What is java?

Java is a programming language and a platform from oracle corporation. It’s a high-level language having following features:

Features of Java

1) Simple

Simple syntax. No pointers, no multiple inheritance with the classes which causes ambiguity error. For almost every task API (Application Programming Interface) is available; Programmer just need to know how to use that API.

2) Object Oriented

Java is strong object oriented as it does not allow features like global data, friend function which are against OOP principles.

3) Automatic Garbage Collection:

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed.
Garbage collection cannot be forced in java. U can just make a request for Garbage Collection, by invoking a method "System.gc()". or "Runtime.getRuntime().gc()".

4) Robust

Robust means strong. Java puts a lot of emphasis on early checking for possible errors, as Java compilers are able to detect many problems that would first show up during execution time in other languages.
It provides the powerful exception handling and type checking mechanism as compare to other programming languages.

5) Platform Independent

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be compile-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine. Any machine with Java Runtime Environment can run Java Programs

6) Secure

If a bytecode contains any virus or malicious code, JVM will not execute it. This feature saves your system especially when u download java code and try to execute.

7) Multi-Threading

Java multithreading feature makes it possible to write program that can do many tasks simultaneously.

8) Portable

Java Byte code can be carried to any platform.

9) Architectural Neutral

No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

10) High Performance

Java enables high performance with the use of Just-In-Time (JIT) compiler.

Local Variable: A variable that is declared inside the method is called local variable.

Instance Variable: A variable that is declared inside the class but outside the method is called instance variable. It is not declared as static.

Static variable: A variable that is declared as static is called static variable. It cannot be local.

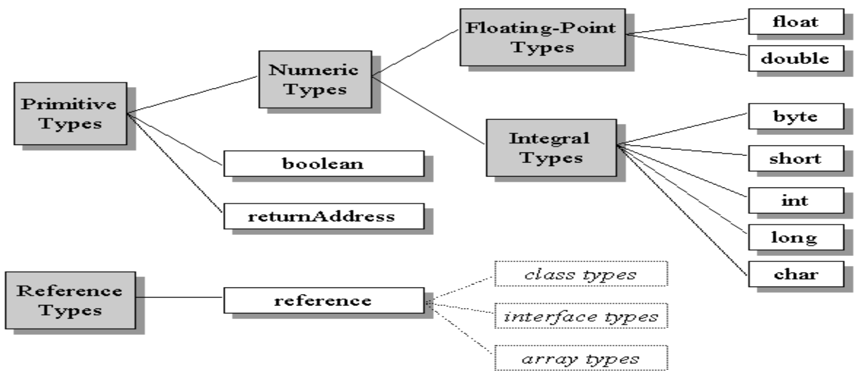
Why char datatype takes 2 bytes in java?

Since Java is designed to support Internationalization (I18N), it makes sense that it would use Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets such as Latin, Greek, Arabic, Russia and many more. Developer will convert this application in such a way that clients from different countries can see the labels in their languages.

For this purpose, it requires 16 bits. Thus, char in java is 16-bit type. The range of char is 0 to 65535. There are no negative chars.

Variable is name of reserved area allocated in memory.

Data types in java



Note: java is platform independent. Compile once run anywhere [on any platform]
Every platform has its own jdk.

Ascii values

0-9	= 48-57
A-Z	= 65 – 90
a-z	= 97-122

JDK, JRE, JVM and API

JDK – java development kit in which there is JRE. JRE (java runtime environment) is a combination of JVM (java virtual machine) and API (application programming interface).

JRE is an acronym for *Java Runtime Environment*. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

JVM - A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine1 (Java VM). JVM that provides runtime environment in which java bytecode can be executed.

JVMs are available for many platforms. hardware and software

JVM, JRE and JDK are platform dependent because configuration of each OS differs. But Java is platform independent.

JVM perform following 3 tasks:

1. Load code
2. Verifies code
3. Execute code.

API - Java Application Programming Interface

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as packages.
API mainly consists of jar and dll files.

Internationalization: is the process of designing an application so that it can be adapted to **various** languages and regions without engineering changes. Sometimes the term **internationalization** is abbreviated as **i18n**, because there are 18 letters between the first "i" and the last "n."

what is "this"?
"this" is a reference which refers to the current or invoking object.
What is the importance of "this"?
when u create multiple objects, u have those many copies of instance members however u have only one copy of member function/s. In that case how will member function keep a track of invoking object.
member function/s will come to know about invoking or current object through "this" reference.

Why we used non-static block?
Syntax:
```
            {
            }
```
NON-STATIC BLOCK: is used for - if we have many constructors inside a class and those constructors need to have some common statements.
instead of repeated those statements in each constructor, we place those statements in non-static block.
This block is called that many times as many times constructor gets created.
e.g., counter which is incremented in each constructor, to keep a track of number of objects created.

static block

syntax

```
                    static
                    {
                    }
```

a) static block is used to access static variables.
b) static block is executed implicitly as soon as class gets loaded.
c) u can define more than one static blocks inside class definition. They will be executed in the order in which they defined.
d) it’s a one-time activity. We use it when we want to perform a task just once.

static block vs static member function
static member function needs to be called explicitly whereas static blocks are called implicitly, as soon as class gets loaded.

Null Pointer Exception
When a reference contains null and if u invoke a method on it, u get Null Pointer Exception.
solution: - before invoking method on any reference just check it does not contain null.

static member
static members are allocated memory as soon as class gets loaded in the memory. They are not associated with the instance.
Since they are not associated with instance, they are having only one copy in the memory, irrespective of no of instances created.
They can be accessed by class name. They are also called as "class variables".
static member function is used to access private static member.
Static members can be share by all objects
e.g.
Account object
non-static members - accid, name, balance
static member - rateofinterest.

Can static member function access non-static data?
No. because static member function can be called without creating object. When object is not created, non-static member is not allocated memory.

Can non-static member function access static data?
Yes. Because in order to invoke non-static member function u need to create object and by that time static members are already allocated memory

what happens when object is created.
a) memory is allocated for instance member/s.
b) constructor is called, which initialises instance member/s.

constructor: It is a special member function.
Special because
a) it is used to initialize (construct) the instance member/s.
b) it has got the same name as of class
c) it does not have a return type.

OOPS CONCEPT
Object-oriented programming (OOP) refers to software design in which programmers define not only the data for a data structure, but also the types of operations (functions) that can be applied to the data.
It has 4 pillars:
1. abstraction
2. encapsulation
3. polymorphism
4. inheritance

Object-Oriented Programming System (OOPs) is a programming concept that works on the principles of abstraction, encapsulation, inheritance, and polymorphism. ... The basic concept of OOPs is to create objects, re-use them throughout the program, and manipulate these objects to get results

The advantages of OOP are as following:

1. The complexity is reduced and the program structure is very clear.
2. Object oriented programming can easily extensible. New features or changes in operating environment can be easily done.
3. Objects can be maintained separately, making locating and fixing problems easier.
4. Objects can be reused in different programs.
5. It implements real life scenario.

ABSTRACTION

Abstraction is the process of extracting the relevant properties of an object that are relative to the perspective of the viewer, while ignoring nonessential details.

1) For E.g. **Yahoo Mail...**
Layman's perspective:
provide the username and password
click on submit button.
select Compose, Inbox, Outbox, Sent mails so on.
developer's perspective:
look and feel of web pages……
how to validate user name and password…
which server it should connect to ….
ensuring successful mail sent as well as receipt.

2) For E.g. **ATM**
User Perspective:
swipe your card to withdraw money,
some instructions to be followed displayed on the screen,
provide some info like pin code of your account,
withdraw amount
get the balance
change the pin
get the statement
Application Developer perspective:
how queries are being generated based on user's info in ATM machine,
which database these queries are going to, Oracle or MySQL etc.
where that database is located
which language has been used for these implementations i.e., java or c# or c++,

ENCAPSULATION

Encapsulation is the process of separating the aspects of an object into external and internal aspects. The external aspects of an object need to be visible or known, to other objects in the system. The internal aspects are details that should not affect other parts of the system. Hiding the internal aspects of an object means that they can be changed without affecting external aspects of the system.

EX. ATM
User: EXTERNAL ASPECTS
Insert card
Withdraw money
get the balance
change the pin
get the statement
INTERNAL ASPECTS:
Algorithms connect to the database
Check sufficient balance
Debit the balance.

Main purpose of encapsulation is External aspects remain same even though Internal aspects change.

EX. CAR

Abstraction
From the driver's perspective, acceleration and braking actions which can be performed on the car.

Encapsulation
When the driver presses the accelerator pedal (external aspect), there are a lot of things that happen within the car which actually cause the rpm (rotations per minute of the wheel) to increase (Internal aspect). Is the driver concerned about what actually happens within the engine? No. The driver just wants the car to accelerate on pressing the pedal (external aspect) and is least bothered about the underlying mechanisms used by the manufacturers to achieve this. He doesn't care about how the engine is designed or as to how the piston is moving to achieve acceleration (internal aspect). All he knows (and wants to know generally) is that the car should accelerate when he presses the pedal.

Inheritance
The ability for a new class to be created from an existing class by extending its properties is known as inheritance. It provides reusability.
Basically, you go for inheritance when u realize that "new type is same as existing type".
Java allows only 3 types of inheritance
  a) single level
  b) multi-level
  c) Hierarchical

Inheritance is inbuilt in java i.e. if your class is not derived from any base class, "java.lang.Object" is the base class of it.

Reusability

means using existing type while defining a new type. It can be achieved in two ways:
a) composition/aggregation [has-a relationship]
b) inheritance [is-a relationship]

composition/aggregation: you go for composition/aggregation when you want to use some of the functionalities of existing type inside new type.
Ex. while designing "Car" you would reuse "Engine" by composition/aggregation, because "Car" is not an "Engine" it just needs some functionalities of Engine.

Inheritance: you go for inheritance when you realize that new type is "same as" existing type.
Ex. while designing "Car" you would reuse "Four-wheeler" because Car is same as Four-wheeler.

Finalize method:

What happen when main block overs?
a) scope of m1 gets over
b) m1 gets removed
c) object to which m1 was referring becomes unreferred so that object is going to be garbage collected.
d) just before the object gets garbage collected, "finalize" method gets called.

finalize method can be defined in order to release resources such as File, Database connection or Socket.

Since there is no guarantee as to when exactly object gets garbage collected, we cannot completely rely upon "finalize" method for releasing resources. We have to have some alternate mechanism in order to release resources.

What is class "Class" in java?
In java whenever any class gets load it is represented by instance of class "Class".
This instance holds information about loaded class such as member variables, methods, constructors etc.

`getClass().getName();` // used to get a name of a class

SingletonDemo.java

Singleton class

1. only one object created, that too inside the class.
2. clients are not allowed to create object- private constructor.
3. non-static methods, client can call.
4. static method which will return the reference of the only object which u have created inside the class.

package

1. package is a collection of related classes and interfaces.
2. package is mainly used to avoid name conflicts.
3. package concept is similar to "namespace" concept of C++.
4. java has so many in-built packages. e.g.
    - java.awt
    - java.io
    - java.util
    - java.lang

by def. java.lang is available to all the java applications.
in order to use package, you need to use the keyword "import".
when u say "import", nothing is physically included (unlike #include of c and c++). It is only for compiler. Runtime performance is not at all affected with "import" statements.

Class Loaders:

There are 3 types of class loaders in java

1. Bootstrap Class Loader or Primordial Class Loader
2. Extension Class Loader
3. System Class Loader

Bootstrap or Primordial class loader: -
Bootstrap class loader loads those classes which are essential for JVM to function properly. Bootstrap class loader is responsible for loading all core java classes for instance java.lang.* ,java.io.* etc. Bootstrap class loader finds these necessary classes from “jdk/jre/lib/rt.jar”

Extension class loader: -
Extensions Class loader is the immediate child of Bootstrap class loader. This class loader loads the classes in lib\ext. directory of the JRE.

System class loader: -
System-Class path class loader is the immediate child of Extensions class loader. It loads the classes and jars specified by the CLASSPATH environment variable, java.class. path system property, -cp or -class path command line settings. If any of the jars specified in one of the above manners have a MANIFEST.MF file with a Class-Path attribute, the jars specified by the Class-Path attribute are also loaded. This class loader is also used to load an application’s entry point class that is the “static void main()” method in a class.

public static void main(String args[])

public: It is an access specifier. We should use a public keyword before the main() method so that JVM can identify the execution point of the program. If we use private, protected, and default before the main() method, it will not be visible to JVM.

static: You can make a method static by using the keyword static. We should call the main() method without creating an object. Static methods are the method which invokes without creating the objects, so we do not need any object to call the main() method.

void: In Java, every method has the return type. Void keyword acknowledges the compiler that main() method does not return any value.

main(): It is a default signature which is predefined in the JVM. It is called by JVM to execute a program line by line and end the execution after completion of this method. We can also overload the main() method.

String args[]: The main() method also accepts some data from the user. It accepts a group of strings, which is called a string array. It is used to hold the command line arguments in the form of string values.

System.out.println("hello"); - SOP

a) "hello" needs to be displayed

b) u need to invoke "print()" or "println()" methods
println("hello");

c) both the methods are non-static in "PrintStream" class, so you need reference of "PrintStream" class

d) "out" is a reference of "PrintStream"
out.println("hello")

e) "out" is also a static member of "System" class which can be accessed as "classname.member" i.e.
System.out.println("hello");

SEQUENCE of things get called in java program

1. Program gets compile into "classname.class". means class gets loaded.
2. Loaded class is represented by instance of class 'Class'. That instance is created.
3. Static block gets called.
4. Main method is invoked.
5. Memory allocated for instance members.
6. Constructor gets called which initialises instance member (if any class have non-static block, then this block 1st called then constructor gets called)

Named object vs Anonymous object.

- Named object once created it will within memory till the main block
- Whereas anonymous object is once created then it will be used only once.
  As control will go to next command it will be already gone.
- Both has its own prone and cons. We use both as per our requirement.

Accessibility modifiers:

Accessibility modifiers	Same class	Sub class in same package	Different class in same package	Sub class in different package	Different class in different package
Private	Yes	No	No	No	no
<default>	Yes	Yes	Yes	No	no
Protected	Yes	Yes	Yes	Yes	no
Public	Yes	Yes	Yes	Yes	yes

JVM Architecture:

1. **Class loader sub system**
   - Loading: loading of class
   - Linking:
     1. Verifying: verify for security
     2. Preparing: initialize static variables with default values.
     3. Resolving: checked resolution if any
   - initializing: initializing static variables with the value given by programmer.

2. **Runtime data areas**

- Method areas: class bytecode, class loader, static variables, various literals, class “Class”.
- Heap: all instances
- Java stacks: local members, references
- Pc registers: multithreads
- Native method stack: when program calls C, C++ methods

3. **Execution Engine**
   Converts byte code into native code of that machine.
   - Interpreter
   - JIT compiler
   - Hotspot profiler: used to sort code for compilers.
   - gc()

Lazy and Eager resolutions:
- An eager algorithm executes immediately and returns a result.
- A lazy algorithm will not execute until it is not necessary to execute and then produces a result. Eager and lazy algorithms both have their own pros and cons.
- in case of static, early resolution
- in case of non-static, lazy resolution

Overriding
we do override to provide specific version to base class method.
Rules:
1) arguments must be same otherwise it becomes "overloading".
2) return type of overriding can be co-variant.
3) overriding method must be having same or more accessibility as compare to overridden method.
4) overriding and checked-exception rule:
   a) overriding method may not declare any checked exception.
   b) overriding method can declare same checked exception or its sub-type declared by overridden method.
   c) overriding method cannot declare checked exception not declared by overridden method.

what is Co-variant return type in case of overriding?
if overridden method has "parent" as a return type then overriding method can have "child" as a return type.

Binding
Resolving function call with function body is called binding.

how many types of Binding are there?
1. early binding / static binding
   - early binding means resolving function call with function body at the compilation time.

2. late binding / dynamic binding

- late binding means resolving function call with function body at the runtime.
By default, it’s always a late binding.
When type of the object is determined at run-time, it is known as dynamic binding.

Abstract class
When to use abstract class in java?
- while designing Parent class, if u realize that there is some functionality compulsorily required in child classes but Parent class is not able to define it.
- This functionality is a contract or abstract function. Since abstract function cannot be declared inside non-abstract class, u have to make class as abstract.
- abstract class cannot be instantiated. because abstract class is incomplete i.e., it has at least one contract [abstract method]
- in java as soon as u define a class with "abstract" keyword, class becomes abstract.

- abstract class cannot be instantiated.
- abstract class can contain abstract as well as non-abstract methods.
- abstract method is a method which is declared with "abstract" keyword. (it cannot be private)
- a child class of an abstract class has to provide implementation to the method which is declared "abstract" in parent class or else make child class also "abstract".
- a class cannot be "abstract" and "final" both.

can abstract class have a constructor?

-yes, it will be invoked from sub class constructor when sub class gets instantiated.
The constructor inside the abstract class can only be called during constructor chaining i.e., when we create an instance of sub-classes. This is also one of the reasons abstract class can have a constructor.

What is the role played by Compiler in case of late binding?

Suppose we have,

```
base ref=new sub1();  // upcasting
ref.disp();  // late binding
```

- when compiler encounters "ref.disp()" ,
- for the compiler "ref" is of type "base".
- So, compiler searches "disp()" inside "base".
- if it's there, accessible and non-final, compiler writes an instruction for runtime to follow.
- That instruction is " during runtime check the content of "ref" and invoke "disp()" accordingly.

Why interface?

- If we want to extend 2 classes for implementation in one class then it is not possible by using abstract class, - we need interface.
- It contains only contract (abstract class) classes.
- By default, all methods in interface are **public and abstract.**
- When we implement parent class in child class then we need to give public accessibility to child methods else we get weaker method exception.
- interfaces are abstract in nature. i.e., they cannot be instantiated.
- variable which is declared in interface is by default **"public", "static" and "final".**
- a class can be derived from one or more interfaces. ("implements" keyword)
- child class has to define all the methods of parent interface/s otherwise u need to make child class as "abstract".
- one interface can be derived from one or more other interfaces. ("extends" keyword)
- if a class is derived from some other class and some interfaces, "extends" keyword should precede "implements".

Do we have constructor in interface?

No, you cannot have a constructor within an interface in Java. You can have only public, static, final variables and, public, abstract methods as of Java7.

When to use abstract and when to use interface?

Interface is used when you want to define a contract and you don't know anything about implementation. (Here it is total abstraction as you don't know anything.)

Abstract class is used when you know something and rely on others for what you don't know. (here it is partial abstraction as some of the things you know and some you don't know.)

What is aggregation?

Aggregation is a specialize form of Association where all object has their own lifecycle but there is ownership and child object cannot belong to another parent object. Let’s take an example of Department and teacher. A single teacher cannot belong to multiple departments, but if we delete the department teacher object will not destroy. We can think about “has-a” relationship.

What is composition?

Composition is again specializing form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. Child object does not have their lifecycle and if parent object deletes all child object will also be deleted. Let’s take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room cannot belong to two different housespoly if we delete the house room will automatically delete. Let’s take another example relationship between Questions and options. Single questions can have multiple options and option cannot belong to multiple questions. If we delete questions options will automatically delete.

Polymorphism

Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So, polymorphism means many forms.
There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overriding method is resolved at runtime rather than compile-time.
In this process, an overriding method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.(upcasting)

example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.
Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Polymorphism is the ability of an object to take on many forms.
e.g.

person behaves like student at college, son at home, customer in the mall.

Compile Time Polymorphism: Whenever an object is bound with their functionality at the compile-time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures. So this is called compile-time polymorphism or static or early binding. Compile-time polymorphism is achieved through method overloading. Method Overloading says you can have more than one function with the same name in one class having a different prototype. Function overloading is one of the ways to achieve polymorphism but it depends on technology that which type of polymorphism we adopt. In java, we achieve function overloading at compile-Time.

what is "instanceof" operator?

instanceof operator checks is-a relationship. It returns true or false.

Virtual function in java

By default, all the instance methods in Java are considered as the Virtual function except final, static, and private methods as these methods can be used to achieve polymorphism.

Association

Association refers to the relationship between multiple objects. It refers to how objects are related to each other and how they are using each other's functionality. Composition and aggregation are two types of association.

Composition

The composition is the strong type of association. An association is said to composition if an Object owns another object and another object cannot exist without the owner object. Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human.

Aggregation

Aggregation is a weak association. An association is said to be aggregation if both Objects can exist independently. For example, a Team object and a Player object. The team contains multiple players but a player can exist without a team.

Association	**Aggregation**	**Composition**
Association relationship is represented using an arrow.	Aggregation relationship is represented by a straight line with an empty diamond at one end.	The composition relationship is represented by a straight line with a black diamond at one end.
In UML, it can exist between two or more classes.	It is a part of the association relationship.	It is a part of the aggregation relationship.
It incorporates one-to-one, one-to-many, many-to-one, and many-to-many association between the classes.	It exhibits a kind of weak relationship.	It exhibits a strong type of relationship.
It can associate one more objects together.	In an aggregation relationship, the associated objects exist independently within the scope of the system.	In a composition relationship, the associated objects cannot exist independently within the scope of the system.
In this, objects are linked together.	In this, the linked objects are independent of each other.	Here the linked objects are dependent on each other.
It may or may not affect the other associated element if one element is deleted.	Deleting one element in the aggregation relationship does not affect other associated elements.	It affects the other element if one of its associated element is deleted.
Example: A tutor can associate with multiple students, or one student can associate with multiple teachers.	Example: A car needs a wheel for its proper functioning, but it may not require the same wheel. It may function with another wheel as well.	Example: If a file is placed in a folder and that is folder is deleted. The file residing inside that folder will also get deleted at the time of folder deletion.

###
Favour composition over inheritance is a one of the popular object-oriented design principles, which helps to create flexible and maintainable code in object-oriented languages.

Inheritance drawbacks:

Tight coupling- if base class (Four-wheeler) is changed, sub class (Car) will break.

inheritance breaks encapsulation.

white-box reuse. That is, with inheritance, the parent class implementation is often visible to the subclasses.

Composition Advantages:

black-box reuse as it does not break encapsulation. “Car” knows only an interface “Engine”. It doesn’t know the implementation.

Loose coupling, program to interface. During runtime any implementations (such as “HondaEngine” or “MarutiEngine” or “BMWEngine” can be passed to “Engine ref” and “on()” method can be invoked polymorphically.
##

Program to interface gives loose coupling (car – engine example)

Here we use ref of parent and call child at runtime. And it’s a Black box reuse

Where program to implementation gives tight coupling

Here we directly made object of parent in side child class which is tight coupling. And it’s a white box.

Why do we use interface?

is used to achieve total abstraction. Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance. It is also used to achieve loose coupling.

Why we do override?

The benefit of overriding is: ability to define a behaviour that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement. In object-oriented terms, overriding means to override the functionality of an existing method.

Why we do upcasting?

Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. This type of initialization is used to access only the members present in the parent class and the methods which are overridden in the child class. This is because the parent class is upcasted to the child class.

what is down casting?
conversion from base type to derived type is called as "down casting".

When to use down casting?
when base reference referring to child object (upcasting) and u want to invoke a method of child class (on the same object) which is not there in the base class, you need to use down casting.

Why we used static keyword to the perform() method?
As we know as soon as class get loaded static blocks of that class gets loaded.
But if we haven’t defined this method as static then we have to be made object of that class and through object only we can call this non static perform method. So unnecessary instance will be made for perform.
But if we do this method static then it already gets memory inside heap so, no need to do any unnecessary invocation of this method.

Class VS Interface

Class	Interface
A class describes the attributes and behaviours of an object.	An interface contains behaviours that a class implements.
A class may contain abstract methods, concrete methods.	An interface contains only abstract methods.
Members of a class can be public, private, protected or default.	All the members of the interface are public by default.

Abstract vs interface

1. An abstract class allows you to create functionality that subclasses can implement or override.
   An interface only allows you to define functionality, not implement it.

2. And whereas a class can extend only one abstract class,
   interface can take advantage of multiple inheritance

Early Binding / static binding

Early binding is happened when method is static or private or final.
In case of static method, static method is not associated with objects. that's why early binding happens.
In case of final method, this method is not override so, no any other option for compiler. That's why early binding.
In case of private, if your parent method is private so, it can be access only in his class, so early binding.
When type of the object is determined at compiled time(by the compiler), it is known as static binding.

Object Class Methods

public String toString();

returns the String representation of an instance.

This method returns o/p like myclass@12135153 which is not readable. So, to make it readable we need to override it.

public boolean equals(Object);

checks the equality of two references. If they are referring to same instance then they are equal otherwise not.

If client wants to check whether to references which are referring to the objects have the same value or not then we need to override this method.

```
public boolean equals(Object ref)

{

Return this.num=(MyNum)ref.num;

}
```

public int hashCode();

every object is given a unique number inside memory. This number is called as hashcode. This method returns the hashcode of caller object.

When we need to show hashcodes of two object same base on content of objects then we need to override the hashCode() method

```
Public int hasCode()

{

Return num;

}
```

When we override the equals() method then we have to override hashCode() method to be in contract rules.

Equals() and Hashcode() contract

1. Whenever Hashcode() method is invoked on the same object more than once during an execution of a Java application, the hash Code method must consistently return the same integer. This integer need not remain consistent from one execution of an application to another execution of the same application.

2. If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

3. It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Intern();

String s1=new String("hello");

String s2=s1.intern();

1) String class instance is created inside heap (not inside pool).
2) "s1" reference is created on the stack.
3) "s1" refers to the instance created inside heap.
4) "intern()" method is called on the String class instance.
5) since there is no String class instance with the value "hello" inside String pool, it gets created.
6) "s2" reference is created on the stack.
7) "s2" refers to the instance inside String pool.

If String class instance with the value "hello" inside String pool is already there then no new instance will create. s2 directly refer to existing one instance.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

What is string pool?

String pool is the memory space in heap memory specially allocated to store the string objects created using string literals. In String pool, there will be no two string objects having the same content

Immutable

It means whenever we perform any action on an instance it does not affect that instance, rather it creates another instance.
String is immutable class.
Immutable does not have setter method.

Why string is immutable?

in case of String there is a possibility that more than one references may refer to same object [from string pool]. here if String class is mutable what will happen is because of one reference other references will suffer. That's why String is immutable.

String VS StringBuffer

String is immutable
StringBuffer is mutable

String s1="hello";
s1+"hi";

s1 will not be modified, rather it will create a new string "hellohi".
thus, increasing memory consumption proving the fact that string is immutable.

StringBuffer sb=new StringBuffer("hello");
if we say sb.append("hi");

a new object is not created, rather existing object gets modified. Thus, reducing memory consumption.

Conclusion:
It is always recommended to work with StringBuffer
a) it is faster
b) less memory consumption

StringBuilder is also same thing as StringBuffer
// StringBuilder class is mutable, that means any operation is //performed on an instance, will modify the same instance.

StringBuilder sb1=new StringBuilder("hello");
System.out.println(sb1); //hello
sb1.append("world");
System.out.println(sb1); //helloworld

StringBuilder is more efficient than StringBuffer.

Why we use this() ?? example

```
public class MyDynamicArray
{
public MyDynamicArray()
{
this(10);
}
}
public MyDynamicArray(int capacity)
{
create dynamic array with the given capacity
}
```

we want to make capacity by default 10 if not provided
MyDynamicArray array1=new MyDynamicArray(50);
MyDynamicArray array2=new MyDyanamicArray();

String methods
s1.equalsIgnoreCase(s2);
s2=s1.concat(“hello”);
go for diary
s1.ComapreTo(s2)
IndexOf(char)
Substring(int)
LowerCase();
UpperCase();
Trim();
Replace(char1,char2);

Exception Handling

What is exception?
An exception is an event, which occurs during the execution of a program, that interrupts the normal flow of the program's instructions.
Exception can arise from different kind of situations such as wrong data entered by user, network connection failure etc.
Whenever any exception occurs while executing a java statement, an exception object is created and then JVM tries to find exception handler to handle the exception. If suitable exception handler is found then the exception object is passed to the handler code to process the exception, known as catching the exception. If no handler is found then the application gets terminated.

All exceptions are inbuild classes. Object is a grand parent of all exception and exception class is parent of all other exceptions.

Advantages of exceptions:

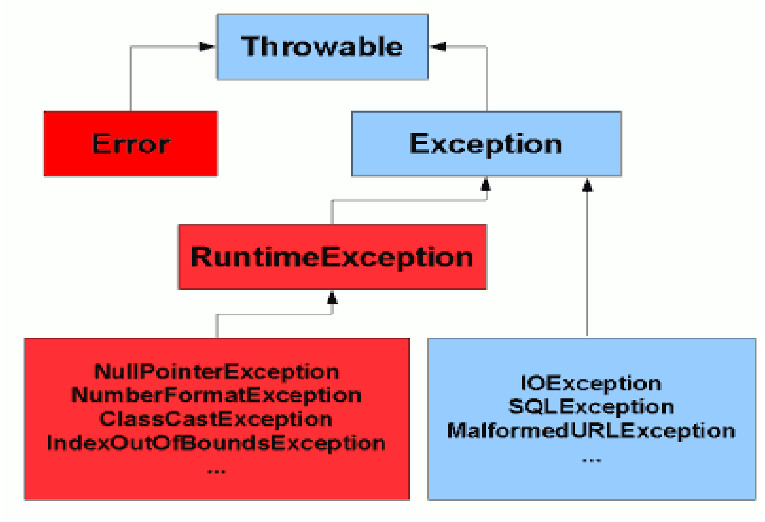
1. In traditional programming, error detection, reporting, and handling often lead to confusing code because programmers would use error code inside the main logic. Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere.

2. A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. i.e., if a method does not want to handle an exception it can propagate it to the caller and so on. we did not have this advantage in traditional programming.

3. because all exceptions thrown within a program are objects, categorizing of exceptions is a natural outcome of the class hierarchy.

How Exception Handling works?
1. problem occurs
2.create exception
3.throw exception
4.handle exception

When exception is raised, it happens in two steps :
a) instantiation of that particular exception class
b) throwing that exception (that instance or object) to the caller.



Exception

[**unchecked exceptions**]
RuntimeException
[due to programmer's logical mistake]

[they can be avoided using simple "if....else" , so, java does not enforce]

1.NullPointerException
2.NumberFormatException
3.IndexOutOfBoundsException
4.ClassCastException

[checked exceptions]
IOException
[these can be raised in a logically correct program]

[java enforces programmer to handle these]

1.FileNotFoundException
2.EOFFileException
3.SQLException

Types of Exceptions in Java
Java defines two kinds of exceptions:

Checked exceptions: Exceptions that inherit from the Exception class are checked exceptions. Checked exceptions are those exceptions which can be raised in a logically correct program. Client code has to handle the checked exceptions thrown by the API, either in a catch clause or by forwarding it outward with the throws clause.

Unchecked exceptions: Unchecked exceptions are those exceptions which can be raised due to programming (logical) mistakes. RuntimeException also extends from Exception. However, all of the exceptions that inherit from RuntimeException get special treatment. There is no requirement for the client code to deal with them, and hence they are called unchecked exceptions.

try and catch block

```
try
{
// possible statement that might cause exception
}
catch(exception ex)
{
// catch the object which thrown by jvm
}
```

In case of one try multiple catch, when u define one try and multiple catch, the rule is most specific catch block should precede most generic catch block, else compiler will throw an error.

In try block only one statement can be handled.

```
finally
{
}
```

We have one more block called as finally block.

finally block gets executed irrespective of whether exception is raised or not.
it can be used to release resources such as file, socket, database connection etc. since you cannot rely upon "finalized" method for the same task.
finally block can follow after catch (try..catch...finally) or even after try (try...finally).
If exception is not handled then 1st finally get executed then program will abort.
finally block will not get executed if
a) System.exit(0) is called inside try, catch or finally block.
b) exception gets raised in finally block itself.
Within finally we can write try and catch blocks.

User defined exception

how to create user defined exception?

for checked exception:
public class MyException extends Exception{}

for unchecked exception:
public class MyException extends RuntimeException{}

on what basis you will decide whether to create checked or unchecked exception?

you create checked exception when you would like your client to take corrective action/s in case of exception. i.e. checked exception somehow enforces client to handle it. (using try...catch) and the corrective action/s can be taken inside catch block.

you create unchecked exception when you feel there is no need of any corrective action by client in case of exception.

What is “handle or declare” rule?

a) whenever any method raises checked exception/s, method has to either handle [try….catch] or declare [throws] that checked exception/s.

b) whenever u invoke a method which has declared [using throws] checked exception/s, caller method has to either handle [try….catch] or declare [throws] that checked exception/s.

throw – used to raise the exception

throws- used to declare the exception

what do you mean by "declaring" exception?

"Declaring" exception means propagating it to the caller and then it's caller's responsibility to "handle or declare"

Automatic Resource Management (ARM)

what is Resource Management?

it means releasing the resources once you used them.

Before java7 programmer had to do Explicit Resource Management

```
try
{
FileInputStream fis=new FileInputStream("abc.txt");
// code to read the file
}
catch(FileNotFoundException e)
{
e.printStackTrace();
}
finally
{
fis.close();
}
```

what is exactly Automatic Resource Management (ARM) ?

it means now (java7 onwards) programmer can write the code as follows:

```
// ARM block
//here we write try with round brackets inside which we declare object.
try(FileInputStream fis=new FileInputStream("abc.txt"))
{
// code to read the file
}
catch(FileNotFoundException e)
{
e.printStackTrace();
}
```

That's it. it means you don't have to define "finally" block.

when you compile above code, compiler will convert this code automatically with the finally block

compiler provide "finally" block which has "fis.close()" statement. So since compiler takes care of releasing resource/s, It is known as "Automatic Resource Management"

can we write any class inside ARM block?

No. inside ARM block you can write only those classes which implement "AutoCloseable" or "Closeable" interface.

Ex. File related classes, database connection related classes, socket related classes.

Does catch(object ob) will work?

-NO.

because if "catch(Object)" is allowed , then people can write

throw new String("hello");
throw new Integer(20);
throw new Sample();
etc.
and the rule is along with "throw" you can have class of type "Throwable" only.

Assertions

Assertions are used to test the program. The advantage is assertions are by default disable, u need to enable them.

Assertions vs System.out.println statements while testing the code:

if u write S.o.p statements while testing the code, u need to remove or comment them while the code goes to client.

if u write assert statements u need not do anything as assertions are by default disable.

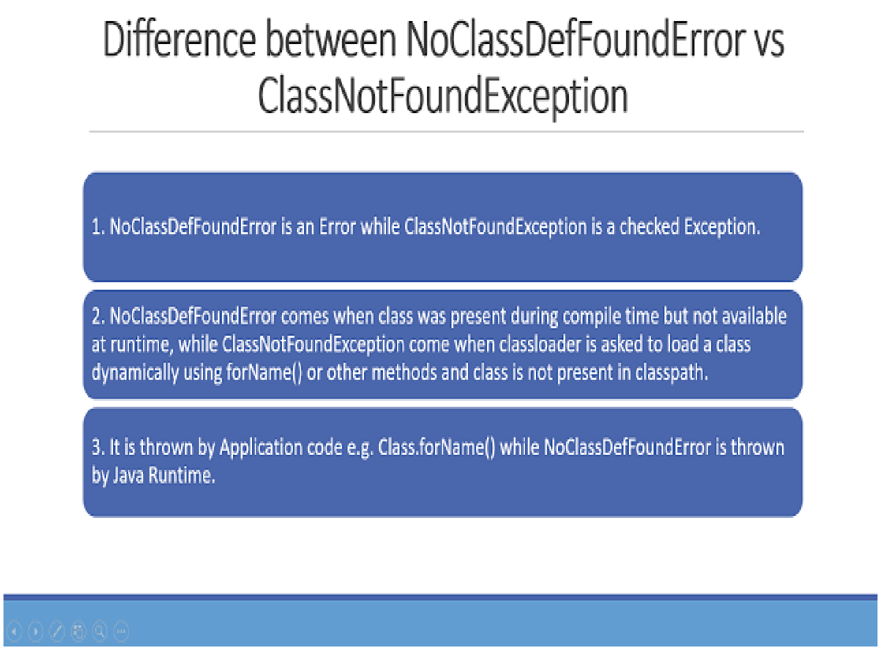
assert boolean expression
ex. **assert(num<=10);**

here if boolean expression is true, remaining code will get executed. if expression is false, then AssertionError will come.

it is recommended not to handle AssertionError (though we can handle it)
because when it comes, we would like to stop the execution or we will come to know where is the problem.

As we know by default assertion is disabled. So, to enable the assertion we need to use command **"-ea classname".**

NoClassDefFoundError vs ClassNotFoundException


Difference between NoClassDefFoundError vs ClassNotFoundException

1. NoClassDefFoundError is an Error while ClassNotFoundException is a checked Exception.

2. NoClassDefFoundError comes when class was present during compile time but not available at runtime, while ClassNotFoundException come when classloader is asked to load a class dynamically using forName() or other methods and class is not present in classpath.

3. It is thrown by Application code e.g. Class.forName() while NoClassDefFoundError is thrown by Java Runtime.

Application should not try to catch Error - Because, in most of cases recovery from an Error is almost impossible. So, application must be allowed to terminate.
Example> VirtualMachineError, IOError, AssertionError,OutOfMemoryError, StackOverflowError.

Let’s say errors like OutOfMemoryError and StackOverflowError occur and are caught then JVM might not be able to free up memory for rest of application to execute, so it will be better if application don’t catch these errors and is allowed to terminate.

Overriding 4th rule: implementation

1. overriding method may not declare any checked exception.
2. overriding method can declare same checked exception or its sub-type declared by overridden method.
3. overriding method cannot declare checked exception not declared by overridden method.

Wrapper classes in java

Wrapper classes are used to wrap primitives. in java Wrapper classes are available for each primitive.

byte -	Byte
short -	Short
int -	Integer
long -	Long
float -	Float
double -	Double
char -	Character
boolean-	Boolean

all the wrapper classes are derived from "java.lang.Object". They all are "final".
All wrapper classes are immutable, once initialised will not change during runtime.

```
public class Demo
{
                    static void show(Object ref)
                    {
                          S.o.p(ref);
                    }
                    main()
                    {
                          show(new String("hello")); // alld
                          show(new ArrayList());  // alld
                          show(new LinkedList()); // alld
                                   int num=10;
                          // show(num); // was not possible in before jdk5
                              // now its is possible in jdk5 onwards, internally compiler converts this code as below code

                          Integer ob=new Integer(num);

                          show(ob); // this is possible before in jdk 5

                    }
}
```

// convert int to Integer – **Autoboxing [assigning primitive to wrapper]**
int num=5;
Integer ob1=new Integer(num); //before jdk5
Integer ob1=num;   //new method
// convert Integer to int **– Unboxing [assigning wrapper to primitive]**
int k=ob1.intValue();   // before jdk5
int k=ob1;   //new method
autoboxing and unboxing concepts are only till compile time. When u compile the code, compiler converts autoboxing and unboxing code into the old code. It is because JVM doesn't understand autoboxing and unboxing.
conclusion: autoboxing and unboxing are only syntactical sugar.

Why do we need wrapper class?
suppose we have a method in java
```
                    void add(Object)
```

in which we want to store any primitive value, we can not pass primitive directly to Object as they are not compatible. e.g. Object class and int are not compatible.

solution here is to convert int to Integer (autoboxing) and pass Integer to "add" method. This is acceptable because Integer is a child class of Object.

One more use of wrapper class:
//used to convert str into int using wrapping

```
String str="125";
Int k = Integer.parseInt(str);
```

Enum

- enum is a user defined data type.
- it is used to define set of predefined values. It helps in making the program more readable and also helps to reduce programming bugs.
- Every enum in java is derived from "Enum" class
- Any wrong input will not compile hence there is no javac risk of unpredictable result. Which is happen in case of static final variables.

Varags (variable arguments)

They internally create an array

```
void disp(int ...set)
{
for(int i=0;i<set.length;i++)
      {
                System.out.println(set[i]);
       }
                for(int i:set)  // foreach loop
  {
                System.out.println(i);
      }
}
public static void main(String args[])
{
     Demo1 d=new Demo1();
 d.disp(10,20);
 d.disp(10,20,30,40,50);
     d.disp();
}
```

Command line arguments

arguments which can be given to the running java program. it also means that arguments given to main function. in Java main function is always ready to accept command line arguments.

public static void main(String args[])

widening (we can say upcasting)

int=short
double=float
base type=sub type

Overloading- considering
a) Widening (upcasting) b) Autoboxing c) Var-args

Widening beats boxing
Widening beats var-args
boxing beats var-args

Rules For Oveloading methods using Widening, boxing, and var-args.

1) Primitive widening uses the "smallest" method argument possible.
2) Used individually, boxing and var-args are compatible with overloading.
3) You cannot widen from one wrapper type to another (IS-A fails).
4) You cannot widen and then box (An int can't become Long).
5) You can box and then widen. (An int can become an Object, via Integer).
6) You can combine var-args with either widening or boxing.

Microsoft Word Document

Bridge methods in java.

In above example (inside word file), return type of overriding method is co-variant. We know this is overriding, compiler also compile this but jvm doesn't know the co-variant rule of the overriding. For jvm both methods are different. So, compiler internally inherit the base disp method inside sub class and inside it calls the sub disp method which has co-variant return type and control goes to that sub disp method.

Why cloning?

The clone() saves the extra processing task for creating exact same copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

If the cost of creating a new object is large and creation is resource intensive, we clone the object. We use the interface Cloneable and call clone() method to clone the object.

Important points about object cloning in java

java.lang.Object class has following method:
protected Object clone()throws CloneNotSupportedException
if you need to **clone** object of a particular class, that class has to
a) implement "Cloneable" interface [Cloneable interface does not have any abstract method. It's a marker interface]
b) override "clone()" method of Object class.

MyClass2.java MyClass1.java

Protected accessibility

Protected members are "**accessible outside the package only through inheritance** ".
protected member will be accessible in the subclass outside the package by calling function name directly.
protected member will not be accessible in the subclass outside the package by using parent class's reference.

How to create immutable class in java?

1) Don't provide "setter" methods — methods that modify fields

2) If the instance fields include references to mutable objects, don't allow those objects to be changed:
i.e., Don't provide methods that modify the mutable objects.

3) If the instance fields include references to mutable objects, don't allow those objects to be changed:
Don't share references to the mutable objects.

4) if necessary, create copies to avoid returning the originals in your methods

5) a class should be final
(Immutable classes need to be final, to avoid a subclass to break the immutability.)
(A subclass cannot actually modify the values of private properties in its parent, but it could behave as though it has)
final Ensure that the class can't be extended. This prevents careless or malicious subclasses from compromising the immutable behaviour of the class by behaving as if the object's state has changed.

The standard argument for making immutable classes final is that if you don't do this, then subclasses can add mutability, thereby violating the contract of the superclass. Clients of the class will assume immutability, but will be surprised when something mutates out from under them.

Innerclass/nested class

Nested classes are divided into two categories: **static and non-static.**
Nested classes that are declared static are called **static nested classes.**
Non-static nested classes are called **inner classes**.
classes which are defined inside any method are known as "**Local Inner Classes**".

```
Outer.inner i=o1.new inner();
```

here when inner class gets instantiated, compiler puts a reference of "Outer" class inside inner class object. That reference refers to the object where "o1" refers to.

Why compiler does that?
because even though they are "outer" and "inner" classes for a developer, for JVM these two are different classes altogether.

How compiler does that?
compiler adds a parameterized constructor accepting "outer" class reference inside "inner" class and discards all the other available constructors.

What is use of inner class?
Inner class makes the code more readable and maintainable.
The inner class shares a special relationship with the outer class i.e. the inner class has access to all members of the outer class and still have its own type is the main advantages of Inner class. Advantage of inner class is that they can be hidden from the other classes in the same package and still have the access to all the members (private also) of the enclosing class. This increases the level of encapsulation.
Inner classes are best used in the event handling mechanism and to implement the helper classes.

How Are Inner Classes translated to the Byte Code?
Inner classes have been implemented only to the compiler level. When the classes are compiled which contain inner classes, the byte code which gets generated does not actually implement inner classes as a class within a class.
"Inner classes are translated into regular class files with $ (dollar signs) delimiting outer and inner class names and the virtual machine does not have any special knowledge about them"
That means when the Outer class gets compiled, we get two .class files
1. Outer.class
2. Outer$Inner.class

Note:
Top level classes only be public and default but nested classes can be private, public, protected, default.
Nested class are used for encapsulation. They separate aspects into outer aspects and inner aspects.

How we can access outer class members inside inner class directly?
When we create a object of inner class compiler adds 1 member secretly called noname reference and that member is nothing but reference of outer class and it refers to that object by using which we have created inner class object. Because of this we can call outer class members directly.

In case of inner class:
```
Outer.inner i=o.new inner();
```

In case of static nested class:
```
Outer.inner i=new Outer.inner();
```

Here static nested class cannot access non-static instance members of outer class. It only accesses static instance member of outer class. Because static nested class does not required object of outer class to initiate the object which is in case of inner class.so when inner class instantiate by that time my be outer class is not instantiated so, how can we access members of outer class.

Anonymous class

Drawback: Anonymous class can either implements a class or extend a class both things can-not be come together

When we say,
```
New emp()                    //anonymous class which implementing class emp or interface emp.
        Void disp()
        {
             //code
        }
}
```

It means this class has no name and it implementing emp class / interface

-Anonymous class when get compiled we get .class file as **outer$1.class**

Reflection API

The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine. You'll want to use the reflection API if you are writing development tools such as debuggers, class browsers, and GUI builders. With the reflection API you can:

1) Determine the class of an object.
2) Get information about a class's modifiers, fields, methods, constructors, and superclasses.
3) Find out what constants and method declarations belong to an interface.
4) Create an instance of a class whose name is not known until runtime.
5) Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
6) Invoke a method on an object, even if the method is not known until runtime.
7) Create a new array, whose size and component type are not known until runtime, and then modify the array's components

It means an ability to find out about classes, methods, properties etc. during runtime.
It can also mean we can instantiate classes during runtime even though we don't know their name till runtime.
This API is used by programmers who designs IDE's, web browsers, web servers, application servers.
reflection does not show parent class methods if parent class is in different package.

- forName()	is a static method of class "Class" which is used to load a given class.
- getDeclaredMethods();	returns method list which (Method)
- getDeclaredConstructors()	returns constructor list (Constructor)
- getParameterTypes();	returns all parameters of method /constructor (class arr[])
- getDeclaringClass()	return class name
- getExceptionTypes()	return all exceptions (class arr[])
- getReturnType()	return return type of method

-	getName()	return method/constructor name
-	getDeclaredFields()	return field/variable name (Field)
-	getType()	return field type ex. Int, double, java.lang.String
-	getModifiers()	return modifiers (int)
		Modifier.toString(mod) ; - to get readable modifier name

```
getDeclaredMethod("disp1",null);
invoke(class reference,null)
setAccessible(true);
getDeclaredField("num1");
value.setAccessible(true);
value.set(class reference, value to be set);
```

where we do use of class "Class" in programming?
We use class "Class" in Reflection API.

Can we access private members outside the class?
Yes, using reflection API.

To give security for no to access private members outside the class even using reflection API
Use constructor

```
Sample ()
{
   System.setSecurityManager(new SecurityManager ());
}
```

Multithreading

- Thread class and Runnable interface have connection between them. Thread class implements Runnable interface.
- In multithreading thread Scheduler always call run method of Runnable interface in case of **implements Runnable.**

```
Public class Thread implements Runnable
{
           public thread()
          {
          }

         public thread (Runnable target)
        {
              This.target=target;
        }

        public void run()
       {
             If (target!=null)
             {
                 target.run();
             }
       }
}
```

Multitasking means performing more than one task simultaneously.

process - is a program in execution.
thread - is one of the parts of program in execution.

process-based multitasking: - more than one processes are running simultaneously. e.g. word and excel applications are running simultaneously.

thread-based multitasking: - more than one threads are running simultaneously. e.g. within a word application, you can start formatting as well as printing.

whether process-based or thread-based, a CPU can handle only one task at a time, unless it is multiprocessor machine. It is just an impression given to the user. what actually CPU does is context switching, i.e., jump from one task to another and vice-versa.

process-based vs thread-based

a) threads can share the memory, processes can not
b) context switching between threads is relatively cheaper as compare to that between processes.
c) cost of communication between threads is also low.
(Cheaper or cost is low actually means less no. of system resources are used.)

- Java supports thread-based multitasking.

Application of multi-threading in java: - due to multithreading feature, java has become effective on server side. e.g. Servlet, JSP etc.

Thread-Schedular
a) pre-emptive
b) time-slice.

JVM can have any schedular, i.e. either pre-emptive or time-slice.
It is because JVM is different for different platforms.

Java has given certain mechanisms (functions) whereby u can make sure, Ur multi-threading application can run more or less same on any os.

Following are the imp. steps required for multithreading application.
1) create thread/s
2) define thread execution body
3) register thread with the thread schedular
4) thread schedular will execute the thread/s

Java's multi-threading support lies in

a) java.lang.Thread
b) java.lang.Runnable (interface)
c) java.lang.Object

Thread class:- this is the most imp. class required in order to create multi-threading application.
Following are its methods.

a) **start**
   is used to register thread with jvm schedular

b) **run**

is used by the programmer to define thread execution body, but will be called by jvm schedular whenever it executes a particular thread.
when the run method is over, thread is dead.

c) **sleep** (static)
is used to make thread sleep for some time

d) **setName**
to set the name of thread

e) **getName**
to get the name of thread

f) **currentThread**
returns the currently running thread

g) **setPriority**
to set the priority
in java priorities are numbers from 1 to 10
1 - minimum priority
5 - normal priority
10 - maximum priority

by default every thread created has normal priority.

(imp:- priorities are not guaranteed across different platforms.)

h) **getPriority**
to get the priority
i) **join**
join() method is used for waiting the thread in execution until the thread on which join is called is not completed.

Runnable interface :- interface which contains abstract method

void run();

There are two ways to create multi-threading application.
1) extends Thread
2) implements Runnable

e.g.
extends Thread

```
public class Th1 extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Hello"+i);
                }
        }
        public static void main(String args[])
        {
                Th1 t1=new Th1();
                t1.start();
        }
}
```

by def. every java application has main thread created by jvm. This thread is used to execute main() function.
In the above code, there are 2 threads
main thread
user defined thread i.e. t1
Hence there are 2 call stacks in the above code. one for main() and other for t1 (run() method).

when main function is over, main thread dies, but user defined thread/s can continue. They will be taken care by JVM.
i.e. in the above code, after "t1.start()" when main() function is over, main thread dies , but t1's execution will be managed by JVM.

```
public class Th2 extends Thread
{
        public void run()
        {
                System.out.println(Thread.currentThread());
                for(int i=0;i<5;i++)
                {
                        System.out.println("Hello"+i);
                }
        }
        public static void main(String args[])
        {
                System.out.println(Thread.currentThread());
                Th2 t1=new Th2();
                t1.setName("first");
                t1.start();
        }
}
```

can we call run() directly ?

```
public class Th3 extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Hello"+i);
                }
        }
        public static void main(String args[])
        {
                Th3 t1=new Th3();
                t1.setName("first");
                t1.run();
        }
}
```

we can call run() directly. But in that case it won't be thread execution , it is a normal method call. That is different call stacks won't be created.

more than one user defined-threads

```
public class Th4 extends Thread
{
	public void run()
	{
		System.out.println(Thread.currentThread());
		for(int i=0;i<5;i++)
		{
			System.out.println("Hello"+i);
		}
	}
	public static void main(String args[])
	{
System.out.println(Thread.currentThread());
		Th4 t1=new Th4();
		Th4 t2=new Th4();
		t1.setName("first");
		t2.setName("second");
		t1.start();
		t2.start();
	}
}



public class Th4_a extends Thread
{
	public void run()
	{
		for(int i=0;i<5;i++)
		{
			System.out.println("Hello  "+Thread.currentThread().getName()+"\t"+i);
			try
			{
				Thread.sleep(100);
			}
			catch(InterruptedException ie)
			{
				ie.printStackTrace();
			}
		}
	}
	public static void main(String args[])
	{
		Th4 t1=new Th4();
		Th4 t2=new Th4();
		t1.setName("first");
		t2.setName("second");
		t1.start();
		t2.start();
	}
}

public class Th5 implements Runnable
{
	public void run()
	{
		for(int i=0;i<5;i++)
		{
			System.out.println("Hello"+i);
		}
	}
	public static void main(String args[])
	{
		Th5 ob=new Th5();
		Thread t1=new Thread(ob);
		Thread t2=new Thread(ob);
		t1.start();
		t2.start();
	}
}
```

a) define a class which implements Runnable
b) define run()
c) instantiate the class which impl. Runnable
d) instantiate Thread class by passing above instance (child of Runnable)
e) register Thread class instance/s

how Thread class is related to Runnable interface?

ans- Thread class implements Runnable

Difference between extends Thread and implements Runnable

1. *When you extend Thread class, after that you can't extend any other class which you required. (As you know, Java does not allow inheriting more than one class). When you implement Runnable, you can save space for your class to extend any other class in the future or now.*

2. *by extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same object instance.*

what is the use of implements Runnable?

if your class is already extending some class, you can't say extends Thread, because multiple inheritance is not allowed in java. In that case you have to go for implements Runnable.

above program also proves that threads can share the memory.

when threads share the memory there is a risk of "race condition".

e.g.

There are 2 threads.
one thread is reading from a file
other thread is writing to a file.

Race condition means
while one thread is reading from a file, other thread might write in a file or vice-versa.

Race condition always leads to Data Corruption.

How do we avoid Race condition?

we will have to make sure that while one thread is working on a data, other thread should not run. Only after first thread completes its job, other thread should start its execution. In java we can achieve this by using "synchronization".

"synchronization" is a solution to the race condition.

```
public class Th6 implements Runnable
{
	synchronized public void run()
	{
		for(int i=0;i<5;i++)
		{
```

```
                        System.out.println("Hello"+i);
                }
        }
        public static void main(String args[])
        {
                Th6 ob=new Th6();
                Thread t1=new Thread(ob);
                Thread t2=new Thread(ob);
                t1.start();
                t2.start();
        }
}
```

Synchronized Block

```
public class Th7 implements Runnable
{
        public void run()
        {
                synchronized(this)
                {
                        for(int i=0;i<5;i++)
                        {
                                System.out.println("Hello"+i);
                        }
                }
        }
        public static void main(String args[])
        {
                Th7 ob=new Th7();
                Thread t1=new Thread(ob);
                Thread t2=new Thread(ob);
                t1.start();
                t2.start();
        }
}
```

synchronized keyword

method :- all the statements are protected.

block :- only those statements are protected which are given inside synchronized block.

what exactly happens when we use synchronized keyword?

There is a concept of object lock.

in java every object has a lock. This lock can be accessed by only one thread at a time. The lock will be released as soon as the thread completes its job and thus another thread can acquire the lock.
This lock comes into picture only when object has got non-static synchronized method/s or block. whichever thread executes the synchronized method first, it acquires the lock. Other thread/s have to be in "**seeking lock state**".

Acquiring and Releasing lock happens automatically.

once a thread acquires a lock on an object, it can have control on all the non-static synchronized methods of that object.

Synchronized is used only and only when threads share the same object. Because synchronise will provide the lock on object and that lock will not give permission to interrupt any other thread to work on same object while already 1st thread is working on that. So synchronized is related to single object only to avoid race condition.

In Following code, threads are sharing different objects so synchronise has no use in this. Race condition will not occur.

```
public class Th8 implements Runnable
{
	public void run()
	{
		synchronized(this)
		{
			for(int i=0;i<5;i++)
			{
				System.out.println("Hello"+i);
			}
		}
	}
	public static void main(String args[])
	{
		Th8 ob=new Th8();
		Th8 ob1=new Th8();
		Thread t1=new Thread(ob);
		Thread t2=new Thread(ob1);
		t1.start();
		t2.start();
	}
}
```

Even though synchronized method or block is used to avoid "Race Condition", there can be danger of "DeadLock" inside it.
e.g. if one thread is working inside synchronized block or method and if it gets stuck up ! imagine what will happen ?
neither this thread can complete and release the lock, nor other thread can acquire the lock.

Solution to this is to have a Communication bet'n threads.
i.e,. if the thread realizes it cannot continue, it should come out of synchronized method or block and release the lock. Now other thread will acquire the lock, execute the code and allow the first thread to resume.

Following are the methods used for communication bet'n threads.

a) **wait**

it will make thread, release the lock and go to wait pool.

Wait will execute when thread stuck or blocked, then this thread executes wait on himself to come out from lock.

b) **notify**

it will make the thread to move from wait pool to seeking lock state.

c) **notifyAll**

it will make all the threads to move from wait pool to seeking lock state.

In wait pool there are only those threads who called on himself **wait**

These methods are defined in "java.lang.Object" class and are final so u cannot override them.

These methods must be called from synchronized method or block.

Because this communications will call when object gets block and object block comes in picture only in case of synchronization.

difference bet'n wait and sleep

wait releases the lock on an object , sleep does not.

Thread-safety

Thread-safe classes are those classes, which contain synchronized non-static methods.

what is class lock?

We use class lock when we want synchronization between two objects and both objects have independent threads only in case class have static synchronise methods.
They share one instance of class class.
every class has a lock. It is actually a lock on an instance of class Class. This is because , whenever any class is loaded in java, it is represented by instance of class Class.
The class lock comes into picture in case of synchronized static methods.
Thread which gives a call to synchronized static method can acquire a class lock. Only after thread complete that static method, lock is released.

```
public class Th9 implements Runnable
{
	public void run()
	{
		synchronized(this)
		{
			for(int i=0;i<5;i++)
			{
				System.out.println("Hello"+i);
			}
		}
	}
	public static void main(String args[])
	{
		Th9 ob=new Th9();
		Thread t1=new Thread(ob);
		Thread t2=new Thread(ob);
		t1.start();
		t2.start();

		System.out.println("Both the threads are over");
	}
}
```

in the above code "Both the threads are over" will not be displayed in the end because it is a statement of main. It is because as we know , main thread completes first and user defined thread are continue, they are taken care by JVM.
if we want that "Both the threads are over" should be displayed at the end, we have to make sure that main thread will complete only after the completion of "t1" and "t2".

Solution is "join()" method.

join() method

join method makes caller thread (main thread) to wait for called thread (t1 and t2) to complete.

how join works ?

```
public class Th9_a implements Runnable
{
        public void run()
        {
                synchronized(this)
                {
                        for(int i=0;i<5;i++)
                        {
                                System.out.println("Hello"+i);
                        }
                }
        }
        public static void main(String args[])
        {
                Th9 ob=new Th9();
                Th9 ob1=new Th9();
                Thread t1=new Thread(ob);
                Thread t2=new Thread(ob1);
                t1.start();
                t2.start();
                try
                {
                        t1.join();
                        t2.join();
                }
                catch(InterruptedException e)
                {
                }
                System.out.println("Both the threads are over");
        }
}
```

in the above code, when main() function calls "t1.join()" for example, it says "join me at your end".
Since main() is calling "t1.join()" and "t2.join()" , it is added to the end of both t1 and t2. That's why now the statement "Both the threads are over" is getting executed at the end.

whenever thread is in a blocked state ie.due to sleep, join or wait methods, it can get interrupted by other threads. Whenever blocked thread gets interrupted, it throws "InterruptedException".
But this can not be predictable, hence we have to be ready with "try... catch(InterruptedException)"

Thread states

Born – when thread created
Runnable – when we say start()
Running - decision by Jvm scheduler to run the thread
Blocked - in case of sleep(), wait(), join()
Dead – when run block overs and when unhandled exception raised
Note: from blocked state thread always goes to runnable state then jvm scheduler decided when to run thread.

User threads and Daemon threads

User threads
user defined threads
main thread

Daemon thread

e.g. garbage collection thread (low priority thread)

Daemon threads are the threads which are at the mercy (servant) of user thread/s. Their only purpose is to serve user defined thread/s. When there is no user thread alive, Daemon thread will die.

Example of Garbage Collection Thread

```
package core1;

public class Sample
{

	public static void main(String[] args) {
		Sample s1=new Sample();
		s1=null;
		//System.gc();

		//Runtime.getRuntime().gc();
		System.out.println("Done");

	}

	protected void finalize() throws Throwable
	{
		System.out.println("inside finalized method");
		System.out.println(Thread.currentThread());

	}

}
```

output:

```
Done
inside finalized method
Thread[Finalizer,8,system]
```

Difference between

System.gc() and Runtime.getRuntime.gc()

inside "System" class we have following code:

```
public static void gc()
{
     Runtime.getRuntime().gc();
}
```

difference between applying "synchronized" keyword and "synchronized block":

```
synchronized public void run()
```

```
{
        10 statements e.g
}
```

in above case, we have all 10 statements are blocked for a particular thread. i.e. unless a particular thread executes all 10 staements , second thread will not execute. It should be avoided if all 10 statements are not critical.

```
public void run()
{
        synchronized(this)
        {
                4 statements e.g
        }
        other 6 statements [ not so critical ]
}
```

above code will give u performance advantage as compare to previous code because only first 4 statements are blocked. once a thread completes first 4 statements , other thread can execute.

In case of Class lock, When u say cs.wait() cs.nofify() or cs.notifyAll() what does it mean ?

It means that,

wait(), notify() and notifyAll() are in Object class and they are final , so they cannot be overridden.

Class class is a child of Object class.

When we say cs.wait(),cs.nofify() or cs.notifyAll() it means that these methods are inherited in Class class which we are invoking.

Difference between implements Runnable and extends Thread in Java

1) The first and most important difference between extending Thread and implementing Runnable comes from the fact that a class can only extend one class in Java. So, if you extend the Thread class then your class lose that option and it cannot extend another class, but if you implement Runnable then your class can still extend another class e.g. Applet. It's a common pattern in Java GUI programming that your class extends Applet and implements Runnable, EventListener etc.

2) The second difference between extends Thread and implements Runnable is that using the Runnable instance to encapsulate the code which should run in parallel provides better reusability. You can pass that Runnable to any other thread or thread pool. [Reusability]

3) when you implement Runnable, [Thread creation and Thread Execution body can be loosely coupled] but if you extend Thread then they are tightly coupled.

4) Another difference between Thread and Runnable comes from the fact that you are extending Thread class just for run() method but you will get overhead of all other methods which come from Thread class. So, if your goal is to just write some code in run() method for parallel execution then use Runnable instead of extending Thread class.

5) The fifth difference between extending Thread and implementing Runnable also comes from OOP perspective. In Object oriented programming you extend a class to enhance it, to put some new features on it. if you just want to execute thread/s and get the work done, then stick with implementing the Runnable interface rather than extending Thread class.

Conclusion: -

When to use "extends Thread" over "implements Runnable"

The only time it makes sense to use "extends Thread" is when you have a more specialized version of Thread class. In other words, because you have more specialized thread specific behaviour.

But practically we just would like to get the work done by thread/s so in that case you need to use "implements_Runnable" which also leaves your class free to extend some other class.

Class lock vs object block

when u want to ensure that non-static method/s should be called by only one thread at a time, u apply lock on an object which is shared by more than one threads. The technique to apply "object lock" is to have synchronized non-static method/s or block/s.

What if u want to ensure that static method/s should be called by only one thread at a time?

here object lock will not help at all because static methods are not at all associated with the object. Hence u need to apply "class lock" i.e. lock on an instance of class "Class". The technique to apply "class lock" is to have synchronized static method/s or block/s.

What is ThreadPool in Java?

A thread pool contains previously created threads to execute a given task. These threads are initially idle. Once any task comes a randomly selected thread from thread pool will be made active and asked to perform the task. Once the task is over the thread is not destroyed but simply sent back to the pool for reuse.

Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive

Why thread pool comes in picture?

Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

For thread pool

```
Import java.util.concurrent.* ;
```

```
ExecutorService exec=Executors.newCachedThreadPool();
```

It creats a threads which will store inside threadpool which will be in ideal state
Where ExecutorService is a interface and Executors is a class

```
ExecutorService exec=Executors.newFixedThreadPool(2);
```

It creates fixed number of threads

```
exec.execute(new myapp());
```

gives a task to jvm

```
exec.Shutdown();
```

If u write this then after u won't give any task to jvm
shutdown() prevents new tasks from being submitted to that Executor. The current thread (e.g. main thread) will continue to run all tasks submitted before shutdown() was called.

Reentrant

Condiations

```
import java.util.concurrent.locks.*;
ReentrantLock mylock=new ReentrantLock();
Condition value=mylock.newCondition();
mylock.lock();
mylock.unlock();
```

Imp note:- call to “signalAll()” does not immediately activate a waiting thread. It only unblocks the waiting threads so that they can compete for entry into the object after the current thread has exited the “synchronized” method.

“signal()” method unblocks only a single thread from the wait state, chosen at random.

Imp note:- a thread can only call “await()”,”signal()” and “signalAll()” on a “Condition” object when it owns lock of the condition.

Lock Fairness (longest waiting thread get chance to acquire a lock).
locks favour granting access to the longest-waiting thread

What is ReentrantLock in Java?

ReentrantLock is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by Thread until a call to unlock() method. Fairness parameter is provided while creating instance of ReentrantLock in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().

Difference between ReentrantLock and synchronized keyword in Java?

1) acquiring and releasing lock needs to be done explicitly.

2) extended feature like fairness, which can be used to provide lock to longest waiting thread.

3) ability to trying for lock with or without timeout. Thread doesn’t need to block infinitely, which was the case with implicit synchronization.

Though ReentrantLock provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differ in certain aspect. main difference between synchronized and ReentrantLock is ability to trying for lock with or without timeout. Thread doesn’t need to block infinitely, which was the case with synchronized. Let’s see few more differences between synchronized and Lock in Java.

1) Another significant difference between ReentrantLock and synchronized keyword is fairness. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of conflict.

2) Second difference between synchronized and Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduce blocking of thread waiting for lock in Java application.

Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.

3) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

In short, Lock interface adds lot of power and flexibility and allows some control over lock acquisition process, which can be influenced to write highly scalable systems in Java.

Disadvantages of ReentrantLock in Java

Major drawback of using ReentrantLock in Java is , wrapping method body inside try-finally block, which makes code unreadable and error-prone. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

Things to remember:

- Release all locks in finally block.
- Beware of thread starvation! The fair setting in ReentrantLocks may be useful if you have many readers and occasional writers that you don’t want waiting forever. It’s possible a writer could wait a very long time (maybe forever) if there are constantly read locks held by other threads.
- Use synchronized wherever possible. You will avoid bugs and keep your code cleaner.
- Use tryLock() if you don’t want a thread waiting indefinitely to acquire a lock.

boolean tryLock()
tries to acquire the lock without blocking, returns true if it was successful.

boolean tryLock(long time, TimeUnit wait)
tries to acquire the lock, blocking no longer than the given time. Returns true if it was success.
If(mylock.tryLock(100,TimeUnit.MILLISECONDS))
TimeUnit is an enumeration with values SECONDS, MILLISECONDS, MICROSECONDS, and NANOSECONDS.

Note:

Case : synchronise

in case of synchronise execution of thread if any exception is raised then lock will automatically release and second thread get the chance to run.

Case : re-entrant lock

in case of ReentrantLock when exception is raised , lock is not released.
hence it is very imp to unlock inside finally block

##

FILE HANDLING

##

stream

stream is a communication path bet'n source and destination.

There are two types of streams
a) input stream :- for reading
b) output stream :- for writing

if u want to read, u need to open input stream on the source (e.g. file, array, network)
if u want to write ,u need to open output stream on destination (e.g. file, array, network)

Java has two categories of streams
byte streams :- for reading and writing bytes , image or sound files also. It is also used to read and write java Objects.
unicode character streams :- for reading and writing unicode characters.

File class in Java:

File class in java allows you to check properties of a given file such as size, last modified date, name, read/write permission, whether file is existing or not.

Additionally, File class also allows you to display information about a particular folder.

Entire io support is available from "**java.io" package.**

Hierarchy of stream

byte streams

```
        InputStream                              OutputStream    [ abstract classes ]
       [ read ]                                    [ write ]

     FileInputStream                                 FileOutputStream
```

unicode character streams

```
     Reader                                      Writer            [ abstract classes ]
     [ read ]                                   [ write ]

   FileReader                                     FileWriter
```

Syntax (class File)

```
File f=new File("c:\\temp\\FileDemo.java");
```

Or

```
FileOutputStream fos=new FileOutputStream("path ",true);
fos.write(b,0,k);

FileInputStream fis=new FileInputStream("path");
fis.read(b);

f.getName();
f.getPath();
f.getAbsolutePath();
f.exists()?"Exists":"Doesn't Exists";
f.canWrite()?"Can Write":"Can Not Write";
f.canRead()?"Can Read":"Can Not Read";
f.isDirectory()?"It is Directory":"It is not Directory";
f.isFile()?"Yes File":"No File";
f.isAbsolute()?"is Absolute":"It is Not Absolute";
new Date(f.lastModified());
f.length();

FileWriter fw=new FileWriter("path ");  //for Unicode writing in filesystem
FileReader fr=new FileReader("path ");  // for Unicode reading in filesystem
```

RandomAccessFile

A random-access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the file pointer; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read.

```
RandomAccessFile rf=new RandomAccessFile("path","rw");

rf.seek(rf.length());
//method sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
rf.seek(0);
```

Program to write and read primitives in file

DataOutputStream lets an application write primitive Java data types to an output stream in a portable way.

DataInputStream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.

```
DataOutputStream dos=new DataOutputStream(fos)

                                        dos.writeInt(10);
                                        dos.writeChar('A');
                                        dos.writeFloat(3.9f);
                                        dos.writeBoolean(true);
                                        dos.writeUTF("hello world");

DataInputStream dis=new DataInputStream(fis)

                               System.out.println(dis.readInt());
                              System.out.println(dis.readChar());
                              System.out.println(dis.readFloat());
                              System.out.println(dis.readBoolean());
                              System.out.println(dis.readUTF())
```

object persistence

it means saving the state of an object inside either filesystem or database so that it can be retrieved back in future.
When we save the state of an object inside filesystem, it is known as "**Serialization**".
When we read the state of an object from filesystem, it is known as "**Deserialization**".
in java there are two rules for Serialization:
a) a class has to implement either Serializable or Externalizable interface.
b) class must have all the instance members of type serialized. (Serialized type means which can be easily converted into sequence of bytes).

By default, all the primitives are of serialized type.
Reference type can be made serialized type by implementing either Serializable or Externalizable interface.

Serializing object means writing Object inside filesystem as follows:

```
FileOutputStream fos=new FileOutputStream("filename");
ObjectOutputStream oos=new ObjectOutputStream(fos);
        oos.writeObject(<object to be written>);
```

Deserializing object means reading object from filesystem as follows:

```
FileInputStream fis=new FileInputStream("filename");
ObjectInputStream ois=new ObjectInputStream(fis);
 MyClass m1=(MyClass)ois.readObject();
```

What happen in Deserialization?

1. The object is read from the stream.
2. The JVM determines (through info stored with the serialized object) the object's class type.
3. The JVM attempts to find and load the object's class. If the JVM can't find and / or load the class, the JVM throws an exception and deserialization fails.
4. A new object is given space on the heap, but the serialized object's constructor does not run! obviously, if the constructor ran, it would restore the state of the object back to its original 'new' state and that's not what we want. We want the object to be restored to the state it had when it was serialized, not when it was first created.

5. If the object has a non-serializable class somewhere up its inheritance tree, the constructor for that non-serializable class will run along with any constructors above that. Once the constructor chaining begins, you can’t stop it, which means all superclass’s, beginning with the first non-serializable one, will reinitialize their state.
6. The object’s instance variables are given the values from the serialized state. Transient variables are given a value of null for object references and defaults (0, false, etc.) for primitives.

Why object streams are used in Java?

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

Object Graph

when we implement Serializable, entire object graph is saved inside file.

What are the things get written when u serialize an object using Serializable?

- it writes out the metadata (description) of the class associated with an instance such as length of the class, the name of the class, serialVersionUID (or serial version), the number of fields in this class.

- Then it recursively writes out the metadata of the superclass until it finds java.lang.object.

- Once it finishes writing the metadata information, it then starts with the actual data associated with the instance. But this time, it starts from the top most superclass.

- Finally it writes the data of objects associated with the instance starting from metadata to actual content recursively.
  (has-a relationship objects)

How do writeObject() and readObject() works?

oos.writeObject(s)

when u call "writeObject" , it will check whether "s" implements "Serializable" or "Externalizable"
if it implements "Serializable" it will check whether u have defined "private writeObject

```
private void writeObject(ObjectOutputStream out) throws IOException;
{
                    System.out.println("in writeObject method");
                    char arr[]=pwd.toCharArray();
                    for(int i=0;i<arr.length;i++)
                    {
                              arr[i]=(char) ~arr[i];
                    }
                    String encriptpwd=new String(arr);
                    o.writeUTF(encriptpwd);
                    o.defaultWriteObject();
}
```

note: both object methods are use for customized serialization both methods are declared private and of course they must be declared private, proving that neither method is inherited and overridden or overloaded. The trick here is that the virtual machine will automatically check to see if either method is declared during the corresponding method call . if it is their then jvm call this methods.

if yes
  it will invoke it
if no
  it will go for def. serialization

if it implements "Externalizable" it will invoke "writeExternal()"

```
public void writeExternal(ObjectOutput out) throws IOException
 {
```

```
                System.out.println("in writeExternal method");
                char arr[]=pwd.toCharArray();
                for(int i=0;i<arr.length;i++)
                {
                        arr[i]=(char) ~arr[i];
                }
                String encriptpwd=new String(arr);
                out.writeUTF(encriptpwd);
                out.writeUTF(username);

        }
```

ois.readObject()

when u call "readObject",it will check whether class has implement "Serializable" or "Externalizable"
if it implements "Serializable" it will check whether u have defined "private readObject"

```
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
{
        System.out.println("in readObject method");
        String str=o.readUTF();
        char arr[]=str.toCharArray();
        for(int i=0;i<arr.length;i++)
        {
                arr[i]=(char) ~arr[i];
        }
        pwd=new String(arr);
        o.defaultReadObject();
}
```

if yes

it will invoke it

if no

it will go for def. deserialization

if it implements "Externalizable"
invoke default constructor
invoke "readExternal()" method

```
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
 {

                System.out.println("in readExternal method");
        String str=in.readUTF();
        char arr[]=str.toCharArray();
        for(int i=0;i<arr.length;i++)
        {
                arr[i]=(char) ~arr[i];
        }
        pwd=new String(arr);
        username=in.readUTF();
        }
```

Externalization

- Here we have control on serialization. Like in case of serialization we have a control on has-a but don't have control on is-a relationship.
- In this case while deserialization default constructor gets called back-to-back and then read external method gets called.

- Class should have default constructor else externalization will fail. It will serialise but at the time of deserialize it will fail and give InvalidClassException
- Both the methods are public. And need to define compulsory.
- externalization is fast than serialization.

  In case of Externalizable when we deserialized object:
- 1) A new object gets created in heap area.
- 2) Instance members are allocated memory.
- 3) Default constructor gets invoked.
- 4) readExternal() method gets invoked which initializes instance members with the help of file info.

why in case of Serializable "default constructor" does not get called during deserialization?
Because if here default constructor gets called then we will get default values and not those values which were there when we stored that object inside filesystem.

But then why default constructor gets called in case of Externalizable ?
It is true that default constructor gives us default initial values, But Externalizable gives us one more chance to reinitialize our members with the help of file info ie. using "readExternal()" method.

What is serialVersionUID ?
As per java docs, during serialization, runtime associates with each serializable class a version number, called a serialVersionUID, which is used during de-serialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

Simply put, the serialVersionUID is a unique identifier for Serializable classes. This is used during the deserialization of an object, to ensure that a loaded class is compatible with the serialized object. If no matching class is found, an InvalidClassException is thrown.

what exactly happens when you serialize an object?
serialization mechanism generates serialversionuid (if you have not defined it explicitly) for the class , whose object you are trying to serialize. This serialversionuid is then stored inside the file.

what exactly happens when you deserialize an object?
during deserialization the class whose object you want to deserialize , needs to be loaded.
Here, deserialization mechanism generates serialversionuid (if you have not defined it explicitly) for the class , whose object you are trying to deserialize. This serialversionuid is then checked with the serialversionuid stored inside file. if they match then deserialization becomes successful else you get an exception "InvalidClassException" and deserialization fails.

Since the default serialVersionUID computation differs on different JVM implementations, it is highly recommended for a class which implements Serializable or Externalizable interfaces to declare serialversionuid explicitly in order to ensure successful deserialization across all the platforms.
Ex.
<any access modifier> static final long serialVersionUID = 42L;

Compatible change and incompatible change discussion
- Ideally, after serializing object and distributing a file to client or other person, if you make any changes in your class, you need to serialize again and redistribute the file.
- A compatible change is one that can be made to a new version of the class, which still keeps the stream compatible with older versions of the class

Addition of new fields or classes does not affect serialization	you cannot change the default signatures for readObject() and writeObject()
We can change field access modifiers like private, public, protected or default	We can not change a class which implement serializable to implement externalizable and vice versa
Addition of new methods	we also cannot change the field types within a class
we can change a transient or static field to a non-transient or non-static field	You cannot alter the position of the class in the class hierarchy

we can change the access modifiers for constructors and methods of the class	You cannot change the name of the class or the package it belongs to, as that information is written to the stream during serialization.

Serialization inner class

- Both outer and inner class must implement serialized
- At the time of deserialization along with "Inner" class object, "Outer" class object is also created.
- In case if we made inner class to implement externalizable then we need to define that both the methods and as per rule we must need default constructor, but due to inner class has ref to outer class compiler provides parametrized constructor inside inner class so for that we need to do inner class as static public class or must provide pubic default constructor.

5 Different Ways to Create Objects in Java

Using new keyword	} → constructor gets called
Using newInstance() method of Class class	} → constructor gets called
Using newInstance() method of Constructor class	} → constructor gets called
Using clone() method	} → no constructor call
Using deserialization (readObject)	} → no constructor call

##

Generics

Why Generic provide type-safety?

before java 5

```
List mylist=new ArrayList();
```
// Arraylist is a implementation of list. List is Interface and arraylist is a child class of list

```
mylist.add(new Integer(100));
mylist.add("hello");
mylist.add(new Double(3.4));
```

because "add()" method argument is in java.lang.Object

```
String str=(String)mylist.get(1);
```
// while reading

because "get()" method return type is java.lang.Object. we need to typecast to string

what is the risk involved in case of above code?

if "Integer" is there at 1 position then we will get ClassCastException.

Generics - java 5 onwards

```
List<String> mylist=new ArrayList<String>();
```

now compiler will see to it that mylist will be used with String only or else it will give error which is much better than ClassCastException.

e.g.
mylist.add("hello");

```
mylist.add("welcome");
mylist.add(100); //  compilation error

String str=mylist.get(1); // no typecasting required
Integer ob=mylist.get(0); // u will get compilation error
```

Hence we can say that Generics provide "type-safety".

Type Eraser

```
public class Generic1<T>
{
            private T first;
            void setVal(T first)
            {
                        this.first=first;
            }
            T getVal()
            {
                        return first;
            }
```

when u compile above class, compiler will remove all the generic information because JVM can't understand Generics. This is known as "Type Erasure". So after compilation the above class will be as follows:

```
public class Generic1
{
            private Object first;
            void setVal(Object first)
            {
                        this.first=first;
            }
            Object getVal()
            {
                        return first;
            }
```

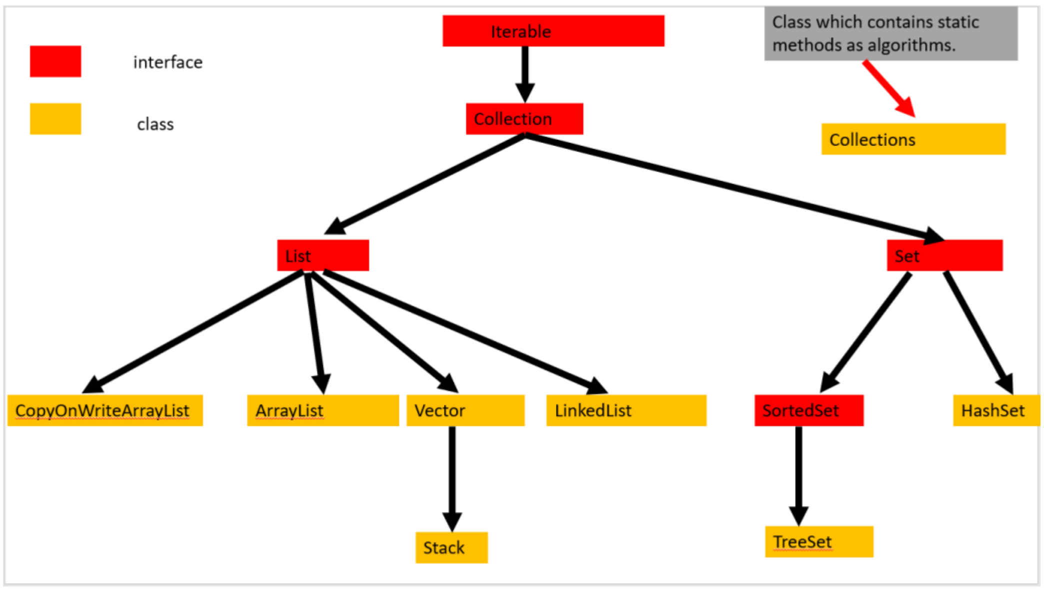
Why Collection API?

Arrays are fixed in size.
Collection API provides us Containers:

- They are dynamic
- Some of them are sorted
- Some of them allow only unique elements
- some of them are thread-safe
- some of them are not thread-safe, so they are efficient
- Some of them allow u to store key and value
- all the containers implement Serializable so that they can be easily persisted inside file system.

Iterator: allows u to traverse through containers.
Algorithms: sort, count, max, min etc.
Entire support for Collection API has been given inside "java.util" package.

Hierarchy:



Iterable: The Java Iterable interface represents a collection of objects which is iterable - meaning which can be iterated.

Collection: Enables you to work with groups of objects. It is at the top of the collection's hierarchy. It is the foundation upon which the collections framework is built.

List: Duplicates allowed. Cares about index. Has methods related to index. All 3 List implementation are ordered by index position.

ArrayList: Fast insertion and fast random access. Ordered collection (by index), but not sorted. Choose this over LinkedList, when you need fast iteration but are not likely to be doing a lot of insertion and deletion.

Vector: Similar to ArrayList but with two differences. Vector is synchronized (hence there is always a performance hit as compare to ArrayList) and it contains many legacy methods that are not part of the collections framework.
Extends AbstractList and implements List.

ArrayList	**Vector**
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the **Iterator** interface to traverse the elements.	A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

LinkedList: It is like ArrayList except elements are doubly-linked to each other. Even though iterates slowly as compare to ArrayList, insertions and deletions in a doubly-linked list are very efficient. I.e., elements are not shifted, as in case for an array. when frequent insertions and deletions occur inside a list, a LinkedList can be worth considering.

Stack: Subclass of Vector that implements a standard last in first out stack.

CopyOnWriteArrayList: Creates a copy of original ArrayList when you open an iterator on it. So that any updates can happen on original list while traversing is going on the copy.

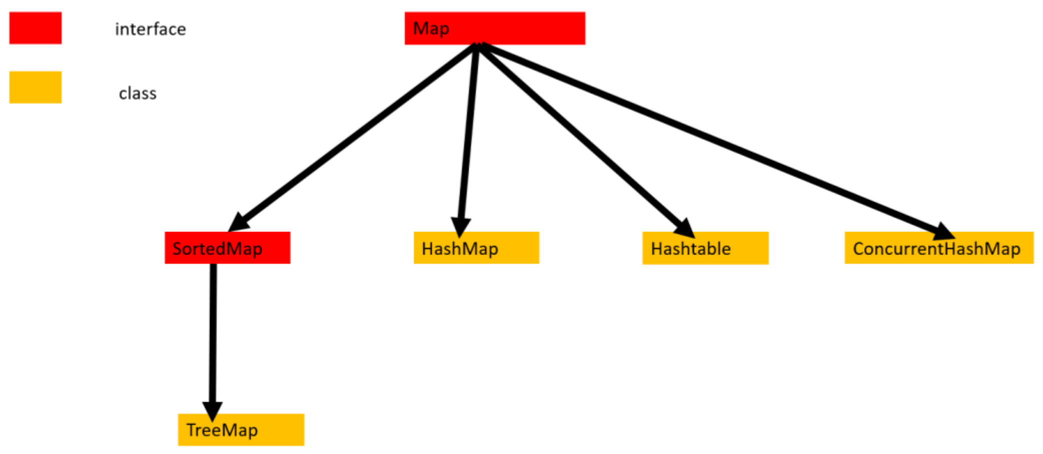
Set: Uniqueness. No duplicates, sometimes sorted.

SortedSet: Extends Set to handle sorted sets, sorted in ascending order.

Hashset: When you put an object into a HashSet, it uses the object's hashcode value to determine where to put the object in the Set. But it also compares the objects hashcode to the hashcode of all the other objects in the HashSet, and if there is no matching hashcode, the HashSet assumes that this new object is not a duplicate.
HashSet does not guarantee the order of its elements. If you need sorted storage, then TreeSet is a better choice.

Hashcode() method: HashSet or LinkedHashSet, the objects you add to them must override hashcode() , otherwise Object's hashcode() method will allow multiple object that u might consider "meaningfully equal" to be added to your "no duplicates allowed set".

TreeSet: Sorted, ascending order, optionally you can set the order using Comparator interface.



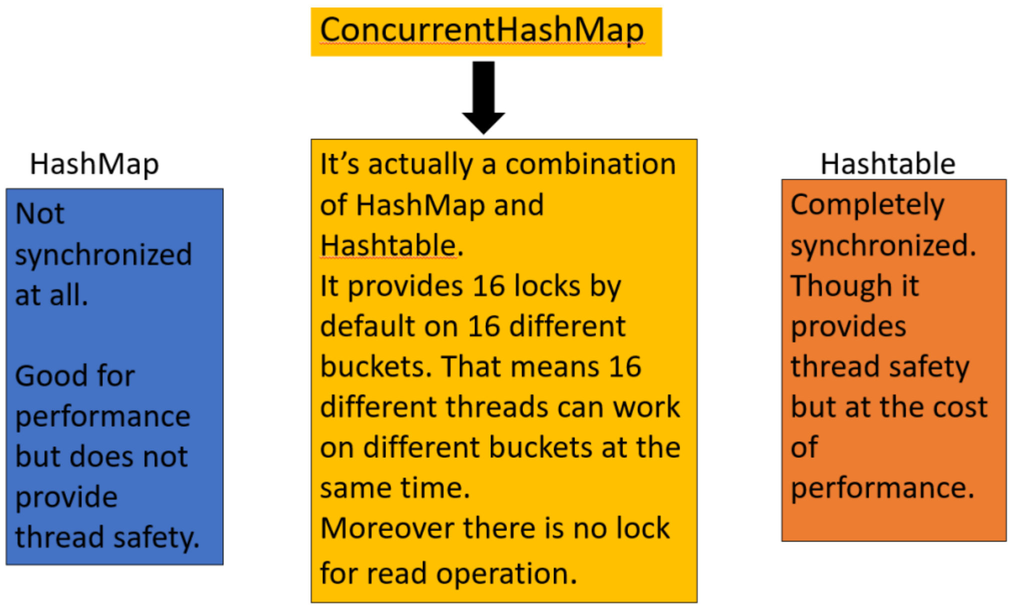
Map Interface: Maps unique keys to values. after the value is stored, you can retrieve it by using its keys.Null is not allowed. Although keys are typically String names, a key can be any object. Map.Entry I.e.Entry is an inner interface of Map.It describes an element (a key/value pair) in a map.

HashMap: Does not guarantee the order of its elements .Therefore ,the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.
Elements are stored as key,value pair.
Compare to this, other maps add a little more overhead. **It is the quickest map**.
Searching in a HashSet or HashMap can be faster than in a TreeSet or TreeMap, as hashing algorithms usually offer better performance than the search algorithms for balanced trees.

Hashtable: Implements Map. Similar to HashMap but it is synchronized. Stores key/value pair.

TreeMap: Sorted map, that is keys are stored in sorted order. Natural order like TreeSet. Custom sort possible using Comparable.

Concurrent HashMap:



ConcurrentHashMap

HashMap

Not synchronized at all.

Good for performance but does not provide thread safety.

It's actually a combination of HashMap and Hashtable.
It provides 16 locks by default on 16 different buckets. That means 16 different threads can work on different buckets at the same time.
Moreover there is no lock for read operation.

Hashtable

Completely synchronized. Though it provides thread safety but at the cost of performance.

Difference Iterator vs ListIterator:

Iterator is an interface use for traversal through List, Set and Map
ListIterator is a child of Iterator, which has two features

1) it allows modification
2) it allows bidirectional traversal

Fail-Fast Iterator

in case of ArrayList while u r traversing through the list using iterator if u try to add inside the list, u get "ConcurrentModificationException". It means iterator of ArrayList is "Fail-Fast".

Fail-Safe Iterator

in case of CopyOnWriteArrayList when u create an iterator, it creates a snapshot of original list so that u can traverse it. If u try to add inside the list, element gets added inside original list. It means iterator of CopyOnWriteArrayList is "Fail-Safe".

How does HashMap put works?

hashCode() is invoked to determine the bucket.

hashcode - first entry

subsequent entries
hashcode - different - different bucket

```
           same [ hash collision case]
                   == true - overwrite the value
                       false
                             equals - true - overwrite the value
                                      false - linked list will be formed within a bucket i.e. same bucket having different Entries
[Entry]
```

How does HashMap get works?

hashcode
bucket is determined for search

```
                    == true - get the value
                        false
                               equals - true - get the value
                                        false - linked list will be traverse and subsequently == and equals are invoked.
```

Hashtable vs Hashmap

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Concurrent HashMap

It's a combination of HashTable and HashMap. It has bucket lock. It's a thread safe and can be shared with multiple threads. This lock's are on write only, not for read.

TreeMap

In case of Comparable <person> //interface

```
Public int compareTo(person ref)
{
   return name.compareTo(ref.name);
}
```

In case of Comparator <person> //different class to implement Comparator interface

```
class NameComparator implements Comparator<Person>
{
	public int compare(Person ref1,Person ref2)
	{
		return ref1.getName().compareTo(ref2.getName());
	}
}
```

Note: collections always stores an references and while serialization copy of collection and copy object both the save in file.

Collections

Is a class which has set of algorithms in the form of static methods.

How to convert non-thread safe collection to safe thread collection?

```
List<String> mylist=new ArrayList<String>();   // non-thread safe list
List<String> mylist1=Collections.synchronizedList(mylist);
```

mylist1- Thread safe list

```
Map<String,Integer> mymap=new HashMap<String,Integer>();   // non-thread safe map
Map<String,Integer> mymap1=Collections.synchronizedMap(mymap);
```

mymap1 - Thread safe map

<? extends vs super>

List<? extends Animal> arr

it means

a) you can pass list of Animal or its sub types and not super types
b) you cannot add using "arr"

List<? super Dog> arr

it means

a) you can pass list of Dog or its super types but not sub types
b) you can add only "Dog or its sub types" using "arr"

```
	disp(List<?> mylist) // u can pass list of any type including Object
	{
		add // not allowed
	}
	vs

	disp(List<Object> mylist) // u can pass list of only Object type
	{
		add // allowed
	}
```

##

DAY 13

##

Why we need Default Methods?

Before Java8 interfaces were too tightly coupled with their implementation classes. I.e., it was not possible to add a method in interface without breaking the implementer class. Once you add a method in an interface, all its implemented classes had to define method body of this new method.
So, for backward compatibility, Java 8 introduced Default Methods.

A default method is a method declared and defined in an interface whose method header begins with the default keyword. Every class that implements the interface inherits the interface's default methods and can override them.

Onwards JAVA 8 - interface

interfaces are abstract in nature. i.e., they cannot be instantiated.
interface can contain

- members ("public", "static" and "final".)
- abstract methods ("public" and "abstract")
- default methods (by default public)
- static methods

child class of an interface has to provide implementation of the method/s which are declared abstract in parent or else child class also has to be declared as "abstract".

default method/s may or may not be overridden by child class. (If overridden, "public" modifier is compulsory)

static methods are like utility methods which can be invoked only on the interface in which they are defined.

what is the difference bet', interface and abstract class in java8?
java8 interface can have method implementation (default methods) and abstract methods.
abstract class can have state (instance members), constructors which interface of even java8 cannot have.

Why can’t a lambda expression work with non-functional interface?
A lambda expression can only be used for a "functional" interface - one that has only one non-default method.
Basically, lambda expressions are a shorthand for single blocks of code.
A lambda expression is an anonymous function. Simply put, it’s a method without a declaration, i.e., access modifier, return value declaration, and name.

Functional Interface: An interface with exactly one abstract method is called Functional Interface. It may have static and default methods. "functional interface" was known as "SAM (Single Abstract Method interface" before Java8.
Note: if we have an abstract class with just one abstract method, lambda does not work. Lambda works only with Functional Interface

When to use Method Reference?
Sometimes lambda does nothing but invokes existing method. In that case we can use "Method Reference" which is more compact than lambda expression. That also means that "Method Reference" can work with "functional interface" only.
Imp rule:
abstract method's argument and the method which you invoke from the interface method must match with the arguments.

###

DAY 14

###

Stream in Java

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure; they only provide the result as per the pipelined methods.
- Each intermediate operation returns a stream as a result, hence various intermediate operations can be pipelined.
- Terminal operations mark the end of the stream and return the result.
- We can use Stream API to implement internal iteration
- Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc

Collection API VS Stream API

- A collection is a data structure that holds elements. Each element is computed before it actually becomes a part of that collection.
- On the other hand, a stream is not a data structure. A stream is a pipeline of operations that compute the elements on-demand. Though we can create a stream from a collection and apply a number of operations, the original collection doesn't change.

```
values.stream().sorted().count();  // Sequential stream
values.parallelStream().sorted().count(); // Parallel stream
UUID uuid = UUID.randomUUID();      // Generate random numbers
TimeUnit.NANOSECONDS.toMillis(t1 - t0); // converts System.nanoTime() to millis
```

Changes in MAP (java 8 onwards)

```
map.putIfAbsent(i, "val" + i);  // it prevents us from writing additional and if null checks then write the value (default method inside map)
map.forEach((id, val) -> System.out.println(val)); // forEach accepts a consumer to perform operations for each value of the map.

map.computeIfPresent(3, (num, val) -> val + num);  //concatenation where num is key & val is value
map.computeIfAbsent(23, num -> "val" + num);     //if key is not there then it will be created with value as given lambda expres.
map.replace(3, null);

System.out.println(map.getOrDefault(42, "not found"));  // if key  is there inside then return the value and if it is not there then
System.out.println(map.getOrDefault(4, "not found"));    // it will return not found
System.out.println(map.merge(9, "concat", (value, newValue) -> value.concat(newValue))); // merge  old value with new value
                                                                  // if key is not there then new entry is put with new given value
```

##

DAY 12 - Socket

##

import java.net.*;

1. UDP Client-Server

2. TCP Client-Server

3. Object passing over network

Fundamental concept of client-server

1) Server should be ready
2) client must know name (IP address) of the server and port no. on which required application is running.
3) how the data travels from Client to Server and vice-versa?

Data gets converted into Packets [marshalling]

a) Packets are delivered on the other side of network.
b) Packets get converted into Data. [unmarshalling]

What is port no?
on Server we have different applications running e.g., tomcat server, oracle server, WebLogic server etc.
These applications are differentiated from each other with the help of port no.
So, when we say Client would like to connect to the server, it is for a particular application (tomcat server, oracle server etc).
And that client can specify with the help of port no.

Layers of Network:

1. Application Layer – HTTP, FTP, SMTP, JRMP (Java Remote Method Protocol), IIOP (Internet Inter-ORB protocol)
2. Transport Layer (UDP, TCP) – how communication happens
3. Internet Layer (IP) – how data should go from 1 end to another end.
4. Physical Layer – wires, cables

TCP - Transmission Control Protocol
It is similar to telephone service. i.e., when sender sends the message, he gets to know whether receiver received the message or not.
It is reliable. i.e., it gives guarantee that all the packets will be delivered at the other end and that too in the order in which they are delivered. However, this reliability comes at the cost of performance.

UDP - User Datagram Protocol
it is similar to postal service where sender does not know whether message is received by receiver or not.
it is not reliable as TCP. i.e., it does not give you guarantee of delivery of packets. However, it is faster than TCP.

Note: In java when u write socket program you have to use port no.
Total range of port no.: 1 to 65535
out of that,
1 to 1023 = system applications.
1024 to 65535 = java application.

what does DatagramSocket's receive() method do?
receive() method makes server application to wait for client application to send the packet.

Pass object over network
here we have called **"Thread.sleep(200)"**
if we don't call "sleep" then client socket gets closed as soon as client code is terminated and server gets exception **"java.net.SocketException:** Connection reset" while reading object from client.
with "sleep" method what we achieve is that when client invokes **oos.writeObject(mylist)"** , server application gets sufficient time to read that "mylist". " This is because client socket gets closed only after "sleep(200)"

##

DAY 15 – NIO, Stream API with NIO, Java Mail

##

NIO (new I/p o/p)

import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

Differences of NIO (new I/p o/p) vs Traditional Java I/O

First main difference between the standard IO and NIO is, standard IO is based on streams and NIO is buffer oriented. Buffer oriented operations provide flexibility in handling data. In buffer-oriented NIO, data is first read into a buffer and then it is made available for processing. So, we can move back and forth in the buffer. But in case of streams, it is not possible.
Second main difference is, blocking and non-blocking IO operations. In case of streams, a thread will be blocked until it completes the IO operation. Wherein the NIO allows for non-blocking operations. If the data is not available for IO operations, then the thread can do something else and need not stay in blocked mode.

How does Non-Blocking I/O Work?

Very simple to understand. If we want file operations to run in parallel mode, what is the solution in Java? Answer is Threads. This is exactly how it works. Another thread (Daemon Thread) is spawn for the file operation and it continues doing the job we initiated, while main thread goes ahead doing next tasks.

NIO Buffers

Buffers are a cornerstone upon which the NIO operations are built. Basically, all operations that involve NIO use buffers as a staging area for transferring data in and out from the data source to/from target. In the NIO library, data travels into buffers and out of buffers on a regular basis. Anytime you write data, you are writing data into a buffer and when you read data you are reading from a buffer.

NIO Channels

Channels are similar to Streams in traditional Java I/O API with the exceptions that they can provide three modes: input, output or bi-directional.
Streams on the other hand, are uni-directional (you were either using InputStream or OutputStream).
Channels interact directly with buffers and the native IO source, that is, file, or socket.

Channels Working:

for reading purpose:

1. channel writes data into buffer
2. program reads from buffer.

for writing purpose:

1. program writes data into buffer
2. channel reads from buffer.

```
Path path = Paths.get("d:\\FileDemo.java"); // creates path to read from file
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.READ); // created channel in read mode
ByteBuffer buffer = ByteBuffer.allocate(1024);  // creates buffer with given bytes of size


fileChannel.read(  buffer, 0, null, new CompletionHandler<Integer, ByteBuffer>()
{
  public void completed(Integer result, ByteBuffer attachment) {}
  public void failed(Throwable exc, ByteBuffer attachment) {}

});  // due to completion handler if I/O operation completes successfully then completed method is called else failed method is called.
```

Stream API with NIO

1. to get the list of paths from file

```
Stream<Path> stream = Files.list(Paths.get("e:\\temp"))
stream.map(String::valueOf).filter(path -> path.endsWith(".java")).forEach(System.out::println);
```

2. find the directory and get the paths

```
Stream<Path> stream = Files.find(Paths.get("e:\\temp"),3, (path, attr) -> String.valueOf(path).endsWith(".java"))
stream.map(String::valueOf).forEach(System.out::println);
```

3. reads the text lines from text/java file and show on the upper or lower case.

```
Stream<String> stream = Files.lines(Paths.get("e:\\temp\\Example2.java"))) {
stream.map(String::toLowerCase).forEach(System.out::println);
```

Optional Class

Java 8 introduced a new type called `Optional<T>` to help developers deal with null values properly.

What is Optional?

Optional *is a container type for a value which may be absent.*

Optional class enforces programmer to check Null-Pointer exception.

Optional class methods-

ofNullable( )
ifPresent( )
filter( )
map( )
flatmap( )
orElse( )
isPresent( )

1).offNullable :-

```
Optional<MyClass> op=Optional.ofNullable(getMyClass());
```

Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.

That means in our example, if "getMyClass()" returns an instance of "MyClass", ofNullable will return "Optional<MyClass>".

and if "getMyClass()" returns null, ofNullable will return "Optional.empty" i.e. empty Optional.

2). ifPresent :-

```
Optional<MyClass> op=Optional.ofNullable(getMyClass());

op.ifPresent(k->System.out.println(k.disp()));
```

If a value is present, invoke the specified consumer with the value, otherwise do nothing.

it means that if getMyClass() method returns instance of MyClass, then

```
op.ifPresent(k->System.out.println(k.disp()));
```

"k" will represent that instance and we can invoke "k.disp()".

3). filter :-

```
System.out.println(op.filter(s->s.equalsIgnoreCase("sachin")));
                              returns "Optional<Sachin>"

op.filter(s->s.equalsIgnoreCase("rahul"))
                           returns "Optional.empty"

System.out.println(op.filter(s->s.equalsIgnoreCase("rahul")).isPresent());
```

If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.

4). Map :-

System.out.println(op.map(MyClass::disp).orElse("could not get MyClass instance"));

If a value is present(ref not null), apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result.

Otherwise if a value is not present, returns an empty Optional i.e. Optional.empty

5). orElse :-

Optional<MyClass> op=Optional.ofNullable(getMyClass());

System.out.println(op.map(MyClass::disp).orElse("could not get MyClass instance"));

Return the value if present, otherwise return other.

Explanation:
here if getMyClass() returns instance of MyClass,
op will have reference for that instance.

op.map(MyClass::disp)
will invoke "disp" on that instance of MyClass and return "Optional<String>
now you are invoking "orElse("could not get MyClass instance")" on "Optional<String>"

since Optional<String> has a value "in disp of MyClass", it will be returned by "orElse()" method.

if getMyClass() returns null,
op will be empty.

op.map(MyClass::disp)
will not invoke "disp" and return "Optional.empty".
now you are invoking "orElse("could not get MyClass instance")" on "Optional.empty".

since Optional.empty does not have a value , "orElse()" method will return "could not get MyClass instance".

6). isPresent :-

Optional<MyClass> op=Optional.ofNullable(getMyClass());

boolean isPresent()

Return true if there is a value present, otherwise false.

so if getMyClass() returns instance of MyClass, then "op.isPresent()" will return true and if getMyClass() returns null then "op.isPresent()" will return false.

Program-

```
import java.util.Optional;

class MyClass
{
                    public String disp()
                    {
                      return "in disp of MyClass";
                    }
}
public class Demo {
                    private static MyClass ref=new MyClass();
                    // private static MyClass ref;

                    public static MyClass getMyClass()
                    {
                      return ref;
                    }
                    public static void main(String[] args)
                    {

                      Optional<MyClass> op=Optional.ofNullable(getMyClass());
                      System.out.println(op.map(MyClass::disp).orElse("could not get MyClass instance"));
                      //System.out.println(op.map(MyClass::disp));

                    }

}
```