtraction three different types of methods have been used. The difficulty here is for the user to remember different names of the methods and the user friendliness misses.

That is, if a developer creates the method names it is a problem so the best would be to create a contract and mention the method names that have to be used by all three developers with the body of the method written as per their requirement.

Hence they came up with the contract/ Interface.

Interfaces are represented with rectangular box with angular braces inside it.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.



implements means MyCalculator classes will give body to the methods of add() and sub() of Calculator interface but will never change the name of the methods.

The logic of operations of three developers may be different but the method names must be same.

Note:

Interface promotes polymorphism. An interface type reference can point to implementing class objects. This achieves loose coupling and hence code reduction and code flexibility.

Methods within an interface are automatically public abstract.

Let us now write code to understand interface:

```
import java.util.Scanner;
interface Calculator
{
    void add();
    void sub();
}

class MyCalculator1 implements Calculator
{
    public void add()
    {
        int a = 20;
        int b = 10;
        int c = a+b;
        System.out.println(c);
    }
    public void sub()
    {
        int a = 20;
        int b = 10;
        int c = a-b;
        System.out.println(c);
    }
}
class MyCalculator2 implements Calculator
{
    public void add()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        int c = a+b;
        System.out.println(c);
    }
    public void sub()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        int c = a-b;
        System.out.println(c);
    }
}
```



```
class MyCalculator3 implements Calculator
{
    public void add()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        if(b == 0)
        {
            System.out.println("Second number is zero");
        }
        else
        {
            int c = a+b;
            System.out.println(c);
        }
    }
    public void sub()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        if(b == 0)
        {
            System.out.println("Second number is zero");
        }
        else
        {
            int c = a-b;
            System.out.println(c);
        }
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyCalculator1 mc1 = new MyCalculator1();
        MyCalculator2 mc2 = new MyCalculator2();
        MyCalculator3 mc3 = new MyCalculator3();

        mc1.add();
        mc1.sub();
        mc2.add();
        mc2.sub();
        mc3.add();
        mc3.sub();
    }
}
```

Abstract Classes

1. What is an abstract class?

An abstract class is a class which may or may not contain abstract methods.

2. What is an abstract method?

An abstract method is a method which does not have a body or implementation.

3. Is it compulsory for the abstract class to include abstract methods?

No. An abstract class may contain completely concrete methods as well.

4. Can abstract class be instantiated?

No. Abstract class is normally incomplete. Hence, it cannot be instantiated.

5. Can abstract class be subclassed?

Yes.

(Refer Shape example in class notes)

6. Is it compulsory for the subclass of an abstract class to provide all the method implementations?

No.

7. If the subclass of an abstract class do not provide all the method implementations then what must be done?

Such a subclass must be declared as an abstract class.

8. When should you consider using abstract classes in your project?

In the class hierarchy, if at a given level, the method body cannot be provided, then in such cases, the method must be declared as abstract.

9. What is a Concrete class?

Concrete class is a class which contains concrete methods i,e all the methods would be having implementation or body.

10. Can we declare abstract class as final?

No. Illegal combination of modifiers error would occur.

11. Can an abstract class have a super class which is abstract?

Yes.

12. Can an abstract class have a super class which is concrete?

Yes.

13. Can an abstract class have a sub class which is abstract?

Yes.

14. Can an abstract class have a sub class which is concrete?

Yes.

15. Can a concrete class have a super class which is abstract?

Yes.

16. Can a concrete class have a super class which is concrete?

Yes.

17. Can a concrete class have a sub class which is abstract?

Yes.

18. Can a concrete class have a sub class which is concrete?

Yes.

19. Can we have a final abstract method? Why?

No. Illegal combination of modifiers error would occur.

20. Do abstract methods have method definition?

No.

21. Do abstract methods have method declaration?

Yes.

22. Can an abstract class completely contain concrete methods?

Yes.

23. Can a concrete class completely contain abstract methods?

No.

24. Can an abstract class contain main() method?

Yes.

25. When should we declare a method as abstract?

In the class hierarchy, if at a given level, the method body cannot be provided, then in such cases, the method must be declared as abstract.

26. What is the difference between pure abstract class and impure abstract class?

A pure abstract class is a class which contains only abstract methods. It would not contain any concrete method.

Impure abstract class is a class which could contain a few concrete methods along with abstract methods.

27. What is the advantage of abstract classes?

Polymorphism can be achieved.

28. Can you create an object of an abstract class? Why?

No, because abstract class is incomplete. An object of an incomplete class can not be created.

29. Can an abstract class be defined without any abstract methods?

Yes.

30. What role does an abstract class enforce?

Abstraction.

31. Can we have an abstract variable?

No. Only abstract method is permitted.

32. Can an abstract class have static variables?

Yes

33. Can an abstract class have static methods?

Yes.

34. Can we have constructor within the abstract class?

Yes.

35. Which design pattern do abstract classes promote?

template method design pattern

Exception handling continued

In the previous session we saw 3 cases, here let's see the same 3 cases with the code.

Case1:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection is estd");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the numerator:");
        int a = scan.nextInt();
        System.out.println("Enter the denomenator:");
        int b = scan.nextInt();
        int c = a/b;
        System.out.println(c);
        System.out.println("Connection is terminated");
    }
}
```



www.clipartof.com · 1246277

Output:

```
Connection is estd
Enter the numerator:
100
Enter the denomenator:
5
20
Connection is terminated
Press any key to continue . . .
```

This is the ideal case where without any mistakes the program is typed and the inputs given are also as expected by the programmer.

Case 2:

```
1 import java.util.Scanner;
2 class Demo
3 {
4     public static void main(String[] args)
5     {
6         System.out.println("Connection is estd");
7         Scanner scan = new Scanner(System.in);
8         System.out.println("Enter the numerator:");
9         int a = scan.nextInt();
10        System.out.println("Enter the denominator:");
11        int b = scan.nextInt();
12        int c = a/b;//<--Exception would occur due to faulty input.
13        System.out.println(c);
14        System.out.println("Connection is terminated");
15    }
16 }
```

Output:

```
Connection is estd
Enter the numerator:
100
Enter the denominator:
0
Exception in thread "main" java.lang.ArithmetricException: / by zero
        at Demo.main(Demo.java:12)
Press any key to continue . . .
```

Here in the output we see that we have got an exception. This exception clearly states that it's arithmetic type exception which has occurred due to division by zero. And it is pointing to line 12. Note that the lines after 12th line did not execute, that's because the program got abruptly terminated when exception occurred and when the runtime system checked for a special code inside the program and didn't find any. Then the control was given to default exception handler and then the program was abruptly terminated without execution of the remaining lines.

Case 3:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection is estd");
        Scanner scan = new Scanner(System.in);
        try //lines which might generate exception are placed here
        {
            System.out.println("Enter the numerator:");
            int a = scan.nextInt();
            System.out.println("Enter the denomenator:");
            int b = scan.nextInt();
            int c = a/b;//<--Exception would occur due to faulty input.
            System.out.println(c);
        }
        catch (Exception e) //If the error is occurred in try block it is caught here
        {
            System.out.println("Some problem occurred.");
        }
        System.out.println("Connection is terminated");
    }
}
```

Output:

```
Connection is estd
Enter the numerator:
100
Enter the denomenator:
0
Some problem occurred.
Connection is terminated
Press any key to continue . . .
```



The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block. The **try** and **catch** keywords come in pairs

Providing try-catch is referred to as **user defined exception handler**. And prevents the program from abrupt termination.

Now let's see different types of Exception that can generate for the following example:

```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection is estd");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the numerator:");
        int a = scan.nextInt();
        System.out.println("Enter the denomenator:");
        int b = scan.nextInt();
        int c = a/b;
        System.out.println(c);

        System.out.println("Enter the size of the array:");
        int size = scan.nextInt();
        int arr[] = new int[size];
        System.out.println("Enter the element to be stored");
        int ele = scan.nextInt();
        System.out.println("Enter the index at which the element must be stored:");
        int index = scan.nextInt();
        arr[index] = ele;
        System.out.println(arr[index]);
        System.out.println("Connection is terminated");
    }
}
```

We have seen arithmetic exception in the previous example. Let's see some other exceptions

Output 1:

```
Connection is estd
Enter the numerator:
100
Enter the denomenator:
2
50
Enter the size of the array:
-5
Exception in thread "main" java.lang.NegativeArraySizeException: -5
    at Demo.main(Demo.java:17)
Press any key to continue . . .
```

In the above output we see that the input given to an array size was negative. And when the exception object was given to the default exception handler it clearly showed type of exception created is **NegativeArraySizeException** and where is it pointing to.

Output 2:

```
Connection is estd
Enter the numerator:
100
Enter the denominator:
2
50
Enter the size of the array:
5
Enter the element to be stored
false
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Demo.main(Demo.java:19)
Press any key to continue . . .
```



In the above output we see that the input given to the element stored is a string but the program expects integer type. And when the exception object was given to the default exception handler it clearly showed type of exception created is **InputMismatchException** and where is it pointing to.

Output3:

In the below output we see that the index at which the element must be stored is exceeding the size of array. And when executed the exception object was given to the default exception handler it clearly showed type of exception created is **ArrayIndexOutOfBoundsException** and where is it pointing to.

```
Connection is estd
Enter the numerator:
100
Enter the denominator:
2
50
Enter the size of the array:
5
Enter the element to be stored
99
Enter the index at which the element must be stored:
9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 5
    at Demo.main(Demo.java:22)
Press any key to continue . . .
```



So now let us put all the statements that might create exception in try-catch block

```
1 import java.util.Scanner;
2 class Demo
3 {
4     public static void main(String[] args)
5     {
6         System.out.println("Connection is estd");
7         Scanner scan = new Scanner(System.in);
8         try
9         {
10             System.out.println("Enter the numerator:");
11             int a = scan.nextInt();
12             System.out.println("Enter the denominator:");
13             int b = scan.nextInt();
14             int c = a/b;
15             System.out.println(c);
16             System.out.println("Enter the size of the array:");
17             int size = scan.nextInt();
18             int arr[] = new int[size];
19             System.out.println("Enter the element to be stored");
20             int ele = scan.nextInt();
21             System.out.println("Enter the index at which the element must be stored:");
22             int index = scan.nextInt();
23             arr[index] = ele;
24             System.out.println(arr[index]);
25         }
26         catch (Exception e)
27         {
28             System.out.println("Invalid Input");
29         }
30         System.out.println("Connection is terminated");
31     }
32 }
```

Now if we give the same inputs then exception object generated will not be given to default exception handler instead it would be given to user defined exception handler.

Let's see the outputs with the same inputs given earlier.

Output1:

```
Connection is estd
Enter the numerator:
100
Enter the denomenator:
2
50
Enter the size of the array:
-5
Invalid Input
Connection is terminated
Press any key to continue . . .
```

Output2:

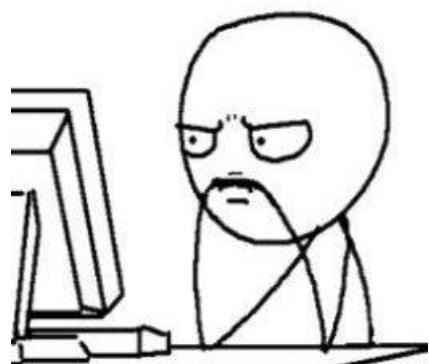
```
Connection is estd
Enter the numerator:
100
Enter the denomenator:
2
50
Enter the size of the array:
5
Enter the element to be stored
flase
Invalid Input
Connection is terminated
Press any key to continue . . .
```

Output3:

```
Connection is estd
Enter the numerator:
100
Enter the denominator:
2
50
Enter the size of the array:
5
Enter the element to be stored
99
Enter the index at which the element must be stored:
9
Invalid Input
Connection is terminated
Press any key to continue . . .
```

In all the above outputs we saw that there was no abrupt termination, all lines were executed and program got terminated. But the disadvantage here was every exception object was going to the same exception handler and the type of exception occurring was not clear. Which can be overcome by using multiple exception handler or in other words multiple catch blocks.

Below is how multiple catch blocks to be used to catch multiple exceptions.



```

        catch (ArithmaticException ae)
        {
            System.out.println("Enter non zero denominator");
        }
        catch (NegativeArraySizeException nas)
        {
            System.out.println("Enter a positive size");
        }
        catch (InputMismatchException ime)
        {
            System.out.println("Enter the valid input");
        }
        catch (ArrayIndexOutOfBoundsException ao)
        {
            System.out.println("Enter the elements within limit");
        }
        catch (Exception e)
        {
            System.out.println("Some problem occured");
        }
    System.out.println("Connection is terminated");
}

```

Now whenever the exception object is generated it checks for all the exception handler or all the catch blocks and the respective message is displayed.

If by chance an exception occurred but it doesn't belong to any of the above mentioned exceptions then it goes to general exception handler.

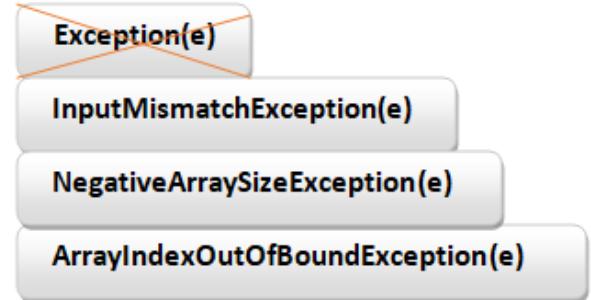


Exception handling continued....

Note:

Can you place a generic catch block on top of specific catch blocks??

In Java, the generic catch block can never be placed on top of the specific catch blocks. If attempted, it would result in an Error.



Now let us see how Runtime System works when multiple method calls are involved.

let us consider a program to understand:



```
import java.util.Scanner;
class Demo1
{
    void fun1()
    {
        System.out.println("Connection4 is estd");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the numerator");
        int a = scan.nextInt();
        System.out.println("Enter the denominator");
        int b = scan.nextInt();
        int c = a/b;
        System.out.println(c);
        System.out.println("Connection4 is terminated");
    }
}
class Demo2
{
    void fun2()
    {
        System.out.println("Connection3 is estd");
        Demo1 d1 = new Demo1();
        d1.fun1();
        System.out.println("Connection3 is terminated");
    }
}
```

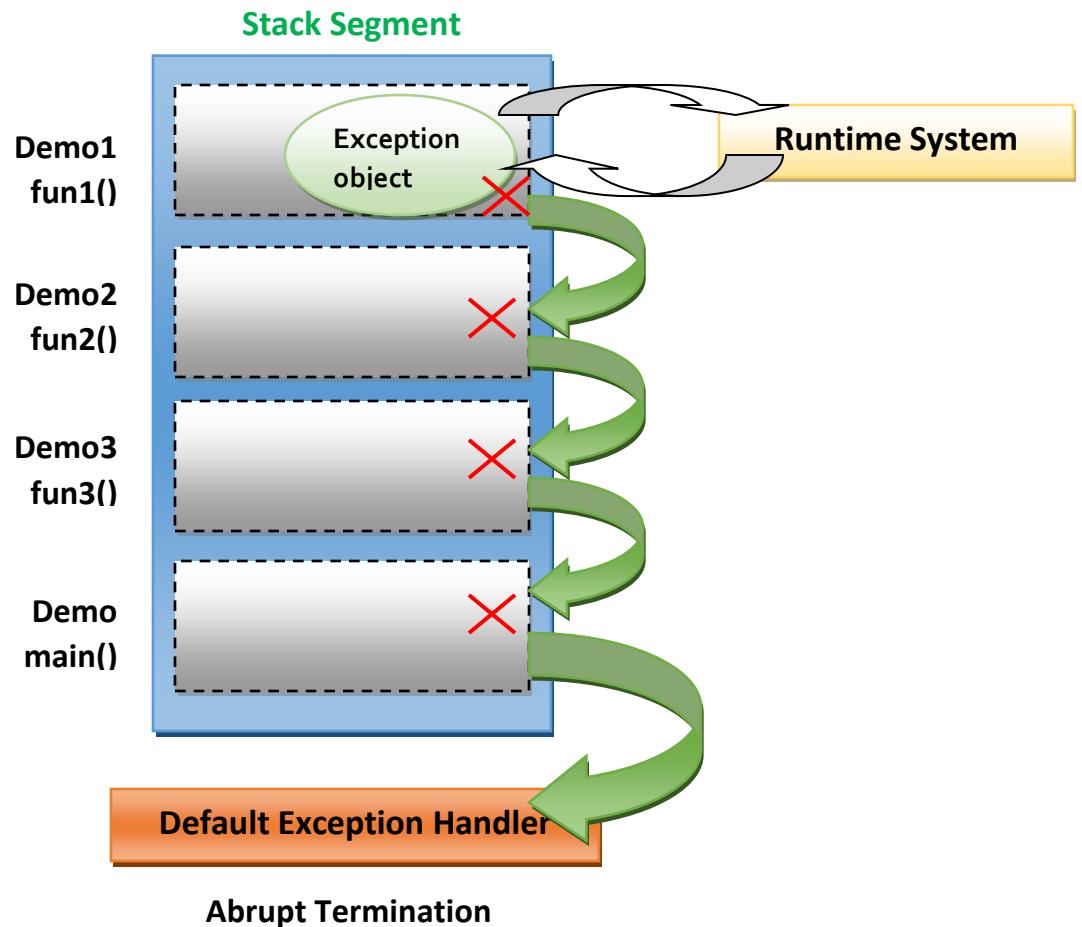
```

class Demo3
{
    void fun3()
    {
        System.out.println("Connection2 is estd");
        Demo2 d2 = new Demo2();
        d2.fun2 ();
        System.out.println("Connection2 is terminated");
    }
}

class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection1 is estd");
        Demo3 d3 = new Demo3();
        d3.fun3();
        System.out.println("Connection1 is terminated");
    }
}

```

Let us make use of **Stack segment** to understand the above program.



Explanation:

The first method that would get executed is the main method. Whenever a method is called their stack frame gets created as **Demo main()** in the Stack Segment. Next fun3() method is called and its stack frame gets created as **Demo3 fun3()**. In the next step, fun3() method calls fun2() method and its stack frame gets created as **Demo2 fun2()**. Later, fun2() method calls fun1() method and its stack frame gets created as **Demo1 fun1()**.

Whenever there are multiple methods and if within a method an exception object gets generated it handles it to the Run time System and the Run time System checks if there is any try-catch block to handle the exception. If it is not there, then the Exception object does not directly goes to the default exception handler instead it gets propagated below the stack.

If the caller of the method also does not handle the exception with try-catch then it propagates below the stack and this continues until any of the method handles the exception.

If none of the methods handles the exception then it reaches the Default Exception handler and Abrupt Termination happens.

Normal Termination

```
Connection1 is estd
Connection2 is estd
Connection3 is estd
Connection4 is estd
Enter the numerator
100
Enter the denominator
2
50
Connection4 is terminated
Connection3 is terminated
Connection2 is terminated
Connection1 is terminated
```

Abrupt Termination

```
Connection1 is estd
Connection2 is estd
Connection3 is estd
Connection4 is estd
Enter the numerator
100
Enter the denominator
0
Exception in thread "main" java.lang.ArithmetricException:
    at Demo1.fun1(Demo.java:12)
    at Demo2.fun2(Demo.java:24)
    at Demo3.fun3(Demo.java:35)
    at Demo.main(Demo.java:46)
```

Let us see another example now with try-catch block given.

```
import java.util.Scanner;
class Demo1
{
    void fun1()
    {
        System.out.println("Connection4 is estd");
        Scanner scan = new Scanner(System.in);
        try
        {
            System.out.println("Enter the numerator");
            int a = scan.nextInt();
            System.out.println("Enter the denominator");
            int b = scan.nextInt();
            int c = a/b;
            System.out.println(c);
        }
        catch (Exception e)
        {
            System.out.println("Some problem occured");
        }

        System.out.println("Connection4 is terminated");
    }
}

class Demo2
{
    void fun2()
    {
        System.out.println("Connection3 is estd");
        Demo1 d1 = new Demo1();
        d1.fun1();
        System.out.println("Connection3 is terminated");
    }
}

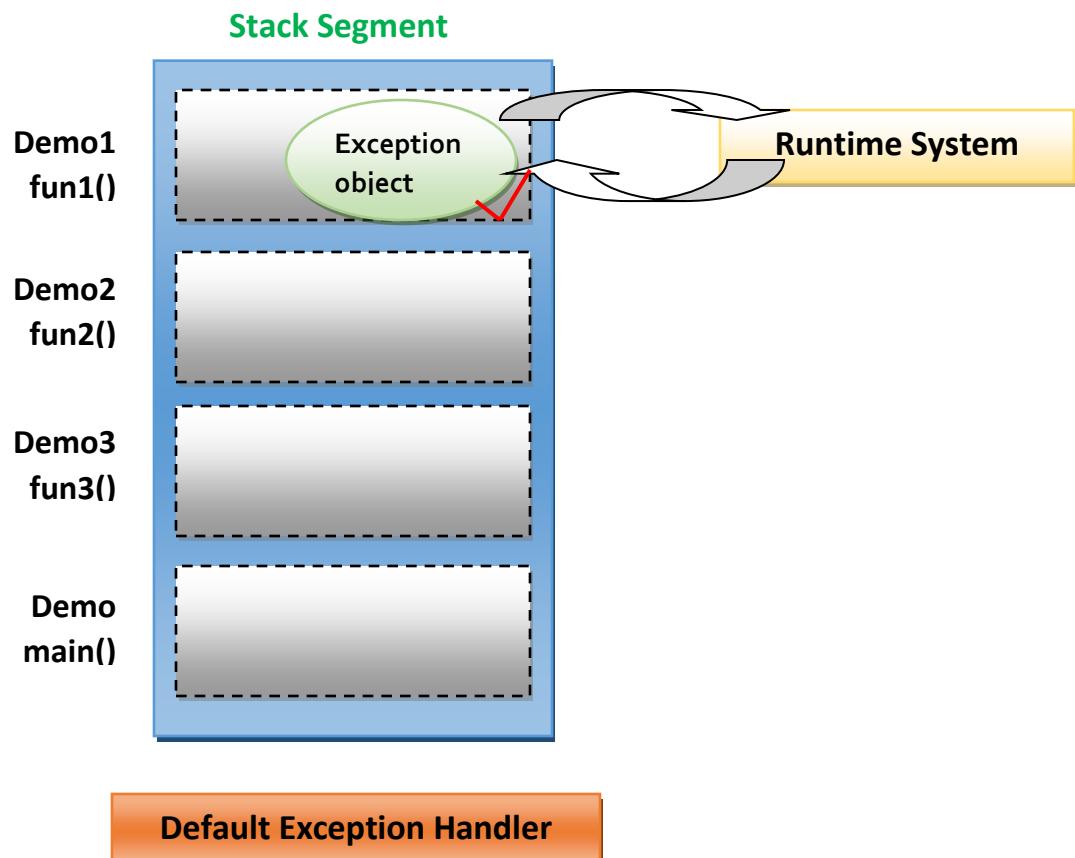
class Demo3
{
    void fun3()
    {
        System.out.println("Connection2 is estd");
        Demo2 d2 = new Demo2();
        d2.fun2 ();
        System.out.println("Connection2 is terminated");
    }
}
```

```

class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection1 is estd");
        Demo3 d3 = new Demo3();
        d3.fun3();
        System.out.println("Connection1 is terminated");
    }
}

```

Let us make use of **Stack segment** to understand the above program.



Explanation:

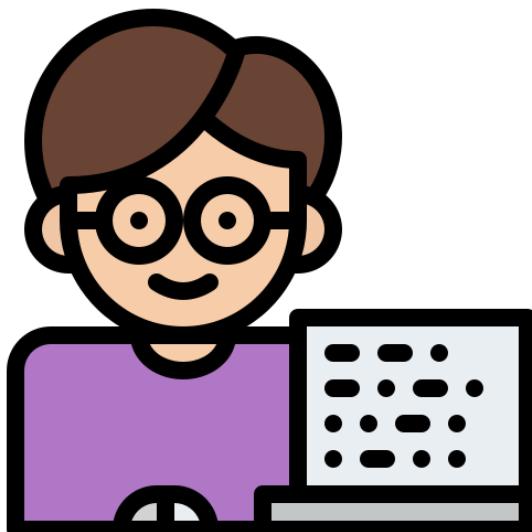
In this example we have given try-catch block inside a fun1() method with an assumption that an exception might occur.

When an exception objects get generated in a method it is handed over to the Runtime System. Then the Runtime System checks if there is any try-catch block to handle the exception. In this example, the exception is caught by the catch block of fun1() method.

Once an exception is caught it does not gets propagate down to the caller methods and normal termination of the program happens.

Output:

```
Connection1 is estd
Connection2 is estd
Connection3 is estd
Connection4 is estd
Enter the numerator
100
Enter the denominator
0
Some problem occurred
Connection4 is terminated
Connection3 is terminated
Connection2 is terminated
Connection1 is terminated
```



HE GOT THE OUTPUT. DID YOU??

Different ways of handling the Exception.

- 1) Handling the Exception(try-catch)
- 2) Re-throwing the Exception(try-catch-throw-throws-finally)
- 3) Ducking the Exception(throws)

Case-1: Handling the exception.

If the methods in which exception occurs and the method in which handling occurs are one and the same then it is called Handling the exception.

Let us understand this with an example

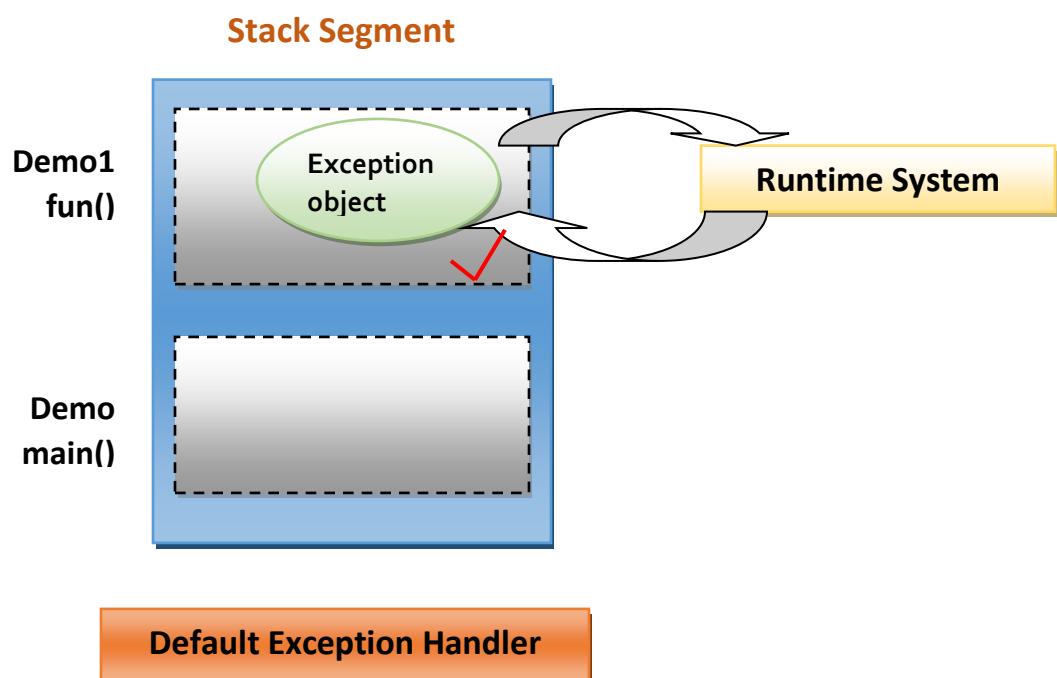
```
import java.util.Scanner;
class Demo1
{
    void fun()
    {
        System.out.println("Connection2 is estd");
        Scanner scan = new Scanner(System.in);
        try
        {
            System.out.println("Enter the numerator");
            int a = scan.nextInt();
            System.out.println("Enter the denominator");
            int b = scan.nextInt();
            int c = a/b;
            System.out.println(c);
        }
        catch (Exception e)
        {
            System.out.println("Exception handled in fun()");
        }
        System.out.println("Connection2 is terminated");
    }
}
```

```

class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection1 is estd");
        Demo1 d1 = new Demo1();
        d1.fun();
        System.out.println("Connection1 is terminated");
    }
}

```

Let us understand the program through Stack Segment Diagram.



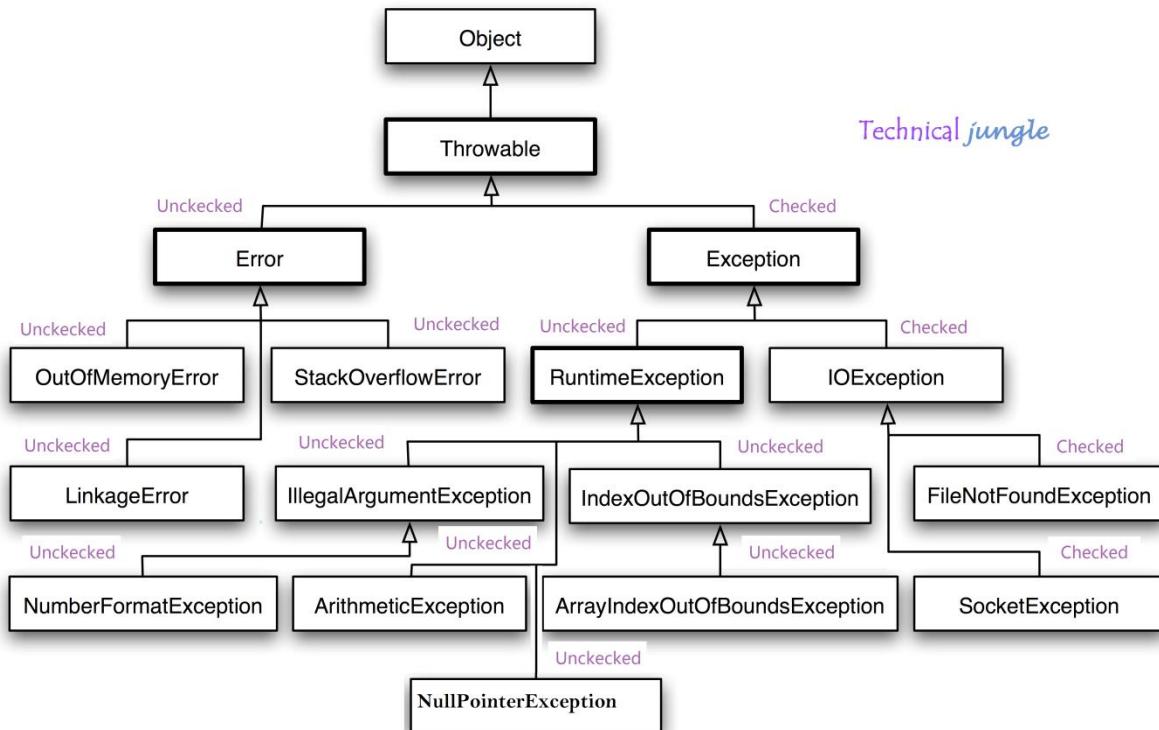
Here in this program the exception gets generated inside `fun()` method and then it goes to the Runtime System. The Runtime System comes back to `fun()` methods and checks can the exception be handled in the same method.

Since there is a try-catch block to handle the exception it doesn't goes to the Default exception handler.

So according to our Case-1, exception occurs and gets handled in the same method.

Exception Hierarchy

The possible exceptions in a Java program are organized in a hierarchy of exception classes. The **Throwable class**, which is an immediate **subclass of Object**, is at the root of the exception hierarchy. **Throwable has two immediate subclasses: Exception and Error.**



All of the subclasses of **Exception** represent exceptional conditions that a normal Java program may want to handle. Many of the standard exceptions are also subclasses of **Runtime Exception**. **Runtime exceptions represent runtime conditions that can generally occur in any Java method, so a method is not required to declare that it throws any of the runtime exceptions.** However, if a method can throw any of the other standard exceptions, it must declare them in its throws clause.

A Java program should try to handle all of the standard exception classes, since they represent routine abnormal conditions that should be anticipated and caught to prevent program termination.



All exception classes are subtypes of the `java.lang.Exception` class. The exception class is a subclass of the `Throwable` class. Other than the exception class there is another subclass called `Error` which is derived from the `Throwable` class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

StackOverflowError:

When a function call is invoked by a Java application, a **stack frame** is allocated on the call stack. The stack frame contains the parameters of the invoked method, its local parameters, and the return address of the method. The return address denotes the execution point from which, the program execution shall continue after the invoked method returns. If there is no space for a new stack frame then, the `StackOverflowError` is thrown by the Java Virtual Machine (JVM).



The most common case that can possibly exhaust a Java application's stack is **recursion**. In recursion, a method invokes itself during its execution. Recursion is considered as a powerful general-purpose programming technique, but must be used with caution, in order for the `StackOverflowError` to be avoided.

OutOfMemoryError:

Usually, this error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

OutOfMemoryError usually means that you're doing something wrong, either holding onto objects too long, or trying to process too much data at a time.

Sometimes, it indicates a problem that's out of your control, such as a third-party library that caches strings, or an application server that doesn't clean up after deploys. And sometimes, it has nothing to do with objects on the heap.



Similarity between Exception and Error

- Error and Exception both occur during runtime.
- In both cases Abrupt termination happens.

Now let's see Difference

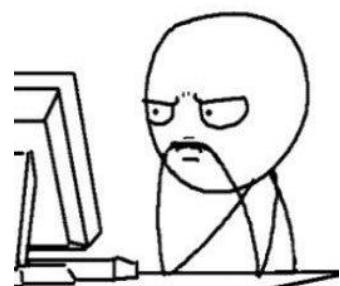
Exception	Error
1) Exceptions can be recovered	1) Errors cannot be recovered.
2) Exceptions can be classified in to two types: a) Checked Exception b) Unchecked Exception	2) There is no such classification for errors. Errors are always unchecked.
3) In case of checked Exceptions compiler will have knowledge of checked exceptions and force to keep try catch blocks.	3) In case of Errors compiler won't have knowledge of errors.

Different ways to write try-catch-finally

Before looking at the valid combinations let us see the invalid combinations..



- If you are trying to write only try, only catch, or only finally then it is not valid. Let's think logically. Without knowing if certain statements will give exception, how will catch block know what to catch. And if anything is in try block which might throw exception then in the absence of catch block who will catch it??



Valid combinations of try-catch-finally

1st possibility:

```
try
{
    //some statements
}
catch ()
{
    //some statements
}
```

3rd possibility:

```
try
{
    //some statements
}
catch ()
{
    //some statements
}
finally
{
    //some statements
}
```

2nd possibility:

```
try
{
    //some statements
}
catch ()
{
    //some statements
}
catch ()
{
    //some statements
}
```

4th possibility:

```
try
{
    try
    {
        //some statements
    }
    catch ()
    {
        //some statements
    }
    finally
    {
        //some statements
    }
}
catch ()
{
    //some statements
}
finally
{
    //some statements
}
```



5th possibility:

```
try
{
    //some statements
}
catch ()
{
    try
    {
        //some statements
    }
    catch ()
    {
        //some statements
    }
    finally
    {
        //some statements
    }
}
finally
{
    //some statements
}
```

6th possibility:

```
try
{
    //some statements
}
catch ()
{
    //some statements
}
finally
{
    try
    {
        //some statements
    }
    catch ()
    {
        //some statements
    }
    finally
    {
        //some statements
    }
}
```

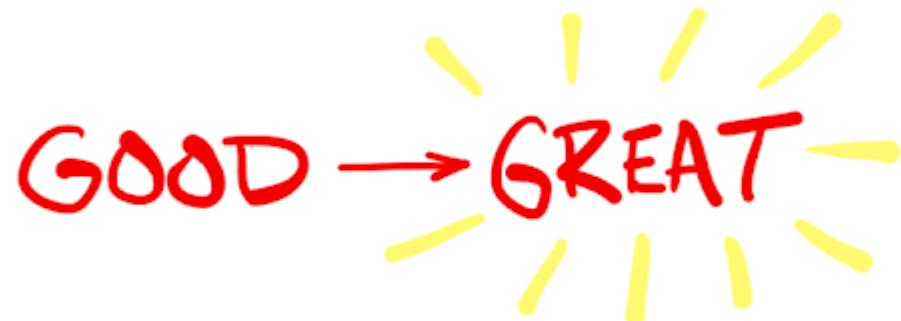
7th possibility:

```
try
{
    //some statements
}
finally
{
    //some statements
}
```



Rules for method overriding, considering the Exceptions

- If the parent class method throws an exception then the child class overridden method can either throw the same exception or not throw any exception at all.
- If the parent class method throws an exception, then the child class overridden method can throw a different exception provided is-a relationship exists between the exceptions.
- If the parent class method throws an exception then the child class overridden method can throw a different exception even though is-a relationship does not exist provided both the exceptions are RuntimeException.



Different ways of handling the Exception continued...

Case-2: Re-throwing the Exception (try-catch-throw-throws-finally)

If an exception occurs within a method and is also handled, but the exception must also **propagate to the caller of the method**, then it is referred to as **re-throwing the exception**.

To achieve this, we must use- try-catch-throw-throws-finally.

Throw keyword is used to **explicitly throw an exception** from a method or any block of code.

The **disadvantage** of throw keyword is that **statements below throw keyword do not execute**. This can be **overcome** by placing the statements within **finally** block.

Throws keyword is used in the **signature of method** to indicate that this method might throw an exception. The caller of this method must handle the exception using try-catch block.

Let us try writing code to understand this in detail

```
import java.util.Scanner;
class Demo1
{
    void fun() throws Exception
    {
        System.out.println("Connection2 is established");
        Scanner scan = new Scanner(System.in);
        try
        {
            System.out.println("Enter numerator");
            int a = scan.nextInt();
            System.out.println("Enter denominator");
            int b = scan.nextInt();
            int c = a/b;
            System.out.println(c);
        }
        catch (Exception e)
        {
            System.out.println("Exception handled in fun()");
            throw e;
        }
    }
}
```



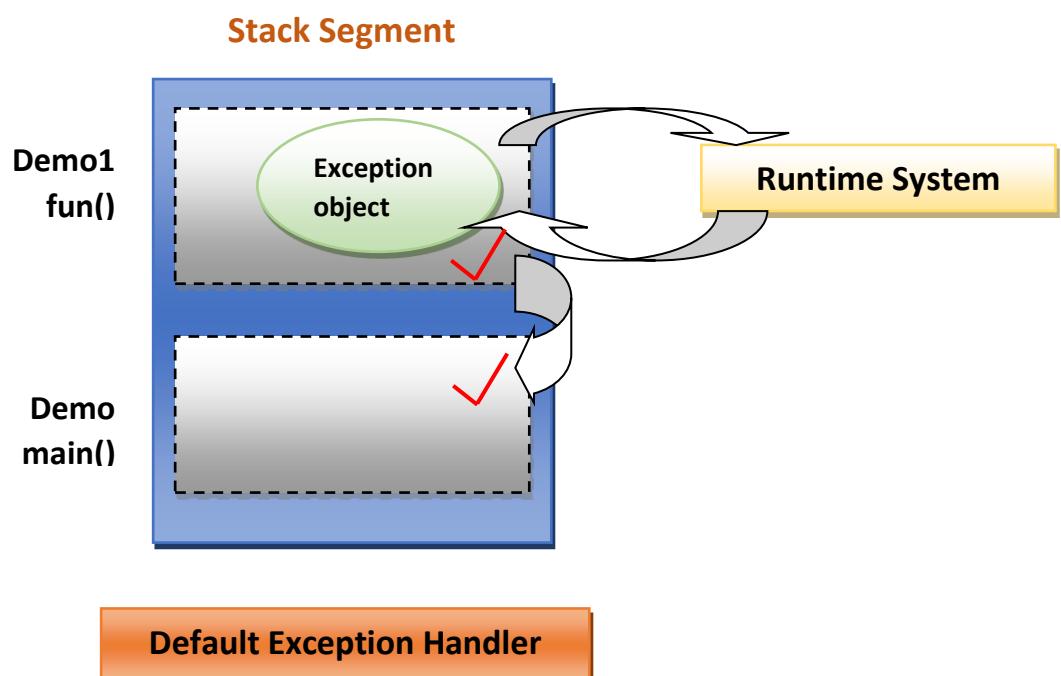
```

        finally
        {
            System.out.println("Connection2 is terminated");
        }
    }
}

class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection1 is established");
        try
        {
            Demo1 d1 = new Demo1();
            d1.fun();
        }
        catch (Exception e)
        {
            System.out.println("Exception handled in main()");
        }
        System.out.println("Connection1 is terminated");
    }
}

```

Let us understand the program through Stack Segment Diagram shown below:



Explanation:

Here in this program the exception gets generated inside fun() method in Demo1 class and then it goes to the Runtime System. The Runtime System comes back to fun() method and checks can the exception be handled in the same method.

Since there is a try-catch block to handle the exception it doesn't go to the Default exception handler. We are handling the exception and throwing it to main() in class Demo. Before the exception is thrown, the statements inside finally block get executed and then exception is handled again in main() using try-catch.

So according to our Case-2, **exception occurs, gets handled in the same method and is thrown to the caller method, which is nothing but rethrowing the exception.**

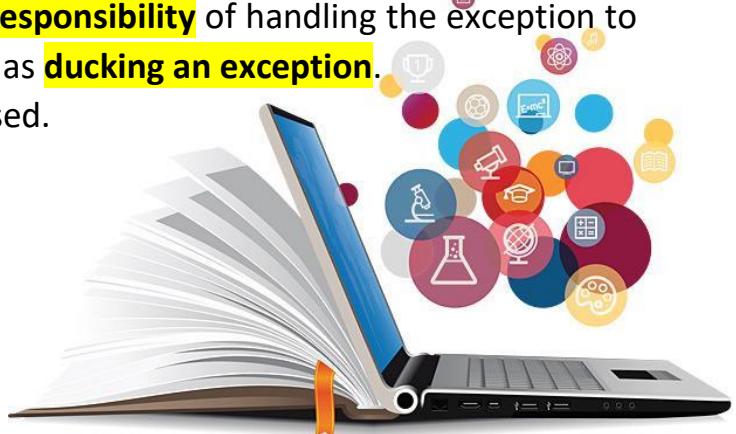
Output:

```
Connection1 is established
Connection2 is established
Enter numerator
100
Enter denominator
0
Exception handled in fun()
Connection2 is terminated
Exception handled in main()
Connection1 is terminated
```

Case-3: Ducking the Exception(throws)

If an exception occurs within a method and the **method does not want to handle** it, instead it wants to **handover the responsibility** of handling the exception to **caller of the method**, it is referred to as **ducking an exception**.

To achieve this, **throws** keyword is used.



Let us try writing code to understand this case

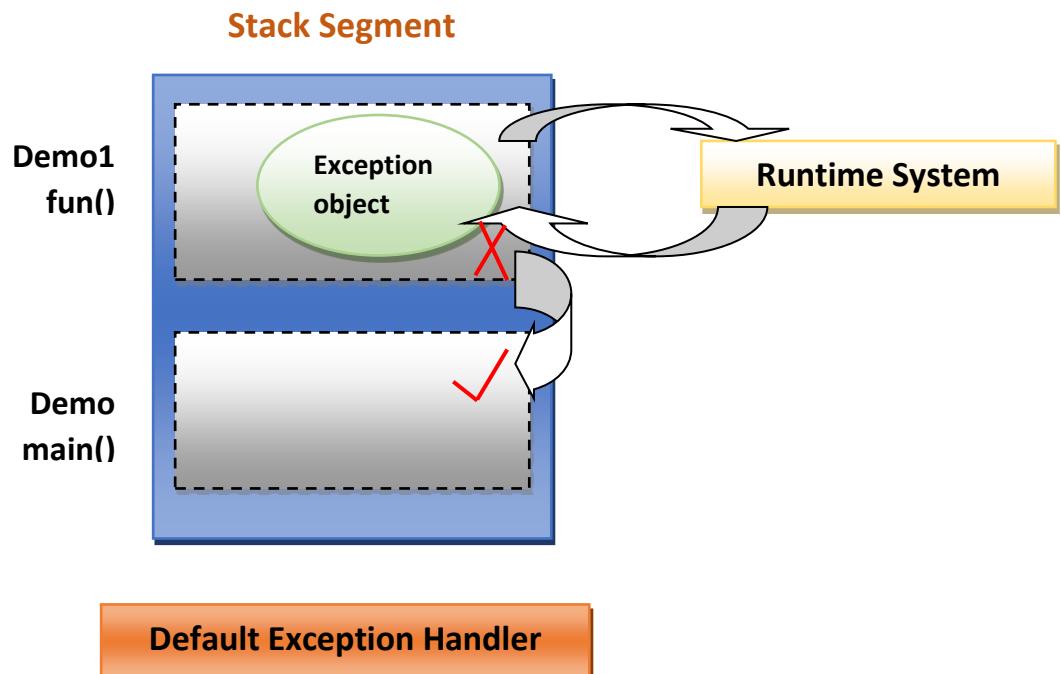
```
import java.util.Scanner;
class Demo1
{
    void fun() throws Exception
    {
        System.out.println("Connection2 is established");
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter numerator");
        int a = scan.nextInt();
        System.out.println("Enter denominator");
        int b = scan.nextInt();
        int c = a/b;
        System.out.println(c);

        System.out.println("Connection2 is terminated");
    }
}
class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Connection1 is established");
        try
        {
            Demo1 d1 = new Demo1();
            d1.fun();
        }
        catch (Exception e)
        {
            System.out.println("Exception handled in main()");
        }
        System.out.println("Connection1 is terminated");
    }
}
```



Let us understand the program through Stack Segment Diagram shown below:



Explanation:

Here in this program the exception gets generated inside `fun()` method in `Demo1` class and then it goes to the Runtime System. The Runtime System comes back to `fun()` method and checks can the exception be handled in the same method.

Since there is `throws` keyword used in method signature it doesn't go to the Default exception handler, the control is given to `main()` and it is handled in `main()` using try-catch. Due to which **connection2 is terminated** is not printed as the control is given to `main()` once the exception occurs.

So according to our Case-3, **using throws we are not handling the exception in the method in which it occurs instead we are ducking it.**

Output:

```
Connection1 is established
Connection2 is established
Enter numerator
100
Enter denominator
0
Exception handled in main()
Connection1 is terminated
```

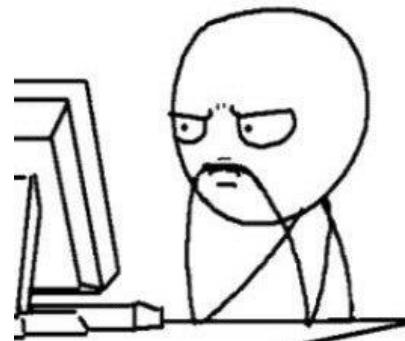
Custom Exceptions in Java

If you are **creating your own Exception** that is known as custom exception.
Java custom exceptions are used to customize the exception according to user need.

Before knowing how to create custom exception let us see how java cannot make out some exceptions.

Example: Let us consider atm example

```
import java.util.Scanner;
class ATM
{
    int acc_num=1234;
    int password=9999;
    int an,pwd;
    void acceptInput()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the acc_num");
        an = scan.nextInt();
        System.out.println("Enter the password");
        pwd = scan.nextInt();
    }
    void verify()
    {
        if(acc_num==an && password==pwd)
        {
            System.out.println("Colletc your money");
        }
        else
        {
            System.out.println("Invalid card details. Try again!!");
        }
    }
}
class Bank
{
    void initiate()
    {
        ATM atm = new ATM();
        atm.acceptInput();
        atm.verify();
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Bank b = new Bank();
        b.initiate();
    }
}
```



Output:

```
Enter the acc_num  
1234  
Enter the password  
9999  
Colletc your money  
Press any key to continue . . .
```



```
Enter the acc_num  
1234  
Enter the password  
8888  
Invalid card details. Try again!!  
Press any key to continue . . .
```



In ideal atm case if the entered password/pin is not correct then we get two more chances but here that is not happening... So this can be overcome by using custom exception which next we'll be seeing.

Java provides us facility to create our own exceptions which are basically derived classes of Exception.

Example using custom exception:

```
import java.util.Scanner;
class InvalidUserException extends Exception
{
    public String getMessage()
    {
        return("Invalid card details. Try again!!");
    }
}
class ATM
{
    int acc_num=1234;
    int password=9999;
    int an,pwd;
    void acceptInput()
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the acc_num");
        an = scan.nextInt();
        System.out.println("Enter the password");
        pwd = scan.nextInt();
    }
    void verify() throws Exception
    {
        if(acc_num==an && password==pwd)
        {
            System.out.println("Collect your money");
        }
        else
        {
            InvalidUserException iue = new InvalidUserException();
            System.out.println(iue.getMessage());
            throw iue;
        }
    }
}
class Bank
{
    void initiate()
    {
        ATM atm = new ATM();
        try
        {
            atm.acceptInput(); //first attempt
            atm.verify();
        }
        catch (Exception e)
        {
            try
            {
                atm.acceptInput(); //second attempt
                atm.verify();
            }
        }
    }
}
```



www.clipartof.com · 1246277

```

        'catch (Exception f)
        {
            try
            {
                atm.acceptInput(); //third and final attempt
                atm.verify();
            }
            catch (Exception g)
            {
                System.out.println("Card blocked!!!");
                System.exit(0);
            }
        }
    }
}

class Demo
{
    public static void main(String[] args)
    {
        Bank b = new Bank();
        b.initiate();
    }
}

```

Output for invalid entries:

Enter the acc_num
1234
Enter the password
7777
Invalid card details. Try again!!
Enter the acc_num
1234
Enter the password
3333
Invalid card details. Try again!!
Enter the acc_num
1234
Enter the password
5555
Invalid card details. Try again!!
Card blocked!!
Press any key to continue . . .



GOOD → GREAT

Key Points

- Create a class and extend the Exception class.
- Override the getMessage() and provide a suitable message.
- Explicitly create an object of the class wherever there is need for an exception to be generated.

If you are wondering why or where exactly in real life the custom exception are being used then let us see that:

Java exceptions cover almost all general exceptions that are bound to happen in programming.

However, we sometimes need to supplement these standard exceptions with our own.

The main reasons for introducing custom exceptions are:

- **Business logic exceptions** – Exceptions that are specific to the business logic and workflow. These help the application users or the developers understand what the exact problem is.
- To catch and provide specific treatment to a subset of existing Java exceptions.
- **Custom exceptions are very much useful when we need to handle specific exceptions related to the business logic.** When used properly, they can serve as a useful tool for better exception handling and logging.

But 'Why?'



**GREAT
WORK!**

Exception Handling

1. What are exceptions?

Exception refers to such mistakes that would occur during the execution time of an app due to faulty inputs given by the user.

2. What happens if exceptions are not handled?

It would result in abrupt termination of the app due to the exception object reaching the default exception handler.

3. Does C language support exception handling?

No.

4. Does C++ language support exception handling?

Yes. But not robust.

5. Does Java language support exception handling?

Yes.

6. What is meant by exception handling?

It refers to the process of managing the exception object in a manner that it does not result in abrupt termination of an app.

7. Are exceptions detected during compilation time?

No.

8. Are exceptions detected during execution time?

Yes.

9. What is an error?

Error refers to such mistakes that would occur during execution time due to faulty coding by the programmer. Errors cannot be handled using try and catch.

10. What is the difference between exception and error?

Exception	Error
Occurs due to faulty inputs given by the user.	Occurs due to faulty coding by the developer.
Can be handled using try and catch	Cannot be handled using try and catch
Abrupt termination of an app can be prevented	Abrupt termination of an app cannot be prevented
Examples include ArithmaticException, NegativeArraySizeException, ArrayIndexOutOfBoundsException etc.,	Examples include StackOverFlowError, OutOfMemoryError etc.,

11. Why are exceptions caused normally in a program?

Due to faulty inputs given by the user.

12. Why do errors occur normally in a program?

Due to faulty coding of an app by the developer.

13. What is the difference between compile time error and run time error? Compilation time errors are syntax errors which can be detected by the compiler.

Eg. missing semicolon, undefined symbol etc.

Run time errors are such errors which cannot be detected by the compiler.

They would be detected during execution time.

Eg. StackOverFlowError , OutOfMemoryError etc.

14. Which construct is used in Java for exception handling?

try-catch-throw-throws-finally.

15. What is a role of try keyword in Java?

Such statements which could possibly result in an exception would be placed inside the try block.

16. What is a role of catch keyword in Java?

If any exception object is generated in the try block, then it would be caught by the catch block.

17. What is a role of throw keyword in Java?

It is used to re-throw an exception object and hence forcefully enables a handled exception object to trickle down the stack hierarchy.

18. What is a role of finally keyword in Java?

The most critical statements that have to be compulsory executed such as statements releasing resources present at the end of a method must be enclosed within the finally block.

19. What is a role of throws keyword in Java?

It is used to duck an exception. It is also used to alert the caller of the method that this method could possibly throw an exception object.

20. Does Java language provide any exception handler?

Yes. It provides default exception handler.

21. If Java language provides exception handler then why should programmer handle exceptions?

Because the default exception handler provided by java ensures only system safety. It does not ensure termination of connections and release of acquired resources. Hence programmer should handle an exception.

22. Who creates the exception object?

The method in which a run time mistake has occurred would create an exception object.

23. What would be present in an exception object?

- i) The type of exception
- ii) Line no. on which exception occurred

- iii) Stack trace
24. **Whom is the exception object given to soon after its creation?**
Exception object would be given to Run Time System(RTS).
25. **What is run time system?**
It is a part of JVM which deals with exception handling.
26. **What is a role of run time system in exception handling?**
It receives the exception object from the method and verifies if a user defined exception handler is present. If the user defined exception handler is present then the exception object would be handed over to it. Otherwise the exception object would be handed over to the default exception handler.
27. **Can we have multiple catches for a single try block?**
Yes.
28. **Why should an Exception type reference be placed at the bottom of multiple catch blocks?**
In order to ensure that if the specific catch blocks present at the top of the catch hierarchy fails to handle an exception, then the generic catch present at the bottom would be able to handle it.
By doing this,
- 1) The exception object is prevented from reaching the default exception handler and hence prevents abrupt termination.
 - 2) By using log4j tool in the generic catch block, a log report can be sent to the developer which would be useful to upgrade the app.
29. **How is it that Exception reference can catch (refer to) all types of exceptions?**
Because, in the Exception hierarchy it is a parent type. In java parent type reference can refer to any child object. Hence the Exception type reference can catch any of the child class objects.
30. **What are checked exceptions?**

They are such exceptions for which java compiler forces to handle it using a try catch block.

Eg: SQLException, ClassNotFoundException etc.,

31. What are unchecked exceptions?

They are such exceptions for which java compiler does not force the programmer to handle it using try catch.

Eg: ArrayOutOfBoundsException, ArithmeticException etc.,

32. Why should we not have exception type reference at the top of multiple catch blocks?

Because if it is placed at the top of the catch hierarchy, it would catch all exception that are generated by the try block without giving any opportunity for the specific catch blocks present below in the hierarchy to catch the exception object.

33. What is meant by ducking?

It refers to the process in which a method would pass an exception object without handling it. “throws” keyword is used for ducking.

34. What is meant by rethrowing?

It is the process of a method passing the exception object down the stack hierarchy after handling it. “throw” and “throws” keywords are used for rethrowing.

35. What is the problem associated with throw keyword? How is it resolved? Disadvantage of “throw” keyword is that statements below throw keyword would not get executed. This problem can be overcome by using “finally” block.

36. Is using throws compulsory in exception handling?

Not upto JDK 1.7. However, from JDK 1.8 onwards if a method ducks or re-throws an exception object, then it has been made compulsory to use “throws” keyword.

37. How does the run time system respond in case of multiple method calls?

If the exception handler is not present in the method that generated the exception object, then the Run Time System would not send the exception object directly to the Default Exception Handler (DEH). Rather, the exception object would trickle down the stack hierarchy till a handler is found in one of the methods (activation record). If no handler is found then the exception object would be handed over to Default Exception Handler.

38. Can you give an example for an exception?

ArithmaticException, ArrayIndexOutOfBoundsException,
NegativeArraySizeException, ArrayStoreException etc.,

39. Can you give an example for an error?

StackOverFlowError, OutOfMemoryError etc.,

40. Is there any way in Java by which the finally block would not get executed?

Yes. By using System.exit(0).

41. When should a method duck an exception object?

It is decided by the designer of the project depending upon the specific project requirements.

42. When should a method rethrow an exception object?

It is decided by the designer of the project depending upon the specific project requirements.

43. Can we manually create exception objects?

Yes.

44. Is a try-catch combination valid?

Yes.

45. Is a try-catch-finally combination valid?

Yes.

46. **Is a try-finally combination valid?**

Yes.

47. **Is a catch-finally combination valid?**

No.

48. **Can we have a try block without any other block?**

No.

49. **Can we have a catch block without any other block?**

No.

50. **Can we have a finally block without any other block?**

No.

51. **Can we have try-catch-finally within a try block?**

Yes.

52. **Can we have try-catch-finally within a catch block?**

Yes.

53. **Can we have try-catch-finally within a finally block?**

Yes.

54. **Name a few checked exceptions?**

FileNotFoundException, ClassNotFoundException, SQLException etc.,

55. **Name a few unchecked exceptions**

ArithmaticException, ArrayIndexOutOfBoundsException,
NegativeArraySizeException, ArrayStoreException etc.,

56. **Which class is the superclass of Exception?**

Throwable.

57. **Which class is the superclass of Error?**

Throwable.

58. What is the role of getMessage() in exception handling?

It enables programmer to create custom exception.

59. What is the role of printStackTrace() in exception handling?

It gives the developer an idea about the method in which exception occurred and also the subsequent propagation of the exception object.

60. Differentiate between throw and throws?

throw	throws
Used to rethrow an Exception object.	Used for ducking and also to rethrow an exception object.
It is associated with the Exception type reference.	It is associated with the signature of the method.
It would be present within the body of the method.	It would be present in the signature of the method.

61. What is meant by LSP?

LSP stands for Liskov Substitution Principle. It speaks about the rules associated with Exceptions and overriding.

62. What is try region also called as?

Guarded region.

63. What is catch region also called as?

Handling region.

64. What is finally region also called as?

Cleanup region.

65. What are the rules of exception that must be followed during inheritance?

LSP rules. (Eg: Refer class notes)

66. Can the base class method and derived class method throw different types of exception?

Yes. Provided

- 1) Is-a relationship exists between them
- 2) Even though Is-a relationship does not exists, it is possible provided the exceptions being thrown are of Runtime Exceptions (unchecked Exceptions).

67. If the base class method throws an exception is it mandatory for the derived class method also to throw an exception?

No.

68. How do we create custom exceptions?

- i) By sub classing Exception class ,
- ii) By overriding public String getMessage() to provide suitable diagnostic message.

69. If I want an object of my class to be thrown as an exception object then what should I do?

If I want an object of my class to be thrown as Exception object, then a class should be made as subclass of an Exception class.

70. If my class already extends from some other class what should I do if I want an instance of my class to be thrown as an exception object?

Not possible, because multiple inheritance is not permitted in java.

71. What are the different ways to manage exceptions?

- i) Handling an Exception,
- ii) Re-throwing an Exception,
- iii) Ducking an Exception.

72. If I write return at the end of the try block, will the finally block still execute? Yes.**73. What is the difference between final, finally and finalize()?**

- final - used to create constants and to prevent inheritance
finally - is an exception handing control construct in which most

critical statements are placed.

`finalize()` - `finalize()` method would be called by garbage collector in java

to perform clean up operations on an object before it is removed from the memory.

multithreading

Before getting to know what is multithreading, let us first understand why multithreading?



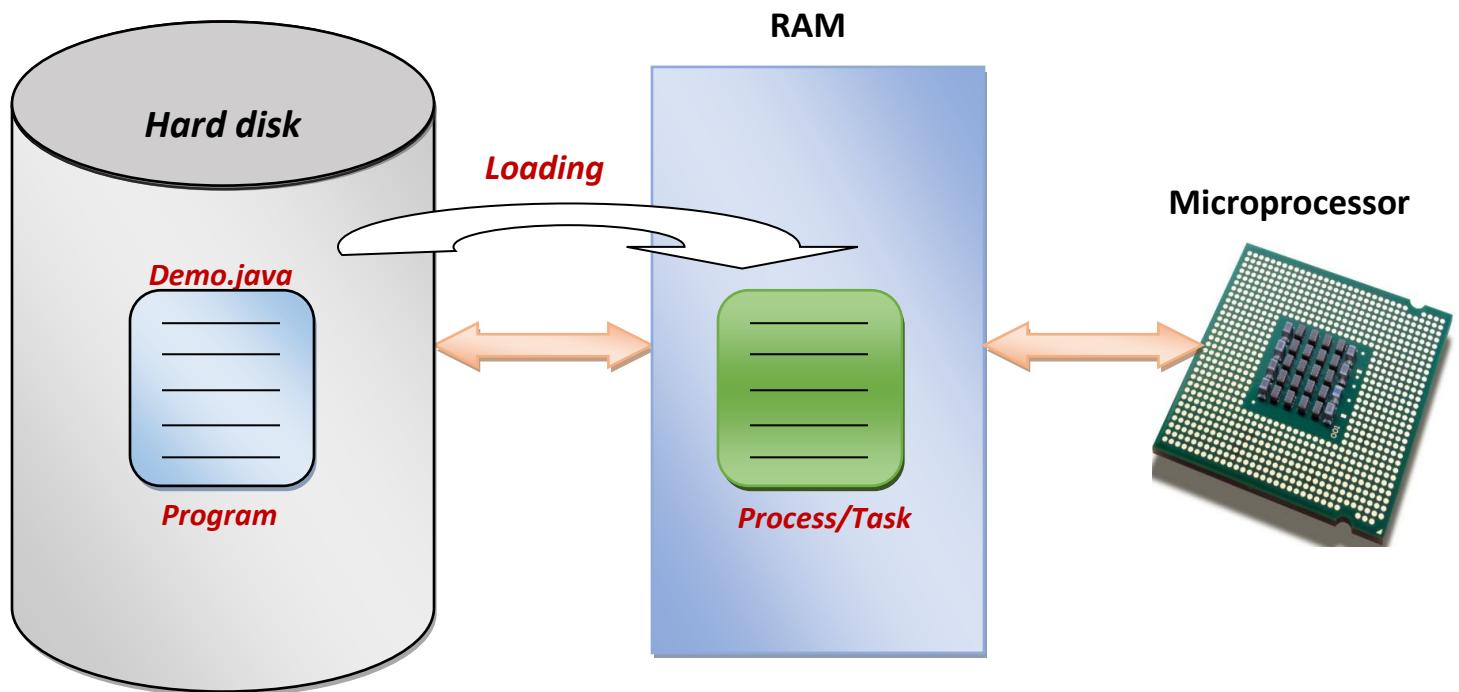
Why Multithreading??

Any developer should understand the concept of Multithreading to make their program efficient and for faster execution.

Before going into the depth of Multithreading let us understand what does process/task means?

Let us assume we have a java program named **Demo.java** saved on the hard disk. The execution doesn't happen in Hard disk so for the program to get executed it should be placed on the RAM. The process of taking the program and placing it on the RAM is called as **Loading**. Once it is placed on RAM it is called as **process** as it is available to the microprocessor for execution. Process provides an execution environment for our program.

Therefore, a program under execution is technically called as Process.

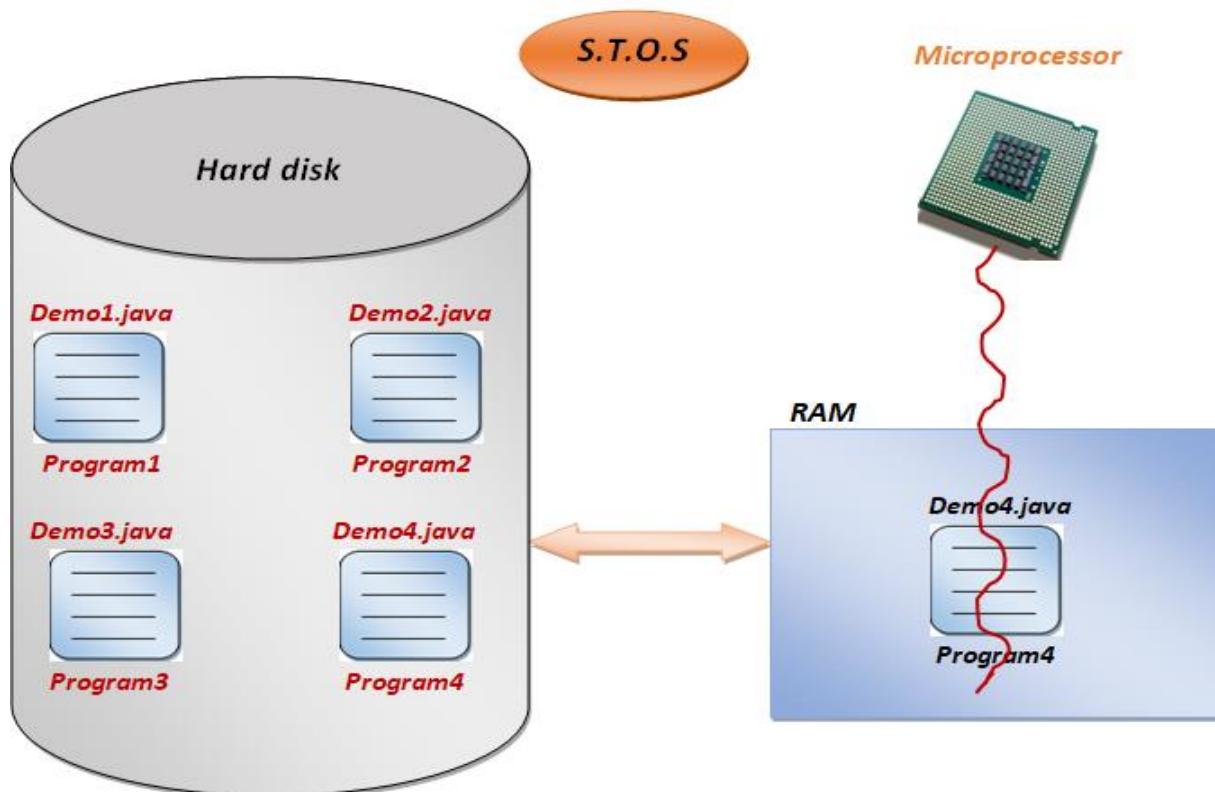


Now Let us understand what happens with the Operating System.

Let us assume that the Operating System we used was **Single Tasking Operating System (S.T.O.S)**. STOS as the name implies, per processor it gives only one task. It is designed to manage the computer so that one user can effectively do one thing at a time.

Let us imagine there are multiple programs on the Hard disk but since we are working with **STOS** with use of **single microprocessor**, only one program gets loaded onto the RAM. And now there is one process on the RAM and microprocessor starts executing.

If the other program's has to get executed then the process which is under execution in the RAM should complete its execution and it is removed from the RAM. Then the next program gets loaded onto the RAM and execution continues in this way. **This style of execution is called as Sequential Execution.**



Single task is achieved using single processor with the use of S.T.O.S

Disadvantage of STOS with single processor approach:

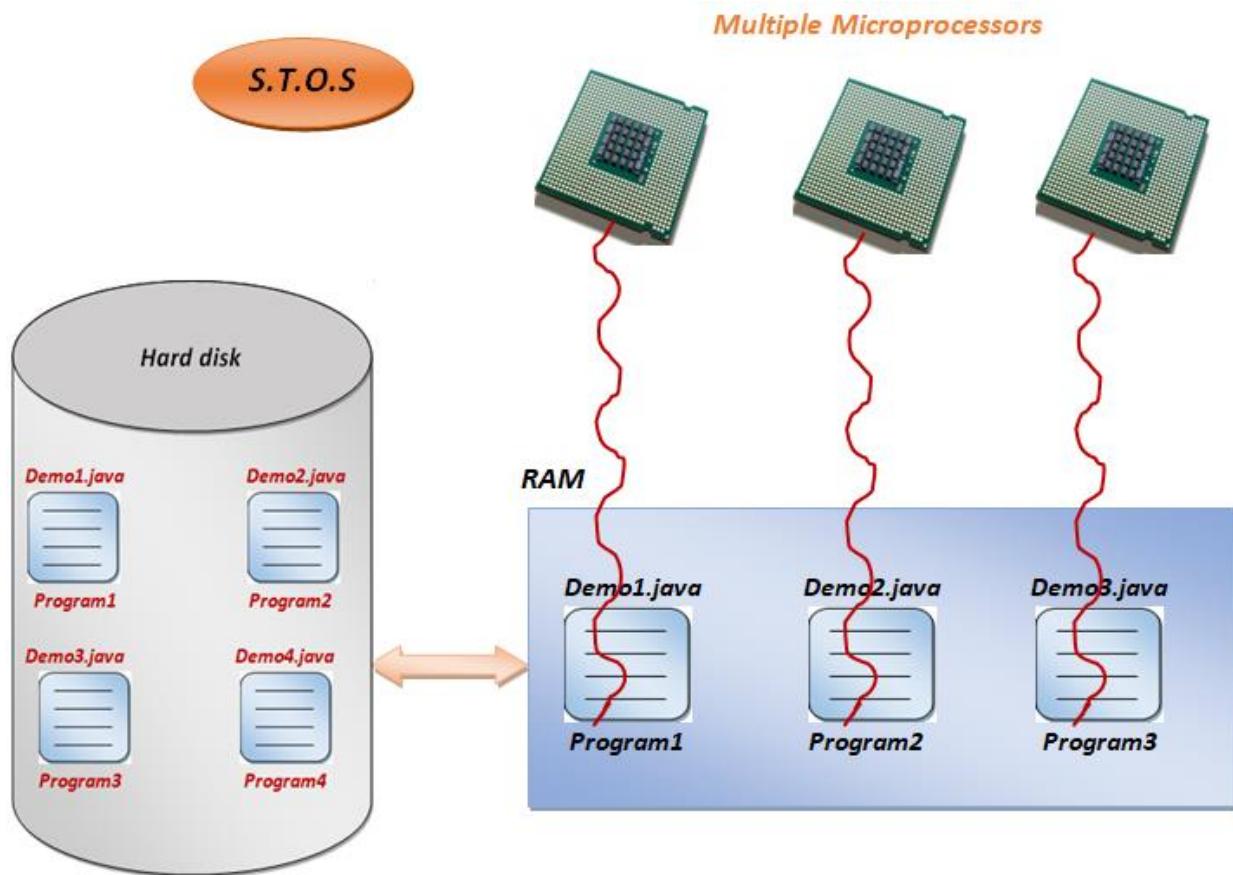
- **Tasks take longer time to complete:** As we know if no multiple tasks run at a time then many tasks are waiting for the CPU. This will make system slow.

Now if we want multiple processes to work parallel what should be done??



Let us consider the same **Single Tasking Operating System (S.T.O.S)**.

As we can see there are multiple programs on the Hard disk. But now let us consider **multiple microprocessors** which means multiple processes can be loaded onto the RAM. Since there are multiple processors each processor started executing one task. Therefore, multiple task is been done.



Multitasking is achieved using multiple processors with the use of S.T.O.S

Advantage of STOS with multiple processor approach:

- Execution is fast as multiple programs are executing at the same time. This style of execution is called as **Parallel Execution**.

Disadvantage of STOS with multiple processor approach:

- Extremely expensive as it requires multiple processors.

Now we should perform multitasking but with overcoming the disadvantage of using multiple processors which makes the cost expensive, but how???



The solution for this was they thought of changing the software.

That is they replaced the S.T.O.S with **M.T.O.S (Multi Tasking Operating System)**.

Multitasking Operating System provides the interface for executing the multiple program tasks by single user at a same time on the one computer system.

This operating system is able to keep track of current task and can go from one to the other without losing information.

Example: user can open Gmail and Power Point same time.

The unit of the CPU depends upon the microprocessor. It can be nanoseconds, milliseconds, microseconds.

Here, the Operating System takes one unit of CPU time and divides into equal multiple slices.

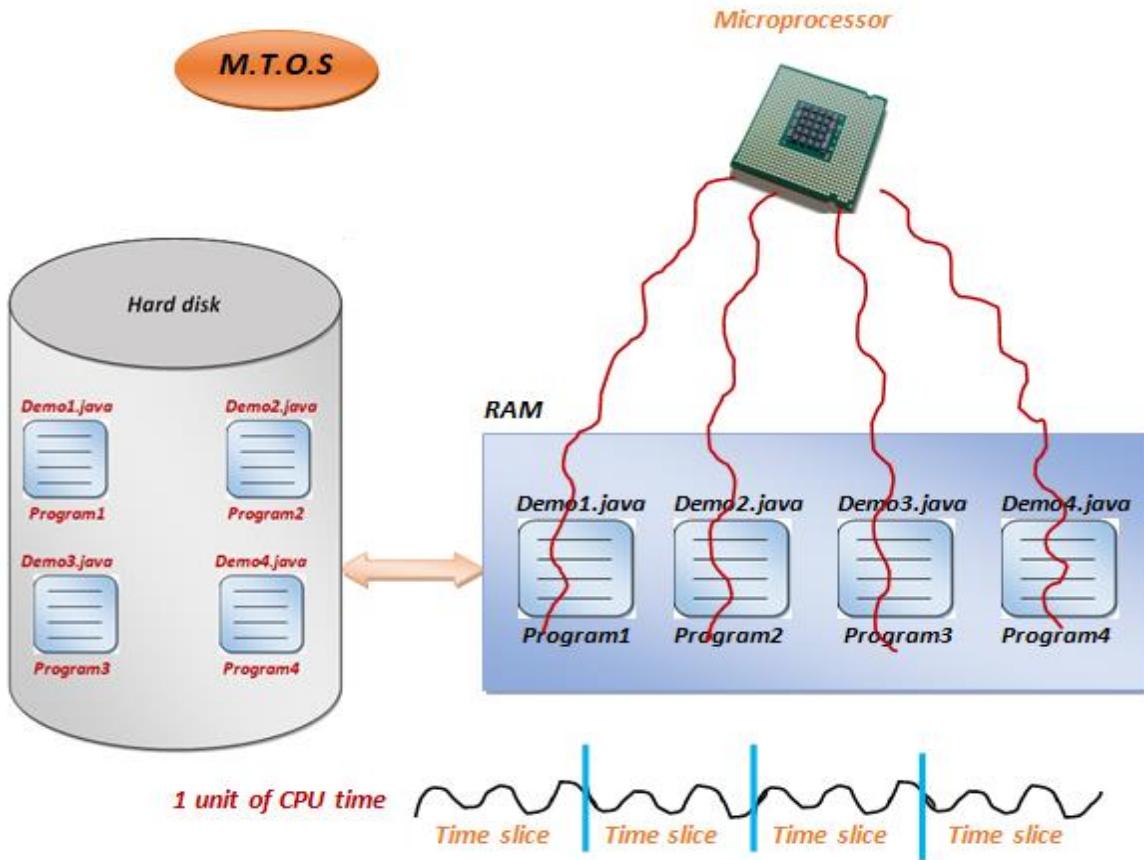
Then multiple processes are loaded onto the RAM from the hard disk as it is working with M.T.O.S. Now there is a single processor with multiple tasks.



How does that work:

The first slice of the CPU time is given to processor to execute one task, the moment the time is done then the control is taken away from the first task and the next slice of CPU time is given to the processor to execute the second task for that period of time. This switching continues with all the processes by giving time slice of CPU time till the task of all the processes is done.

This switching creates an illusion that multiple processes are executing at the same time. This kind of working is called ***Time-sharing Operating System or Time-slicing OS***. ***This style of execution is called as concurrent execution.***



Multitasking is achieved using single processor with the use of M.T.O.S

Advantage of M.T.O.S:

- Since multiple processors are not used cost is inexpensive.
- Concurrent execution that is time sharable in which, all tasks are allocated specific piece of time, so they do not need for waiting time for CPU.

Let us see whether our Java program efficiently utilise the CPU time.

In this example, let us perform **three activities**.

The first activity is **adding two numbers**, second is **printing characters** and the **third activity is printing numbers from 1 to 10**. All these activities are independent of each other.

Let us see which type of execution happens with this program.



```
import java.util.Scanner;
class Demo
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("Addition task started");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        int c = a+b;
        System.out.println(c);
        System.out.println("Addition task completed");

        System.out.println("Character printing task started");
        for(int i=65;i<=75;i++)
        {
            System.out.println((char)i);
            Thread.sleep(4000);
        }
        System.out.println("Character printing task completed");

        System.out.println("Number printing task started");
        for(int i=1;i<=10;i++)
        {
            System.out.println(i);
            Thread.sleep(4000);
        }
        System.out.println("Number printing task completed");
    }
}
```

In this program, when it enters the main method it gets started with the first activity of addition of two numbers.

But here according to the program since we have given Scanner input, the execution doesn't go forward to the next activities until and unless the input is given for the first activity of addition and this gets complete.

So don't you think there is loss of CPU time? Therefore, with this analysis we can understand that the type of execution that takes place here is **Sequential execution which makes the program slow and inefficient.**

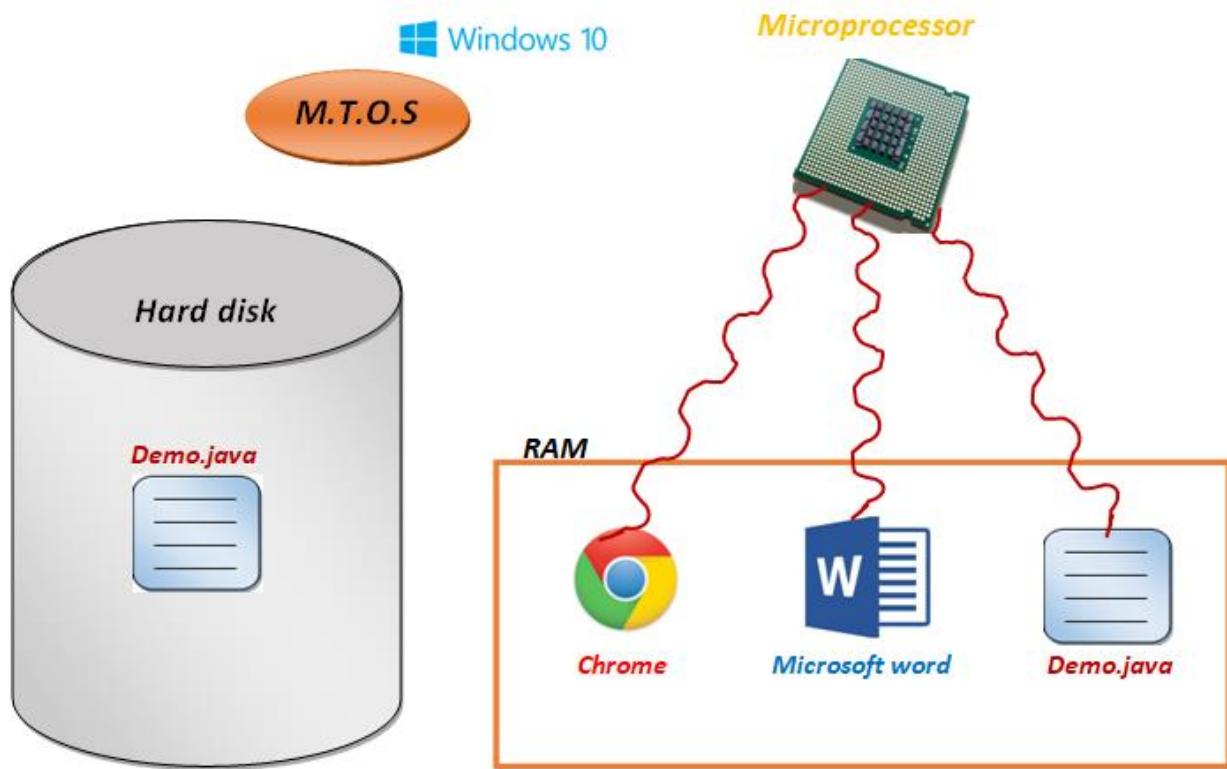
This is the disadvantage of single threaded approach.

Output:

```
Addition task started
Enter the first number
10
Enter the second number
20
30
Addition task completed
Character printing started
A
B
C
D
E
F
G
H
I
J
K
Character printing completed
Number printing started
1
2
3
4
5
6
7
8
9
10
Number printing completed
```



Consider the example shown below:

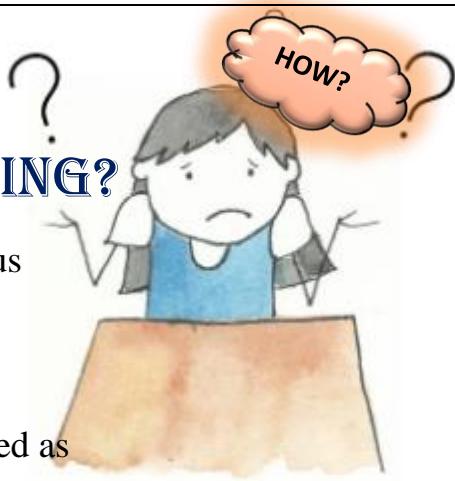


Let us assume our Java program is saved in the file Demo.java which is stored on the hard disk. The Operating System which we are using M.T.O.S which may be Windows 10 OS. Now our program is loaded onto the RAM and now it's called as a process. But in our computer we might have multiple processes on the RAM which are shown here as Google chrome, Microsoft Word. But all these have a single processor for execution. With the use of Time slicing method, one slice of time is given to chrome and when the time completes the control is given to next task i.e., Microsoft word and in this way it continues. This is **concurrent execution**. Now the java program must efficiently utilize the CPU time given to it but in the above code the execution not concurrent rather it was sequential and hence CPU time was getting wasted.



HOW TO ACHIEVE MULTITHREADING?

Before understanding how to achieve multithreading, let us Understand the concept of threads with the help of code.



The stack which is **automatically created by java** is called as **Main stack**.

To execute the contents of main stack, **thread scheduler automatically creates a line of execution called as main thread**.

A thread is a part of a process or is the path followed when executing a program.

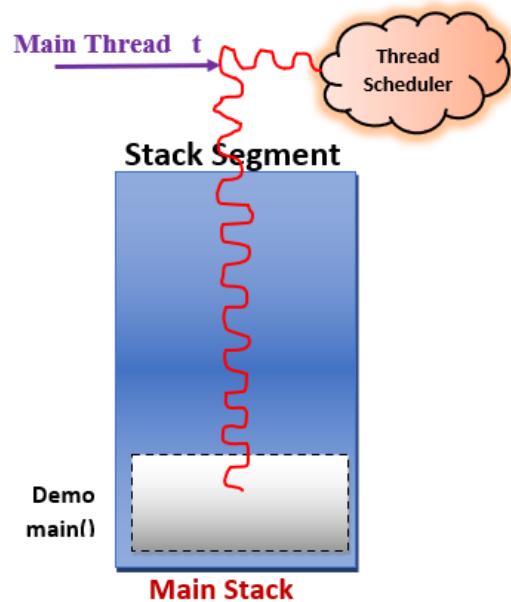
Thread scheduler in java is a software **that decides which thread should run**.

Only one thread at a time can run in a single process.

The code below consists of **main thread only** and hence it is a **single threaded program**.

Code Segment

```
class Demo
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println(t);
    }
}
```



OUTPUT:

Thread [main ,5, main]

Thread Name

Priority

Method Name

Can we achieve multithreading using multiple methods?

Let us write code with three different activities in three different methods and look at its output to understand this.

```
import java.util.Scanner;
class Demo1
{
    void fun1()
    {
        System.out.println("Addition task started");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        int c = a+b;
        System.out.println(c);
        System.out.println("Addition task completed");
    }
}
class Demo2
{
    void fun2() throws Exception
    {
        System.out.println("Character printing started");
        for(int i=65;i<=75;i++)
        {
            System.out.println((char)i);
            Thread.sleep(1000);
        }
        System.out.println("Character printing completed");
    }
}
class Demo3
{
    void fun3() throws Exception
    {
        System.out.println("Number printing started");
        for(int i=1;i<=10;i++)
        {
            System.out.println(i);
            Thread.sleep(1000);
        }
        System.out.println("Number printing Completed");
    }
}
```

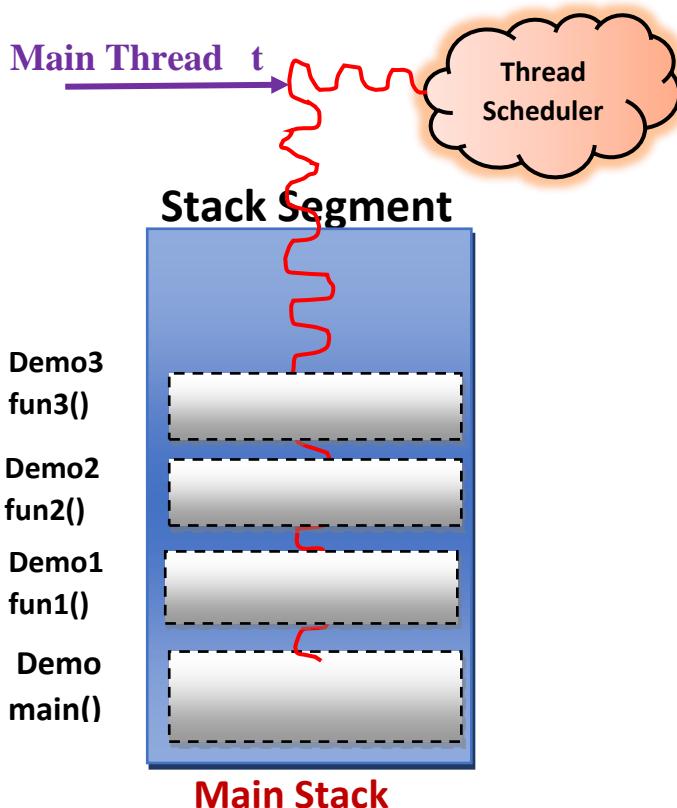


```

class Demo
{
    public static void main(String[ ] args) throws Exception
    {
        Demo1 d1 = new Demo1();
        Demo2 d2 = new Demo2();
        Demo3 d3 = new Demo3();
        d1.fun1();
        d2.fun2();
        d3.fun3();
    }
}

```

Let us see how stack segment of this code looks like.

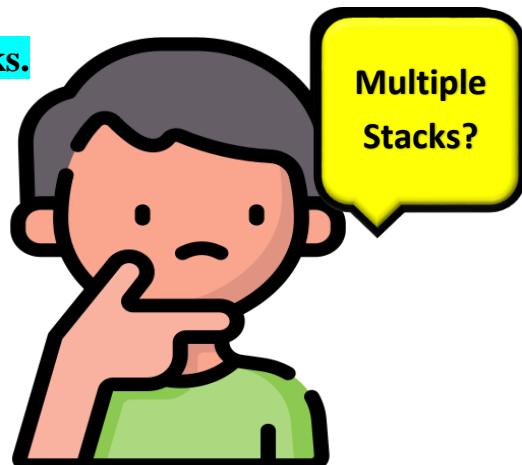


The stack segment above consists **of main stack** only which in turn consists of **main thread only** and hence it is a **single threaded program**.

In conclusion, **multithreading cannot be achieved using multiple methods**.

Then how do we achieve it???

We achieve by creating multiple-stacks.



OUTPUT:

```
Addition task started
Enter the first number
10
Enter the second number
20
30
Addition task completed
Character printing started
A
B
C
D
E
F
G
H
I
J
K
Character printing completed
Number printing started
1
2
3
4
5
6
7
8
9
10
Number printing Completed
```

Multithreading can be achieved by creating multiple stacks

Extra stack can be created by creating an object of thread class.

For every new stack, the thread scheduler will create a new thread.

**The independent activities which must be concurrently executed
should be placed within run().**

The programmer must never directly call run().

It must always be indirectly called using start().

**When start() is called, a new thread is created and code inside run()
method is executed in new thread while if you call run() directly no
new thread is created and code inside run() will execute on current
thread or main thread.**

Let us write code to understand this in detail



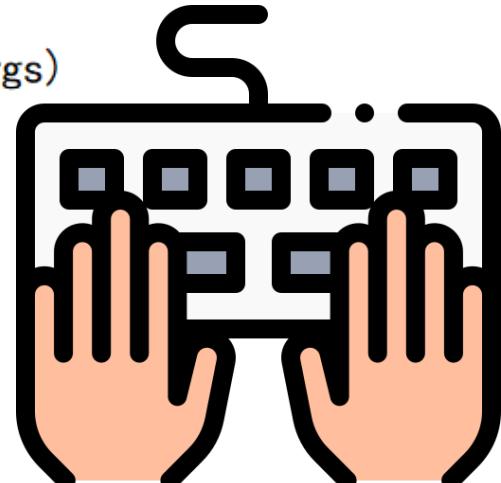
```
import java.util.Scanner;
class Demo1 extends Thread
{
    public void run()
    {
        System.out.println("Addition task started");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the first number");
        int a = scan.nextInt();
        System.out.println("Enter the second number");
        int b = scan.nextInt();
        int c = a+b;
        System.out.println(c);
        System.out.println("Addition task completed");
    }
}
class Demo2 extends Thread
{
    public void run()
    {
        System.out.println("Character printing started");
        for(int i=65;i<=75;i++)
        {
            System.out.println((char)i);
            try
            {
                Thread.sleep(4000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
        System.out.println("Character printing completed");
    }
}
class Demo3 extends Thread
{
    public void run()
    {
        System.out.println("Number printing started");
    }
}
```

```

        for(int i=1;i<=10;i++)
        {
            System.out.println(i);
            try
            {
                Thread.sleep(4000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occured");
            }
        }
        System.out.println("Number printing Completed");
    }

class Demo
{
    public static void main(String[ ] args)
    {
        Demo1 d1 = new Demo1();
        Demo2 d2 = new Demo2();
        Demo3 d3 = new Demo3();
        d1.start();
        d2.start();
        d3.start();
    }
}

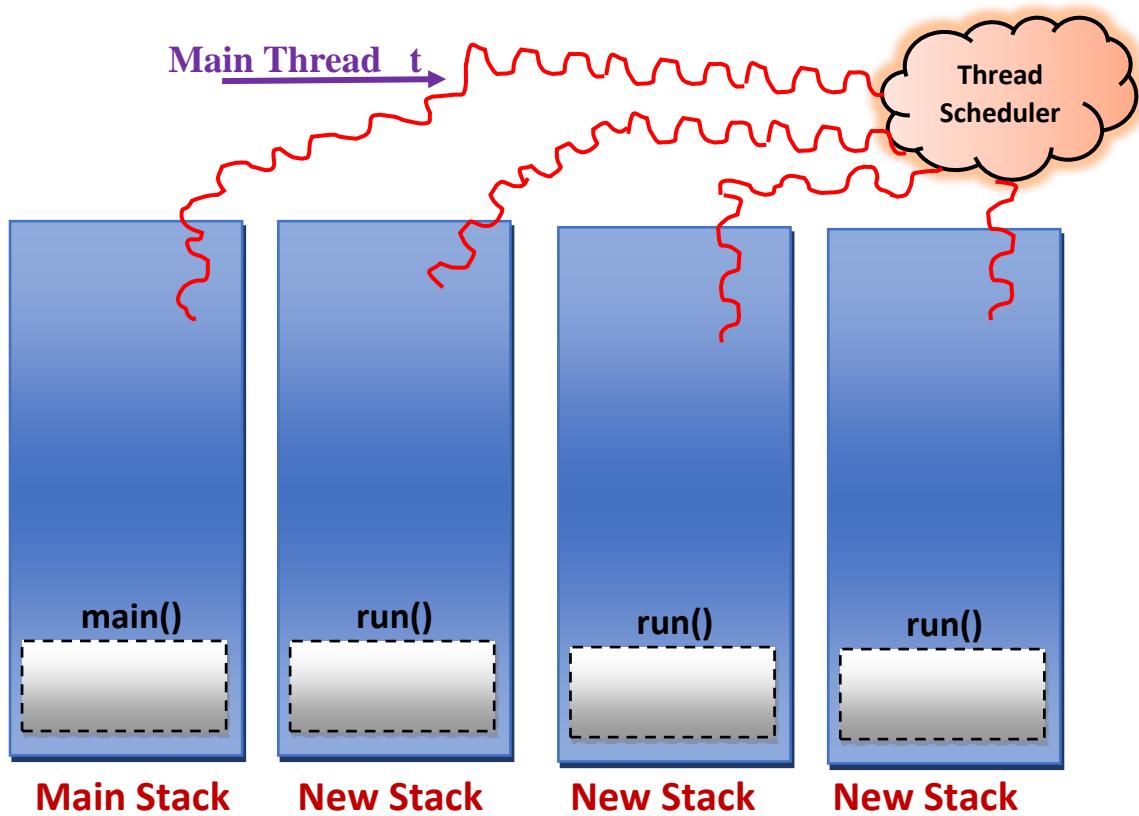
```



In the above code, all the **three activities** that is addition, printing alphabets and **printing numbers** are written **inside run()** and **every class is extending thread class**. Since every class is extending thread class, when an object is created of these thread classes, **new stack gets created** and now **thread scheduler creates new thread** for each of these stacks due to which **concurrent execution takes place and CPU time is utilised efficiently**.

Let us look at its output and stack segment.

STACK SEGMENT:



OUTPUT:

```
Addition task started
Number printing started
Character printing started
A
1
Enter the first number
2
Enter the second number
B
2
C
3
D
4
E
5
F
6
400
402
Addition task completed
G
7
H
8
I
9
J
10
K
Number printing Completed
Character printing completed
```



Types of Thread

User thread

Daemon Thread

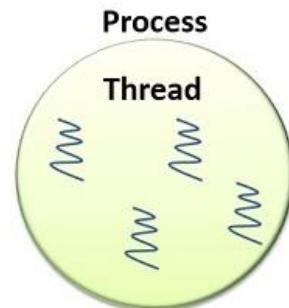


A thread is used to perform parallel execution in Java e.g. while rendering screen your program is also downloading the data from the internet in the background. There are two types of threads in Java, **user thread and daemon thread**, both of which can be used to implement parallel processing in Java depending upon priority and importance of the task. The main difference between a user thread and a daemon thread is that your Java

program will not finish execution until one of the user thread is live. JVM will wait for all active user threads to finish their execution before it shutdown itself. On the other hand, a **daemon thread doesn't get that preference, JVM will exit and close the Java program even if there is a daemon thread running in the background.** They are treated as **low priority threads** in Java, that's why they are more suitable for non-critical background jobs.

Let's see with an example:

```
class Demo1 extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("User thread executing");
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
    }
}
```



```

class Demo2 extends Thread
{
    public void run()
    {
        for(;;)
        {
            System.out.println("Daemon thread executing");
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
    }
}
class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Main() started execution");
        Demo1 d1 = new Demo1();
        Demo2 d2 = new Demo2();
        d2.setDaemon(true);
        d1.start();
        d2.start();
        System.out.println("Main() completed execution");
    }
}

```

Output:

Main() started execution
Main() completed execution
User thread executing
Daemon thread executing
Daemon thread executing
User thread executing
Daemon thread executing
Press any key to continue . . .



www.clipartof.com · 1246277



In the above example we see that the daemon thread stopped executing right after user thread completed it's execution.

Difference between User thread and Daemon thread

- **JVM doesn't wait for daemon thread to finish**

First and foremost difference is that JVM will not wait for daemon thread to finish their work but it will wait for any active user thread.

- **JVM itself creates Daemon thread**

Another difference between them is that daemon thread is mostly created by JVM e.g. for some garbage collection job. On the other hand user thread is usually created by the application for executing some task concurrently.

- **Daemon thread is not used for critical task**

Another key difference between daemon and user thread is that daemons are not used for any critical task. Any important task is done by non-daemon or user thread. A daemon thread is generally used for some background tasks which are not critical.

- **Thread Priority**

As compared to user thread, the daemon threads are low priority thread. Which means they won't get CPU as easily as a user thread can get.

- **JVM closes daemon thread**

The user thread is closed by application or by itself but JVM will force daemon thread to terminate if all user threads have finished their execution. A daemon thread cannot keep the JVM running but a user thread can. This is the most critical difference between daemon and user thread which you should remember.

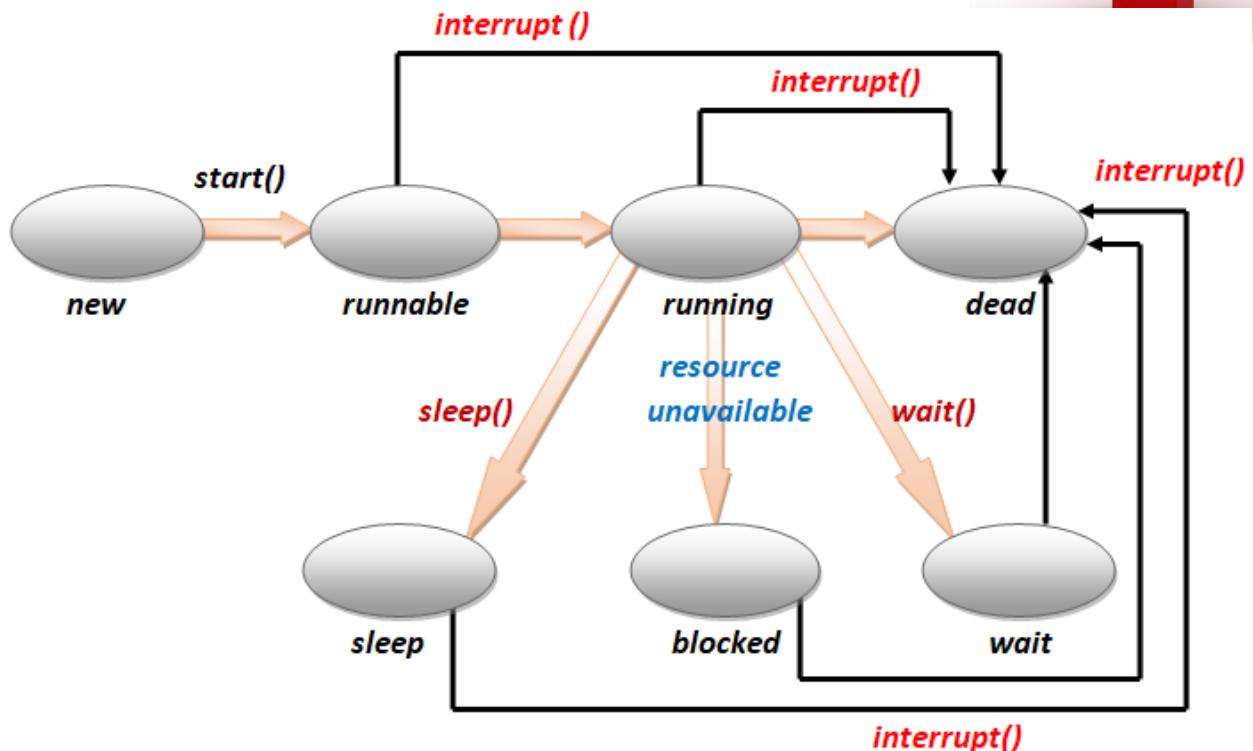


Multithreading Continued

Which States?

Life Cycle of a thread (Thread State diagram)

The moment we create a thread till the moment the thread finishes its execution, a thread goes through different phases or states.



- **New:** As soon as, you create a thread, it's in *New state*. It remains in this state until the program starts the thread using its **start()** method.
- **Runnable:** A thread that is ready to run is moved to the *Runnable state* after invocation of **start()** method, but the thread scheduler has not selected it to be the running thread.
- **Running:** The thread is in *running state* if the thread scheduler has selected it and given time to run.
- **Blocked:** A *Runnable thread* transitions to the *blocked state* when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.

- **Waiting:** Usually program put a thread in *wait state* by calling **`wait()`** method because something else needs to be done prior to what current thread is doing.
- **Dead:** A thread enters this state when it successfully completes its task or otherwise terminated due to any error or even if it is forcefully killed.

Threads from runnable, running, sleep, blocked and wait state can be moved to dead state by calling **`interrupt()`** method.

Now let us understand single thread program using thread state diagram.

```
class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("main thread started execution");

        System.out.println("main thread is executing");
        Thread.sleep(5000);
        System.out.println("main thread is executing");
        Thread.sleep(5000);
        System.out.println("main thread is executing");
        Thread.currentThread().interrupt();
        Thread.sleep(5000);

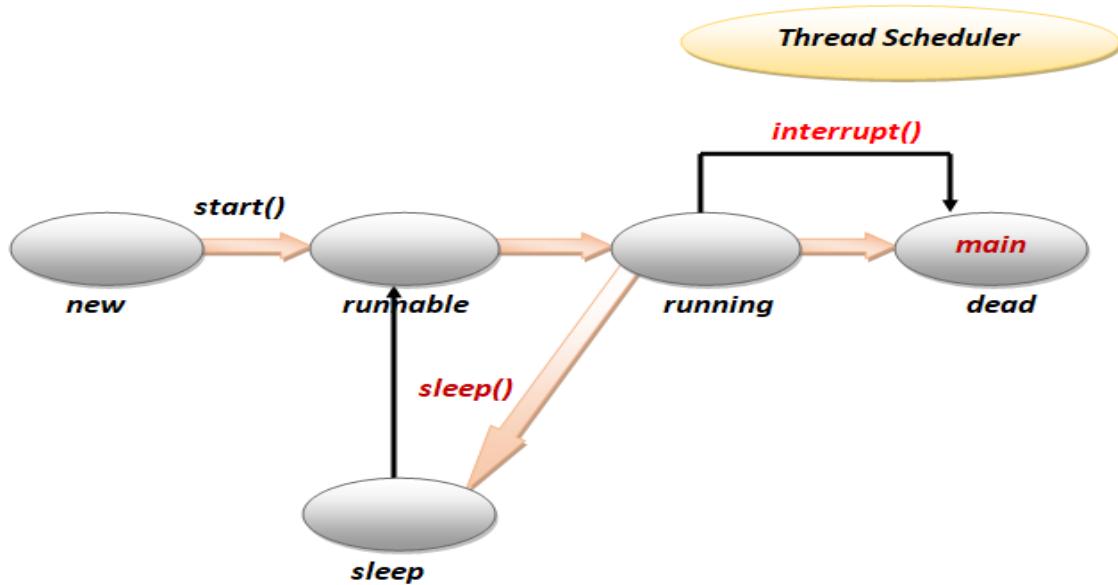
        System.out.println("main thread completed execution");
    }
}
```

The first thread that is created in every Java program is the main thread. As soon as, the thread is created it exists in the ***new state***. Before the thread start executing it should be brought to the ***runnable state*** by calling **`start()`** method making the thread ready to run.

The duty of the thread scheduler is to ensure whether the CPU time is efficiently utilized or not. If the running state is empty and no thread is executing it will take the main thread from the runnable state to the ***running state***.

Whenever the thread is executing is called by the **`sleep()`** method, then the running thread stop execution and moves to the ***sleep state***.

When the sleep time is finished it again moves back to the runnable state and the flow repeats till it completes its execution. If an *interrupt()* method is called then the main thread which is alive is killed and moved to the ***dead state***. And when this happens Java creates an exception.



Since this is a single threaded program when the running state is empty there would be no other thread to execute and **therefore the CPU time is not been used efficiently.**

Output:

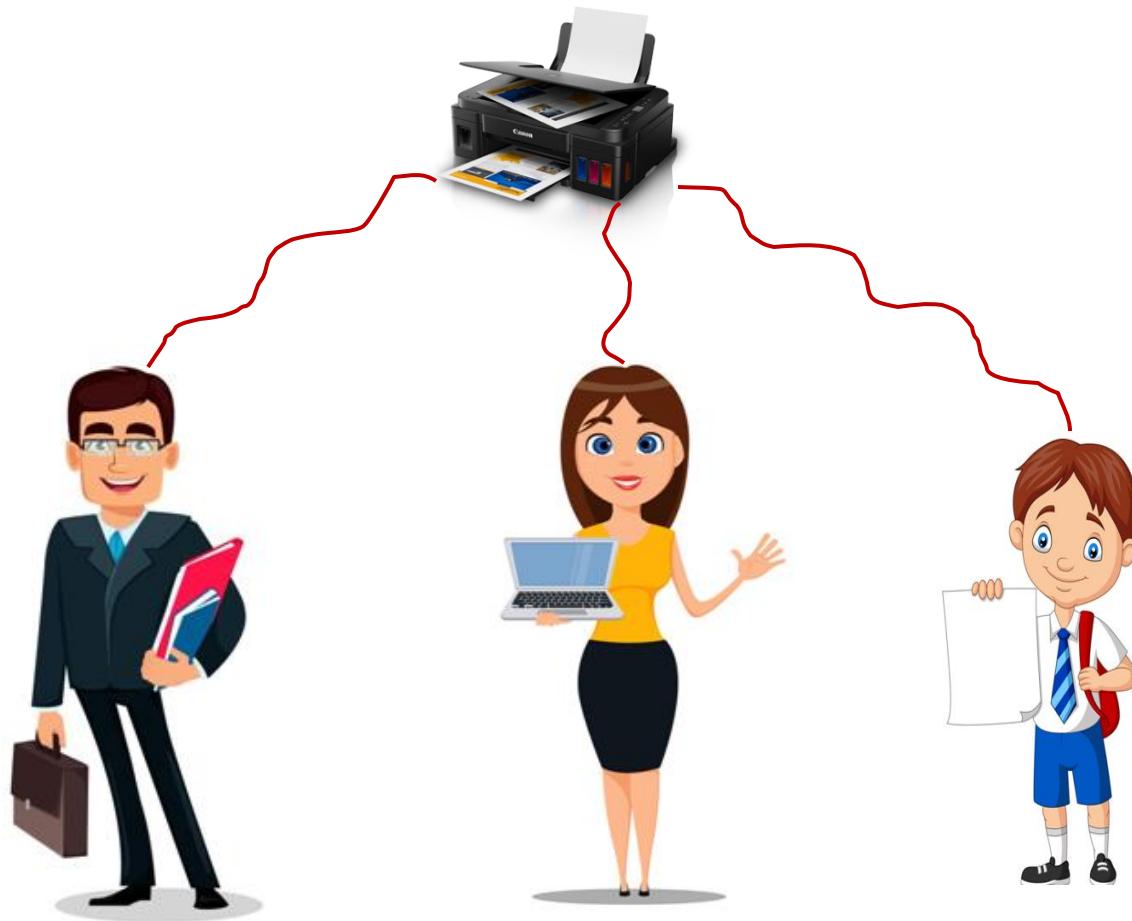
```
main thread started execution
main thread is executing
main thread is executing
main thread is executing
Exception in thread "main" java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at Demo.main(Demo.java:13)
```

Disadvantage of Multithreading.

Applying the approach of multithreading at all cases is not a good practice. In some cases we have to use Single threaded approach.

Let's understand this with a scenario.

Now imagine we have a single printer and there are three people as Men, Women and a Child and they want to print a document. Now should this one resource be used by all these people at the same time?



If this was the case, then when print is given, the printer takes the input from all these people and concurrent execution happens.

The printer at any given time should take the input from one person for a proper document to be printed.

Let us code this Scenario

```
class Printer implements Runnable
{
    public void run()
    {
        String name = Thread.currentThread().getName();

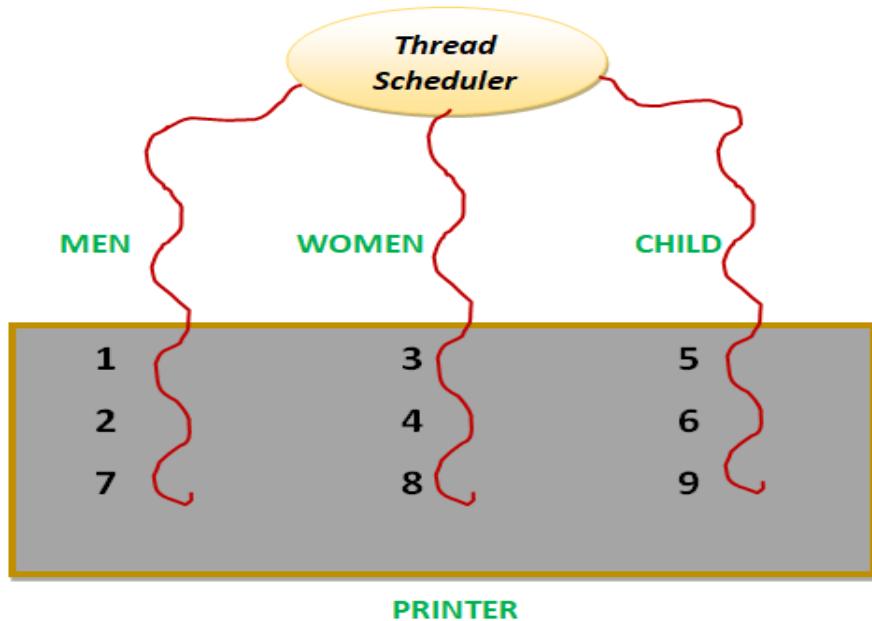
        System.out.println(name+ " started printing");
        for(int i=1;i<=3;i++)
        {
            System.out.println(name+ " is printing");
            try
            {
                Thread.sleep(4000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
        System.out.println(name+ " completed printing");
    }
}

class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        Printer p = new Printer();

        Thread t1 = new Thread(p);
        Thread t2 = new Thread(p);
        Thread t3 = new Thread(p);
        t1.setName("MEN");
        t2.setName("WOMEN");
        t3.setName("CHILD");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

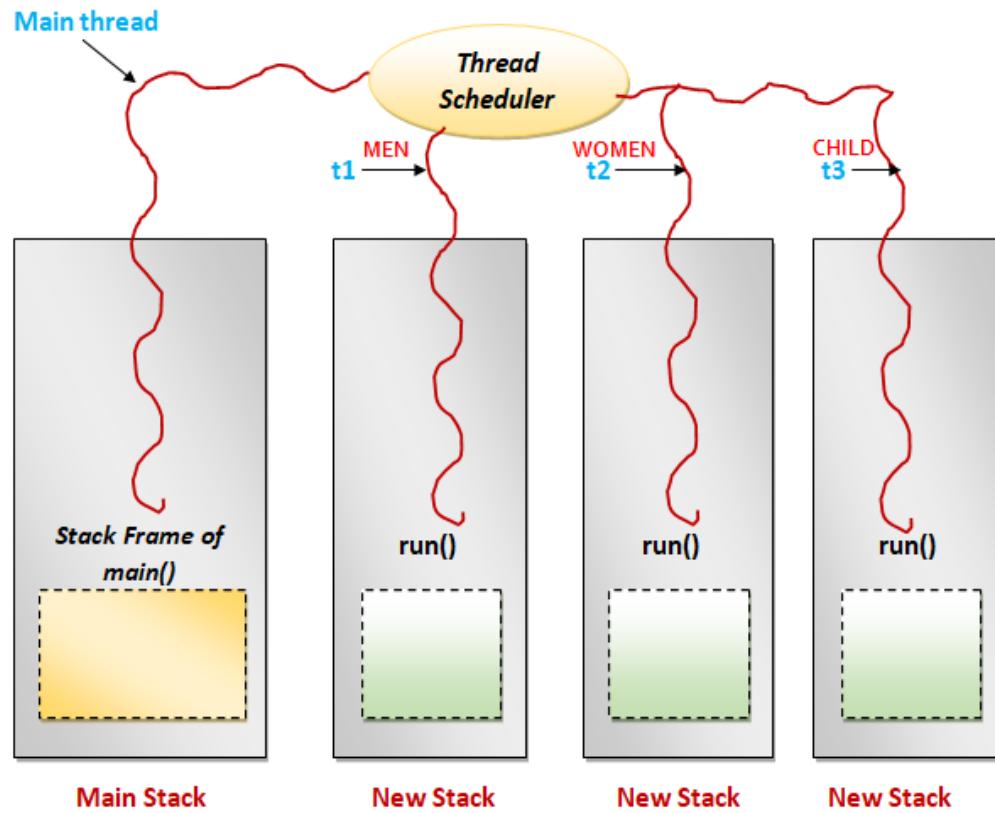




In this program, in the Stack segment the stack frame of main method gets created and the Thread scheduler will create the main thread.

When we create an object of a class which is implemented a runnable interface then a new Stack will not get created. If we want to create a new thread, then we should create an object of Thread class and we have to mention which run method should it refer to. Similarly we create three new stacks for Men, Women and Child.

Then we start the execution by calling the start() method with their corresponding references.



Output:

```
WOMEN started printing
CHILD started printing
MEN started printing
CHILD is printing
WOMEN is printing
MEN is printing
WOMEN is printing
CHILD is printing
MEN is printing
CHILD is printing
MEN is printing
WOMEN is printing
CHILD completed printing
MEN completed printing
WOMEN completed printing
```

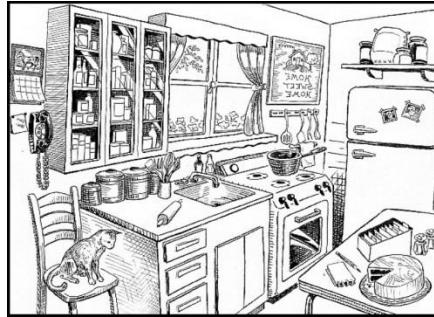
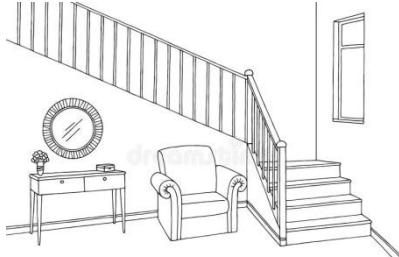
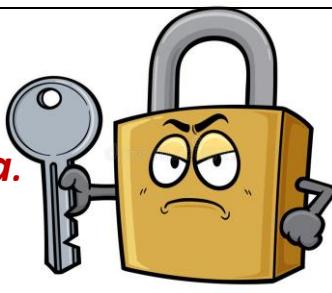


Here in the above program the problem was all the three threads were accessing the same resource at the same time and therefore the final result we got was in the above manner.

Let us understand how to resolve this problem? What is the solution?



Concept of Lock in Java.



Now in these two cases, a single resource (such as the hall or kitchen) can be used by multiple people (threads). This is allowed/permitted/acceptable.

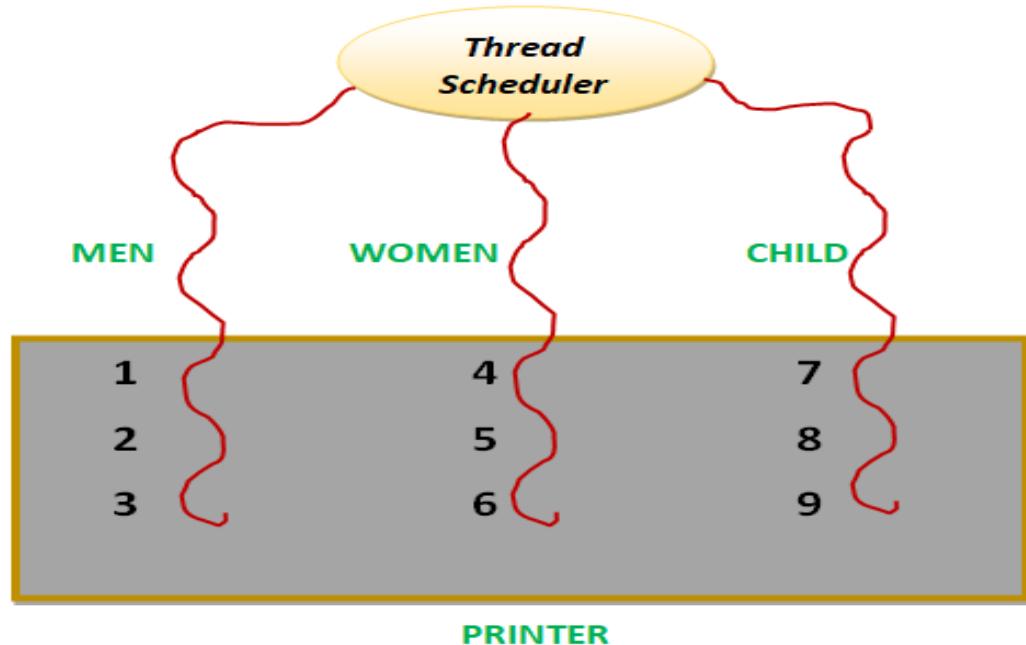
But the same approach is not preferred here. Therefore, we have to restrict people (threads) by applying lock to the resource.

Similarly in Java program, a common resource is accessed by multiple threads at the same time. This will result in unexpected output and hence must be prevented. Such statements which only a single thread must access at any given point of time are referred to as **Monitor or Semaphore**.

A semaphore can be achieved in Java by using the facility of locks. A lock can be implemented by using the synchronized keyword.

Multithreading with Synchronization.

In this case, the thread scheduler should not give access to any other thread until the current thread completes its execution. This can be done by Sequential execution with applying locks to the threads.



Case-1:

```
class Printer implements Runnable
{
    synchronized public void run()
    {
        String name = Thread.currentThread().getName();

        System.out.println(name+ " started printing");
        for(int i=1;i<=3;i++)
        {
            System.out.println(name+ " is printing");
            try
            {
                Thread.sleep(4000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
        System.out.println(name+ " completed printing");
    }
}
```

```

class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        Printer p = new Printer();

        Thread t1 = new Thread(p);
        Thread t2 = new Thread(p);
        Thread t3 = new Thread(p);
        t1.setName("MEN");
        t2.setName("WOMEN");
        t3.setName("CHILD");

        t1.start();
        t2.start();
        t3.start();
    }
}

```

So now by applying lock to *run()* method, no other threads will be able to access it.

Output:

```

MEN started printing
MEN is printing
MEN is printing
MEN is printing
MEN completed printing
CHILD started printing
CHILD is printing
CHILD is printing
CHILD is printing
CHILD completed printing
WOMEN started printing
WOMEN is printing
WOMEN is printing
WOMEN is printing
WOMEN completed printing

```



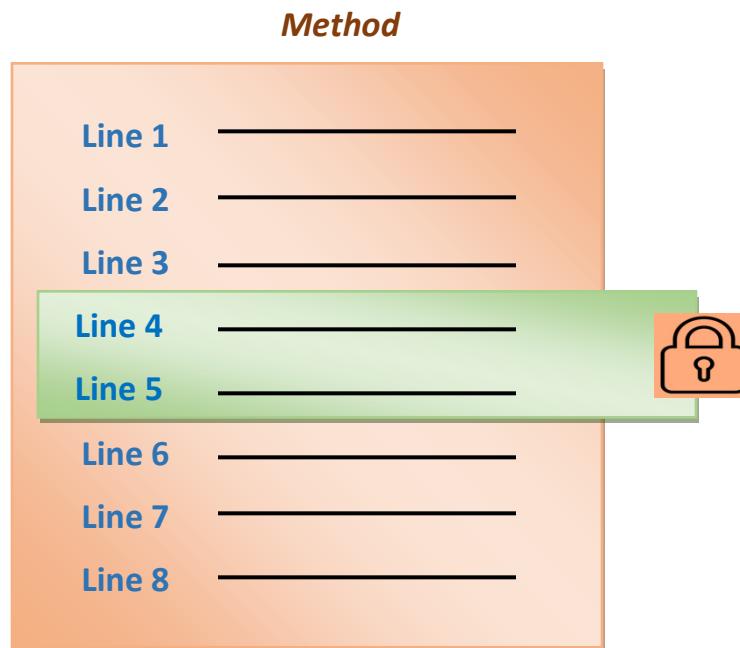
Why Synchronization?

- To prevent thread interference.
- To prevent consistency problem.

Case-2:

If all the lines in method should be locked then use “synchronized” to the entire method.

Whereas, if only certain lines of the method should be locked then use “synchronized” only to those lines by putting it in a block. And name the block as synchronized.



Code:

```
class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        Test t = new Test();

        Thread t1 = new Thread(t);
        Thread t2 = new Thread(t);

        t1.setName("ONE");
        t2.setName("TWO");

        t1.start();
        t2.start();
    }
}
```

```
class Test implements Runnable
{
    public void run()
    {
        String name = Thread.currentThread().getName();

        try
        {
            System.out.println(name+ " is executing first line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing second line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing third line");
            Thread.sleep(3000);

            synchronized(this)
            {
                System.out.println(name+ " is executing fourth line");
                Thread.sleep(3000);
                System.out.println(name+ " is executing fifth line");
                Thread.sleep(3000);
            }

            System.out.println(name+ " is executing sixth line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing seventh line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing eighth line");
            Thread.sleep(3000);
        }

        catch (Exception e)
        {
            System.out.println("Some problem occurred");
        }
    }
}
```

Output:

```
ONE is executing first line
TWO is executing first line
TWO is executing second line
ONE is executing second line
TWO is executing third line
ONE is executing third line
TWO is executing fourth line
TWO is executing fifth line
TWO is executing sixth line
ONE is executing fourth line
ONE is executing fifth line
TWO is executing seventh line
ONE is executing sixth line
TWO is executing eighth line
ONE is executing seventh line
ONE is executing eighth line
```

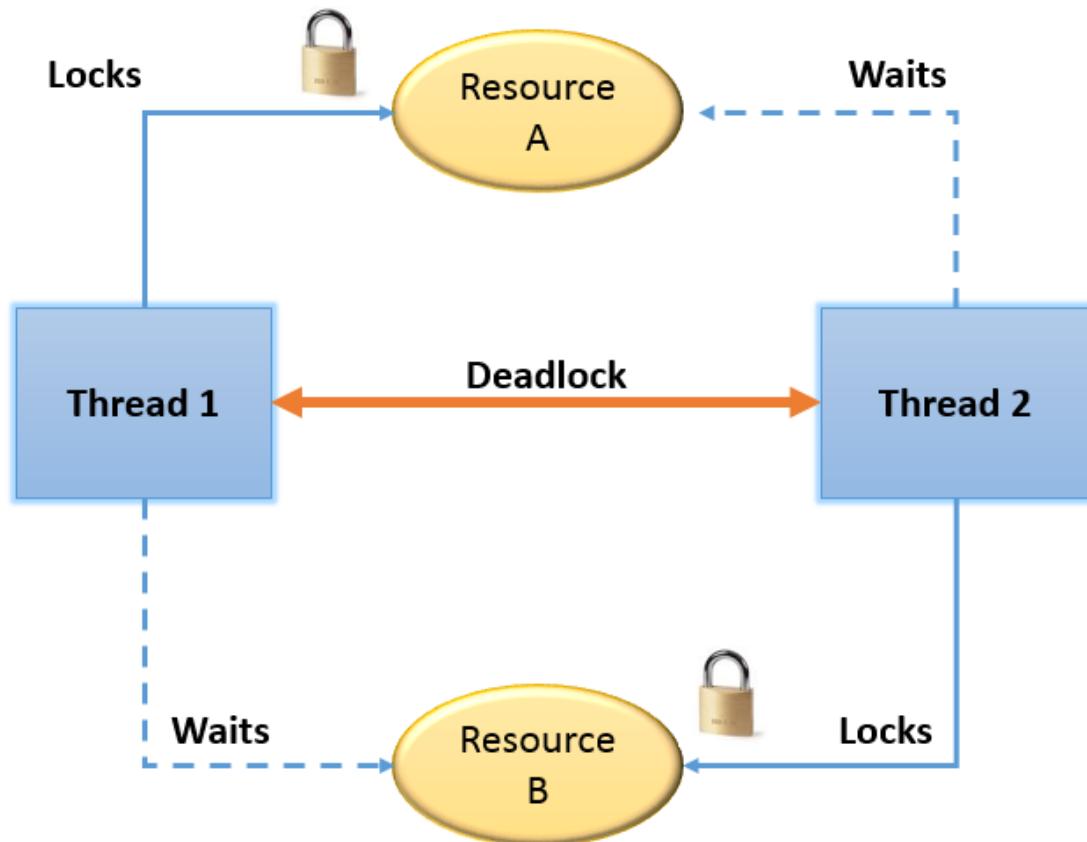


DEADLOCK

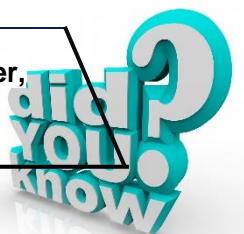
Deadlock in java is a situation where **two or more threads are blocked forever**, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A **Java multithreaded program** may suffer from the deadlock condition because the **synchronized keyword** causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.



Consider the below diagram, thread1 is waiting for an object lock, that is acquired by another thread2 and second thread2 is waiting for an object lock that is acquired by first thread1. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



The QWERTY keyboard was designed to slow you down. If you want to type faster, try the [Dvorak Keyboard](#).

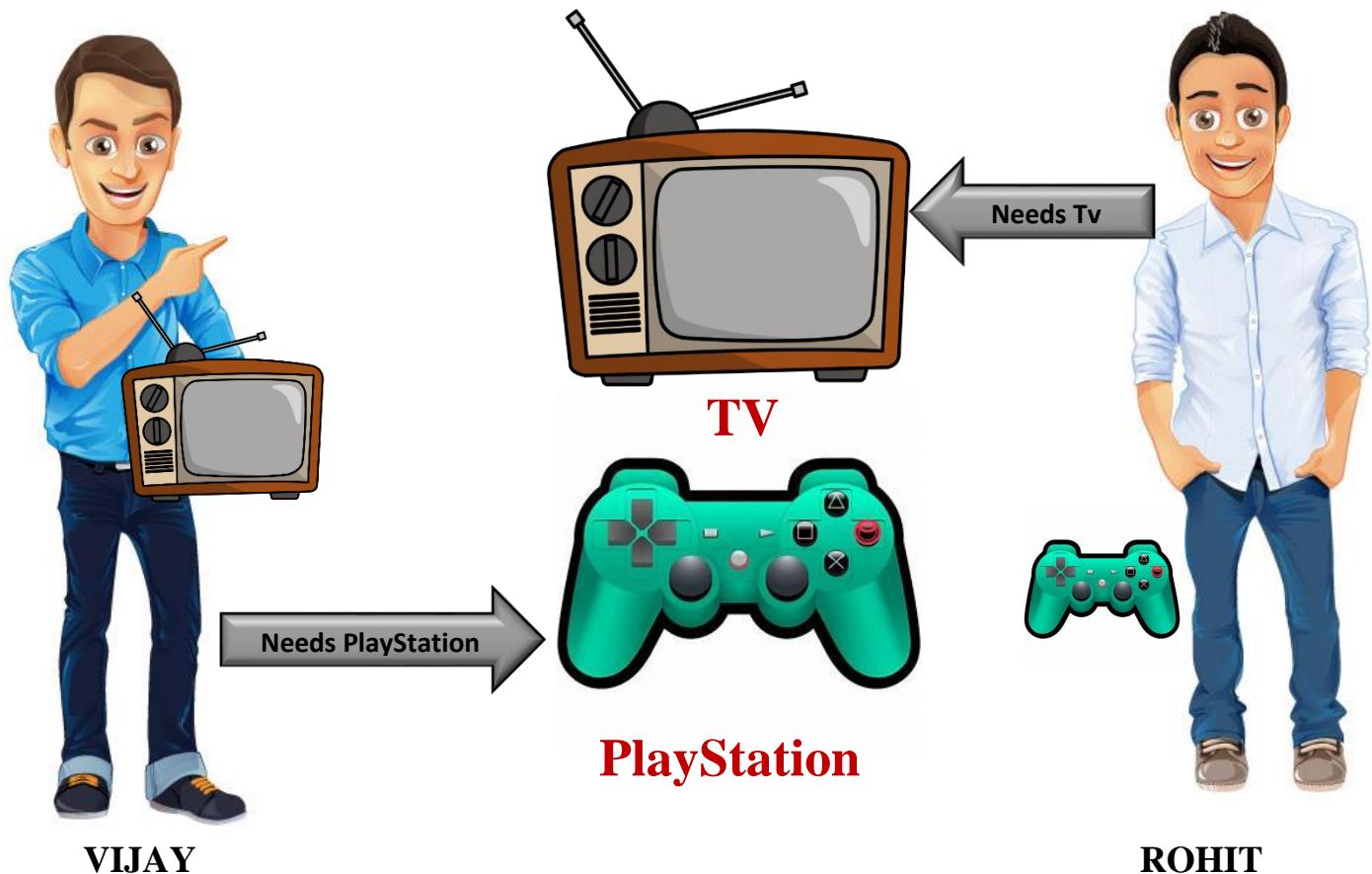


Let us consider a scenario to understand deadlocks in detail



Scenario:

There are two threads **rohit** and **vijay** and there are two resources **TV** and **PlayStation**. Both the threads now want to acquire both the resources. Rohit acquires **resource2** first which is PlayStation. Vijay acquires **resource1** first which is TV. Now **vijay wants playstation** which is already acquired by rohit and **rohit wants Tv** which is already acquired by vijay. In this way, they enter into a state where both the **resources are locked** and threads enter into deadlock.



Let us start coding the above scenario



```

class Family implements Runnable
{
    String resource1 = "Tv";
    String resource2 = "Playstation";
    public void run()
    {
        String name = Thread.currentThread().getName();
        if(name.equals("Rohit") == true)
        {
            rohitAccquiredResource();
        }
        else
        {
            vijayAccquiredResource();
        }
    }
    void rohitAccquiredResource()
    {
        synchronized(resource2)
        {
            try
            {
                System.out.println("Rohit acquired Platstation");
                Thread.sleep(1000);
                synchronized(resource1)
                {
                    System.out.println("Rohit accquired Tv");
                    Thread.sleep(1000);
                }
            }
            catch (Exception e)
            {
                System.out.println("Rohit failed");
            }
        }
    }
    void vijayAccquiredResource()
    {
        synchronized(resource1)
        {
    
```



```

try
{
    System.out.println("Vijay acquired Tv");
    Thread.sleep(1000);
    synchronized(resource2)
    {
        System.out.println("Vijay acquired Playstation");
        Thread.sleep(1000);
    }
}
catch (Exception e)
{
    System.out.println("Vijay failed");
}
}
}

class Launch1
{
    public static void main(String[] args)
    {
        Family f = new Family();
        Thread t1 = new Thread(f);
        Thread t2 = new Thread(f);
        t1.setName("Rohit");
        t2.setName("Vijay");
        t1.start();
        t2.start();
    }
}

```

OUTPUT

Rohit acquired PlayStation
Vijay acquired Tv
 Cursor keeps blinking as both
the threads are locked and
execution never proceeds.

join() in Java

java.lang.Thread class provides the **join()** method which allows one thread to wait until another thread completes its execution. If **t** is a Thread object whose thread is currently executing, then **t.join()** will make sure that **t** is terminated before the next instruction is executed by the program.

If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so **you should not assume that join will wait exactly as long as you specify.**

Let's see one example:

```
class Demo1 extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            try
            {
                System.out.println("JAVA");
                Thread.sleep(2000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred.");
            }
        }
    }
}
class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main thread started");
        Demo1 d1 = new Demo1();
        d1.start();
        d1.join(); //d1 thread is making main thread to wait until it completes execution
        System.out.println("Main thread completed");
    }
}
```



www.clipartof.com · 1246277

Output:

```
Main thread started  
JAVA  
JAVA  
JAVA  
JAVA  
JAVA  
Main thread completed  
Press any key to continue . . .
```

In the above output we see that main thread is waiting for d1 thread to complete execution.

Key Points

There are three overloaded join functions.

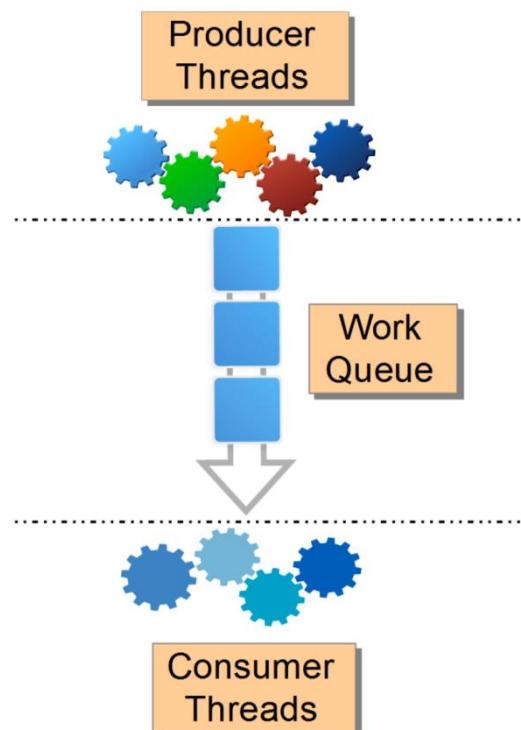
- **join():** It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.
- **join(long millis) :**It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds).
- **join(long millis, int nanos):** It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds + nanos).

understood

Producer-Consumer problem in java

In computing, **the producer-consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which **share a common, fixed-size buffer used as a queue**.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.



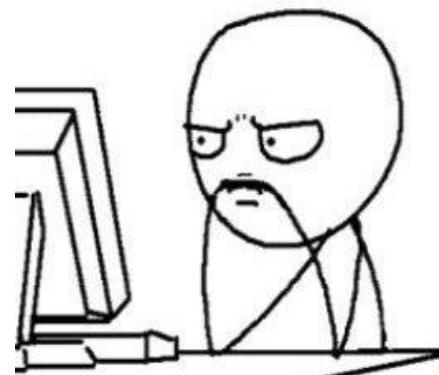
Let us see a case where producer is producing n number of values and Consumer is consuming the values produced.

```
class Queue
{
    int x;
    void store(int j)
    {
        x = j;
        System.out.println("Produced"+x);
    }
    void retrieve()
    {
        System.out.println("Consumed"+x);
    }
}
```

```

class Producer extends Thread
{
    Queue a; // reference to queue class which is pointing to queue object
    Producer (Queue q)//constructor taking queue type reference as input
    {
        a = q;
    }
    public void run()
    {
        int i=1;
        for(;;)
        {
            a.store(i++);
        }
    }
}
class Consumer extends Thread
{
    Queue b;
    Consumer(Queue q)
    {
        b = q;
    }
    public void run()
    {
        for(;;)
        {
            b.retrieve();
        }
    }
}
class Demo
{
    public static void main(String[] args)
    {
        Queue q = new Queue();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);
        p.start();
        c.start();
    }
}

```



Output:

We see in the below output that the process is so fast that sometimes we see a value getting consumed before the produced value is printed. And also all the values that are getting produced are not getting consumed.

```

Produced1921
Consumed1921
Consumed1922
Consumed1922
Consumed1922
Consumed1922
Produced1922
Consumed1922
Produced1923
Produced1924
Produced1925
Produced1926
Produced1927
Consumed1923
Consumed1928
Consumed1928
Terminate batch job (Y/N)?

```



The above code is partially correct let us see how to overcome the problems in the below example:

In the above code you will have to make changes in class Queue, so let's see what are those changes,

```

class Queue
{
    int x;
    boolean is_data_present=false;
    synchronized void store(int j)
    {
        try
        {
            if(is_data_present==false)
            {
                x = j;
                System.out.println("Produced"+x);
                is_data_present = true;
                notify();
            }
            else
            {
                wait();
            }
        }
        catch (Exception e)
        {
            System.out.println("Some problem occured");
        }
    }
}

```



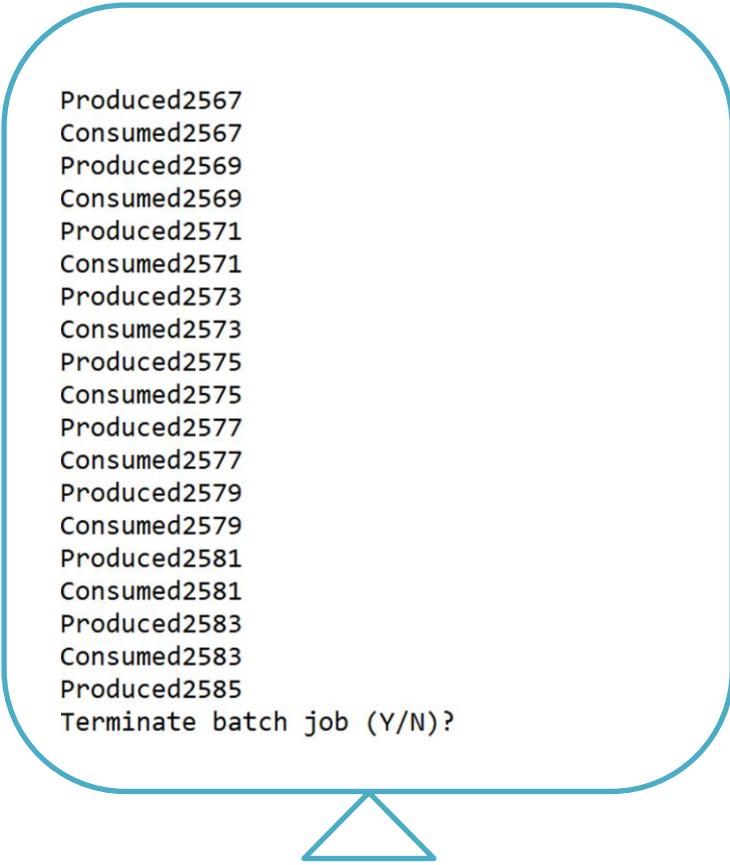
www.clipartof.com · 1246277

```

synchronized void retrieve()
{
    try
    {
        if(is_data_present==true)
        {
            System.out.println("Consumed"+x);
            is_data_present = false;
            notify();
        }
        else
        {
            wait();
        }
    }
    catch (Exception e)
    {
        System.out.println("Some problem occurred");
    }
}
}

```

Let us now see the output for the same code after making the above changes:



```

Produced2567
Consumed2567
Produced2569
Consumed2569
Produced2571
Consumed2571
Produced2573
Consumed2573
Produced2575
Consumed2575
Produced2577
Consumed2577
Produced2579
Consumed2579
Produced2581
Consumed2581
Produced2583
Consumed2583
Produced2585
Terminate batch job (Y/N)?

```

We can see that only after the value is produced, consumer is able to consume. And because of the fast execution some values are not printing but that doesn't mean they are not produced.

Second method to achieve Multi-threading

- **Implementing a runnable Interface**

Java runnable is an interface used to **execute code on a concurrent thread**.

It is an interface which is implemented by any class if we want that the instances of that class should be executed by a thread.

The runnable interface has an undefined method **run()** with void as return type, and it takes in no arguments. The method summary of the run() method is given below-

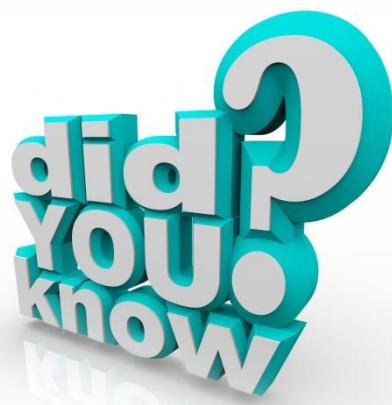
Method	Description
public void run()	This method takes in no arguments. When the object of a class implementing Runnable class is used to create a thread, then the run method is invoked in the thread which executes separately.

The runnable interface provides a standard set of rules for the instances of classes which wish to execute code when they are active. **The most common use case of the Runnable interface is when we want only to override the run method.**

When a thread is started by the object of any class which is implementing Runnable, then it invokes the run method in the separately executing thread.

A class that implements Runnable runs on a different thread without subclassing Thread as it instantiates a Thread instance and passes itself in as the target. This becomes important as classes should not be subclassed unless there is an intention of modifying or enhancing the fundamental behavior of the class.

Runnable class is extensively used in network programming as each thread represents a separate flow of control. **Also in multi-threaded programming, Runnable class is used.** This interface is present in **java.lang** package.



Implementing Runnable

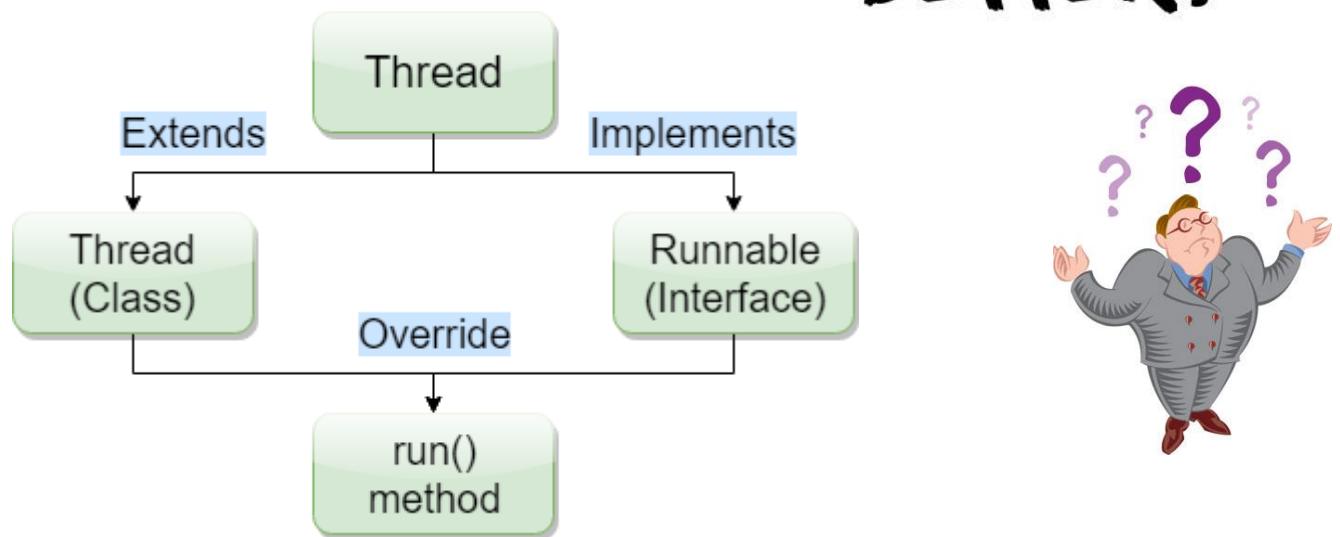
It is the easiest way to create a thread by implementing Runnable. One can create a thread on any object by implementing Runnable. To implement a Runnable, one has only to implement the run method.

```
public void run()
```

In this method, we have the code which we want to execute on a concurrent thread. In this method, we can use variables, instantiate classes, and perform an action like the same way the main thread does. The thread remains until the return of this method. The run method establishes an entry point to a new thread.

Thread vs. Runnable

WHICH ONE IS BETTER?



There are several differences between Thread class and Runnable interface based on their performance, memory usage, and composition.

- By extending thread, there is overhead of additional methods, i.e. they consume excess or indirect memory, computation time, or other resources.

- Since in Java, we can only extend one class, and therefore **if we extend Thread class, then we will not be able to extend any other class**. That is why we should **implement Runnable interface to create a thread**.
- **Runnable makes the code more flexible**, if we are extending a thread, then our code will only be in a thread whereas, **in case of runnable, one can pass it in various executor services, or pass it to the single-threaded environment**.
- Maintenance of the code is easy if we implement the Runnable interface.
- We can **achieve basic functionality of a thread** by extending Thread class because it **provides some inbuilt methods like yield(), interrupt()** etc. that are not available in Runnable interface.

We can clearly see from the above points that implementing runnable is better approach.

Multi-threading using single run()

Previously we saw how to achieve multi-threading with multiple run() or by overriding run(). Now let's see how to achieve it by making use of single run().

Let us try to understand using the same example,

```
import java.util.Scanner;
class Demo1 implements Runnable
{
    public void run()
    {
        Thread t = Thread.currentThread();
        String name = t.getName();
        if(name.equals("ADD")==true)
        {
            add();
        }
        else if(name.equals("CHAR")==true)
        {
            charPrint();
        }
        else
        {
            numPrint();
        }
    }
}
```



```
void add()
{
    System.out.println("Addition task started");
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter the first number");
    int a = scan.nextInt();
    System.out.println("Enter the second number");
    int b = scan.nextInt();
    int c = a+b;
    System.out.println(c);
    System.out.println("Addition task completed");
}

void charPrint()
{
    System.out.println("Character printing task");
    for(int i=65;i<=75;i++)
    {
        System.out.println((char)i);
        try
        {
            Thread.sleep(4000);
        }
        catch (Exception e)
        {
            System.out.println("Some problem occurred");
        }
    }
    System.out.println("Character printing task completed");
}

void numPrint()
{
    System.out.println("Number printing task started");
    for(int i=1;i<=10;i++)
    {
        System.out.println(i);
        try
        {
            Thread.sleep(4000);
        }
        catch (Exception e)
        {
            System.out.println("Some problem occurred");
        }
    }
    System.out.println("Number printing task completed");
}
```

```
class Demo
{
    public static void main(String[] args)
    {
        Demo1 d1 = new Demo1();

        Thread t1 = new Thread(d1);
        Thread t2 = new Thread(d1);
        Thread t3 = new Thread(d1);

        t1.setName("ADD");
        t2.setName("CHAR");
        t3.setName("NUM");

        t1.start();
        t2.start();
        t3.start();
    }
}
```



**GREAT
WORK!**

Multi Threading

1. What is a thread?

Thread refers to an alternative sequence of execution. It refers to a separate stack which is created in the stack segment.

2. What is a process?

A process is a program which has been loaded on to the RAM and is under execution.

3. What is the difference between process and thread?

Process	Thread
Process refers to a program which has been loaded on to the RAM and is in execution.	Thread refers to an alternative sequence of execution
Scheduling of a process is under the control of O.S	Scheduling of threads is under the control of thread scheduler
Switching between processes is performed by OS	Switching between threads is performed by thread scheduler
A process is a heavy weight entity. Internally, it would contain either a single thread or multiple threads	Thread is a light weight entity

4. What are the advantages of threads?

It enables efficient utilization of CPU time.

5. What is meant by multi threading?

Multi threading refers to the process of creating multiple stacks on the stack segment and loading multiple independent sub tasks into the stack segments and hence creating multiple sequence of execution. This would result in efficient utilization of CPU time.

6. Who manages multi tasking?

Operating System.

7. How can a programmer create his own thread?

- i) By creating an object of the thread class
- ii) By creating an object of the class which implements the **Runnable interface** and giving this object as a parameter to the Thread class object.

8. What is the name of the thread that executes the main method called as? Main thread.

9. What is a default priority of main thread?

Default priority of main thread is 5.

10. Can we change priority of main thread?

Yes. Using setPriority() method.

11. Which is the default method that the main thread executes?

Main thread always executes main() method.

12. Can we make the main thread to begin execution from any method of our own choice other than the default method it executes?

No.

13. Can we change the name of main thread?

Yes. Using setName() method.

14. Who creates the main thread?

JVM creates main thread and hands it over to the thread scheduler.

15. What is a lock?

Lock is a mechanism available in java to ensure that only a single thread makes use of the shared resource. Once any thread applies a lock on a resource, then other threads would not be able to access that resource until and unless the thread which has applied the lock releases it. Lock can be applied on a shared resource in java using synchronized keyword.

16. Does the priority of the thread matter during scheduling?

If the application is executing on a preemptive OS, then the priority matters. In such a case, the higher priority thread tends to get more CPU cycles than a lower priority threads.

If the application is executing on a non-preemptive OS, then irrespective of the priority all threads tend to get almost the same CPU time.

17. What is the problem associated with single threaded programs?

In a single threaded program, there is a possibility that the CPU cycle would not be efficiently utilized.

18. What is the advantage of single threaded program?

Multiple stacks on the stack segment need not be created. The thread scheduler need not get into the activity of scheduling multiple threads. In general, the burden on the thread scheduler would be relatively less.

19.What are the two ways of creating Thread?

- i) By extending the Thread class and creating an object of Thread class.
- ii) By implementing the Runnable interface and giving this object as a parameter to the Thread class object.

20. Which is a better way to achieve multithreading?

Implementing the Runnable interface and giving this object as a parameter to the Thread class object is the better way of creating a

thread since the programmer would have an option of extending another class if required.

21. Is Runnable a class or interface?

Runnable is an Interface.

22. Is Thread a class or interface?

Thread is a class.

23. What is the relationship between Thread and Runnable?

Thread class implements Runnable interface.

24. Why should we override the run () method of the Thread class?

Because the run() method which is provided by the Thread class is an empty run() method. It is the duty of the programmer to override an empty run() method and provide a suitable body as per the project requirements.

25. Why should we start a Thread?

When the thread is created, it would be in the **new state**. By applying start() method on thread, the thread can be shifted from new state to **runnable state** so that the thread scheduler can schedule it to the **running state**.

26. Which state would the Thread be when it is created?

New state.

27. What are the components of a program on which lock can be applied?

Lock can be applied on,

- i) method
- ii) variable
- iii) blocks

28. Can we start a Thread twice?

No because **IllegalThreadStateException** occurs.

29. What happens if we call start () method on a Thread twice?

IllegalThreadStateException occurs.

30. Can we explicitly call a run () method on a Thread?

The programmer should not call run() method since it would prevent multithreading functionality of the thread scheduler. Threads get executed in the same sequence in which their run() methods are called. It goes against the purpose/ethics of multithreading.

31. What happens if we explicitly call run () method on a Thread?

The programmer should not call run() method since it would prevent multithreading functionality of the thread scheduler. Threads get executed in the same sequence in which their run() methods are called. It goes against the purpose/ethics of multithreading.

32. Ideally who must call the run () method of a Thread?

Thread Scheduler.

33. How do we verify if a Thread is alive?

Using **isAlive()** method.

34. When the Thread is executing the run () method in which state it is present?

It would be present in **running state**.

35. In which state should the Thread be present in order to execute the run () method?

It should be readily available in **Runnable state**.

36. Which are the possible states that the Thread may go to while it is in the running state?

From the running state, the thread may go to

- i) Runnable state by executing `yield()` method

- ii) Sleep state by executing sleep() method,
- iii) Blocked state when the required resource is not available ,
- iv) Dead state when the execution of the thread is completed or if interrupt() method is executed,
- v) Wait state by executing wait()

37. When does the Thread enter sleep state?

When the sleep() method is executed.

38. When does the Thread enter wait state?

When the wait() method is executed.

39. When does the Thread enter blocked state?

When the resource is not currently available because some other thread has applied a lock on that resource, then the thread would enter into blocked state.

40. Which keyword is used to apply a lock on a resource?

“synchronized” keyword would be used.

41. Is the Thread alive when it is created?

No.

42. How can we bring the Thread to life?

By executing a start() method.

43. When does the Thread enter dead state?

Thread would enter into dead state when the thread completes its execution or if the interrupt() is executed.

44. How many Threads can be in the running state at any given point of time?

Only one thread can be present in the running state.

45. Does the Thread class provide an implementation to the abstract run () method of the Runnable interface?

Yes. It provides empty body. It is for the programmer to provide body for run() method based on the project requirements.

46. What are the possible states from which a Thread can enter Runnable state?

- i) sleep state to runnable state after sleep duration is completed.
- ii) new state to runnable state using start() method.
- iii) blocked state to runnable state when the previously unavailable resource becomes available.
- iv) wait state to runnable state when notify() or notifyAll() method is executed by another thread.
- v) running state to runnable state using yield() method.

47. Can we have multiple Threads in the new state?

Yes.

48. Can we have multiple Threads in the runnable state?

Yes.

49. Can we have multiple Threads in the running state?

No.

50. Can we have multiple Threads in the sleep state?

Yes.

51. Can we have multiple Threads in the blocked state?

Yes.

52. Can we have multiple Threads in the wait state?

Yes.

53. Can we have multiple Threads in the dead state?

Yes.

54. How can we manually kill a Thread?

Using interrupt() method.

55. Which operator is used to create a Thread of execution?

“new” operator would be used.

56. When a Thread is killed are all its lock relinquished (released)?

Yes.

57. Does a Thread relinquish locks when it enters the sleep state?

No.

58. Does a Thread relinquish locks when it enters the wait state?

No.

59. Does a Thread relinquish locks when it enters the blocked state?

No.

60. What is the difference between yield() and sleep()?

yield() method takes a thread from the running state to the runnable state whereas sleep() method takes the thread from the running state to the sleep state.

61. What is the difference between wait() and sleep()?

wait() method takes the thread from running state to the wait state whereas sleep() method takes a thread from running state to the sleep state.

62. Can we achieve synchronization effect by using join () method?

Yes. But it is not recommended.

63. How does a Thread come out of a sleep state?

When the sleep duration is completed, thread comes out of sleep state and enters into runnable state.

64. In which state can a Thread be interrupted?

Except new state and dead state, if the thread is in any other state, it can be interrupted.

65. How does a Thread come out of a block state?

Whenever the required resource which was previously unavailable becomes available, then a thread comes out of the block state.

66. How does a Thread come out of a wait state?

When some other thread calls notify() or notifyAll() method, then the thread comes out of the wait state.

67. How does a Thread come out of a dead state?

Thread cannot come out of the dead state.

68. How does a Thread come out of a runnable state?

When the thread scheduler decides to schedule the thread.

69. What are the possible reasons for the Thread to move out of running state?

- i) If the thread has been executed for a longer duration, then the thread scheduler would execute the yield() method and send it back to the runnable state.
- ii) If the thread completes its execution, it would move to the dead state.
- iii) If the thread is interrupted using interrupt() method, it would move to the dead state.
- iv) If sleep() method is encountered, it would move to the sleep state.
- v) If wait() method is encountered, it would move to the wait state.
- vi) If the required resource is not available, then the thread would enter into blocked state.

70. When should we synchronize a resource?

If the resource is a shared resource where only a single thread must be able to access the resource at a time, then the resource should be synchronized.

71. What is the advantage of synchronization?

Shared resource can be efficiently used without race condition.

72. What is the disadvantage of synchronization?

It goes against the purpose of multithreading.

73. How do we make one Thread wait for the other Thread?

Using join() method.

74. Is a Thread alive when it is in the new state?

No.

75. Is a Thread alive when it is in the running state?

Yes.

76. Is a Thread alive when it is in the runnable state?

Yes.

77. Is a Thread alive when it is in the sleep state?

Yes.

78. Is a Thread alive when it is in the block state?

Yes.

79. Is a Thread alive when it is in the wait state?

Yes.

80. Is a Thread alive when it is in the dead state?

No.

81. Is interrupting a Thread equal to killing a Thread?

Yes.

82. What is a deadlock?

Deadlock is a situation in a multithreaded programming where multiple threads would be stuck in the blocked state, mutually waiting for one another to release the resource in order to proceed with their execution. In the process, none of the threads would release the resource and hence would not be able to come out of the blocked state. Because of this,

execution of program would reach a standstill. This is called as deadlock.

83. When does a deadlock occur?

Deadlock occurs whenever there is a cyclic dependency for shared resources between multiple threads.

84. How can a deadlock be prevented?

- i) By ensuring that cyclic dependency does not occur between threads
- ii) By assigning different priorities to the threads. However this approach works only on pre-emptive OS and it is not a widely recommended choice.

85. What is daemon Thread?

A daemon thread is a thread that keeps on executing in the background at fixed intervals of time. They do not complete their execution until and unless the important thread of the process completes its execution. Daemon threads would always execute subsidiary activities.

86. What role does the daemon Thread play?

It performs subsidiary activities such as releasing the acquired resources, cleaning up memory i.e. garbage collection, spell check , auto save etc.

87. How do we make a Thread as daemon Thread?

- i) setDaemon() must be made as true.
- ii) The activity that has to be executed by daemon thread must be placed within an infinite loop.
- iii) It must be given a low priority.

88. Should the priority of daemon Thread be low or high?

It should be low priority.

89. Can you name a few daemon Threads?

Examples of daemon threads include Garbage collection thread, spell check thread, auto save thread etc.

90. What is the difference between deadlock and livelock?

In deadlock, the activity would not get completed because the threads would stuck in the blocked state.

In livelock, even though the threads are not in blocked state yet the program would not proceed with the execution and comes to a stand still due to faulty coding.

91. Which method can be used to enable interaction between Threads executing the different subtasks?

“synchronized” key word, wait(), notify() can be used.

92. Which facility can be used to enable interaction between Threads executing the same subtask?

“synchronized” keyword can be used.

93. What is the difference between notify () and notifyAll () methods?

notify() method would take one thread from the wait state to the runnable state, whereas notifyAll() would take all the threads present in the wait state to the runnable state.

94. Which method can be used to set a name to a Thread?

setName() method can be used.

95. Which method can be used to obtain the name of Thread?

getName() method can be used.

96. Can the overridden run() method throw exception object? Why?

No because it changes the signature which is not permitted.

97. Does C support multithreading?

No.

98. What happens if an exception occurs in one of the Thread of a multithreaded program?

If an exception occurs in the multithreaded environment then exception handling would be performed only on that stack in which exception occurred. The other threads (other stacks) would proceed with their execution as normal.

99. How do we verify if the Thread is a daemon Thread?

Using isDaemon() method.

100. Does C++ support multithreading?

No.

101. How many stacks would be present in the stack segment by default?

One stack would be present by default called as main stack.

102. How can we create extra stacks in the stack segment?

By creating the objects of the thread class.

103. Who manages the multiple stacks on the stack segment?

Thread scheduler.

104. From which version of java was Thread pool feature made available?

From jdk 1.7 onwards this feature made available.

105. What happens if a start () method is invoked on a Thread?

The thread moves from the new state to the runnable state. The thread scheduler

becomes aware that there is a thread available in the runnable state and can be scheduled.

106. What is a role of volatile keyword?

(Refer class notes)

107. **What is the datatype of parameter provided to the sleep() method?**

Long type.

108. **If a Thread creates a new Thread then what will be the priority of newly created Thread?**

The child threads priority would be the same as that of its parent thread.

109. **Does the JVM have to wait for all the daemon Threads to complete to wind up its operation?**

Yes.

110. **What are the four variants of constructors present in the Thread class?**

```
Thread t=new Thread(this);  
Thread t=new Thread();  
Thread t=new Thread(job);  
Thread t=new Thread("BANKING");
```

111. **How many locks can we apply on an object?**

Only one lock can be applied on an object. After the lock is relinquished another lock can be applied by another thread.

112. **Can a class contain both synchronized and non-synchronized methods?**

Yes.

113. **Can a class contain both synchronized and non-synchronized variables?**

Yes.

114. **Can a class contain both synchronized and non-synchronized blocks?**

Yes.

115. **Can we apply synchronized keyword on a class?**

No.

116. In which package are the interfaces and classes related to Threading available?

java.lang package.

117. In which version of java was the Callable interface introduced?

From JDK 1.5 onwards.

118. What is the difference between Runnable's run () and Callable's call () methods?

Runnable's run ()	Callable's call ()
Needs to implement run() method	Needs to implement call() method
Runnable interface is present from JDK 1.0 onwards	Callable interface has been introduced from JDK 1.5 onwards
Run() method does not return any value	call() method returns a value
It cannot throw checked exception	It can throw checked exception
Runnable use execute() method to put in the task queue	Callable use submit() method to put in the task queue

119. Can the run() method return the result of execution?

Since in the signature the return type of run is void, the run() method cannot return the result of execution.

120. What is meant by Thread safety?

Thread safety refers to a phenomenon in multithreaded programming in which a shared resource would be utilized by multiple threads such that improper accessing of the shared resource does not occur. Also ensures race condition does not occur. Thread safety can be achieved using “synchronized”, join() , immutable classes and by using thread safe classes.

121. What is meant by race condition in java?

(Refer class notes)

122. How can we share data between two Threads?

Using shared object concept.

123. Are wait(), notify() and notifyAll() methods belong to the Thread class?

No, they belong to Object class.

124. To which class does wait(), notify() and notifyAll() methods belong to?

They belong to Object class.

125. Why are wait(), notify() and notifyAll() methods members of Object class rather than Thread class?

That is because in java, locks are not applied on threads. Rather they would be applied on Objects.

126. What is a Thread local variable?

(Refer class notes)

127. Why should wait() and notify() methods be called from synchronized block?

It is the expectation from java API. If the “synchronized” key word not associated with wait() and notify() then IllegalMonitorStateException occurs.

Basically wait() and notify() should be placed under “synchronized” keyword to avoid race condition.

128. What is a Thread group?

(Refer class notes)

129. What is the advantage of using Thread group (Thread pool)?

(Refer class notes)

130. How do we verify if a Thread holds a lock or not?

Using hasLock() method. If the thread holds the lock then the method would return true otherwise false would be return.

131. There are three Threads t1, t2 and t3? How do you ensure the sequence t1, t2 and t3 in java?

It can be ensured using join() method.

132. What does the yield () method of Thread class do?

When yield() method is encountered the thread which is running state would be sent back to runnable state.

133. What is a semaphore in java?

Semaphore is a collection of statements which have to be executed by a single thread at any given point of time. It can be achieved using “synchronized” keyword.

134. What is a monitor in java?

Monitor is also called as semaphore. It is a collection of statements which have to be executed by a single thread at any given point of time. It can be achieved using “synchronized” keyword.

135. What happens if all the Threads in the Thread pool are busy and still a new task is submitted?

RejectedExecutionException would occur.

136. What is meant by busy spin in multithreading?

Busy spin is a phenomenon in multi-threaded environment where loop of threads would be continuously waiting for another to complete its execution so that they can start with their execution.

Busy spin leads to a wastage of CPU cycles.

137. Mention any three multithreading practices?

1. Threads should be given such names which clearly represents the activities that they perform, rather than calling the threads as t1,t2,t3 etc. It is better to name them as typing, spellcheck, autosave.
2. Avoid applying locks (Using synchronized) as much as possible, because locks go against the principle of multithreading.
3. If lock has to be inevitably applied then it should be done using only synchronized keyword.

138. Is there any means by which a programmer can forcefully take a Thread to the running state?

System.gc() method.

139. What is the difference between preemptive scheduling and time slicing?

preemptive scheduling	time slicing (non pre-emptive scheduling)
It is priority-based scheduling. It gives importance to the priority of a thread.	It is non-priority-based scheduling. It does not give importance to the priority of a thread.
Higher priority threads tend to get more CPU time.	All threads almost get equal CPU time irrespective of their priority.
Dead lock condition does not occur.	Dead lock condition occurs.
Unix OS.	Windows OS.

140. Can we make the user Thread as daemon Thread after starting it?

NO.

141. Can we make the user Thread as daemon Thread before starting it?

Yes.

142. What is shutdown hook?

JVM would perform cleanup operations at the end of program execution using “shutdownhook” thread. After execution of this thread the JVM stops its execution and gets popped from the stack.

143. What is starvation?

Starvation refers to a situation in multi-threaded environment where a thread in the block state which is waiting for the resource would never gets the resource because another thread which is holding the resource would not relinquish it.

144. Does the priority of a Thread play a vital role in its scheduling?

Priority of a thread matters only in case of pre-emptive OS.
However, in
non-pre-emptive OS priority of a thread does not play any role.

145. Name the methods available in Runnable interface?

public void run() method.

146. Name the methods available in Thread class?

sleep(), start(), yield(), join(), isAlive() etc.

147. Name the methods available in Object class?

toString(), hashCode(), notify(), notifyAll() etc.

148. Name the methods used for inter-Thread communication?

wait(), notify(), notifyAll().

149. What are the values of maximum priority, minimum priority and normal priority in java?

Minimum priority – 1

Maximum priority – 10

Normal priority – 5

150. How many Threads can access a monitor at a time?

Only one thread.

151. What is the priority of garbage collector Thread?

Garbage collector thread would be having low priority because it is daemon thread.

Introduction to Collections

In the previous classes we have seen there are different ways to store a data. We have seen the first method called as variable approach, then Arrays.

But each of these had their own limitations. Let us briefly look into what were those limitations.



Variable Approach: In variable approach we had difficulty in creating the references for each variable separately. And not only creating them, in fact accessing them was difficult too because it's definitely not easy to remember all the references. These limitations were overcome by the next approach which is Array approach.

Array Approach: In array approach we had the advantages of storing the same type of data in series one after the other, which made it easy to store the values as well as access. But this approach came with its own limitations which were,

- Only homogeneous can be stored.
- Require contiguous memory allocations in RAM.
- Cannot grow or shrink in size.
- Do not have built-in methods for data manipulation.

Why Collections???

To overcome the limitations that we came across with variable, array type approach we are now going to see what Collections consists of...



The **Collection in Java** is a framework that **provides an architecture to store and manipulate the group of objects**.

Java Collections can achieve all the operations that you perform on a data such as **searching, sorting, insertion, manipulation, and deletion**.

Java Collection means **a single unit of objects**. Java Collection framework provides many interfaces (**Set, List, Queue, Deque**) and classes (**ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet**).

What is a framework?

- It provides readymade architecture.
- It represents a set of classes and interfaces.

What is Collection framework?

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm



ArrayLists

The ArrayList class extends AbstractList and implements the List interface.
ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Let us see if actually ArrayList can store heterogeneous data with a simple example below



```

import java.util.ArrayList;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add("ROHIT");
        al.add(true);
        al.add(99.9f);
        System.out.println(al);
    }
}

```

Output:

[10, 20, 30, ROHIT, true, 99.9]
Press any key to continue . . .



Following is the list of the constructors provided by the ArrayList class.

Sr.No.	Constructor & Description
1	ArrayList() This constructor builds an empty array list.
2	ArrayList(Collection c) This constructor builds an array list that is initialized with the elements of the collection c .
3	ArrayList(int capacity) This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

**WHAT
ELSE ?**

dreamtime.



Apart from the methods inherited from its parent classes, **ArrayList** defines the following methods –

Sr.No.	Method & Description
1	void add(int index, Object element) Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 index > size()).
2	boolean add(Object o) Appends the specified element to the end of this list.
3	boolean addAll(Collection c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException, if the specified collection is null.
4	boolean addAll(int index, Collection c) Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
5	void clear() Removes all of the elements from this list.
6	Object clone() Returns a shallow copy of this ArrayList.
7	boolean contains(Object o) Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
8	void ensureCapacity(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

9	Object get(int index) Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 index >= size()).
10	int indexOf(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
11	int lastIndexOf(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
12	Object remove(int index) Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range (index < 0 index >= size()).
13	protected void removeRange(int fromIndex, int toIndex) Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
14	Object set(int index, Object element) Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 index >= size()).
15	int size() Returns the number of elements in this list.
16	Object[] toArray() Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
17	Object[] toArray(Object[] a) Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
18	void trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.

Difference between add() and set() methods

- **add(index, object):**

If add(index, object) method is used to add an element at a specified index and if an element already exists at the specified position, then all the elements are shifted by one position to the right and then the new element will be added to the specified position. In other words, the existing element will not be replaced.

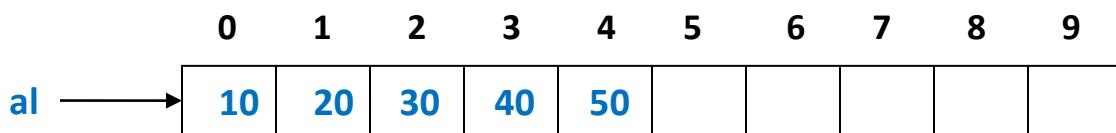
Let us understand this with the help of Code:

```
import java.util.ArrayList;
class Launch
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        System.out.println(al);

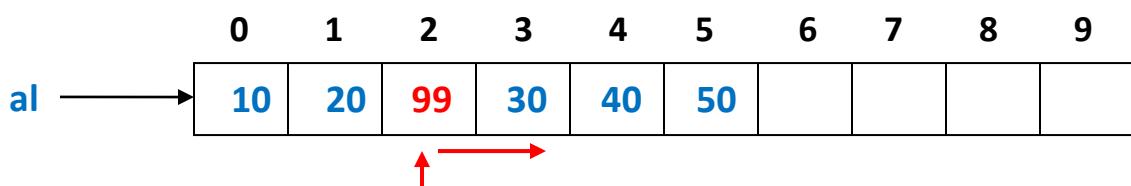
        al.add(2,99);
        System.out.println(al);
    }
}
```



Before using add() method:



After using add() method:



Output:

```
[ 10, 20, 30, 40, 50 ]  
[ 10, 20, 99, 30, 40, 50 ]  
Press any key to continue...
```



- ***set(index, object):***

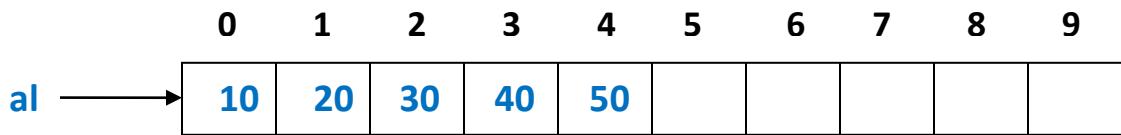
If `set(index, object)` method is used to add an element at a specified index and if an element already exists at that specified position, then the existing element is replaced by the new element.

Code:

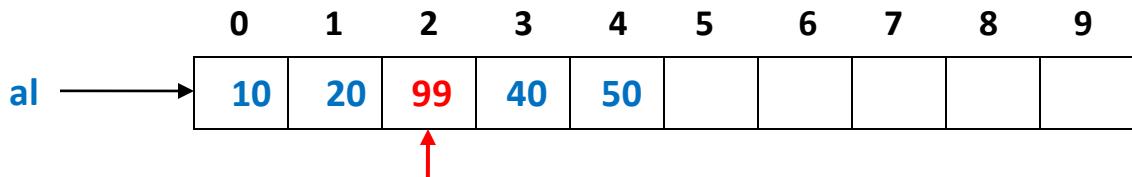
```
import java.util.ArrayList;  
class Launch  
{  
    public static void main(String[] args)  
    {  
        ArrayList al = new ArrayList();  
        al.add(10);  
        al.add(20);  
        al.add(30);  
        al.add(40);  
        al.add(50);  
        System.out.println(al);  
  
        al.set(2,99);  
        System.out.println(al);  
    }  
}
```



Before using set() method:



After using set() method:



Output:

[10, 20, 30, 40, 50]

[10, 20, 99, 40, 50]

Press any key to continue...



Dynamic Array

Standard Java arrays are of fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array can hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk. ArrayList supports dynamic arrays that can grow as needed.

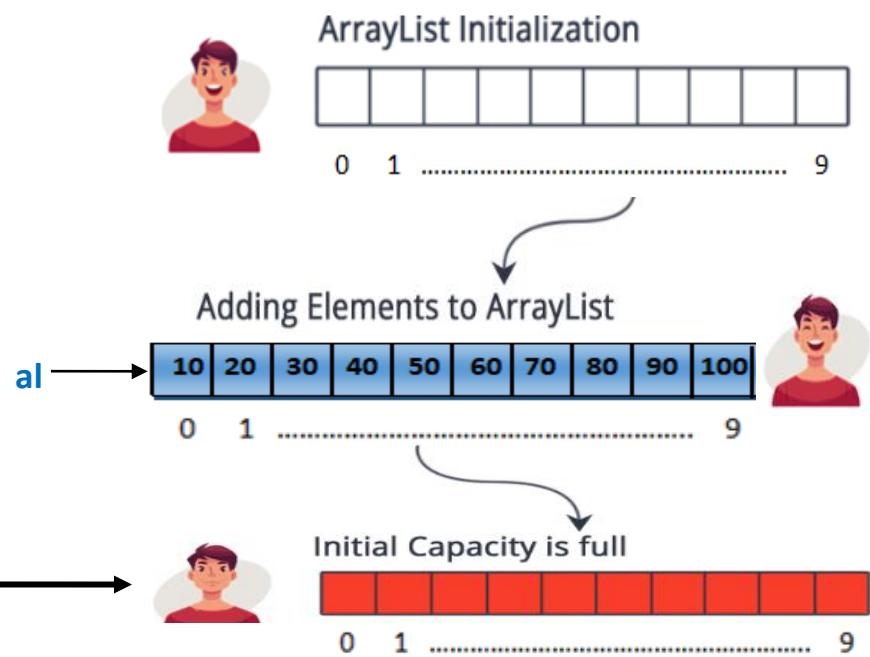
An array which can grow in size or shrink in size is called as **Resizable array** or **Dynamic array**.

But how does this happen??

Whenever we create an **ArrayList** object automatically a regular array is created whose size initially will be of 10 (0 to 9).

```
ArrayList al = new ArrayList();
```

```
al.add(10);  
al.add(20);  
al.add(30);  
.  
.  
al.add(100);  
al.add(110);
```



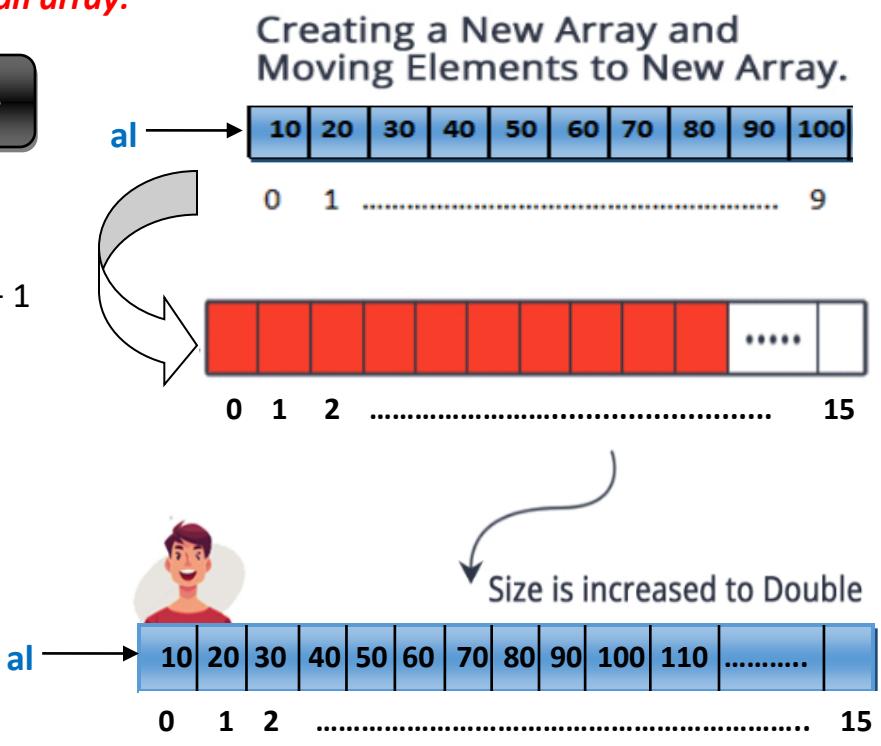
Now the size of array is filled how do we add the value 110 and where do we??

The initial capacity of the resizable array is 10. Once this capacity is filled a new array is created whose size is determined by using a formula.

Formula to find the new size of an array:

$$\text{New Size} = \frac{3}{2} * \text{Old Size} + 1$$

Therefore, New Size = $\frac{3}{2} * 10 + 1$
New Size = 16



All the elements of the previous array are copied to this new array and the reference is reassigned to this array. The old array becomes a garbage object and is deallocated.

The use of size() and trimToSize() method

size(): The `size()` method of `java.util.ArrayList` class is used to get the number of elements in this list.

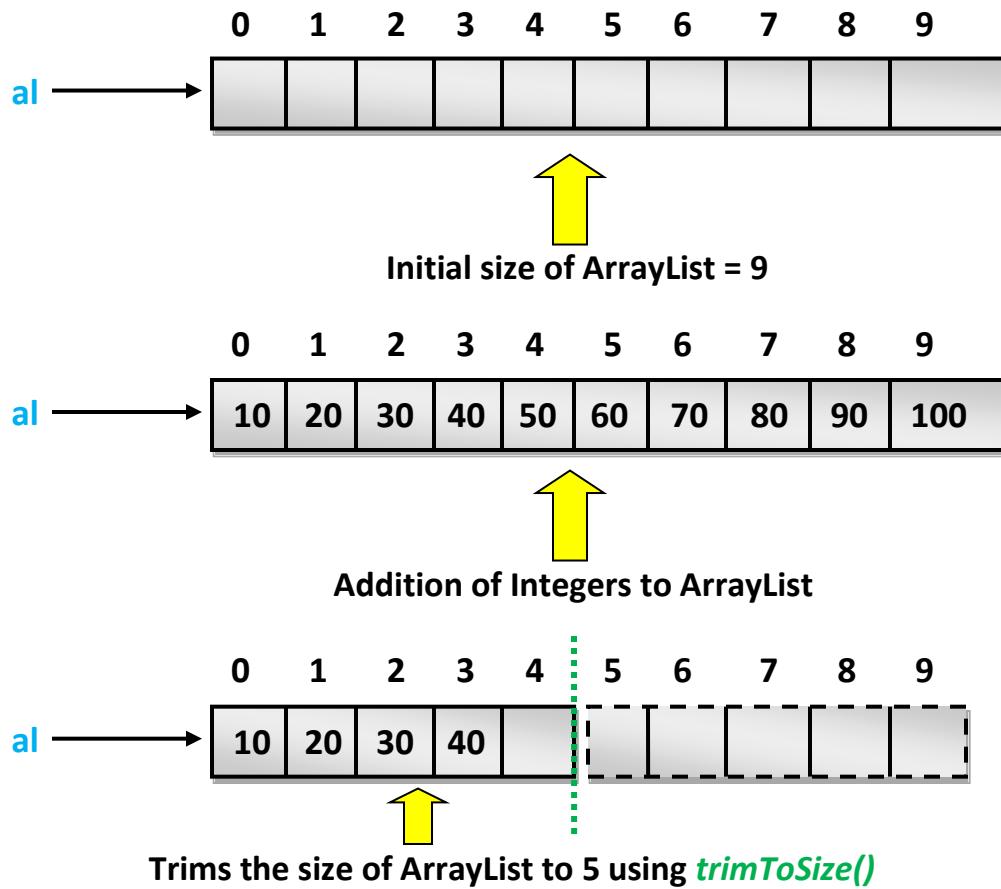
Whenever we create an array the initial capacity by default will be of 10.

trimToSize(): This method is used to make the capacity of the arraylist equal to its size (number of elements stored). This is a useful method to avoid wastage of memory.

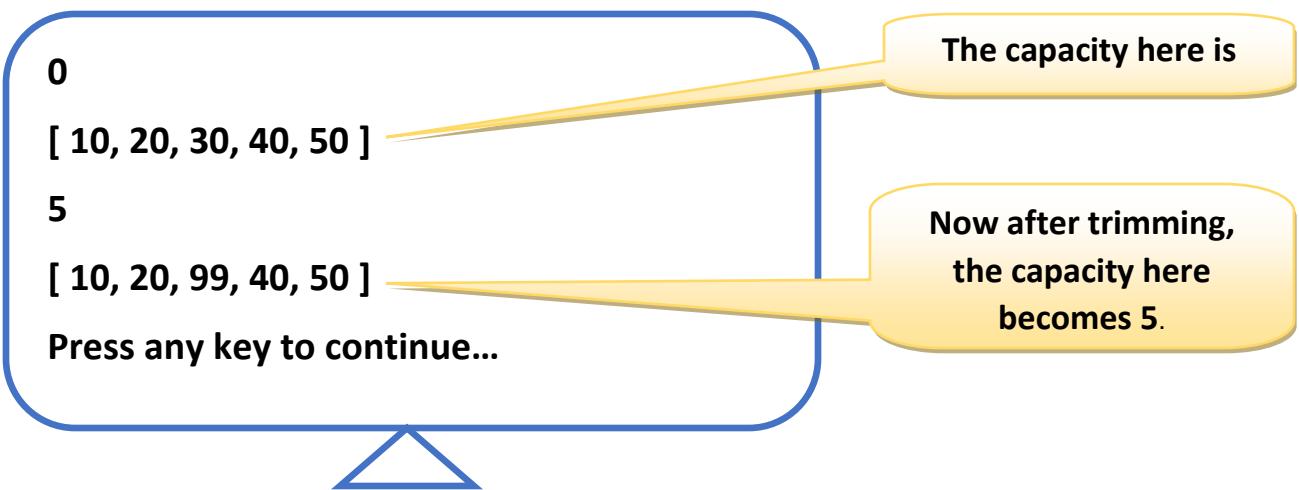
Example:

```
import java.util.ArrayList;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        System.out.println(al.size());
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        System.out.println(al);

        System.out.println(al.size());
        al.trimToSize();
        System.out.println(al);
    }
}
```



Output:



When to use Array over ArrayList??

We know that ArrayList is advantageous when compared to Array. But at few conditions Array is more preferable when compared to ArrayList.

They are:

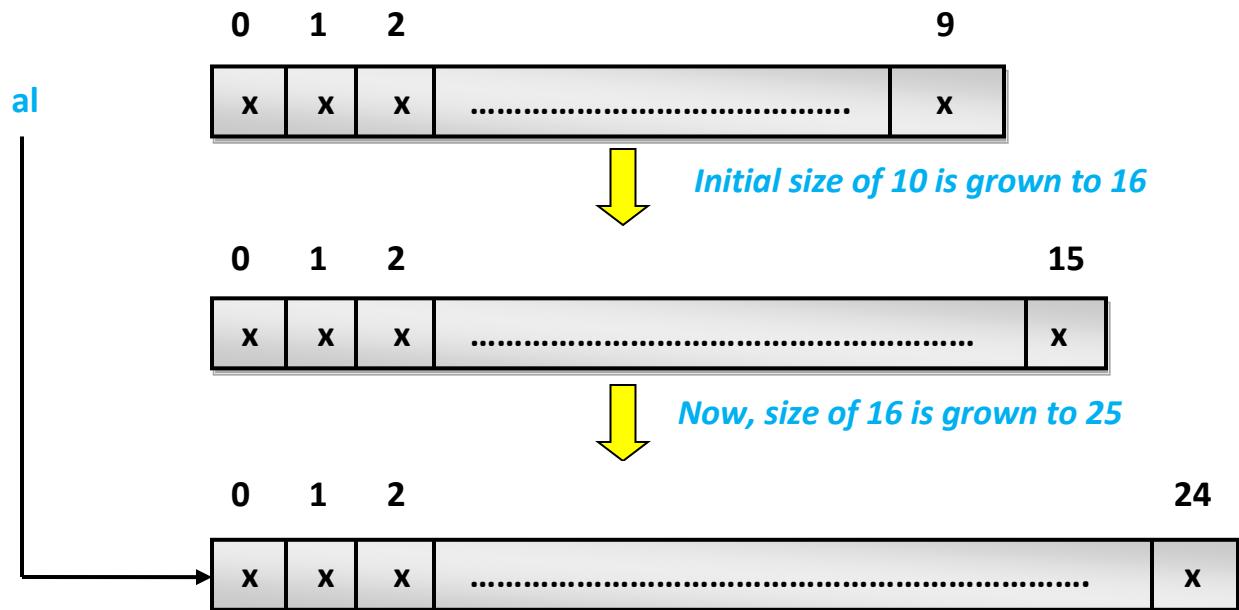
- **When the size of the data is known before.**
- **The data we are storing are homogeneous data.**

ArrayList Approach

For example, when we want to store marks of 25 students.

When create an AraryList object initially the size that gets allocated is 10. To store the extra data the array has to grow in size and all the elements of an array has to get copied to the new array and the reference has to get changed, all these has to happen twice.





Array Approach

Since, we know the number of students and the type of marks to be stored, directly we can create an array.

```
int a[ ] = new int[ 25 ] ;
```

Using the above statement we can directly create an array of 25 students.



Therefore, Arrays are faster in execution when compared to arraylists. Hence whenever the above two conditions are satisfied it is always better to choose an array over the arraylist to store the data.

The use of `contains()`, `containsAll()`, `getClass()` methods.

The `contains()` method is used to determine whether an element exists in an `ArrayList` object. Returns true if this list contains the specified element.

The `containsAll()` method of List interface in Java is used to check if this List contains all of the elements in the specified Collection. So basically it is used to check if a List contains a set of elements or not.

The `getClass()` is the method of Object class. This method returns the runtime class of this object. The class object which is returned is the object that is locked by static synchronized method of the represented class.

```
import java.util.ArrayList;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        System.out.println(al);
        System.out.println(al.contains(50));
        System.out.println(al.contains(99));
    }

    ArrayList b = new ArrayList();
    b.add(10);
    b.add(20);
    b.add(30);
    b.add(40);
    b.add(50);
    System.out.println(b);
    System.out.println(b.containsAll(b));
    System.out.println(b.getClass());
}
}
```

Output

[10, 20 , 30, 40, 50]

true

false

[10, 20 , 30, 40, 50]

true

class java.util.ArrayList

When to use clone() method ?



ArrayList **clone()** method is used to create a shallow copy of the list.
In the new list, only object references are copied.

```
import java.util.ArrayList;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        System.out.println(al);

        ArrayList b = (ArrayList)al.clone();
        System.out.println(b);
    }
}
```

Output:

```
[ 10, 20, 30, 40, 50 ]
[ 10, 20, 30, 40, 50 ]
Press any key to continue...
```

LINKEDLIST

LinkedList internally makes use of doubly linked list data structure to store data where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

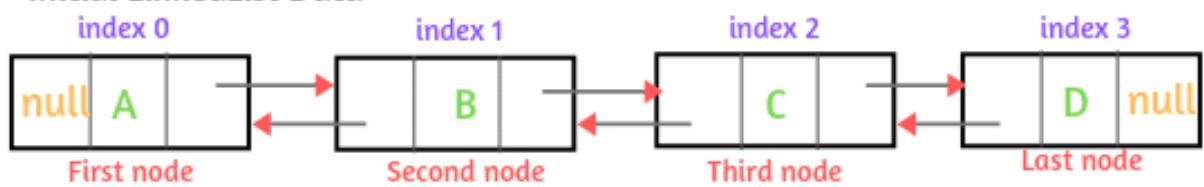


How LinkedList is better than ArrayList?

Just like arrays, ArrayList also expects contiguous Memory locations in RAM. But since LinkedList Makes use of doubly LinkedList, it can make use Of dispersed memory location thus overcoming all the disadvantages of arrays.

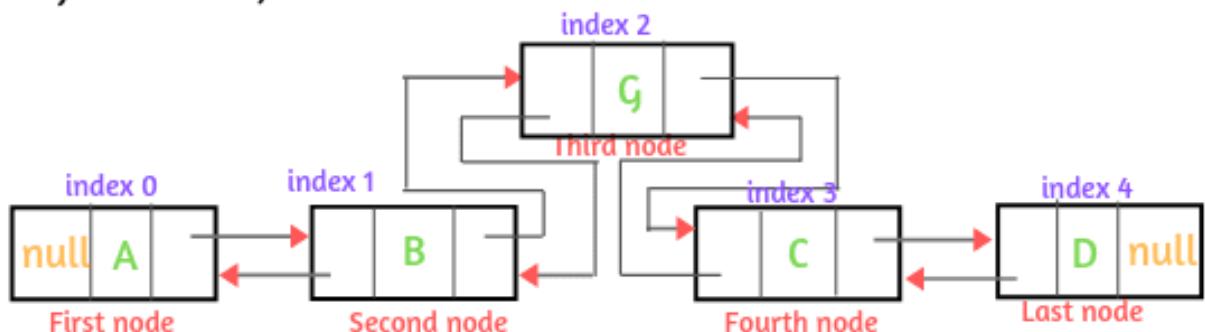
Internal Implementation of LinkedList

Initial LinkedList Data



`linkedlist.add(2,"G");`

After Insertion, LinkedList Data will look like this.



You can see that one node is created with element G and simply changes the next and previous pointer only. No shift of operation has occurred.

Built-in Methods in LinkedList:

1. **add(int index, E element)**: This method Inserts the specified element at the specified position in this list.
2. **add(E e)**: This method Appends the specified element to the end of this list.
3. **addAll(int index, Collection c)**: This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
4. **addAll(Collection c)**: This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
5. **addFirst(E e)**: This method Inserts the specified element at the beginning of this list.
6. **addLast(E e)**: This method Appends the specified element to the end of this list.
7. **clear()**: This method removes all of the elements from this list.
8. **getFirst()**: This method returns the first element in this list.
9. **getLast()**: This method returns the last element in this list.
10. **indexOf(Object o)**: This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
11. **lastIndexOf(Object o)**: This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
12. **peek()**: This method retrieves, but does not remove, the head (first element) of this list.
13. **peekFirst()**: This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
14. **peekLast()**: This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
15. **poll()**: This method retrieves and removes the head (first element) of this list.
16. **pollFirst()**: This method retrieves and removes the first element of this list, or returns null if this list is empty.
17. **pollLast()**: This method retrieves and removes the last element of this list, or returns null if this list is empty.
18. **pop()**: This method Pops an element from the stack represented by this list.
19. **push(E e)**: This method Pushes an element onto the stack represented by this list.
20. **remove()**: This method retrieves and removes the head (first element) of this list.
21. **remove(int index)**: This method removes the element at the specified position in this list.

22. **removeFirst()**: This method removes and returns the first element from this list.
23. **removeLast()**: This method removes and returns the last element from this list.
24. **set(int index, E element)**: This method replaces the element at the specified position in this list with the specified element.
25. **size()**: This method returns the number of elements in this list.

Let us understand these built-in methods with the help of code

```
import java.util.LinkedList;
class Demo
{
    public static void main(String[] args)
    {
        LinkedList ll = new LinkedList();
        ll.add(10);
        ll.add(20);
        ll.add(30);
        ll.addFirst(40);
        ll.addFirst(50);
        ll.addLast(60);
        ll.addLast(70);
        System.out.println(ll);
        System.out.println(ll.getFirst());
        System.out.println(ll.getLast());
        System.out.println(ll.peekFirst());
        System.out.println(ll);
        System.out.println(ll.pollFirst());
        System.out.println(ll);
        System.out.println(ll.peekLast());
        System.out.println(ll);
        System.out.println(ll.pollLast());
        System.out.println(ll);
    }
}
```



OUTPUT

```
[50, 40, 10, 20, 30, 60, 70]
50
70
50
[50, 40, 10, 20, 30, 60, 70]
50
[40, 10, 20, 30, 60, 70]
70
[40, 10, 20, 30, 60, 70]
70
```



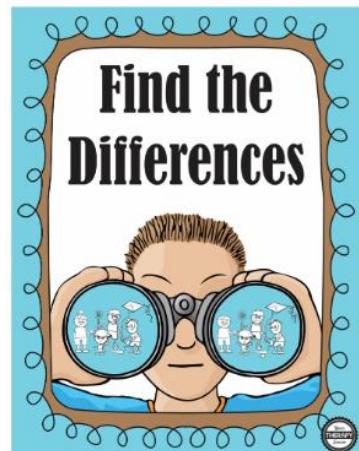
Difference between peekFirst() and getFirst()

If the List is empty and **getFirst()** is called, it would result in **NoSuchElementException**.

If the List is empty and **peekFirst()** is called, it would not result in any **Exception**.

Let us understand the difference with the help of code

```
import java.util.LinkedList;
class Demo
{
    public static void main(String[] args)
    {
        LinkedList ll = new LinkedList();
        System.out.println(ll);
        //System.out.println(ll.getFirst()); -----> NoSuchElementException
        System.out.println(ll.peekFirst());
        System.out.println(ll);
    }
}
```



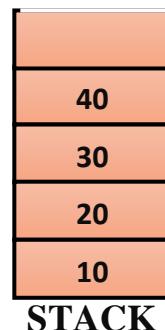
OUTPUT

```
[ ]  
null  
[]  
Press any key to continue . . .
```

Implementation of Stack using LinkedList

Stack can be implemented by using **push()** and **pop()** in **LinkedList** as shown below:

```
import java.util.LinkedList;  
class Demo  
{  
    public static void main(String[] args)  
    {  
        LinkedList ll = new LinkedList();  
        ll.push(10);  
        ll.push(20);  
        ll.push(30);  
        ll.push(40);  
        System.out.println(ll);  
        System.out.println(ll.pop());  
        System.out.println(ll.pop());  
        System.out.println(ll.pop());  
        System.out.println(ll.pop());  
    }  
}
```



OUTPUT

[40, 30, 20, 10]

40

30

20

10

ARRAYDEQUE

ArrayDeque in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as **Array Double Ended Queue or Array Deck**. This is a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.

Few important features of ArrayDeque are as follows:

- Array deques have no capacity restrictions and they grow as necessary to support usage.
- They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads.
- Null elements are prohibited in the ArrayDeque.
- ArrayDeque class is likely to be faster than Stack when used as a stack.
- ArrayDeque class is likely to be faster than LinkedList when used as a queue.



Built-in Methods in ArrayDeque:

1. [add\(Element e\)](#) : The method inserts particular element at the end of the deque.
2. [addFirst\(Element e\)](#) : The method inserts particular element at the start of the deque.
3. [addLast\(Element e\)](#) : The method inserts particular element at the end of the deque. It is similar to add () method
4. [clear\(\)](#) : The method removes all deque elements.
5. [size\(\)](#) : The method returns the no. of elements in deque.
6. [clone\(\)](#) : The method copies the deque.
7. [contains\(Obj\)](#) : The method checks whether a deque contains the element or not.
8. [element\(\)](#) : The method returns element at the head of the deque
9. [getFirst\(\)](#) : The method returns first element of the deque
10. [getLast\(\)](#) : The method returns last element of the deque
11. [isEmpty\(\)](#) : The method checks whether the deque is empty or not.
12. [toArray\(\)](#) : The method returns array having the elements of deque.
13. [offer\(Element e\)](#) : The method inserts element at the end of deque.
14. [offerFirst\(Element e\)](#) : The method inserts element at the front of deque.
15. [offerLast\(Element e\)](#) : The method inserts element at the end of deque.
16. [peek\(\)](#) : The method returns head element without removing it.
17. [peekFirst\(\)](#) : The method returns first element without removing it.
18. [peekLast\(\)](#) : The method returns last element without removing it.
19. [poll\(\)](#) : The method returns head element and also removes it
20. [pollFirst\(\)](#) : The method returns first element and also removes it
21. [pollLast\(\)](#) : The method returns last element and also removes it
22. [pop\(\)](#) : The method pops out an element for stack represented by deque
23. [push\(Element e\)](#) : The method pushes an element onto stack represented by deque
24. [remove\(\)](#) : The method returns head element and also removes it
25. [removeFirst\(\)](#) : The method returns first element and also removes it
26. [removeLast\(\)](#) : The method returns last element and also removes it
27. [removeFirstOccurrence\(Obj\)](#) : The method removes the element where it first occur in the deque.
28. [removeLastOccurrence\(Obj\)](#) : The method removes the element where it last occur in the deque.

Accessing ArrayDeque from front-end.

```
import java.util.ArrayDeque;
class Demo
{
    public static void main(String[] args)
    {
        ArrayDeque dq = new ArrayDeque();
        dq.add(10);
        dq.add(20);
        dq.add(30);
        dq.add(40);
        System.out.println(dq);
        System.out.println(dq.pollFirst());
        System.out.println(dq.pollFirst());
        System.out.println(dq.pollFirst());
        System.out.println(dq.pollFirst());
    }
}
```



OUTPUT

```
[10, 20, 30, 40]
10
20
30
40
Press any key to continue . . .
```

Accessing ArrayDeque from rear-end.

```
import java.util.ArrayDeque;
class Demo55
{
    public static void main(String[] args)
    {
        ArrayDeque dq = new ArrayDeque();
        dq.add(10);
        dq.add(20);
        dq.add(30);
        dq.add(40);
        System.out.println(dq);
        System.out.println(dq.pollLast());
        System.out.println(dq.pollLast());
        System.out.println(dq.pollLast());
        System.out.println(dq.pollLast());
    }
}
```

OUTPUT

```
[10, 20, 30, 40]
40
30
20
10
Press any key to continue . . .
```

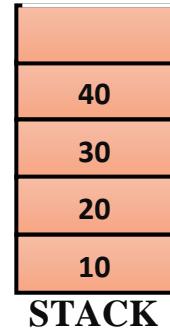
Implementation of Stack using ArrayDeque

Stack can be implemented by using **push()** and **pop()** in **ArrayDeque** as shown below:

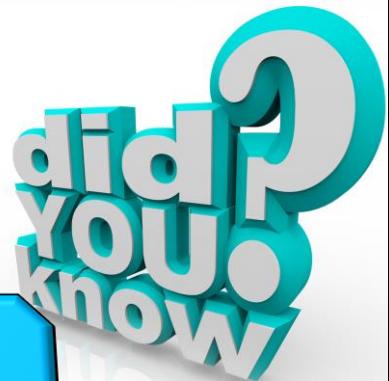
```
import java.util.ArrayDeque;
class Demo55
{
    public static void main(String[] args)
    {
        ArrayDeque dq = new ArrayDeque();
        dq.push(10);
        dq.push(20);
        dq.push(30);
        dq.push(40);
        System.out.println(dq);
        System.out.println(dq.pop());
        System.out.println(dq.pop());
        System.out.println(dq.pop());
        System.out.println(dq.pop());
    }
}
```

OUTPUT

```
[40, 30, 20, 10]
40
30
20
10
```



Technophobia is the fear of technology,
Nomophobia is the fear of being without a mobile phone,
Cyberphobia is the fear of computers.



LINKEDLIST

LinkedList internally makes use of doubly linked list data structure to store data where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

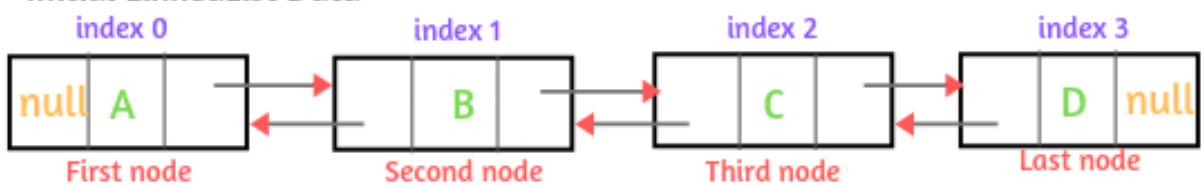


How LinkedList is better than ArrayList?

Just like arrays, ArrayList also expects contiguous Memory locations in RAM. But since LinkedList Makes use of doubly LinkedList, it can make use Of dispersed memory location thus overcoming all the disadvantages of arrays.

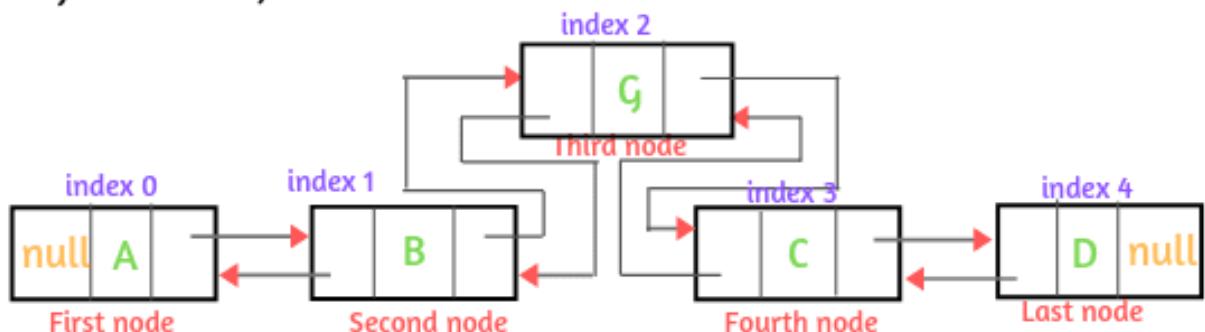
Internal Implementation of LinkedList

Initial LinkedList Data



`linkedlist.add(2,"G");`

After Insertion, LinkedList Data will look like this.



You can see that one node is created with element G and simply changes the next and previous pointer only. No shift of operation has occurred.

Built-in Methods in LinkedList:

1. **add(int index, E element)**: This method Inserts the specified element at the specified position in this list.
2. **add(E e)**: This method Appends the specified element to the end of this list.
3. **addAll(int index, Collection c)**: This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
4. **addAll(Collection c)**: This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
5. **addFirst(E e)**: This method Inserts the specified element at the beginning of this list.
6. **addLast(E e)**: This method Appends the specified element to the end of this list.
7. **clear()**: This method removes all of the elements from this list.
8. **getFirst()**: This method returns the first element in this list.
9. **getLast()**: This method returns the last element in this list.
10. **indexOf(Object o)**: This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
11. **lastIndexOf(Object o)**: This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
12. **peek()**: This method retrieves, but does not remove, the head (first element) of this list.
13. **peekFirst()**: This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
14. **peekLast()**: This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
15. **poll()**: This method retrieves and removes the head (first element) of this list.
16. **pollFirst()**: This method retrieves and removes the first element of this list, or returns null if this list is empty.
17. **pollLast()**: This method retrieves and removes the last element of this list, or returns null if this list is empty.
18. **pop()**: This method Pops an element from the stack represented by this list.
19. **push(E e)**: This method Pushes an element onto the stack represented by this list.
20. **remove()**: This method retrieves and removes the head (first element) of this list.
21. **remove(int index)**: This method removes the element at the specified position in this list.

22. **removeFirst()**: This method removes and returns the first element from this list.
23. **removeLast()**: This method removes and returns the last element from this list.
24. **set(int index, E element)**: This method replaces the element at the specified position in this list with the specified element.
25. **size()**: This method returns the number of elements in this list.

Let us understand these built-in methods with the help of code

```
import java.util.LinkedList;
class Demo
{
    public static void main(String[] args)
    {
        LinkedList ll = new LinkedList();
        ll.add(10);
        ll.add(20);
        ll.add(30);
        ll.addFirst(40);
        ll.addFirst(50);
        ll.addLast(60);
        ll.addLast(70);
        System.out.println(ll);
        System.out.println(ll.getFirst());
        System.out.println(ll.getLast());
        System.out.println(ll.peekFirst());
        System.out.println(ll);
        System.out.println(ll.pollFirst());
        System.out.println(ll);
        System.out.println(ll.peekLast());
        System.out.println(ll);
        System.out.println(ll.pollLast());
        System.out.println(ll);
    }
}
```



OUTPUT

```
[50, 40, 10, 20, 30, 60, 70]
50
70
50
[50, 40, 10, 20, 30, 60, 70]
50
[40, 10, 20, 30, 60, 70]
70
[40, 10, 20, 30, 60, 70]
70
```



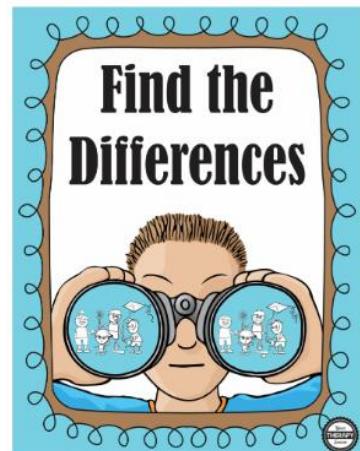
Difference between peekFirst() and getFirst()

If the List is empty and **getFirst()** is called, it would result in **NoSuchElementException**.

If the List is empty and **peekFirst()** is called, it would not result in any **Exception**.

Let us understand the difference with the help of code

```
import java.util.LinkedList;
class Demo
{
    public static void main(String[] args)
    {
        LinkedList ll = new LinkedList();
        System.out.println(ll);
        //System.out.println(ll.getFirst()); -----> NoSuchElementException
        System.out.println(ll.peekFirst());
        System.out.println(ll);
    }
}
```



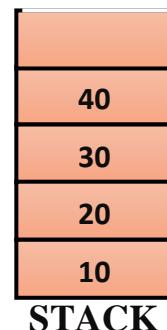
OUTPUT

```
[ ]  
null  
[]  
Press any key to continue . . .
```

Implementation of Stack using LinkedList

Stack can be implemented by using **push()** and **pop()** in **LinkedList** as shown below:

```
import java.util.LinkedList;  
class Demo  
{  
    public static void main(String[] args)  
    {  
        LinkedList ll = new LinkedList();  
        ll.push(10);  
        ll.push(20);  
        ll.push(30);  
        ll.push(40);  
        System.out.println(ll);  
        System.out.println(ll.pop());  
        System.out.println(ll.pop());  
        System.out.println(ll.pop());  
        System.out.println(ll.pop());  
    }  
}
```



OUTPUT

[40, 30, 20, 10]

40

30

20

10

ARRAYDEQUE

ArrayDeque in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as **Array Double Ended Queue or Array Deck**. This is a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.

Few important features of ArrayDeque are as follows:

- Array deques have no capacity restrictions and they grow as necessary to support usage.
- They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads.
- Null elements are prohibited in the ArrayDeque.
- ArrayDeque class is likely to be faster than Stack when used as a stack.
- ArrayDeque class is likely to be faster than LinkedList when used as a queue.



Built-in Methods in ArrayDeque:

1. [add\(Element e\)](#) : The method inserts particular element at the end of the deque.
2. [addFirst\(Element e\)](#) : The method inserts particular element at the start of the deque.
3. [addLast\(Element e\)](#) : The method inserts particular element at the end of the deque. It is similar to add () method
4. [clear\(\)](#) : The method removes all deque elements.
5. [size\(\)](#) : The method returns the no. of elements in deque.
6. [clone\(\)](#) : The method copies the deque.
7. [contains\(Obj\)](#) : The method checks whether a deque contains the element or not.
8. [element\(\)](#) : The method returns element at the head of the deque
9. [getFirst\(\)](#) : The method returns first element of the deque
10. [getLast\(\)](#) : The method returns last element of the deque
11. [isEmpty\(\)](#) : The method checks whether the deque is empty or not.
12. [toArray\(\)](#) : The method returns array having the elements of deque.
13. [offer\(Element e\)](#) : The method inserts element at the end of deque.
14. [offerFirst\(Element e\)](#) : The method inserts element at the front of deque.
15. [offerLast\(Element e\)](#) : The method inserts element at the end of deque.
16. [peek\(\)](#) : The method returns head element without removing it.
17. [peekFirst\(\)](#) : The method returns first element without removing it.
18. [peekLast\(\)](#) : The method returns last element without removing it.
19. [poll\(\)](#) : The method returns head element and also removes it
20. [pollFirst\(\)](#) : The method returns first element and also removes it
21. [pollLast\(\)](#) : The method returns last element and also removes it
22. [pop\(\)](#) : The method pops out an element for stack represented by deque
23. [push\(Element e\)](#) : The method pushes an element onto stack represented by deque
24. [remove\(\)](#) : The method returns head element and also removes it
25. [removeFirst\(\)](#) : The method returns first element and also removes it
26. [removeLast\(\)](#) : The method returns last element and also removes it
27. [removeFirstOccurrence\(Obj\)](#) : The method removes the element where it first occur in the deque.
28. [removeLastOccurrence\(Obj\)](#) : The method removes the element where it last occur in the deque.

Accessing ArrayDeque from front-end.

```
import java.util.ArrayDeque;
class Demo
{
    public static void main(String[] args)
    {
        ArrayDeque dq = new ArrayDeque();
        dq.add(10);
        dq.add(20);
        dq.add(30);
        dq.add(40);
        System.out.println(dq);
        System.out.println(dq.pollFirst());
        System.out.println(dq.pollFirst());
        System.out.println(dq.pollFirst());
        System.out.println(dq.pollFirst());
    }
}
```



OUTPUT

```
[10, 20, 30, 40]
10
20
30
40
Press any key to continue . . .
```

Accessing ArrayDeque from rear-end.

```
import java.util.ArrayDeque;
class Demo55
{
    public static void main(String[] args)
    {
        ArrayDeque dq = new ArrayDeque();
        dq.add(10);
        dq.add(20);
        dq.add(30);
        dq.add(40);
        System.out.println(dq);
        System.out.println(dq.pollLast());
        System.out.println(dq.pollLast());
        System.out.println(dq.pollLast());
        System.out.println(dq.pollLast());
    }
}
```

OUTPUT

```
[10, 20, 30, 40]
40
30
20
10
Press any key to continue . . .
```

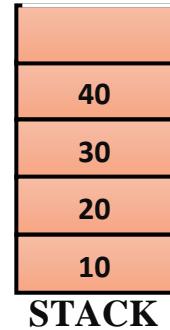
Implementation of Stack using ArrayDeque

Stack can be implemented by using **push()** and **pop()** in **ArrayDeque** as shown below:

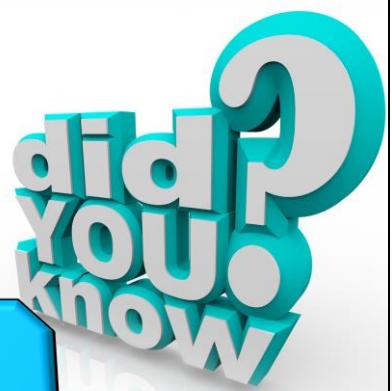
```
import java.util.ArrayDeque;
class Demo55
{
    public static void main(String[] args)
    {
        ArrayDeque dq = new ArrayDeque();
        dq.push(10);
        dq.push(20);
        dq.push(30);
        dq.push(40);
        System.out.println(dq);
        System.out.println(dq.pop());
        System.out.println(dq.pop());
        System.out.println(dq.pop());
        System.out.println(dq.pop());
    }
}
```

OUTPUT

```
[40, 30, 20, 10]
40
30
20
10
```



Technophobia is the fear of technology,
Nomophobia is the fear of being without a mobile phone,
Cyberphobia is the fear of computers.



Priority Queue in java

A PriorityQueue is used when the objects are supposed to be processed based on the priority.

It is known that a queue follows First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play. The PriorityQueue is **based on the priority heap**. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.



Key Points

- PriorityQueue doesn't permit null.
- We can't create PriorityQueue of Objects that are non-comparable
- PriorityQueue are unbound queues.
- **The head of this queue is the least element with respect to the specified ordering.** If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

Basic Operations on PriorityQueue:

- **boolean add(E element):** This method inserts the specified element into this priority queue.
- **public peek():** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- **public poll():** This method retrieves and removes the head of this queue, or returns null if this queue is empty.

Let us understand with the help of an example

Data:120,60,160,30,80,140,180

```
import java.util.PriorityQueue;
class Demo
{
    public static void main(String[] args)
    {
        PriorityQueue pq = new PriorityQueue();
        pq.add(120);
        pq.add(60);
        pq.add(160);
        pq.add(30);
        pq.add(80);
        pq.add(140);
        pq.add(180);
        System.out.println(pq);
    }
}
```

Output:

[30, 60, 140, 120, 80, 160, 180]
Press any key to continue . . .



Note: The order of storing the values is explained in the video with more clarity.

Methods in PriorityQueue class:

1. **boolean add(E element):** This method inserts the specified element into this priority queue.
2. **public remove():** This method removes a single instance of the specified element from this queue, if it is present
3. **public poll():** This method retrieves and removes the head of this queue, or returns null if this queue is empty.

4. **public peek():** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
5. **Iterator iterator():** Returns an iterator over the elements in this queue.
6. **boolean contains(Object o):** This method returns true if this queue contains the specified element
7. **void clear():** This method is used to remove all of the contents of the priority queue.
8. **boolean offer(E e):** This method is used to insert a specific element into the priority queue.
9. **int size():** The method is used to return the number of elements present in the set.
10. **toArray():** This method is used to return an array containing all of the elements in this queue.
11. **Comparator comparator():** The method is used to return the comparator that can be used to order the elements of the queue.



TreeSet in java

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. **The objects of the TreeSet class are stored in ascending order.**

Key Points

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non-synchronized.
- Java TreeSet class maintains ascending order.

Let us see an example for treeset,

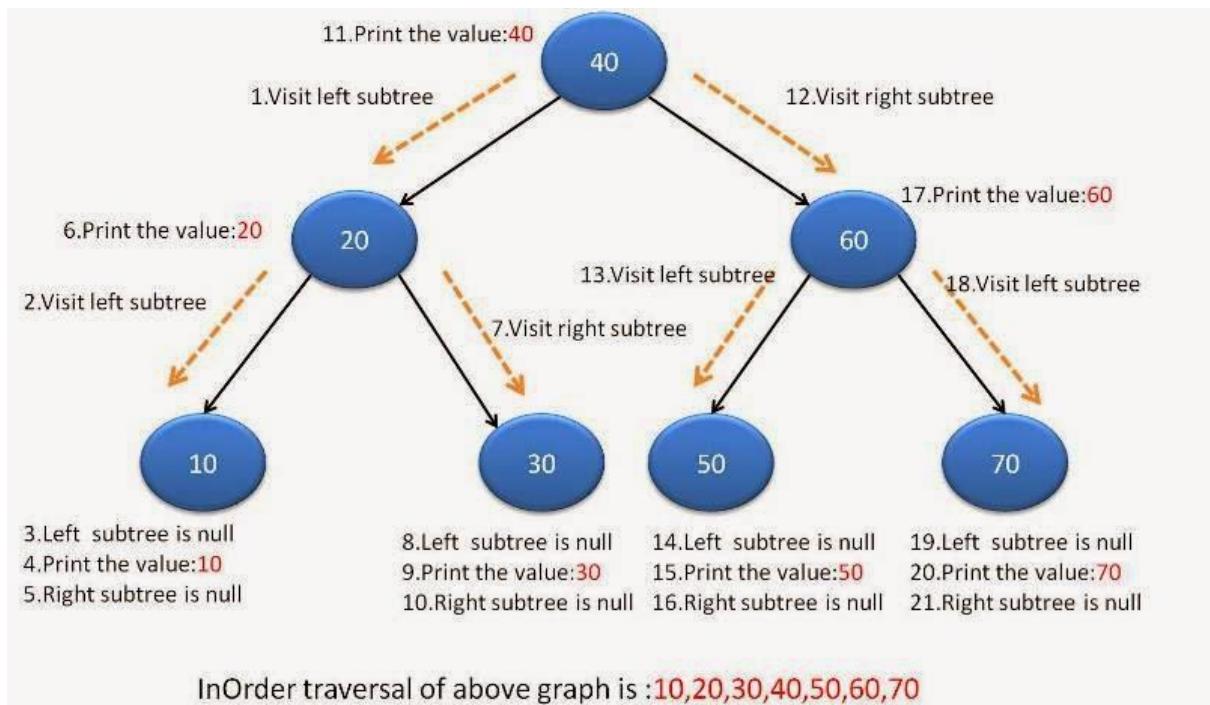
```
import java.util.TreeSet;
class Demo
{
    public static void main(String[] args)
    {
        TreeSet ts = new TreeSet();
        ts.add(120);
        ts.add(60);
        ts.add(160);
        ts.add(30);
        ts.add(80);
        ts.add(140);
        ts.add(180);
        System.out.println(ts);
    }
}
```

Output:

[30, 60, 80, 120, 140, 160, 180]
Press any key to continue . . .

Let us see how exactly we got this order





Methods of TreeSet in java

Methods	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.
E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns comparator that arranged elements in order.
Iterator descendingIterator()	It is used iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending

	order.
E lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.
E pollLast()	It is used to retrieve and remove the highest(last) element.
Spliterator spliterator()	It is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	It returns a set of elements that lie between the given range.
SortedSet subSet(E fromElement, E toElement))	It returns a set of elements that lie between the given range which includes fromElement and excludes toElement.
SortedSet tailSet(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It returns a shallow copy of this TreeSet instance.
E first()	It returns the first (lowest) element currently in this sorted set.
E last()	It returns the last (highest) element currently in this sorted set.
int size()	It returns the number of elements in this set.



Time Complexity

Usually, when we talk about time complexity, we refer to Big-O notation. Simply put, the notation describes how the time to perform the algorithm grows with the size of the input.



Let us see the time complexity with respect to each collection framework we have seen

1. ArrayList

So, let's first focus on the time complexity of the common operations, at a high level:

- **add()** – takes O(1) time
- **add(index, element)** – in average runs in O(n) time
- **get()** – is always a constant time O(1) operation
- **remove()** – runs in linear O(n) time. We have to iterate the entire array to find the element qualifying for removal
- **indexOf()** – also runs in linear time. It iterates through the internal array and checking each element one by one. So the time complexity for this operation always requires O(n) time
- **contains()** – implementation is based on indexOf(). So it will also run in O(n) time



2. LinkedList

LinkedList is a linear data structure which consists of nodes holding a data field and a reference to another node.

Let's present the average estimate of the time we need to perform some basic operations:

- **add()** – supports O(1) constant-time insertion at any position
- **get()** – searching for an element takes O(n) time

- **remove()** – removing an element also takes $O(1)$ operation, as we provide the position of the element
- **contains()** – also has $O(n)$ time complexity

3. PriorityQueue



In Java programming, Java Priority Queue is implemented using Heap Data Structures and Heap

- $O(\log(n))$ time complexity to **insert and delete** element.
- **Offer()** and **add()** methods are used to insert the element in the priority queue java program.
- **Poll()** and **remove()** is used to delete the element from the queue.
- Element retrieval methods i.e. **peek()** and **element()**, that are used to retrieve elements from the head of the queue is constant time i.e. $O(1)$.
- **contains(Object)** method that is used to check if a particular element is present in the queue, have leaner time complexity i.e. $O(n)$.
- Time complexity for the methods **offer & poll** is $O(\log(n))$ and for the **peek()** it is Constant time $O(1)$ of java priority queue.

4. TreeSet

Let's present the average estimate of the time we need to perform some basic operations:

- **add()** – $O(\log n)$ with a base 2 is the time complexity to ass an element.
- **contains()**- $O(\log n)$ with a base 2 is the time complexity to search for an element.
- **next()**- $O(\log n)$ with base 2.

In collections we basically have List, Set, Queue, Map. Below is the table that represents the time complexity for each of them.



List: A list is an ordered collection of elements.

	Add	Remove	Get	Contains	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	Linked List
CopyOnWriteArrayList	O(n)	O(n)	O(1)	O(n)	Array

Set: A collection that contains no duplicate elements.

	Add	Contains	Next	Data Structure
HashSet	O(1)	O(1)	O(h/n)	Hash Table
LinkedHashSet	O(1)	O(1)	O(1)	Hash Table + Linked List
EnumSet	O(1)	O(1)	O(1)	Bit Vector
TreeSet	O(log n)	O(log n)	O(log n)	Red-black tree
CopyOnWriteArraySet	O(n)	O(n)	O(1)	Array
ConcurrentSkipList	O(log n)	O(log n)	O(1)	Skip List

Queue: A collection designed for holding elements prior to processing

	Offer	Peak	Poll	Size	Data Structure
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedList	O(1)	O(1)	O(1)	O(1)	Array
ArrayDeque	O(1)	O(1)	O(1)	O(1)	Linked List
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)	Linked List
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)	Array
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
SynchronousQueue	O(1)	O(1)	O(1)	O(1)	None!
DelayQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)	Linked List

Map: An object that makes keys to values. A map cannot have duplicate keys, each key can map to at most one value.

	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(h / n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(h / n)	Array
WeakHashMap	O(1)	O(1)	O(h / n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	Red-black tree
ConcurrentHashMap	O(1)	O(1)	O(h / n)	Hash Tables
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	Skip List

HASHING

Java **HashSet class** is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface. HashSet contains unique elements only. HashSet allows **null value**.

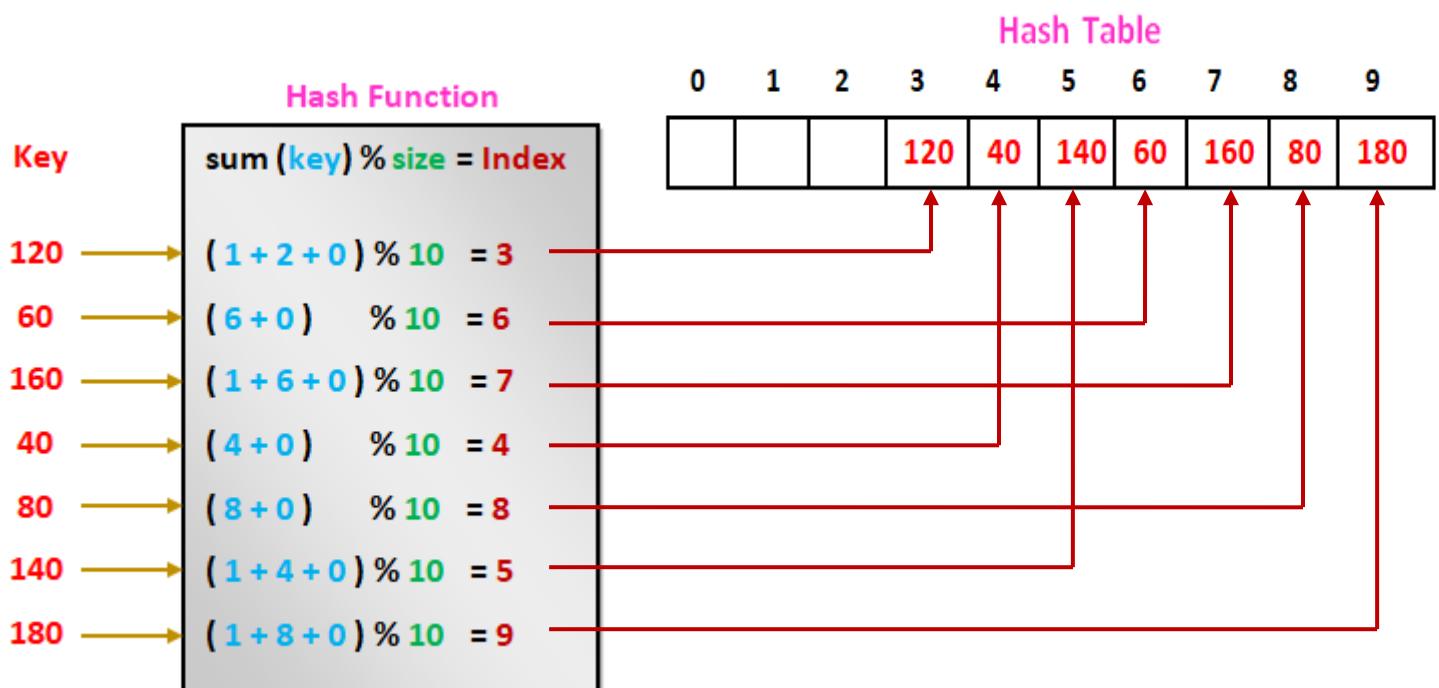
HashSet doesn't maintain any order, the elements would be returned in any random order. Here, elements are inserted on the basis of their hashCode. **HashSet is the best approach for search operations.**

HashSet class in java makes use of the **hashing algorithm** internally to store the data. Hashing makes use of a **hash function** and a **hashtable** to store the data.

The duty of **hashfunction** is to take the data as input and calculate the location on the **hashtable** where the data must be stored.

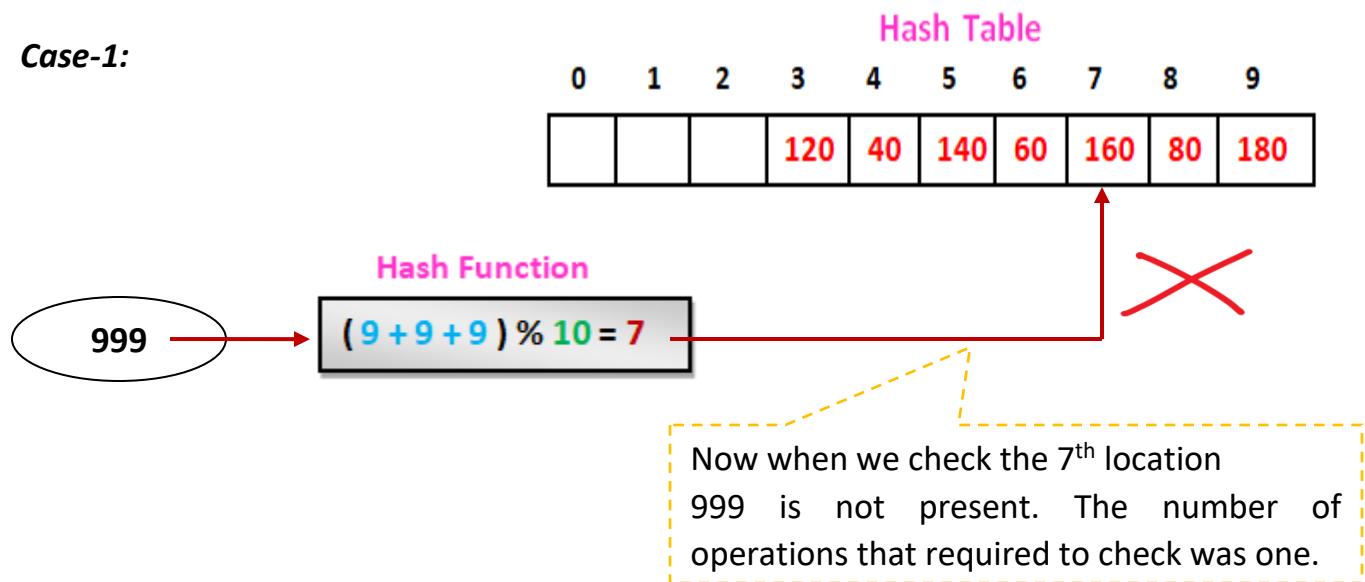
Let's create a hash function, such that our hash table has 10 numbers of buckets. To insert a code into the hash table, we need to find the **hash index** for the given key. And it could be calculated using the hash function.

$$\text{hashIndex} = \text{sum(key)} \% \text{size}$$



Imagine we take some random value to check whether it is present or not in hash table, we make use of hash function with the formula.

Case-1:

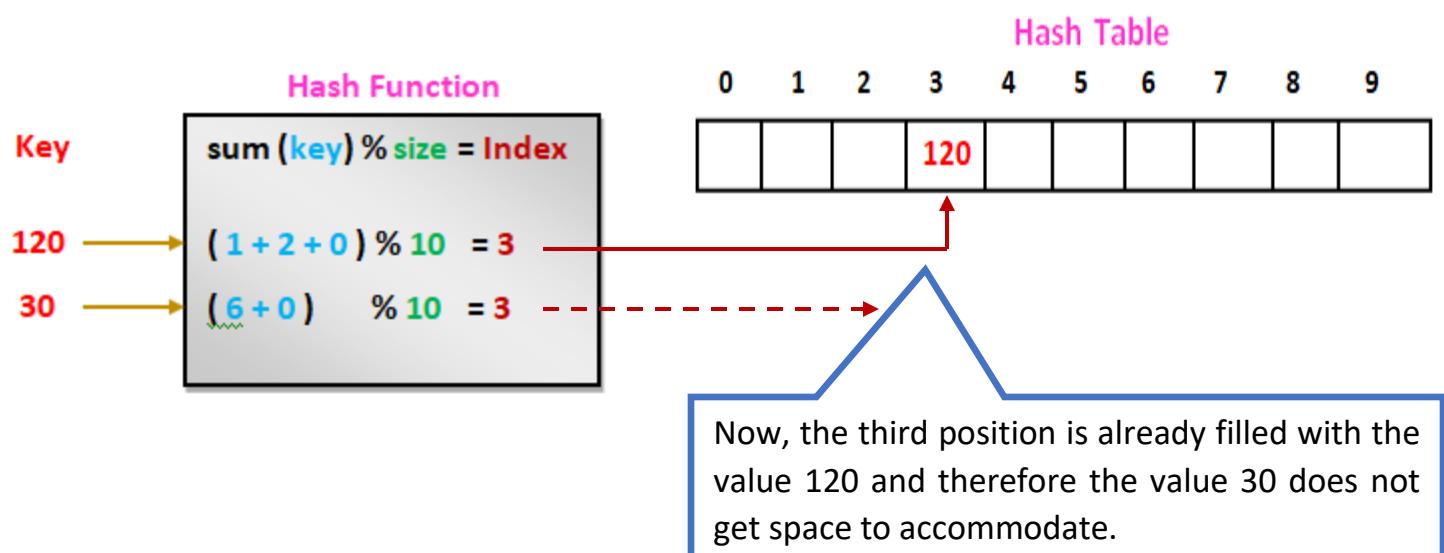


Number of Operation = 1

Hashing is the most efficient algorithm for performing search with a **time complexity of O(1)**.

Let's look at the next Case-2:

Let us imagine we have two data which we pass it to the hash function to find its position in the hash table.



Sometimes two different pieces of data may generate at the same index given to the hash function and this is referred to as ***collision***.

To prevent collision “**Collision resolution technique**” is used such as making use of linked list at a point of collision etc.

Further the moment the capacity of the hashtable is filled to 75% its size is automatically doubled to reduce the chances of collision.

Code:

```
import java.util.HashSet;
class Demo
{
    public static void main(String[] args)
    {
        HashSet hs = new HashSet();
        hs.add(120);
        hs.add(60);
        hs.add(160);
        hs.add(40);
        hs.add(80);
        hs.add(140);
        hs.add(180);
        System.out.println(hs);
    }
}
```



Output

[160, 80, 180, 120, 40, 60, 140]

Press any key to continue...

As we see the output we can clearly understand that HashSet doesn't maintain any order. The elements would be returned in any random order.

But, how to maintain the order??



Well, for that we make use of *LinkedHashSet*.

The LinkedHashSet class extends HashSet class which implements Set interface. Java LinkedHashSet class contains unique elements only like HashSet. Java LinkedHashSet class provides all optional set operation and permits null elements.

LinkedHashSet class in Java makes use of the hashing algorithm internally to store the data.

Java LinkedHashSet class also preserves the order of insertion.

Code:

```
import java.util.LinkedHashSet;
class Demo
{
    public static void main(String[] args)
    {
        LinkedHashSet hs = new LinkedHashSet();
        hs.add(120);
        hs.add(60);
        hs.add(160);
        hs.add(40);
        hs.add(80);
        hs.add(140);
        hs.add(180);
        System.out.println(hs);
    }
}
```

Output:

[120, 60, 160, 40, 80, 140, 180]

Press any key to continue...

Differences and Similarities between HashSet and LinkedHashSet.

HashSet	LinkedHashSet
1) HashSet internally uses HashMap for storing objects.	LinkedHashSet uses LinkedHashMap internally to store its elements.
2) HashSet doesn't maintain any order of elements.	In LinkedHashSet elements are placed as they are inserted.
3) HashSet gives performance of order O(1)	LinkedHashSet gives performance of order O(1)
4) HashSet requires less memory than LinkedHashSet as it uses only HashMap internally to store elements.	LinkedHashSet requires more memory than HashSet as it maintains LinkedList along with HashMap internally to store elements.
5) Use HashSet if you don't want to maintain any order of elements.	Use LinkedHashSet if you want to maintain insertion order of elements.



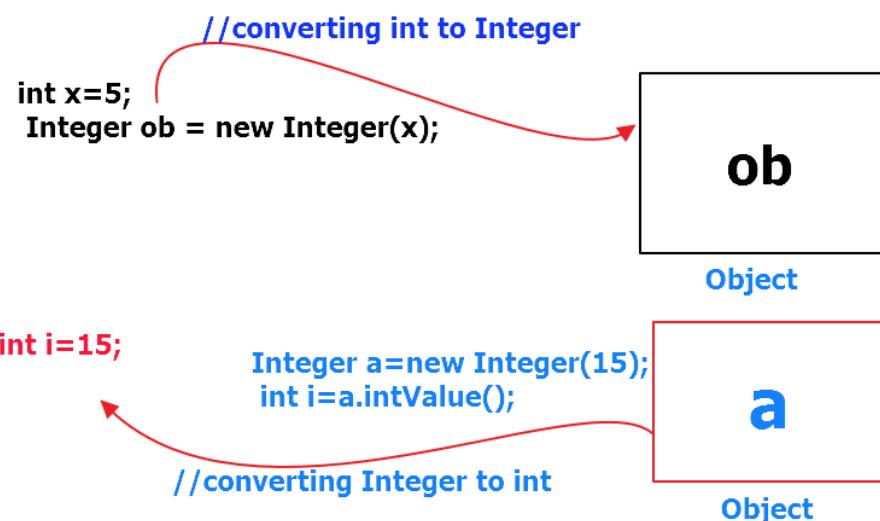
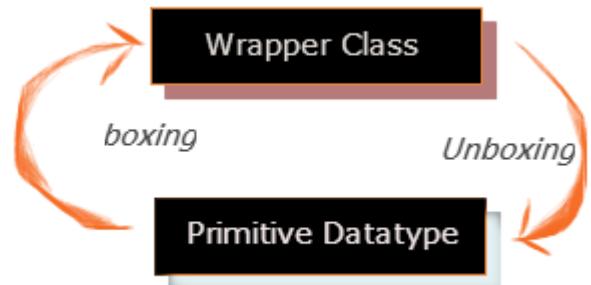
BOXING AND UNBOXING

Definition:

Boxing is the **process of converting a primitive data into its object form** using wrapper class.

Unboxing is the process of **converting an object into its primitive form** using primitive data types.

Both boxing and Unboxing can also be automatically performed by compiler.



boxing and Unboxing



Why should one learn boxing and Unboxing?

All the **collection-based classes can only store objects** and not primitive data types. Even if primitive data is provided, it is **converted into objects** using autoboxing and stored.

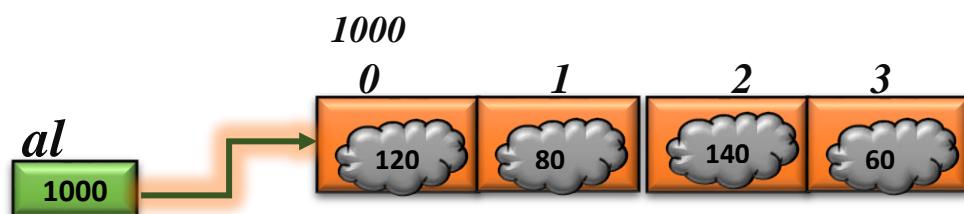


Let us understand this in detail with the help of ArrayList.

```
ArrayList al = new ArrayList();
al.add(120);
al.add(new Integer(120))
al.add(80);
al.add(new Integer(80))
al.add(140);
al.add(new Integer(140))
al.add(60);
al.add(new Integer(60))
```



Internally stored as objects



CODE:

```
import java.util.ArrayList;
class Sis
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(new Integer(120));
        al.add(60);
        al.add(new Double(40.5));
        al.add(80);
        al.add(new Boolean(false));
        al.add("java");
        System.out.println(al);
        Integer a = (Integer)al.get(1);
        System.out.println(a);
    }
}
```



OUTPUT

[120, 60, 40.5, 80, false, java]

60

Press any key to continue . . .



COLLECTIONS HIERARCHY

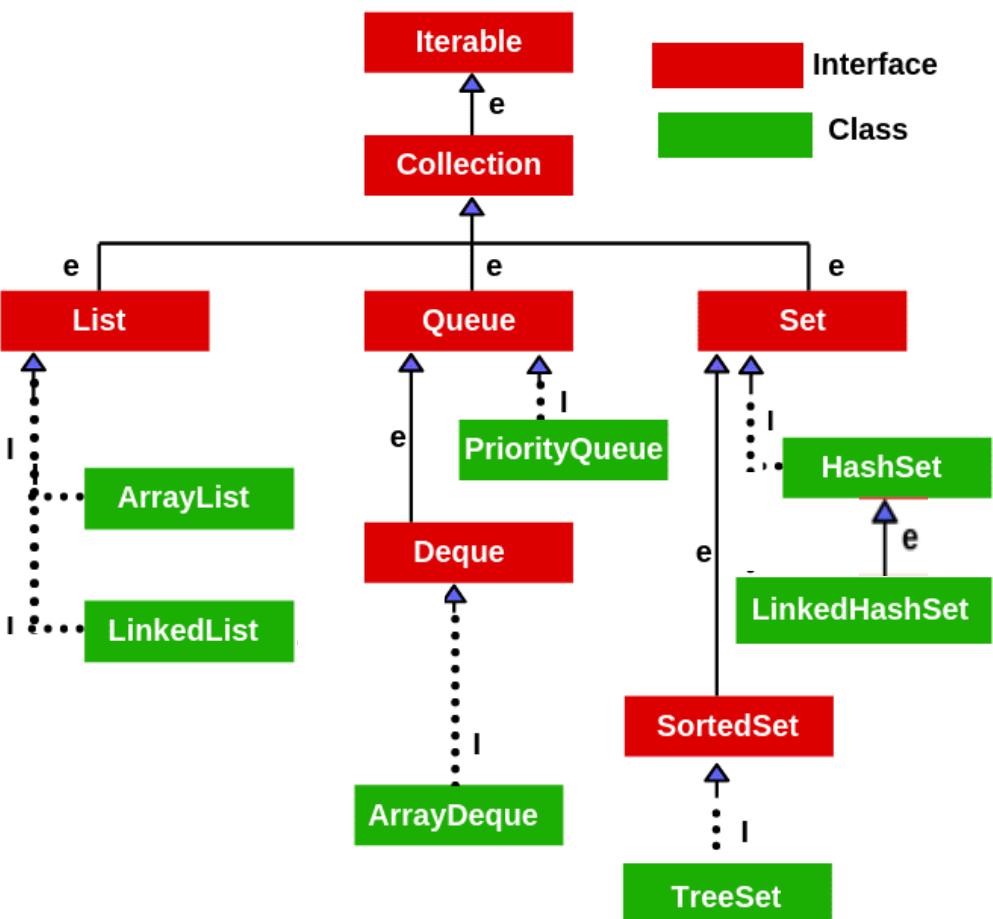


Fig: Collection Hierarchy in Java

Collections Contd...

We have seen several frameworks like ArrayList, LinkedList, ArrayDeque, PriorityQueue, TreeSet, HashSet, LinkedHashSet let us see how to access each one of these

ArrayList and LinkedList

It is clear from previous sessions that values in both these framework have indexes, so indexes can be accessed using for loop. But let's be sure about it by actually seeing the output

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        ArrayList x = new ArrayList();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(int i=0;i<=x.size()-1;i++)
        {
            System.out.println(x.get(i)); //individual value is accessed.
        }
    }
}
```



Output:

```
[120, 40, 60, 180, 120, 160]
120
40
60
180
120
160
Press any key to continue . . .
```

Now let's see for LinkedList

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        LinkedList x = new LinkedList();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(int i=0;i<=x.size()-1;i++)
        {
            System.out.println(x.get(i)); //individual value is accessed.
        }
    }
}
```

Output:

```
[120, 40, 60, 180, 120, 160]
120
40
60
180
120
160
Press any key to continue . . .
```

But that's not the case with other frameworks as they don't have indexes assigned to the values stored in them. So let's see how to access them

For each loop

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java. It

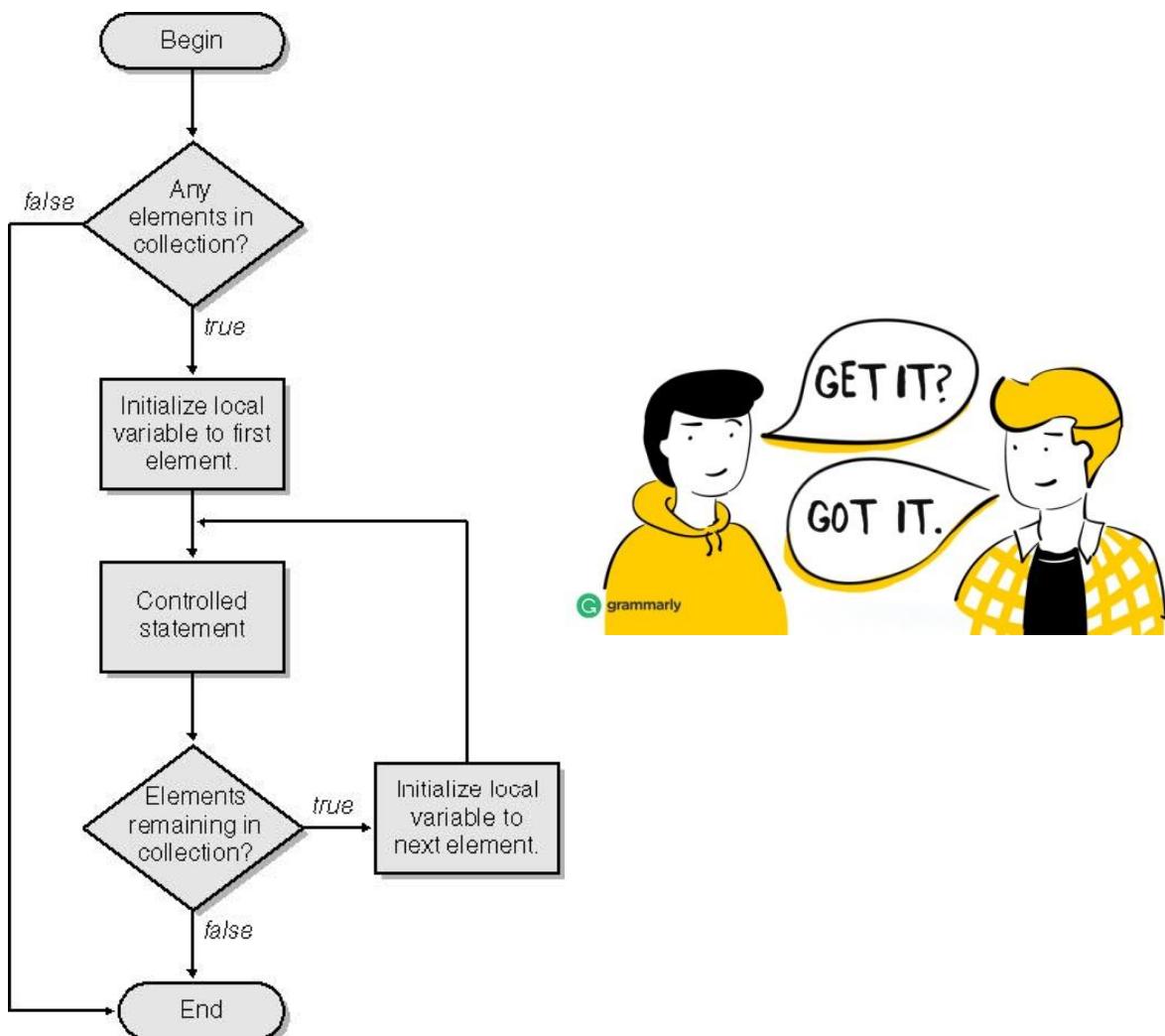
is mainly used to traverse the array or collection elements. The **advantage** of the **for-each loop** is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one.

The **drawback** of the enhanced for loop is that it **cannot traverse the elements in reverse order**.

Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.

But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Flowchart for foreach loop in java



Syntax:

```
for(data_type variable : array | collection)
{
    //body of for-each loop
}
```

Note: foreach loop can we used for array as well as collections.

How does it work?

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.



Let's now see for how many frameworks does this loop works. We will take one example from each List, Queue, Set.

ArrayList from Lists

```
import java.util.*;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList x = new ArrayList();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(Object temp:x)
        {
            System.out.println(temp); //individual value is accessed.
        }
    }
}
```



ArrayDeque from Queue

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        ArrayDeque x = new ArrayDeque();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(Object temp:x)
        {
            System.out.println(temp); //individual value is accessed.
        }
    }
}
```

TreeSet from Set

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        TreeSet x = new TreeSet();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(Object temp:x)
        {
            System.out.println(temp); //individual value is accessed.
        }
    }
}
```

Output:

```
[120, 40, 60, 180, 120, 160]
120
40
60
180
120
160
Press any key to continue . . .
```

Let us now see the next method to access the values.

Iterator

In Java, Iterator is an interface available in Collection framework in `java.util` package. It is a Java Cursor used to iterate a collection of objects.

Key Points

- It is used to traverse a collection object elements one by one.
- It is available since Java 1.2 Collection Framework.
- It is applicable for all Collection classes. So it is also known as Universal Java Cursor.
- It supports both READ and REMOVE Operations.
- Compare to Enumeration interface, Iterator method names are simple and easy to use.

Let us see one example and understand in better way

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        TreeSet x = new TreeSet();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        Iterator itr = x.iterator();
        while(itr.hasNext()==true)
        {
            System.out.println(itr.next());
        }
    }
}
```

Output:

```
[40, 60, 120, 160, 180]
40
60
120
160
180
Press any key to continue . . .
```

Iterator vs ListIterator:

- 1) Iterator is used for traversing **List and Set both**.

We can use ListIterator to traverse **List only**, we cannot traverse Set using ListIterator.

- 2) We can traverse in **only forward direction** using Iterator.

Using ListIterator, we can **traverse a List in both the directions** (forward and Backward).

- 3) We **cannot obtain indexes** while using Iterator

We **can obtain indexes** at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

- 4) We **cannot add element to collection while traversing it using Iterator**, it throws **ConcurrentModificationException** when you try to do it.

We **can add element at any point of time while traversing a list using ListIterator**.



5) We **cannot** replace the existing element value when using **Iterator**.

By using **set(E e)** method of **ListIterator** we can replace the last element returned by **next()** or **previous()** methods.

6) Methods of Iterator:

- **hasNext()**
- **next()**
- **remove()**

Methods of ListIterator:

- **add(E e)**
- **hasNext()**
- **hasPrevious()**
- **next()**
- **nextIndex()**
- **previous()**
- **previousIndex()**
- **remove()**
- **set(E e)**



Collections

1. What is the Collection framework?

Collection framework represents an architecture for storing and manipulating a group of objects. Collection framework has interfaces, classes and also few algorithms.

2. What is the root interface in Collection hierarchy?

Iterable interface is the root interface in collection hierarchy.

3. What is an Iterable interface?

Iterable interface facilitates to iterate the elements. It iterates in forward direction only. It provides a generic way for traversal through the elements of a Collections and implements Iterator design pattern.

4. Which design pattern is followed by Iterable interface?

Iterable interface follows iterator design pattern.

5. What was the java's Collections framework equivalent in C++ called as?

Java's Collections framework equivalent in C++ is called as Standard Template Library.

6. What was the java's Collections framework equivalent in smalltalk called as?

Java's collection framework equivalent in Smalltalk is called as Small Talk Collection hierarchy.

7. What are the benefits of java's Collection framework?

Advantages of Collections framework are as follows:

- i) Reduces the effort in programming.
- ii) Increases the speed of programming and quality.
- iii) Allows inter-operability.

iv) Reduces effort to design new software.

8. When was Collections framework introduced in java?

Collections framework was introduced into Java from JDK 1.2 onwards.

9. Which were the adhoc classes that were provided in java for dealing with collection of objects before java's Collection framework was introduced?

Prior to Collections framework was introduced in Java, James Gosling had provided four classes in order to perform storage and retrieval of the objects namely,

- i) Vector
- ii) Stack
- iii) HashTable and
- iv) Properties

10. What was the problem with these adhoc classes?

Although these legacy classes were useful, they lacked a central unifying theme. This created a lot of confusion for the programming community because the way Vector class used was different from the way Stack class or HashTable class was used. This adhocness lead to the emergence of the Collections hierarchy.

11. What is a Dictionary?

Dictionary is a legacy abstract class. It contains <key-value> pair. It is much like today's Map class.

12. What is a Hashtable?

HashTable is a legacy class. HashTable is the concrete implementation of Dictionary.

13. What is a Properties class?

Properties class is a legacy class. It is a subclass of HashTable class. It

is used to maintain <key-value> pair such that both key and value are String.

Eg. <"Dhoni", "Mahi">

14. When does a ClassCastException occur?

Whenever in a program we attempt to typecast incompatible types then ClassCastException will occur.

15. When does an UnsupportedOperationException occur?

Whenever we attempt to modify an unmodifiable Collection, then UnsupportedOperationException will occur.

16. Name the three static variables present in Collection framework?

- 1) EMPTY_SET
- 2) EMPTY_LIST
- 3) EMPTY_MAP

17. Differentiate between Array and ArrayList?

Arrays	ArrayList
Can store only primitive data.	Can store only Objects
Arrays are fixed in size. They cannot grow or shrink during program execution.	ArrayList can grow or shrink in size during program execution
Can store only homogeneous data.	Can store heterogeneous data.
Creation takes less time	Creation would take more time because auto boxing has to take place using wrapper classes
Auto boxing is not required.	Auto boxing is required.

18. When will we use Array over ArrayList?

- 1) When the size of the array is either fixed or known earlier.
- 2) When we have only primitive data have to be stored.
- 3) When we have multi-dimensional data have to be stored.

19. What are the similarities between ArrayList and Vector?

- 1) ArrayList and Vector maintains the order of insertion.
- 2) ArrayList and Vector are both fail-fast.
- 3) ArrayList and Vector allow null values, both provide index based accessing.

20. Differentiate between ArrayList and Vector?

ArrayList and Vector are almost similar. However, Vector class is synchronized whereas ArrayList if not synchronized.

21. Differentiate between ArrayList and LinkedList?

ArrayList	LinkedList
Internally makes use of dynamic array	Internally makes uses of linked list
Suitable for performing rear end insertion	Suitable for performing insertion at all given position
Cannot utilize dispersed memory location	Can utilize dispersed memory location

22. Differentiate between Iterator and ListIterator?

Iterator	ListIterator
Can be used to access objects present in all the seven Collections classes	Can be used to access objects present in only list based classes

Iterator can iterate upon Collections only in the forward direction	ListIterator can iterate upon Collections both in forward and reverse direction
---------------------------------------------------------------------	---------------------------------------------------------------------------------

23. Differentiate between Iterator and Enumeration?

Iterator	Enumeration
Iterator can traverse legacy and non-legacy elements	Enumeration can traverse only legacy elements
Iterator is fail-fast	Enumeration is not fail-safe
Iterator is slower than Enumeration	Enumeration is faster than Iterator

24. Differentiate between List and Set?

In List index based access is provided and also duplicates are allowed. Where as in set, index based access is not permitted and duplicates are not allowed.

25. Differentiate between HashSet and TreeSet?

HashSet	TreeSet
Internally makes use of Hashing algorithm	Internally makes use of balanced binary search tree using red-black algorithm
Objects would not get displayed in the sorted order	It would display the objects in sorted order because it would make use of in-order traversal.
Search time complexity is O(1)	Search time complexity is O(log ₂ n)

26. Differentiate between Set and Map?

Set	Map
Set stores only values.	Map stores <key, value> pair.
Set is a member of Collections hierarchy.	Map is not a member of Collection hierarchy.
As the complexity of an object increases, efficiency reduces.	Efficiency would not be reduced even though complexity increases because hashing would be applied on “key” and not on value.

27. Differentiate between HashSet and HashMap?

HashSet	HashMap
It is an implementation of Set interface	It is an implementation of Map Interface
Contains only values (objects)	Contains object as <key,value> pair
Duplicates are not permitted	Duplicate keys are not permitted. However, duplicate values are permitted
add() method would be used to insert an object	put() method would be used for insertion operation
One parameter has to be passed while inserting an object. Eg. add(Object arg0);	Two parameters have to be passed for put() method. Eg. put(Object arg0, Object arg1) put(key , value)
Slow in operation	Fast in operation

28. Differentiate between HashMap and TreeMap?

HashMap	TreeMap
Internally makes use of hashing algorithm.	Internally makes use of balanced binary search tree using red-black algorithm.
Data cannot be retrieved in a sorted manner.	Data can be retrieved in a sorted manner.

29. Differentiate between HashMap and Hashtable?

HashMap	HashTable
It is not a synchronized class.	It is a synchronized class.
It is not a legacy class.	It is a legacy class.
It can contain an empty Key or Value pair.	It cannot contain an empty Key or value pair.
Suitable for multithreaded programming	Not suitable for multithreaded programming

30. Differentiate between Collection and Collections?

“Collection” is an interface whereas “Collections” is a class.

31. Differentiate between Comparable and Comparator?

Comparable	Comparator
It expects implementation for compareTo() method	It expects implementation for compare() method

The first object is referred to as "this"	The first object is not referred to as "this"
It is present in java.lang package	It is present in java.util package
Target class has to be modified	Target class need not be modified
It is useful in such scenarios where the target class is modifiable	It is useful in all the scenarios

32. Are the List, Set and Map elements automatically synchronized in java's Collection framework?

No. They are not automatically synchronized.

33. How do we synchronize List, Set and Map elements?

Using Collections.synchronizeCollection(Collection c)

34. What is the advantage of generic Collection?

Advantages of generic collections are:

- i) It is type safe
- ii) It is checked at compilation time
- iii) It prevents ClassCastException
- iv) It makes the code clean since we need not use casting.

35. What is collision in Hashtable and how is it handled?

While computing bucket number for storing an object in hash table, if hash function allocates the same bucket for two different objects, then collision occurs. Therefore, hash function maintains a load factor which is 0.75 or 75% which means, if the hash table gets filled by the objects about 75% of its capacity, then automatically the size of the hash table would increase so that collision would be avoided.

36. What is load factor in hashing base Collection?

While computing bucket number for storing an object in hash table, if hash function allocates the same bucket for two different objects, then collision occurs. Therefore, hash function maintains a load factor which is 0.75 or 75% which means, if the hash table gets filled by the objects about 75% of its capacity, then automatically the size of the hash table would increase so that collision would be avoided.

37. Why is Map not a part of Collection framework?

Map is not a part of Collections framework because map contains <key, value> pair and does not fit into the Collections framework paradigm which contains only values.

38. What is the difference between fail-fast and fail-safe?

Fail-fast	Fail-safe
It would result in termination of the program by generating an Exception if structural modification is attempted	Does not terminate the program by generating an Exception if structural modification is attempted
No clone is used	Clone is used
All the Collections based classes present in java.util package exhibit fail-fast behavior	All the Collections based classes present in java.util.concurrent package exhibit fail-safe behavior

39. How do we avoid ConcurrentModificationException while iterating Collection?

To avoid ConcurrentModificationException we can make use of the classes present in java.util.concurrent package.

40. What are the different Collection views provided by Map interface?

The different Collection views are:

- 1) Keyset
- 2) EntrySet
- 3) values.

41. How do you decide between using HashMap and TreeMap?

In the project, if insertion, deletion and locating elements operations are of utmost important, then HashMap must be used. However, if traversing the keys in a sorted order is important then TreeMap must be used.

42. Which Collection classes are Thread-safe?

The legacy classes such as Vector, Hashtable, Properties and Stack are thread safe classes because they are synchronized.

However, from jdk 1.5 onwards, Collections based classes that are present in java.util.concurrent package such as CopyOnWriteArrayList, CopyOnWriteArraySet, ConcurrentHashMap etc. are also thread safe classes.

43. What is a blocking Queue?

Blocking Queue is a queue in which retrieval is possible if and only if the queue is non-empty and insertion is possible only when space is available. It is used for implementing producer-consumer problem.

44. How can we sort a list of objects?

We can sort a list of objects by :

- i) Using TreeSet class
- ii) Using Collections.sort() method.

45. While passing a Collection as argument to a function how can we make sure that a function will not be able to modify it?

By creating read only collection using:

Collections.unmodifiableCollection(Collection c)

46.What are the common algorithms implemented in Collections class?

- i) Sorting
- ii) Shuffling
- iii) Routine Data Manipulation
- iv) Searching
- v) Composition
- vi) Finding Extreme Values etc.

47. What are the best practices related to java Collections framework?

- i) Choose the best type of Collection depending upon objects to be stored.
- ii) If the number of objects to be stored is known in advance then preferably we
 - can use Collections classes that allow us to specify the initial capacity.
- iii) We can always use generics for type safety and to avoid ClassCastException at
 - run time.
- iv) We can also use Collections utility class methods since it enhances code
 - re-usability, greater stability and maintainability.

48. Which are the classes that have implemented the List interface?

ArrayList and LinkedList classes.

49. Which are the classes that have implemented the Set interface?

HashSet and LinkedHashSet classes.

50. Which are the classes that have implemented the Queue interface?

ArrayDeque and PriorityQueue classes.

51. Which are the classes that have implemented the Map interface?

HashMap, TreeMap and LinkedHashMap classes.

52. Which methods do we have to override in order to use an object as a key in HashMap?

We have to override hashCode() and equals() methods.

53. What is the difference between Stack and Queue?

Stack	Queue
Uses LIFO (Last-In-First-Out) or FILO (First-In-Last-Out) to add and access data.	Uses FIFO (First-In-First-Out) or LILO (Last-In-Last-Out) to add and access data.
Insertion and deletion would happen only at one end	Insertion and deletion would happen at separate ends. For insertion, rear end would be used. For deletion, front end would be used.
Implementation is relatively easy as Stack has no variants.	Implementation is a bit complex as Queue is having variants such as Circular Queue, Priority Queue, Double ended Queue etc.

54. How do we reverse the List in Collection?

We can reverse the list in collection by using Collections.reverse() method.

55. How do we convert an array of String into a List?

We can convert an array of String into a list by using
Collections.addAll() or Arrays.asList() method.

**56. What is the difference between peek(), poll() and remove()
methods of Queue interface?**

peek() and poll() would return null if the queue is empty whereas
remove() would throw NoSuchElementException.

**57. What is the difference between HashMap and
ConcurrentHashMap?**

ConcurrentHashMap is synchronized and does not allow null key and
null value, whereas HashMap is not synchronized and allows null key
and null value.

**58. Arrange the following in the descending order of
performance- ConcurrentHashMap, Hashtable, HashMap,
Collections.SynchronizedMap.**

HashMap, ConcurrentHashMap, Collections.SynchronizedMap,
Hashtable.

59. How does HashMap work?

(Refer class notes).

60. What is identity HashMap?

Identity HashMap implements Map interface by using reference
equality instead of object equality.

61. What is weakHashMap?

WeakHashMap is an implementation of Map interface with weak keys.

62. What is a Data Structure?

A data structure is a specialized format for organizing and storing the data.

Eg. Array, Queue, LinkedList, Tree etc.

63. Can any of the Collection class store primitive data?

No. In all the Collections classes data would be stored in the object format.

64. What happens when a primitive data is given to a Collection class?

When primitive data is given to a collections class, it would be automatically converted into an object format. This process is called as auto boxing.

65. What is meant by boxing?

(Refer class notes).

66. What is meant by auto-boxing?

(Refer class notes).

67. What is meant by unboxing?

(Refer class notes).

68. What is meant by auto-unboxing?

(Refer class notes).

69. What is the initial capacity of a Vector?

Initial capacity of a vector is 10.

70. By what factor does the capacity increase in case of a Vector if the existing capacity is filled?

If the existing capacity is filled then the capacity doubles in existing

size.

71. What are the different ways of printing a Vector?

We can print a vector by using Enumeration.

72. What are the different ways of printing a ArrayList?

We can print ArrayList by using:

- i) for loop
- ii) enhanced for loop
- iii) iterator and
- iv) ListIterator.

73. What are the different ways of printing a LinkedList?

We can print LinkedList by using:

- i) for loop
- ii) enhanced for loop
- iii) iterator
- iv) ListIterator and
- v) DescendingIterator

74. What are the different ways of printing a ArrayDeque?

We can print ArrayDeque by using:

- i) enhanced for loop
- ii) iterator and
- iii) DescendingIterator

75. What are the different ways of printing a PriorityQueue?

We can print a PriorityQueue by using:

- i) enhanced for loop and
- ii) iterator.

76. What are the different ways of printing a TreeSet?

We can print a TreeSet using:

- i) enhanced for loop
- ii) iterator and
- iii) DescendingIterator.

77. What are the different ways of printing a HashSet?

We can print a HashSet using:

- i) enhanced for loop and
- ii) iterator.

78. What are the different ways of printing a LinkedHashSet?

We can print a HashSet using:

- i) enhanced for loop and
- ii) iterator.

79. What are the different ways of printing a TreeMap?

We can print a TreeMap using:

- 1) enhanced for loop
- 2) iterator and
- 3) DescendingIterator.

80. What are the different ways of printing a LinkedHashMap?

We can print a LinkedHashMap using

- 1) enhanced for loop and
- 2) iterator.

81. What are the different ways of printing a LinkedHashMap?

We can print a LinkedHashMap using:

- 1) enhanced for loop and

2) iterator

82. What is the role of clone()?

A clone() method is used to create a new Collections object. It is an alternative method to new operator and factory method to create the objects.

83. How can an Array be converted to a List?

We can convert an Array into a List by using Arrays.asList() method.

84. How can a List be converted to an Array?

We can convert a List into an Array by using toArray() method.

85. What is the difference between add() and set()?

add() method is used to insert a new object at the specified position. If any other object already exists in that position, then it would be shifted to the right so that empty location would be created to insert a new object.

In case of set() method, if any other object exists in the specified position, then shifting towards right would not take place. Rather, new object would be replaced with old object by deleting it.

86. What is the role of retainAll()?

The **retainAll()** method retains all matching elements in the current **ArrayList** that match all elements from the derived Collection list argument. In other words, retainAll() method is equivalent to intersection operation.

87. Which are the Collection class in which duplicates are permitted?

Duplicates are permitted in ArrayList, LinkedList, ArrayDeque and PriorityQueue classes.

88. Which are the Collection class in which duplicates are not permitted?

Duplicates are not permitted in TreeSet, HashSet and LinkedHashSet classes.

89. Which are the Collection class in which null are permitted?

Null is permitted in ArrayList, LinkedList, HashSet and LinkedHashSet classes.

90. Which are the Collection class in which null is not permitted?

Null is not permitted in ArrayDeque, PriorityQueue and TreeSet.

91. What are the two ways in which a LinkedList can be accessed in the reverse order?

- i) Using ListIterator
- ii) Using DescendingIterator

(Eg. Refer class notes)

92. What is the difference between add() and offer()?

add() method would throw **IllegalStateException** if the space is not available to insert an object into a Collections.

offer() method would not throw any Exception if the space is not available to insert an object into a Collections. Rather, it would return false.

93. Which Data Structure is used internally by the Vector class?

Vector uses Dynamic array data structure.

94. Which Data Structure is used internally by the ArayList class?

ArrayList uses Dynamic array data structure.

95. Which Data Structure is used internally by the LinkedList class?

Doubly linked list data structure.

96. Which Data Structure is used internally by the ArrayDeque class?

Double ended queue data structure.

97. Which Data Structure is used internally by the PriorityQueue class?

PriorityQueue uses min-heap data structure.

98. Which Data Structure is used internally by the TreeSet class?

TreeSet uses balanced binary search tree data structure using red-black algorithm.

99. Which Data Structure is used internally by the HashSet class?

HashSet uses Hashing algorithm.

100. Which Data Structure is used internally by the LinkedHashSet class?

LinkedHashSet uses Hashing algorithm.

101. Which Data Structure is used internally by the TreeMap class?

min-heap data structure.

102. Which Data Structure is used internally by the HashMap class?

Hashing algorithm.

103. Which Data Structure is used internally by the LinkedHashMap class?

Hashing algorithm.

104. Under what circumstances should we use the Vector class?
(Refer class notes).

105. Under what circumstances should we use the ArrayList class?
(Refer class notes).

- 106. Under what circumstances should we use the LinkedList class?**
(Refer class notes).
- 107. Under what circumstances should we use the ArrayDeque class?**
(Refer class notes).
- 108. Under what circumstances should we use the PriorityDeque class?**
(Refer class notes).
- 109. Under what circumstances should we use the TreeSet class?**
(Refer class notes).
- 110. Under what circumstances should we use the HashSet class?**
(Refer class notes).
- 111. Under what circumstances should we use the LinkedHashSet class?**
(Refer class notes).
- 112. Under what circumstances should we use the TreeMap class?**
(Refer class notes).
- 113. Under what circumstances should we use the HashMap class?**
(Refer class notes).
- 114. Under what circumstances should we use the LinkedHashMap class?**
(Refer class notes).
- 115. What is the role of ceiling () and floor ()?**
(Refer class notes).

116. **What is the role of higher () and lower ()?**
(Refer class notes).
117. **What is the role of headSet() and tailSet()?**
(Refer class notes).
118. **Give the Collection interface hierarchy?**
(Refer class notes).
119. **Give the Map interface hierarchy?**
(Refer class notes).
120. **How can we perform reverse sorting on a Vector?**
(Refer class notes).
121. **How do you sort the data present in non-indexing data structure?**
(Refer class notes).
122. **How do you sort a Map?**
(Refer class notes).

CORE JAVA - Day 56

Agenda

- Introduction to Map
- Introduction to HashMap



MAP: A map is an object that stores associations between keys and values, or key/value pairs. The Map interface maps unique keys to values. Both keys and values are objects. The Map is declared as shown here:

Interface **Map<k,v>**

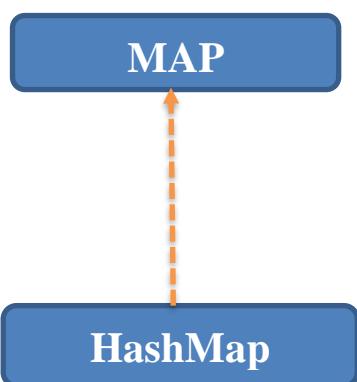
Here, K specifies the type of keys, and V specifies the type of values.

Since Map is an interface you need to instantiate a concrete implementation of the interface in order to use it. HashMap class will implement Map interface. HashMap declaration is:

HashMap<k,v>

It stores the data in (Key, Value) pairs. Inside the angular braces you have to mention the data type of key and value.

Now let's understand creation and few operations performed on HashMap with an example.



Example-1: Creating HashMap

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Print the elements inside the hashMap
        System.out.println(myDetails);
    }
}
```

Output:

Command Prompt
\$java MapIntro.java
{}

Here an empty HashMap is created and no elements are added inside the HashMap so if you try to print the HashMap you are getting an empty HashMap in the output.

put(key, value) method in HashMap: It is used to insert a mapping into a map. This means we can insert a specific key and the value it is mapping to into a particular map. If an existing key is passed then the previous value gets replaced by the new value. If a new pair is passed, then the pair gets inserted as a whole. Let's now understand with examples.

Example-2: Adding the elements inside HashMap

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562L");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Print the elements inside the hashMap
        System.out.println(myDetails);
    }
}
```

Output:

```
Command Prompt  
$java MapIntro.java  
{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562L, Gender=Male, Surname=Somanna, Password=##90$$}
```

put(key, value) is used to **add** elements to the HashMap, it will accept two-parameter key and value. **put(key, value)** will put the elements in the HashMap only if the given key is not present inside the HashMap because duplicate keys are not allowed in the HashMap, duplicate values can be stored. **put(key,value)** will not only put the element inside the HashMap it will **return** something. If you observe the output you can clearly see that HashMap doesn't follow the insertion order. Now will see what **put(key, value)** will return with an example.

Example-3:

```
import java.util.HashMap;  
  
public class MapIntro {  
  
    public static void main(String[] args) {  
        //Create an empty HashMap  
        HashMap<String, String> myDetails = new HashMap<String, String>();  
  
        //Adding elements to the HashMap and print the value returned by HashMap  
        System.out.println(myDetails.put("FirstName", "Alex"));  
        System.out.println(myDetails.put("FirstName", "Somanna"));  
  
    }  
}
```

Output:

```
Command Prompt  
$java MapIntro.java  
null  
Alex
```

In this example, the first time you are getting null as output because initially, HashMap was empty when **myDetails.put("FirstName", "Alex")** line executes it will check whether FirstName i.e given **key** is present inside the

HashMap or not as HashMap was empty it will return **null** and put FirstName and Alex inside the HashMap. When **myDetails.put("FirstName", "Somanna")** executes it will again go and check inside the HashMap whether FirstName i.e given **key** is present inside the HashMap or not, now FirstName is present inside the HashMap so it will return it's associated value i.e **Alex** and replace Somanna in place of Alex(Duplicate keys are not allowed).

Property of HashMap:

- Cannot store duplicate keys.
- Can store duplicate values.
- Doesn't maintain the insertion order.

get(key) method in HashMap: Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. Now let's understand get(key) with an example.

Example-4

```
import java.util.HashMap;

public class MapIntro {

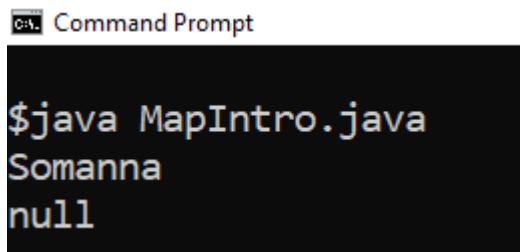
    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562L");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Print the value returned by get()
        System.out.println(myDetails.get("FirstName"));

        System.out.println(myDetails.get("Lastname"));
    }
}
```

Output:



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window contains the following text:
\$java MapIntro.java
Somanna
null

Here first, you are getting Somanna as the output as the key FirstName is present inside the HashMap it's associated value is returned. Next, you are getting null as the output as the key LastName is not present inside the HashMap.

CORE JAVA - Day 57

Agenda

- **values() method**
- **keyset() method**
- **entrySet() method**
- **getKey(), getValue(), setValue() method**



values () method in HashMap: values() method returns a view of all the values present in entries of the HashMap.

Example: To fetch all the values present inside the HashMap.

```
import java.util.Collection;
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562L");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using values() to get the set view of values
        Collection<String> values = myDetails.values();

        //printing the set of values
        System.out.println(values);
    }
}
```

Output:

```
Command Prompt
$java MapIntro.java
[Somanna, 20/02/1947, 8965474562L, Male, Somanna, ##90$$]
```

In this example, values() method will return collection of values stored in the HashMap and we are storing this in a variable values which is of type collection.

Example: values () method in for each loop

```
import java.util.Collection;
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562L");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using values() to get the set view of values
        Collection<String> values = myDetails.values();

        // for-each loop access each value from the view
        for(String value : values) {
            //printing each value
            System.out.println(value);
        }
    }
}
```

Output:

```
Command Prompt
$java MapIntro.java
Somanna
20/02/1947
8965474562L
Male
Somanna
##90$$
```

Here, the **values()** method returns a view of all values. The variable value access each value from the view using for each loop.

keySet() method in HashMap: Returns a Set view of all the keys present in entries of the HashMap.

Example:

```
import java.util.HashMap;
import java.util.Set;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562L");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using keySet() to get the set view of keys
        Set<String> keys = myDetails.keySet();

        // for-each loop access each key from the view
        for(String key : keys) {
            //printing each key
            System.out.println(key);
        }
    }
}
```

Output:



```
Command Prompt
$java MapIntro.java
FirstName
DOB
Phone number
Gender
Surname
Password
```

Here, the **keySet()** method returns a view of all keys. The variable key access each value from the view using for each loop.

Example: To print key and value present in the HashMap

```
import java.util.HashMap;
import java.util.Set;

public class MapIntro {

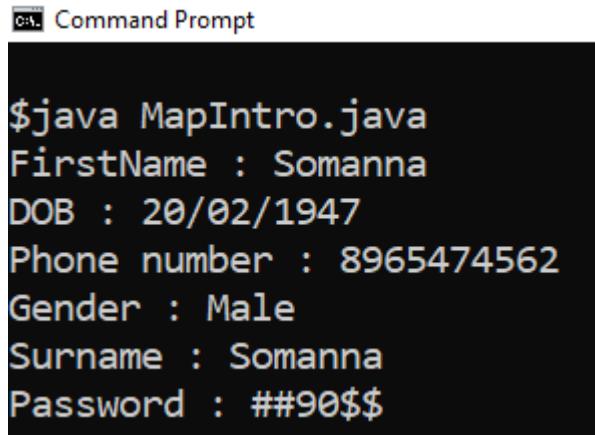
    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using keySet() to get the set view of keys
        Set<String> keys = myDetails.keySet();

        // for-each loop access each key from the view
        for(String key : keys) {
            //printing each key and value
            System.out.println(key + " : " + myDetails.get(key));
        }
    }
}
```

Output:



The screenshot shows a terminal window titled 'Command Prompt'. The command '\$java MapIntro.java' is entered at the prompt. The output displays six key-value pairs: FirstName : Somanna, DOB : 20/02/1947, Phone number : 8965474562, Gender : Male, Surname : Somanna, and Password : ##90\$\$.

```
$java MapIntro.java
FirstName : Somanna
DOB : 20/02/1947
Phone number : 8965474562
Gender : Male
Surname : Somanna
Password : ##90$$
```

Here, we are printing the keys using keySet() method to view all the keys present in the HashMap. Key is used to fetch the key present in the view of keys using key we are fetching the value i.e using get(key) method.

Understanding entry and entrySet in the HashMap.

Entry is simple **key-value** pair, **entrySet** is a set of key-value pairs present inside the **HashMap**. **Map** is an interface inside which we have so many abstract methods, inside the map interface there is another interface called **Entry**. Inside the entry interface we have few more abstract method, let's understand all these methods with an example.

MAP

put()
get()
values()
keyset()

.

.

.

.

.

.

.

Entry

getKey()
getValue()
setValue()

Example: entrySet() method

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class MapIntro {

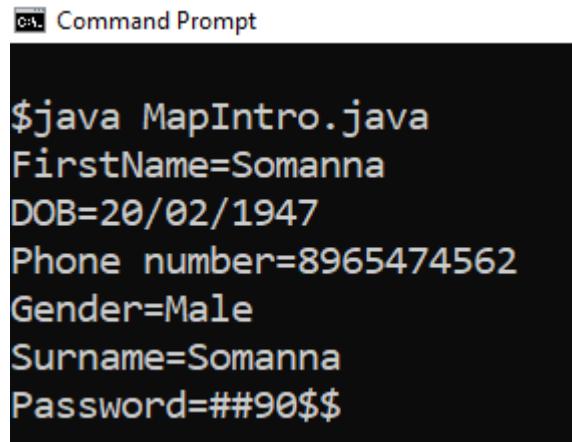
    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using entrySet() to get the set view of entries
        Set<Entry<String, String>> entrys = myDetails.entrySet();

        // for-each loop access each key from the view
        for(Entry<String, String> entry : entrys) {
            //printing each entry
            System.out.println(entry);
        }
    }
}
```

Output:



```
$java MapIntro.java
FirstName=Somanna
DOB=20/02/1947
Phone number=8965474562
Gender=Male
Surname=Somanna
Password=##90$$
```

Here, entrySet() method returns set of entries present in the HashMap. The variable entry access each value from the view of entries using for each loop.

Example: getKey() method

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class MapIntro {

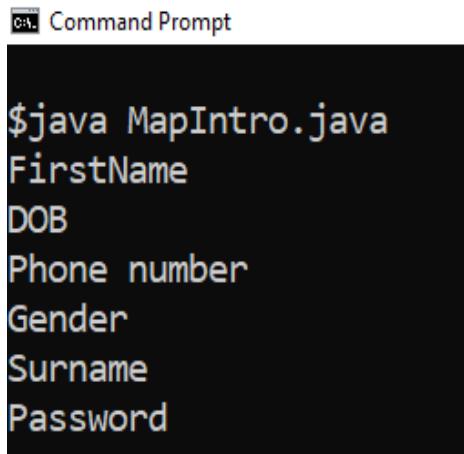
    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using entrySet() to get the set view of entries
        Set<Entry<String, String>> entrys = myDetails.entrySet();

        // for-each loop access each key from the view
        for(Entry<String, String> entry : entrys) {
            //printing each key
            System.out.println(entry.getKey());
        }
    }
}
```

Output:



```
Command Prompt
$java MapIntro.java
FirstName
DOB
Phone number
Gender
Surname
Password
```

Here, getKey() returns the key corresponding to the entry.

Example: getValue() method

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using entrySet() to get the set view of entries
        Set<Entry<String, String>> entrys = myDetails.entrySet();

        // for-each loop access each key from the view
        for(Entry<String, String> entry : entrys) {
            //printing each value
            System.out.println(entry.getValue());
        }
    }
}
```

Output:

```
Command Prompt
$java MapIntro.java
Somanna
20/02/1947
8965474562
Male
Somanna
##90$$
```

Here, getValue() method returns the value corresponding to this entry.

Example: SetValue() method

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //Using entrySet() to get the set view of entries
        Set<Entry<String, String>> entrys = myDetails.entrySet();

        // for-each loop access each key from the view
        for(Entry<String, String> entry : entrys) {
            //setting value of each key to xyz
            String setValue = entry.setValue("xyz");
        }
        System.out.println(myDetails);
    }
}
```

Output:

```
Command Prompt
$java MapIntro.java
{FirstName=xyz, DOB=xyz, Phone number=xyz, Gender=xyz, Surname=xyz, Password=xyz}
```

Here, setValue() method replaces the corresponding to this entry with the specified value.

CORE JAVA - Day 58

Agenda

- **size() method**



size() method: size() method of HashMap class is used to get the size of the map which refers to the number of the key-value pair or mappings in the Map.

Example: size() method in HashMap

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //size() method returns the number of key value pairs
        System.out.println(myDetails.size());
    }
}
```

Output:

ca Command Prompt

```
$java MapIntro.java  
6
```

isEmpty() method: isEmpty() method of HashMap class is used to check for the emptiness of the map. The method returns true if no key-value pair or mapping is present in the map else false.

Example: isEmpty() method

```
import java.util.HashMap;  
  
public class MapIntro {  
  
    public static void main(String[] args) {  
        //Create an empty HashMap  
        HashMap<String, String> myDetails = new HashMap<String, String>();  
  
        System.out.println("Before Adding the elements");  
  
        //Check if HashMap is empty or not  
        System.out.println(myDetails.isEmpty());  
  
        //Adding elements to the HashMap  
        myDetails.put("FirstName", "Somanna");  
        myDetails.put("Surname", "Somanna");  
        myDetails.put("Phone number", "8965474562");  
        myDetails.put("Password", "##90$$");  
        myDetails.put("DOB", "20/02/1947");  
        myDetails.put("Gender", "Male");  
  
        System.out.println("After Adding the elements");  
  
        //Check if HashMap is empty or not  
        System.out.println(myDetails.isEmpty());  
    }  
}
```

Output:

ca Command Prompt

```
$java MapIntro.java  
Before Adding the elements  
true  
After Adding the elements  
false
```

In the above example, we have created a hashmap named myDetails. Here, we have used the isEmpty() method to check whether the hashmap contains any elements or not. Initially, the newly created hashmap does not contain any element. Hence, isEmpty() returns true. However, after we add some elements, the method returns false.

containsKey(key) method: containsKey() method is used to check whether a particular key is being mapped into the HashMap or not. It takes the key element as a parameter and returns True if that element is mapped in the map.

Example: containsKey(key) method

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //checking for key element Surname
        System.out.println(myDetails.containsKey("Surname"));

        //checking for key element Lastname
        System.out.println(myDetails.containsKey("Lastname"));
    }
}
```

Output:

Command Prompt

```
$java MapIntro.java
true
false
```

Here, the Hashmap contains a mapping for the key Surname and doesn't contain a key Lastaname. Hence, the containskey() method returns true first then it will return false.

containsValue(value) method: containsValue() method is used to check whether a particular value is being mapped by a single or more than one key in the HashMap. It takes the Value as a parameter and returns true if that value is mapped by any of the key in the map.

Example: containsValue(value) method

```
import java.util.HashMap;

public class MapIntro {

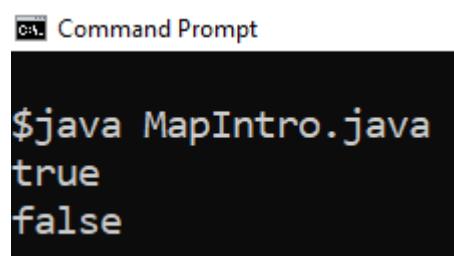
    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //checking for value Somanna
        System.out.println(myDetails.containsValue("Somanna"));

        //checking for value Alex
        System.out.println(myDetails.containsKey("Alex"));
    }
}
```

Output:



```
Command Prompt
$java MapIntro.java
true
false
```

Here, the Hashmap contains a mapping for the value Somanna and doesn't contain a key Alex. Hence, the containsValue() method returns true first then it will return false.

remove(key) method with key: remove() is an inbuilt method of HashMap class and is used to remove the mapping of any particular key from the map. It basically removes the values for any particular key in the Map.

Example: remove() method

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        //removing the mapping with the key
        myDetails.remove("Surname");

        System.out.println();
        System.out.println(myDetails);
    }
}
```

Output:

```
Command Prompt
$java MapIntro.java
{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Surname, Password=##90$$}

{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Password=##90$$}
```

Here, if you observe the output remove() method have removed surname and it's respective value from the map. remove() method not only remove the map it also return the value as shown in below example.

Example:

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //removing the mapping with the key
        String value = myDetails.remove("Surname");

        System.out.println(value);
    }
}
```

Output:

cmd Command Prompt

```
$java MapIntro.java
Somanna
```

Example:

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        //removing the mapping with the key
        String value = myDetails.remove("Lastname");

        System.out.println(value);
    }
}
```

Output:

```
cmd Command Prompt
$java MapIntro.java
null
```

Here, we are getting null as the output because the specified key i.e Lastname is not present in the map so it returns null.

remove(key, value) method with key and value: Removes the entry for the specified key only if it is currently mapped to the specified value.

Example:

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        //removing the mapping with the key and value
        myDetails.remove("Surname", "mg");
        System.out.println();

        System.out.println(myDetails);
    }
}
```

Output:

```
cmd Command Prompt
$java MapIntro.java
{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##90$$}

{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##90$$}
```

Here, remove(key, value) method will remove key and value pair if the specified key value pair present inside the map.

Example:

```
import java.util.HashMap;

public class MapIntro {

    public static void main(String[] args) {
        //Create an empty HashMap
        HashMap<String, String> myDetails = new HashMap<String, String>();

        //Adding elements to the HashMap
        myDetails.put("FirstName", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        //removing the mapping with the key and value
        myDetails.remove("Surname", "Somanna");
        System.out.println();

        System.out.println(myDetails);
    }
}
```

Output:

```
Command Prompt

$java MapIntro.java
{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Surname, Password=##90$$}

{FirstName=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Password=##90$$}
```

Here, remove(key, value) method will remove Surname and Somanna key-value pair in the HashMap.

CORE JAVA - Day 59

Agenda

- **putIfAbsent(key,value)**
- **putAll()**
- **replace(key, value)**
- **replace(key,oldvalue,newvalue)**
- **clear()**



Example 1: If the key is not present inside the HashMap, only then put the entry inside the map.

MapIntro.java

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

public class MapIntro
{
    public static void main(String[] args)
    {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        Scanner s = new Scanner(System.in);
        System.out.println("Enter the key:");
        String key = s.next();
        System.out.println("Enter the value");
        String value = s.next();

        if(myDetails.containsKey(key)==false) {
            myDetails.put(key, value);
        }
        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##90$$}
Enter the key:
Country
Enter the value
India
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Country=India, Gender=Male, Surname=Somanna, Password=##90$$}
```

In the above program `myDetails.containsKey(key) == false` will check whether the key is present inside the HashMap or not, if it is not present only then it will put key value pair inside the HashMap. You can observe from the output also, Country is not present inside the hashmap that is when country=India key-value pair is put into the hashmap. There is an efficient way to check and put the entry inside the HashMap i.e, using `putIfAbsent()` as shown below.

MapIntro.java

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

public class MapIntro
{
    public static void main(String[] args)
    {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        Scanner s = new Scanner(System.in);
        System.out.println("Enter the key:");
        String key = s.next();
        System.out.println("Enter the value");
        String value = s.next();

        myDetails.putIfAbsent(key, value);
        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##90$$}
Enter the key:
Country
Enter the value
India
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Country=India, Gender=Male, Surname=Somanna, Password=##90$$}
```

If you observe the output, you are getting same output when `putIfAbsent()` is used. The **putIfAbsent(key K, value V)** method of `HashMap` is used to map the specified key with the specified value, only if no such key exists in the `HashMap` instance.

Example 2: Write a program where you have to put all the elements present in one hashmap to another hashmap.

MapIntro.java

```
import java.util.HashMap;
import java.util.Set;

public class MapIntro {
    public static void main(String[] args) {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        HashMap<String, String> myData = new HashMap<String, String>();
        myData.put("Email", "Somanna@gmail.com");
        myData.put("Country", "India");
        myData.put("Blood Group", "O+");

        System.out.println(myDetails);

        Set<String> keys = myData.keySet();

        for(String key : keys) {
            String value = myData.get(key);
            myDetails.put(key, value);
        }
        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##9
0$$}
{Firstname=Somanna, Blood Group=O+, Email=Somanna@gmail.com, DOB=20/02/1947, Phone number=8965474562, C
ountry=India, Gender=Male, Surname=Somanna, Password=##90$$}
```

You can see from the above example, to put all the elements from one hashmap to another hashmap first we are using keyset() method using which we are fetching all the keys present inside myDetails HashMap and storing it inside the set. Now using for-each loop value of each key is fetched and put that key value pair inside the myDetails HashMap. There is an efficient way to do the same using putAll() as shown below.

MapIntro.java

```
import java.util.HashMap;

public class MapIntro {
    public static void main(String[] args) {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "#90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        HashMap<String, String> myData = new HashMap<String, String>();
        myData.put("Email", "Somanna@gmail.com");
        myData.put("Country", "India");
        myData.put("Blood Group", "O+");

        System.out.println(myDetails);

        myDetails.putAll(myData);
        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##9
0$$}
{Firstname=Somanna, Blood Group=O+, Email=Somanna@gmail.com, DOB=20/02/1947, Phone number=8965474562, C
ountry=India, Gender=Male, Surname=Somanna, Password=##90$$}
```

As you can see from the output, using putAll() you are getting same expected output. **putAll()** Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

Example 3: Write a Program to replace the value of a specified key

MapIntro.java

```
import java.util.HashMap;

public class MapIntro {
    public static void main(String[] args) {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        myDetails.replace("Password", "abcdef");

        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna
, Password=##90$$}
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna
, Password=abcdef}
```

As you can clearly see from the output, initially the password was ##90\$\$ using replace() we have changed the password to abcdef. **replace(key, value)** will replaces the entry for the specified key only if it is currently mapped to some value.

One more way of replacing the value is if the specified key-value is present then replace the new value with old value, if the specified key-value pair is not present then HashMap remains as it is. Now we will understand this with an example.

MapIntro.java

```
import java.util.HashMap;

public class MapIntro {
    public static void main(String[] args) {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        myDetails.replace("Gender", "Male", "Other");

        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somm
a, Password=##90$$}
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Other, Surname=Somm
a, Password=##90$$}
```

You can see from the above example that, Male i.e old value is replaced with other i.e new value as Gender and Male as a key-value pair is present inside the HashMap. **replace(key, OldValue, NewValue)** will replaces the entry for the specified key only if currently mapped to the specified value.

Example 4: Remove all the elements present inside the HashMap

```
import java.util.HashMap;

public class MapIntro {
    public static void main(String[] args) {
        HashMap<String, String> myDetails = new HashMap<String, String>();
        myDetails.put("Firstname", "Somanna");
        myDetails.put("Surname", "Somanna");
        myDetails.put("Phone number", "8965474562");
        myDetails.put("Password", "##90$$");
        myDetails.put("DOB", "20/02/1947");
        myDetails.put("Gender", "Male");

        System.out.println(myDetails);

        myDetails.clear();

        System.out.println(myDetails);
    }
}
```

Output:

```
$java MapIntro.java
{Firstname=Somanna, DOB=20/02/1947, Phone number=8965474562, Gender=Male, Surname=Somanna, Password=##9
0$$}
{}
```

As you can see from the above example, using `clear()` you can remove all the entry present inside the `HashMap`. `clear()` will remove all of the mappings from this map. The map will be empty after this call returns.

CORE JAVA - Day 60

Agenda

- Hashing Algorithm
- LinkedHashMap



HashMap stores the data in an unordered collections. Now let us understand why HashMap stores the data in an unordered collection with an example.

Example:

MapIntro.java

```
import java.util.HashMap;

public class MapIntro{
    public static void main(String[] args){
        HashMap<String, Integer> names = new HashMap<String, Integer>();
        names.put("anna", 27);
        names.put("bob", 81);
        names.put("elle", 27);
        names.put("otto", 54);
        names.put("arora", 18);
        names.put("ivi", 45);
        names.put("jj", 72);

        System.out.println(names);
    }
}
```

Output:

```
$java MapIntro.java
{jj=72, arora=18, bob=81, otto=54, anna=27, ivi=45, elle=27}
```

As you can see from the above example, the output is not in the same order which is inserted into the HashMap and it is not inserting in some random order it is following **Hashing Algorithm** to insert the value inside the HashMap and **Hashing Algorithm is applied only to the keys not to the values**. Now let's understand the working of Hashing Algorithm using the example.

Whenever you create an object of HashMap, internally HashMap gets created which has an **initial capacity =16** and it will be stored with a value null initially. Once initial capacity is 75% filled then capacity of HashMap doubles every time.

n=16	
0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null
10	null
11	null
12	null
13	null
14	null
15	null

Keys to be inserted inside the HashMap are:

“anna”, “bob”, “elle”, “otto”, “arora”, “ivi”, “jj”

Hashing Algorithm:

hc = hashcode(key)

hash = hc^(hc>>>16)

index =hash & (n-1)

Whenever you try to put the elements inside the HashMap that time Hashing Algorithm is applied.

- When the first key-value pair is inserted inside the HashMap i.e “anna” and 27, hashing algorithm is applied step by step as shown below.

Hashing Algorithm

hc = hashCode(key) = hashCode(“anna”) = 2998944

hash = hc^{^(hc>>>16)} = 2998944^{^(2998944>>>16)} = 2998925

index = hash & (n-1) = 2998925 & (16-1) = 13

First hashCode of “anna” is calculated, **hashCode is present inside the object class and object class is base class for all the classes thus it is inherited in all the class.** In this example, we are trying to insert string value (inside the string class also hashCode method has been inherited and it has been overridden), here it uses formula to calculate the hashCode of given string. Formula to calculate hashCode is

s[0]*31^{^(n-1)}+s[1]*31^{^(n-2)}+.....+s[n-1]

Once hashCode of “anna” is calculated then using the value of hc and calculate the value of hash. Next by substituting the value of hash and n, index value is calculated. After calculating hashing algorithm, index = 13, so in 13th index key-value pair is stored. Now we will see how it will be stored inside the hashMap.

First it will create separate new memory inside which key, value, hash and null will be stored (moving further we will understand why null is inserted over there) the address of that will be stored inside the index 13, now it is referring to that address as shown below.

n=16
0 null
1 null
2 null
3 null
4 null
5 null
6 null
7 null
8 null
9 null
10 null
11 null
12 null
13 7070ff → [2998925 "anna" 27 null]
14 null
15 null

- Next “bob” and 81 is put into the hashMap and hashing Algorithm is calculated as shown below.

Hashing Algorithm

$$hc = \text{hashcode(key)} = \text{hashcode("bob")} = 97717$$

$$\text{hash} = hc^{(hc >>> 16)} = 97717^{(97717 >>> 16)} = 97716$$

$$\text{index} = \text{hash} \& (n-1) = 97716 \& (16-1) = 4$$

Now, in index 4 “bob” and its value is inserted as shown below.

n=16
0 null
1 null
2 null
3 null
4 3000ff → [97716 "bob" 81 null]
5 null
6 null
7 null
8 null
9 null
10 null
11 null
12 null
13 7070ff → [2998925 "anna" 27 null]
14 null
15 null

- Next “elle” and 27 is put into the hashMap and hashing Algorithm is calculated as shown below.

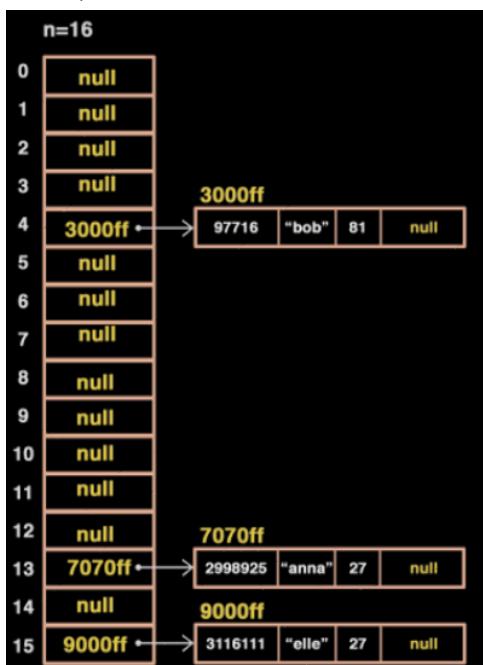
Hashing Algorithm

$hc = \text{hashcode(key)} = \text{hashcode}(\text{"elle"}) = 3116128$

$\text{hash} = hc^{(hc >>> 16)} = 3116128^{(3116128 >>> 16)} = 3116111$

$\text{index} = \text{hash} \& (n-1) = 3116128 \& (16-1) = 15$

Now, in index 15 “bob” and its value is inserted as shown below.



- Next “otto” and 27 is put into the hashMap and hashing Algorithm is calculated as shown below.

Hashing Algorithm

$hc = \text{hashcode(key)} = \text{hashcode}(\text{"otto"}) = 34321984$

$\text{hash} = hc^{(hc >>> 16)} = 34321984^{(34321984 >>> 16)} = 34321972$

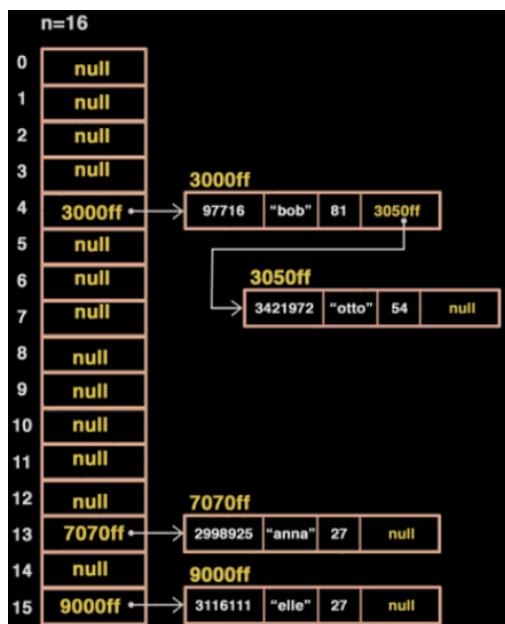
$\text{index} = \text{hash} \& (n-1) = 34321972 \& (16-1) = 4$

Now if you observe you got index value as 4, but in index 4 already “bob” and 81 is present then where will you store the “otto” and 27. This

is the problem which will occur in hashing, the problem is called as **collision**. To overcome this collision there is a collision resolving techniques shown below.

1. Separate chaining(Open Hashing)
2. Open Addressing(closed Hashing)
 - a. Linear probing
 - b. Quadratic probing
 - c. Double Hashing

Among all these collision resolving techniques java makes use of **separate chaining**. Separate chaining internally makes use of Linked list to store the values. First “otto” and 27 is stored in one memory location that address is taken and stored inside the memory of “bob” and 81 where null is present i.e null value present inside “bob” and 81 is now replaced with address of “otto” as shown below.



- Next “arora” and 18 is put into the hashMap and hashing Algorithm is calculated as shown below.

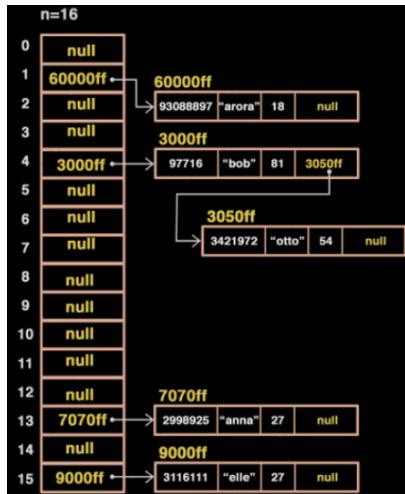
Hashing Algorithm

`hc = hashCode(key) = hashCode("arora") = 93088013`

`hash = hc^(hc>>>16) = 93088013^(93088013>>>16) = 93088897`

`index = hash & (n-1) = 93088897 & (16-1) = 1`

Now, in index 1 “arora” and its value is inserted as shown below.



- Next “ivi” and 45 is put into the hashMap and hashing Algorithm is calculated as shown below.

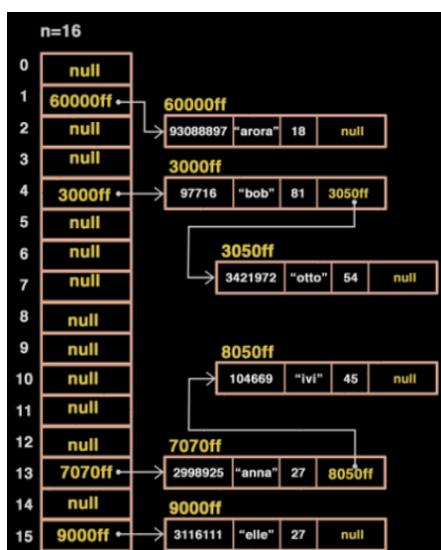
Hashing Algorithm

$$hc = \text{hashcode(key)} = \text{hashcode("ivi")} = 104668$$

$$\text{hash} = hc^{(hc >> 16)} = 104668^{(104668 >> 16)} = 104669$$

$$\text{index} = \text{hash} \& (n-1) = 104669 \& (16-1) = 13$$

Now, in index 13 “ivi” and its value is inserted by making use of linked list because in index 13 already “anna” is present as shown below.



- Next “jj” and 72 is put into the hashMap and hashing Algorithm is calculated as shown below.

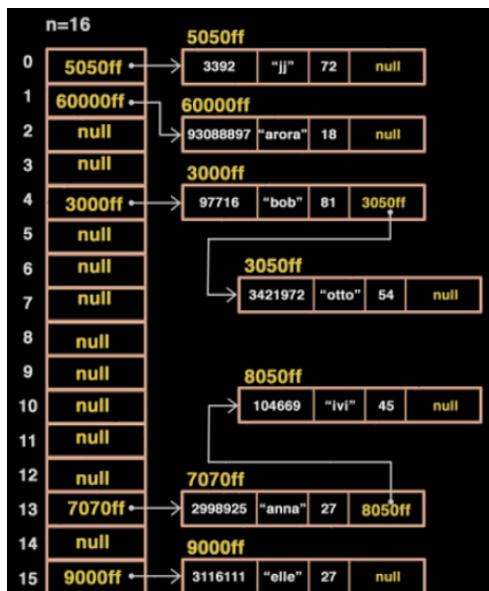
Hashing Algorithm

$hc = \text{hashcode(key)} = \text{hashcode("jj")} = 3392$

$\text{hash} = hc^{(hc >>> 16)} = 3392^{(3392 >>> 16)} = 3392$

$\text{index} = \text{hash} \& (n-1) = 3392 \& (16-1) = 0$

- Now, in index 0 “ivi” and its value is inserted as shown below



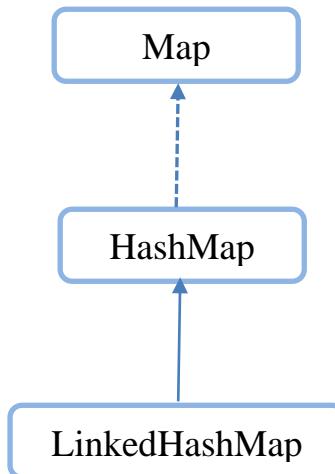
After inserting all the elements inside the hashmap output looks as shown below.

```
$java MapIntro.java
{jj=72, arora=18, bob=81, otto=54, anna=27, ivi=45, elle=27}
```

Now if you observe the value stored internally, the order in which values are stored internally in the same order we are getting the output not in some random order. The advantages of using hashing algorithm is it will be easy to fetch the value present inside the hash map i.e whenever you call `get(key)` it internally applies hashing algorithm such that it will directly go to that particular index and return the value present inside the hashmap.

Now you understood that hashmap doesn't store in the order of insertion but if the user wants the element to be stored in the **insertion order** then you have to make use of **LinkedHashMap** instead of HashMap.

LinkedHashMap inherits HashMap and HashMap implements Map as shown below.



Example:

MapIntro.java

```
import java.util.LinkedHashMap;

public class MapIntro{
    public static void main(String[] args){
        LinkedHashMap<String, Integer> names = new LinkedHashMap<String, Integer>();
        names.put("anna", 27);
        names.put("bob", 81);
        names.put("elle", 27);
        names.put("otto", 54);
        names.put("arora", 18);
        names.put("ivi", 45);
        names.put("jj", 72);

        System.out.println(names);
    }
}
```

Output:

```
$java MapIntro
{anna=27, bob=81, elle=27, otto=54, arora=18, ivi=45, jj=72}
```

As you can see from the output **LinkedHashMap follows insertion order**.

Example-2: To get number of key-value pair present inside the HashMap

MapIntro.java

```
import java.util.LinkedHashMap;

public class MapIntro{
    public static void main(String[] args){
        LinkedHashMap<String, Integer> names = new LinkedHashMap<String, Integer>();
        names.put("anna", 27);
        names.put("bob", 81);
        names.put("elle", 27);
        names.put("otto", 54);
        names.put("arora", 18);
        names.put("ivi", 45);
        names.put("jj", 72);

        System.out.println(names.size());
    }
}
```

Output:

```
$java MapIntro.java
7
```

size() method returns the number of key-value pair present inside the LinkedHashMap.

There are many such methods which we have seen in HashMap which are also available in LinkedHashMap. Why not try by yourself?



File Handling

File handling in Java implies reading from and writing data to a file. The File class from the java.io package, allows us to work with different formats of files. In order to use the File class, you need to create an object of the class and specify the filename or directory name.

- Create an object of file and check whether the file exists or not

```
import java.io.File;

public class Alpha {

    public static void main(String[] args) {

        //      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        //      Create an object of a file
        File file = new File(path);

        //      exists() will return true with the file exists
        //      in that directory else it will return false
        System.out.println(file.exists());
    }

}
```

Now let's see different methods available in File class

- **canRead() and canWrite():**

canRead() method will check whether the file is readable or not. It returns true if it is readable else false.

canWrite() method will check whether the file is writable or not. It returns true if it is readable else false.

```
import java.io.File;

public class Alpha {

    public static void main(String[] args) {

        //      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        //      Create an object of a file
        File file = new File(path);

        //      canRead() will check whether the file is
        //      readable or not
        System.out.println(file.canRead());

        //      canWrite() will check whether the file is
        //      readable or not
        System.out.println(file.canWrite());
    }

}
```

- **getName():**

This method returns the Name of the given file object. The function returns a string object which contains the Name of the given file object.

```
import java.io.File;

public class Alpha {

    public static void main(String[] args) {

        //      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        //      Create an object of a file
        File file = new File(path);

        //      getName() will return the name of the file
        System.out.println(file.getName());
    }

}
```

- **getParent():**

This method returns a String which denotes the pathname string of the parent directory named by this abstract pathname, or null if this pathname does not name a parent.

```
import java.io.File;
```

```
public class Alpha {  
  
    public static void main(String[] args) {  
  
        //      Specify the directory in which file exists  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";  
  
        //      Create an object of a file  
        File file = new File(path);  
  
        //      getName() will return the name of the file  
        System.out.println(file.getParent());  
    }  
  
}
```

Output:

C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles

- **getAbsolutePath():**

The **getAbsolutePath()** method is a part of the file class. This function returns the absolute pathname of the given file object. If the pathname of the file object is absolute then it simply returns the path of the current file object.

```
import java.io.File;  
  
public class Alpha {  
  
    public static void main(String[] args) {  
  
        //      Specify the directory in which file exists
```

```

        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

//      Create an object of a file
File file = new File(path);

System.out.println(file.getAbsolutePath());
}

}

```

Output:

C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt

- **isFile()**

The **isFile()** function is a part of the File class in Java. This function determines whether the is a file or Directory denoted by the abstract filename is File or not. The function returns true if the abstract file path is File else returns false.

```

import java.io.File;

public class Alpha {

    public static void main(String[] args) {

//      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

//      Create an object of a file
        File file = new File(path);

        System.out.println(file.isFile());
    }
}

```

```
    }  
  
}
```

Output:

```
true
```

- **isDirectory()**

This function determines whether the is a file or directory denoted by the abstract filename is Directory or not. The function returns true if the abstract file path is Directory else returns false.

```
import java.io.File;  
  
public class Alpha {  
  
    public static void main(String[] args) {  
  
        //      Specify the directory in which file exists  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";  
  
        //      Create an object of a file  
        File file = new File(path);  
  
        System.out.println(file.isDirectory());  
    }  
}
```

Output:

```
false
```

- **createNewFile():**

The createNewFile() method creates a new and empty file with a specified name. This operation succeeded when the name did not yet exist. Checking for the existence of the file and creation of the file are atomic operations.

```
import java.io.File;
public class Alpha {
    public static void main(String[] args) {

        //      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\aaa.txt";

        //      Create an object of a file
        File file = new File(path);

        file.createNewFile();
    }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  Unhandled exception type IOException
  at Alpha.main(Alpha.java:13)
```

Here you are getting IOException because the createNewFile() will throw IOException so you need to handle this exception using throw or try-catch

```
import java.io.File;
import java.io.IOException;

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

        // Specify the directory in which file exists
        String path =
        "C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\aaa.txt";

        // Create an object of a file
        File file = new File(path);

        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **mkdir():**

The mkdir() function is used to create a new directory denoted by the abstract pathname. The function returns true if the directory is created else returns false.

```
import java.io.File;
public class Alpha {

    public static void main(String[] args) {

        // Specify the directory in which file exists
        String path =
        "C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\aaa.txt";

        // Create an object of a file
    }
}
```

```
File file = new File(path);

System.out.println(file.mkdir());
}

}
```

- **list()**

Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

```
import java.io.File;

public class Alpha {

    public static void main(String[] args) {

        // Specify the directory in which file exists
        String path = "C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles";

        // Create an object of a file
        File file = new File(path);
        String[] list = file.list();
        for (String string : list) {
            System.out.println(string);
        }
    }
}
```

Output:

```
aaa.txt
data.txt
```

- **delete()**

Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.

```
import java.io.File;

public class Alpha {

    public static void main(String[] args) {

        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\aaa.txt";

        // Create an object of a file
        File file = new File(path);
        System.out.println(file.delete());
    }
}
```

Output:

true

File Writer:

Java FileWriter class is used to write character-oriented data to a file. It is a character-oriented class which is used for file handling in java.

- Create an object of file writer
- Call write() and pass string to be written inside the file
- Now call the flush() to push the data to the file from the stream

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        // Create an object of a file
        File file = new File(path);

        FileWriter writer;

        // Create an object of file writer
        try {
            writer = new FileWriter(file);
            // call the write() and pass the string to be written inside
            // the file
            writer.write("Hello World");
            //
            writer.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Now Hello World String is written inside the data.txt file

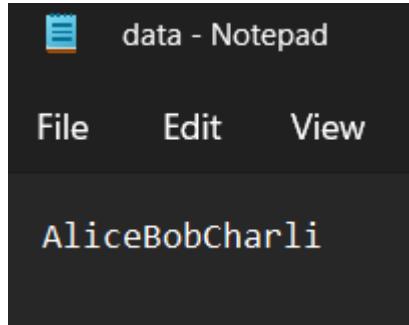
- Write a program to take three words from the user and write that inside the file

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        // Create an object of a file
        File file = new File(path);
        Scanner scan = new Scanner(System.in);
        FileWriter writer;
        // Create an object of file writer
        try {
            int n1 = scan.nextInt();
            int n2 = scan.nextInt();
            int n3 = scan.nextInt();
            writer = new FileWriter(file);
            writer.write(n1);
            writer.write(n2);
            writer.write(n3);
            writer.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

{}

Output:

The user entered data is stored in a data.txt file.

Now if you run the code again the data which is present will be overridden by the new data.

To overcome this problem you need to append the values to the filewriter, to do this you need to pass one more parameter true which will allow you to append

Alpha.java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        // Create an object of a file
        File file = new File(path);
        Scanner scan = new Scanner(System.in);
        FileWriter writer;
```

```
// Create an object of file writer
try {
    String s1 = scan.next();
    String s2 = scan.next();
    String s3 = scan.next();
    writer = new FileWriter(file, true);
    writer.write(s1);
    writer.write(s2);
    writer.write(s3);
    writer.flush();
} catch (IOException e) {
    e.printStackTrace();
}
finally{
    scan.close();
    writer.close();
}
}
```

Output:

```
AliceBobCharliGameOfThrones
```

Now if you observe from the output, the new input is appended to the existing data.

File Reader:

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

It is a character-oriented class which is used for file handling in java.

- Store path of the file in a string
- Create an object of file reader and this will throw an exception of FileNotFoundException if the file is not present in that directory so need to handle that exception
- Call read() which will read the characters present in that file and this method will throw IO Exception need to handle that

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        FileReader reader = null;

        // Create an object of file reader
        try {
            reader = new FileReader(path);
            System.out.println(reader.read());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Output:

65

Here if you observe from the above output, it is giving the ascii value of the first character present inside the file because it reads the character and converts it to integer value. If you want the character instead of integer then you need to do explicit type cast as shown below.

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        FileReader reader = null;

        // Create an object of file reader
        try {
            reader = new FileReader(path);
            System.out.println((char)reader.read());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Output:

A

To read all the string present inside the file you need to create an array using which you can read

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";
        FileReader reader = null;
        char[] ar = new char[15];

        // Create an object of file reader
        try {
            reader = new FileReader(path);
            System.out.println(reader.read(ar));
            System.out.println(ar);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Output:

13

GameOfThrones

Now as you can see from the above example here you need to define the size of the array in which the data present in the file is stored but if you don't know the number of characters present in the file then you cannot read the complete data present inside the file.

Now to read complete data present in the file make use of read() and do this operation repeatedly unless you encounter last character

```
reader = new FileReader(path);
int c = reader.read();
while(c!= -1) {
    System.out.print((char)c);
    c = reader.read();
}
```

Alpha.java

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";
        FileReader reader = null;

        // Create an object of file reader
```

```
try {  
    reader = new FileReader(path);  
    int c = reader.read();  
    while(c!= -1) {  
        System.out.print((char)c);  
        c = reader.read();  
    }  
} catch (FileNotFoundException e) {  
    e.printStackTrace();  
}  
catch(IOException e1) {  
    e1.printStackTrace();  
}  
}
```

Output:

ABCDEFGHIJKLMNOPQRS
INDIA IS MY COUNTRY
JAVA
PYTHON
HTML
SQL

- Now let's create one more file and copy all the data from one file to another file
 - Create an object of filewriter and filereader
 - Read the character present inside one file using read() and write to one more file using write()

```

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";
        String path1 =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\exmp.txt";
        FileReader reader = null;
        FileWriter writer = null;

        // Create an object of file reader and writer
        try {
            reader = new FileReader(path);
            writer = new FileWriter(path1);

            // Read the character from data file and write to exmp file
            int c = reader.read();
            while(c!= -1) {
                writer.write(c);
                c = reader.read();
            }
            writer.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}

```

```
        }  
    }  
}
```

The data is present in the data file to the exmp file, but here if you observe here we are reading each character one by one which is not efficient when there are billions of characters present in the file.

- Now let's see how to read the data from the file using BufferedReader.
BufferedReader will read line by line.
 - Create an object of file reader
 - Convert file reader to bufferedreader by creating an object of buffered reader
 - Call readLine() to read the each line present inside the file

```
import java.io.BufferedReader;  
import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class Alpha {  
    public static void main(String[] args) {  
        // Specify the directory in which file exists  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";  
        FileReader reader = null;  
        BufferedReader reader2 = null;  
        try {  
            reader = new FileReader(path);  
            reader2 = new BufferedReader(reader);  
            String line = reader2.readLine();  
            System.out.println(line);  
            System.out.println(reader2.readLine());  
            System.out.println(reader2.readLine());
```

```
        System.out.println(reader2.readLine());
        System.out.println(reader2.readLine());
        System.out.println(reader2.readLine());
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    catch(IOException e1) {
        e1.printStackTrace();
    }
}
```

Output:

```
ABCDEFGHIJKLMNOPQRS
INDIA IS MY COUNTRY
JAVA
PYTHON
HTML
SQL
```

Here we have called `readLine()` multiple times to read all the lines present inside the file. Instead of that we can make use of a loop and read all lines until the end of the file.

```
import java.io.BufferedReader;
```

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";
        FileReader reader = null;
        BufferedReader reader2 = null;

        try {
            reader = new FileReader(path);
            reader2 = new BufferedReader(reader);
            String line = reader2.readLine();
            while(line != null) {
                System.out.println(line);
                line = reader2.readLine();
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Output:

```
ABCDEFGHIJKLMNPQRS
INDIA IS MY COUNTRY
JAVA
PYTHON
HTML
SQL
```

- Write a program to count number of lines in the file

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class Alpha {
    public static void main(String[] args) {
        // Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";
        FileReader reader = null;
        BufferedReader reader2 = null;

        try {
            reader = new FileReader(path);
            reader2 = new BufferedReader(reader);
            String line = reader2.readLine();
            int count = 0;
            while(line != null) {
                System.out.println(line);
                count++;
                line = reader2.readLine();
            }
            System.out.println(count);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Output:

```
ABCDEFGHIJKLMNPQRS  
INDIA IS MY COUNTRY  
JAVA  
PYTHON  
HTML  
SQL  
6
```

- Write a program to count number of characters in the file

```
import java.io.BufferedReader;  
import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class Alpha {  
    public static void main(String[] args) {  
        // Specify the directory in which file exists  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";  
        FileReader reader = null;  
        BufferedReader reader2 = null;  
  
        try {  
            reader = new FileReader(path);  
            reader2 = new BufferedReader(reader);  
            String line = reader2.readLine();  
            int count = 0;  
            int sum = 0;  
            while(line != null) {  
                count++;  
                int l = line.length();  
                sum += l;  
            }  
            System.out.println("The total number of characters in the file is " + sum);  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        sum +=1;
        line = reader2.readLine();
    }
    System.out.println(count);
    System.out.println(sum);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch(IOException e1) {
    e1.printStackTrace();
}
}

}
```

Scenario 1: Write a program to read the name from the file and phone number from another file, now merge two data and write into file in a format shown below

name : Phone Number

- Create a file to write the **name : Phone Number** combination
- create an object of file reader and buffered reader to read each line from the file
- Run a loop which will help to read all lines until end of the file

```

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Alpha {
    public static void main(String[] args) {
        //      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\name.txt";
        String path1 =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\phone.txt";
        String path2 =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\phone_book.txt";

        //      Creating a reference of FileReader and
        //BufferedReader
        FileReader reader = null;
        BufferedReader reader1 = null;

        FileReader reader2 = null;
        BufferedReader reader3 = null;

        //      Creating a reference of FileWriter
    }
}

```

```
FileWriter writer = null;

try {
    reader = new FileReader(path);
    reader1 = new BufferedReader(reader);
    reader2 = new FileReader(path1);
    reader3 = new BufferedReader(reader2);

    writer = new FileWriter(path2);

    String name = reader1.readLine();
    String phone = reader3.readLine();

    while(name != null && phone != null) {
        writer.write(name + " : " + phone +
"\n");
        name = reader1.readLine();
        phone = reader3.readLine();
    }
    writer.flush();

} catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch(IOException e1) {
    e1.printStackTrace();
}
}
```

Now let's see how you can write into the file using buffered writer

- First Create a path of the file

```
String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";
```

- Create an object of File Writer

```
FileWriter writer = new FileWriter(path);
```

- Create an object of BufferedWriter

```
BufferedWriter bf = new BufferedWriter(writer);
```

- Using bufferedWriter call write() to write the data into the text

```
bf.write("India");
```

Alpha.java

```
import java.io.BufferedReader;  
import java.io.FileNotFoundException;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class Alpha {  
    public static void main(String[] args) {  
        // Specify the directory in which file exists  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";  
  
        // Create a reference of file writer and buffered  
        writer  
        FileWriter writer = null;  
        BufferedWriter bf = null;
```

```
try {
    writer = new FileWriter(path);
    bf = new BufferedWriter(writer);
    bf.write("India");
    bf.flush();

} catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch(IOException e1) {
    e1.printStackTrace();
}
}
```

- Now let's see to take integer value from the user and write into the file

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Alpha {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();

//        Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

//        Create a reference of file writer and buffered
```

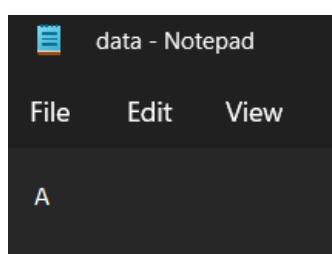
```
writer
    FileWriter writer = null;
    BufferedWriter bf = null;

    try {
        writer = new FileWriter(path);
        bf = new BufferedWriter(writer);
        bf.write(n);
        bf.flush();

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    catch(IOException e1) {
        e1.printStackTrace();
    }
}

}
```

Output:



As you can see from the above output, the user has given 65 but in the file ascii character of the number is stored, it is because write() inside the buffered writer will write the character into the file.

If you try to read float value from the user and try to write into the file using buffered writer and file writer you will get an error. It is because it can read int, String and character array.

To overcome this you can make use of PrintWriter()

- Create an object of PrintWriter and call print() to write the value into the file

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class Alpha {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        float f = scan.nextFloat();

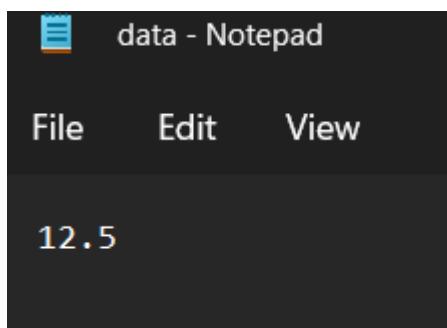
        //      Specify the directory in which file exists
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";

        //      Create a reference of file writer and buffered
        writer
        PrintWriter writer = null;

        try {
            writer = new PrintWriter(path);
            writer.print(f);
            writer.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch(IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

```
    }  
  
}
```

Output:



As you can see from the above output, successfully float value is written inside the file.

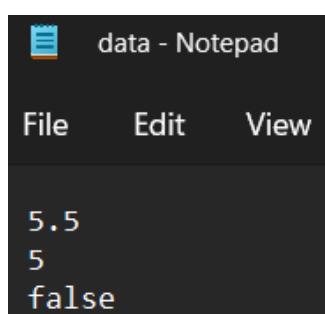
Using PrintWriter you can write all different types of values inside the file by calling print().

- Now let's read integer, float, boolean value from the user and write into the file

```
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.io.PrintWriter;  
import java.util.Scanner;  
  
public class Alpha {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        float f = scan.nextFloat();  
        int n = scan.nextInt();  
        boolean b = scan.nextBoolean();  
  
        // Specify the directory in which file exists  
        String path =
```

```
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\data.txt";\n\n//      Create a reference of file writer and buffered\nwriter\n    PrintWriter writer = null;\n    try {\n        writer = new PrintWriter(path);\n        writer.println(f);\n        writer.println(n);\n        writer.println(b);\n        writer.flush();\n    } catch (FileNotFoundException e) {\n        e.printStackTrace();\n    }\n    catch(IOException e1) {\n        e1.printStackTrace();\n    }\n}\n}\n}
```

Output:



Serialization and Deserialization

Serialization in Java is a mechanism of writing the state of an object into a byte-stream.

The reverse operation of serialization is called *deserialization* where a byte-stream is converted into an object.

Now let us understand seialization with an example:

1. Let us create a class with name, crn number and balance as an attribute whose object needs to be byte-stream.

Customer.java

```
class Customer{  
    private String name;  
    private long crn;  
    private float balance;  
    public Customer(String name, long crn, float  
balance) {  
        super();  
        this.name = name;  
        this.crn = crn;  
        this.balance = balance;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public long getCrn() {  
        return crn;  
    }  
    public void setCrn(long crn) {  
        this.crn = crn;  
    }  
}
```

```

    }
    public float getBalance() {
        return balance;
    }
    public void setBalance(float balance) {
        this.balance = balance;
    }
}

```

Above is the class with name, crn number, balance as an instance variable, we have created constructor, setters and getters.

- Now let's create a main method and perform following operations
- Create an object of Customer class
- Store the path of file inside String variable
- To write the object into the file need to Create an object of FileOutputStream and ObjectOutputStream
- Call writeObject() to write the object into the file

```

public class Alpha {
    public static void main(String[] args) {
        Customer customer = new Customer("Alex",
12345678, 2000);
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\object.txt";

        FileOutputStream fos = new
FileOutputStream(path);
        ObjectOutputStream oos = new
ObjectOutputStream(fos);
        oos.writeObject(customer);
        oos.flush();
    }
}

```

{

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:  
      Unhandled exception type FileNotFoundException  
      Unhandled exception type IOException  
      Unhandled exception type IOException  
      Unhandled exception type IOException  
  
        at Alpha.main(Alpha.java:41)
```

As you can see from the above output you are getting **FileNotFoundException**, **IOException**. Now let's handle these exception using try-catch

```
public class Alpha {  
    public static void main(String[] args) {  
        Customer customer = new Customer("Alex",  
12345678, 20000);  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\object.txt";  
  
        try {  
            FileOutputStream fos = new  
FileOutputStream(path);  
            ObjectOutputStream oos = new  
ObjectOutputStream(fos);  
            oos.writeObject(customer);  
            oos.flush();  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        }  
        catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
java.io.NotSerializableException: Customer
    at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1197)
    at java.base/java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:354)
    at Alpha.main(Alpha.java:46)
```

After handling **FileNotFoundException**, **IOException**, you are still getting the exception which is **NotSerializableException**. It is because whenever you want to write object to the byte-stream that time the **class whose object need to created should implement the Serializable interface**

Now let's **implement serializable interface**

```
class Customer implements Serializable{
    private String name;
    private long crn;
    private float balance;
    public Customer(String name, long crn, float
balance) {
        super();
        this.name = name;
        this.crn = crn;
        this.balance = balance;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getCrn() {
        return crn;
    }
    public void setCrn(long crn) {
        this.crn = crn;
    }
    public float getBalance() {
```

```
        return balance;
    }
    public void setBalance(float balance) {
        this.balance = balance;
    }

}
```

Now we have successfully written the object into byte-stream using serialization.

Deserialization

To read the object from the byte stream we need to do deserialization

Steps to do deserialization

- Create an object of FileInputStream
- Create an object ObjectInputStream
- Call readObject() to read the object from byte stream, this will return the object you need to type cast to the type of class and store it in Customer object
- Using the object reference print name, crn number and balance

```
public class Alpha {
    public static void main(String[] args) {
        String path =
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\object.txt";
        FileInputStream fis = new
FileInputStream(path);
        ObjectInputStream ois = new
ObjectInputStream(fis);
        Customer customer = ois.readObject();
        System.out.println(customer.getName());
        System.out.println(customer.getCrn());
        System.out.println(customer.getBalance());
```

```
    }  
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
  Type mismatch: cannot convert from Object to Customer  
  
  at Alpha.main(Alpha.java:43)
```

Here you have got an exception because you are trying to store the object into the variable of type Customer. Now we need to typecast object to Customer using explicit type casting as shown below

```
Customer customer = (Customer)ois.readObject();
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:  
  Unhandled exception type FileNotFoundException  
  Unhandled exception type IOException  
  Unhandled exception type IOException  
  Unhandled exception type ClassNotFoundException  
  
  at Alpha.main(Alpha.java:41)
```

As you can see from the above output you have got FileNotFoundException, IOException, ClassNotFoundException now to resolve we need to handle these exceptions using try-catch as shown below

```
public class Alpha {  
    public static void main(String[] args) {  
        String path =  
"C:\\\\Users\\\\Megha\\\\Desktop\\\\MyFiles\\\\object.txt";  
        try {  
            FileInputStream fis = new  
FileInputStream(path);
```

```
        ObjectInputStream ois = new  
ObjectInputStream(fis);  
        Customer customer =  
(Customer)ois.readObject();  
        System.out.println(customer.getName());  
        System.out.println(customer.getCrn());  
        System.out.println(customer.getBalance());  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
    catch (ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
}
```

Output:

```
Alex  
12345678  
20000.0
```

As you can see from the above output we have successfully read the data from the file

Serialversionuid:

SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialisation process) is loaded.

At the time of serialization, with every object sender side JVM will save a **Unique Identifier**. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system.

Deserialization at the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise, we will get a Runtime Exception saying InvalidClassException. This unique Identifier is nothing but **SerialVersionUID**.

A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

Syntax:

```
private static final long serialVersionUID=10l;
```

Like this you can avoid InvalidClassException