* What is Java ?

→ Java is a software platform consisting of a development kit (JDK) for building cross platform applications and a runtime enviornment (JRE) for safely executing the binaries of those applications on any platform where it is pre-installed.

It includes support for

① Java programming Language

② Java Virtual Machine
   A. class loader
   B. Hot spot Engine
   C. Garbage Collector
   D. JVM Thread Scheduler

③ Java Class Library.
   A. Runtime
   B. platform

* Difference between C++ & Java.

C++	JAVA
- support both procedural & oop programming model	- only support object oriented programming model.
- platform dependent write once compile Anywhere	- platform independant Write once run Anywhere.
- Support features like operator overloading, Goto, structures, pointers unions etc.	- Does not support features like operator overloading, Goto statements, structures, unions, pointers etc
- Memory Management is manual.	- Memory Management done automatically by Garbage Collector.

OOP (Java & C++)

* Class:

A class in the context of Java is a template used to create objects and to define object data types and methods.

A class is a logical entity

```
class className{
...
}
```

ex. Animal is a class.

* Object:

An object is a real time entity which has its own state and behaviour.

```
ClassName Objectname = new className ();
```

ex. Tiger is a object of Animal class.

object has three characteristics.

1. State
2. Behavior
3. Indentity

An object is a instance of a class.

* There are three different types of variables.

1. Local Variables :

A local variable is a variable declared inside a method body, block or constructor.

Only accessible from where it is created.

2. Instance Variable :

A variable which is created inside the class but outside the method.

Gets memory at runtime when object or instance is created.

They are destroyed when the object is destroyed.

3. Class Variable (Static Variables) :

When a variable declared as a static, then a single copy of the variable is created and shared among all objects of the class level.

Static variables are essentially global variables.

A class variable is accessible from an object instance, while an instance variable is not accessible from a class method.

* Method in Java :

In java, method is like function which is used to expose the behavior of an object.

+ Code Reusability
+ Code Optimization

* new keyword in Java ?

In Java, the new keyword is used to create an instance of a class.

It allocates memory for object at runtime. and returning reference to that memory.

* There are 3 ways to initialize object.

1. By Reference Variable :

```
class Student{
    int id;
}
psvm(String args[]){
    Student s1 = new Student();
    s1.id = 101  //initializing object
```

2. By Method :

```
void insertRecord(int r){
    rollno = r;
}
s1.insertRecord(101); //initializing object
```

3. By Constructor :

* Anonymous Object:

An object which has no reference variable is called anonymous object in Java.

When you want to create only one object in a class then anonymous object is a good approach.

* Access Modifiers:

① public: Visible anywhere

② private: visible only in the class, can not access outside class.

③ protected: Accessible in package & outside package by subclass only.

④ Default: Access only within package.

* Format specifiers:

* Method:

A method is a collection of statements that performs some specific task and return result to the caller. (use for exposing object)

* void:

return type of method which returns nothing.

* main:

Represents starting point of program.

* main method:

```
public static void main (String [] args) {
    ...
    ...        scope starts here ↑
}
↑ scope ends here
```

public
- It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

static
- The main method is static so ~~that~~ making it class related method and JVM can invoke it without instantiating the class.

Saves memory which would have been used by the object declared only for calling the main() method by JVM.

- As main() method doesn't return anything, it's return type is void.
  As soon as main() method terminates, the java program terminates too.
  Hence, it doesn't make any sense to return from the main() method as JVM can't do anything with return value of it.

void

- It is the name of main() method. It is the identifier that the JVM looks for as the starting point of the Java program.
  It's not a keyword.

main

- It stores Java command-line arguments and is an array of type java.lang.String class.
  Here, the name of ~~the~~ the String array is args but it is not fixed and the user can use any name in place of it.

String[] args

* Can main() method be int ? if not, why ?

→ Java does not return int implicitly, even if we declare the return type of main as int.
We will get compile time error.
Even if we do return 0 or integer explicitly ourselves, from int main still we will get runtime error.
But JVM is a process of OS, and JVM can be terminated with a certain exit status. with help of java.lang.Runtime.exit(int status) or system.exit(int status).

Explaination:

Exit Code 0 successful termination
Exit Code 1 indicates an error.

The c and c++ programs which returns int from main are processes of operating system. The int value return from main in c & c++ is exit code or exit status. The exit code illustrates, why program was terminted.
ex. exit code 1 depicts miscellaneous error such as "divide by zero".
The parent process of any child process keeps waiting for the exit status of the child. and after receiving the exit status of the child, cleans up the child process from the process table and frees the resources allocates to it.
This is why it becomes mandatory for c & c++ programs to pass exit status from main explicity or implicitly.
However, the Java program runs as a 'main thread' in JVM. Java program is not even a process of os directy. There is no direct, inter-action between the Java program & OS. There is no direct ~~interaction between~~ allocation of (memory) resources to the Java program directly, or Java program does not occupy any place in process table.
This is why the main() method of Java is designed not to return an int or exit status.

* Object Identity:

Two objects are considered to be identical if they refer to the same instance in the memory.

~~x==y~~ x == y

* Object Equality:

Two objects are considered to be equal if they refer to instances of a same class with matching state in memory.

x.hashCode() == y.hashCode() && x.equals(y)

* 4 Important pillars of oop concepts.

① Data Abstraction

Data Abstraction is defined as the process of identifying only the required characterstics of an object ignoring the irrelevant details.

The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying / grouping the objects.

Abstraction can achieved by interfaces & Abstract classes.

* Abstract class:

An abstract class is a class that is declared with abstract keyword.

- An abstract class may or may not have all abstract methods. Some of them can be concrete methods.
- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- An abstract class can not be directly instantiated with new operator.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

* Abstract Method:

An abstract method is a method that is declared without implementation.

- A method ~~declared~~ defined abstract must always be redefined in the subclass, thus making overriding compulsory or either make the subclass itself abstract.

Note: - Abstract classes can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

* Interface:

The interface in Java is a mechanism to achieve abstraction.

An Interface in Java is defined as an abstract type used to specify behavior of a class.

An Interface in Java is a blueprint of a class.

- Contains static constants and abstract methods.
- There can be only abstract methods in the Java interface, not the method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- Java interface represents the Is-A relationship.

Note: If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

- A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

- To declare Interface use interface keyword.
- All methods in interface are declared with an empty body and are public and all fields are public, static, and final by default.

To implement interface use implements keyword.

* Interfaces are used to implement abstraction. so why use interfaces when we have abstract classes?

→ The reason is, abstract classes may contain non-final variables, whereas variables in the interfaces are final, public and static.

- We can't create an instance of the interface but we can make the reference of it that refers to the object of its implementing class.
- A class can implement more than one interface.

- Abstract class or Interface is used to achieve loose coupling.

Note: Abstract classes can not have final methods because when you make a method final then you can not override it but the abstract methods are meant for overriding.

* Functional Interface:

A functional interface in Java is an interface that contains only one (a single) abstract method.

② Inheritance :

It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class.

The keyword used for inheritance ~~is~~ is extends.

Syntax:

```
class Bycycle {                    // superclass
    ...
}

class MountainBike extends Bycycle {    // subclass.
    ...
}
```

- When an object of MountainBike class is created a copy of all methods and fields of the class - superclass acquire memory in this object.

That is why by using the object of the subclass we can also access the members of a superclass.

Note: During inheritance only the object of the subclass is created, not the superclass.

- Inheritance and polymorphism are used together in Java to achieve fast performance and readability of code.

* Super class :

The class whoose features are inherited is known as superclass.

* Sub class :

The class that inherits the other class is known as sub class.

The subclass can add it's own fields and methods in addition to the superclass fields and methods.

* Reusability :

Inheritance supports the concept of reusability, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can extend our new class from the existing class.

By doing this, we are reusing the fields and methods of the existing class.

* Driver class :

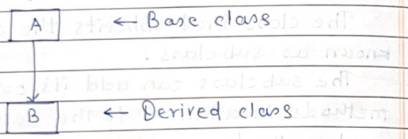
Driver class is a class which contains main() method.

```
class Test {
    public static void main(String args[]){
        ....
    }
}
```

* Types of Inheritances in Java : (C++)

(A) Single Inheritance :

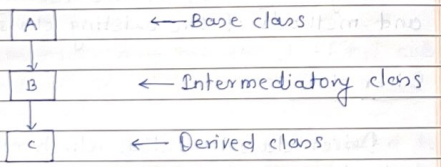
In single inheritance, subclasses inherit the features of only one class.



class A serves as superclass for subclass B.

(B) Multilevel Inheritance :

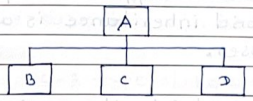
In multilevel inheritance, a derived class will be inheriting a base class as well as the derived class also act as the base class to other class.



class A serves as a base class for the derived class B which in turns serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

(C) Hierarchical Inheritance :

In hierarchical inheritance, one class serves as a superclass for more than one sub-class.



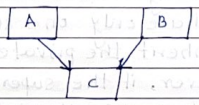
class A serves as a base class for derived class B, C and D.

(D) Multiple Inheritance :

In multiple interfaces, one class can have more than one superclass and inherit features from all parent classes.

Note: Java does not support multiple inheritances with classes.

In Java, we can achieve multiple inheritances only through Interfaces.



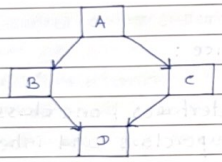
class C is derived from interface A & B.

(E) Hybrid Inheritance :

It is a mix of two or more of above types of inheritance.

Since, Java doesn't support multiple inheritance with classes, Hybrid inheritance is also not possible with classes.

Note: We can achieve hybrid inheritance in Java, only through interfaces.



* Default Superclass :

Except Object class, which has no superclass every class has one and only one direct superclass.

- In the absence of any other explicit superclass, every class is implicitly a subclass of the object class.
- A superclass can have any number of subclasses. But a subclass can have only one superclass.
- A subclass does not inherit the private members of it's parent class. However, if the superclass has public or protected methods (getters & setters) for accessing it's private fields, these can also be used by the subclass.

* Inheriting Constructors :

A subclass does ~~not~~ inherits all the members (fields, methods and nested classes) from it's superclasses.

Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

* Java Is-A type of Relationship :

This object is a type of that object.

Note: We can write a subclass constructor that invokes the constructor of the superclass, either implicitely or by using keyword super.

* Super() :

super() in Java is a reference variable that is used to refer superclass constructor.

If a constructor does not explicitly invoke a superclass constructor by using super(), then the Java compiler automatically inserts a call to the no-argument constructor of the superclass.

③ Data Encapsulation :

Encapsulation is defined as the wrapping of data under a single unit.

It is a mechanism that binds together code and the data it manipulates.

Encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.

- # Technically in encapsulation, the data in a class is hidden from other classes using data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a combination of data-hiding and abstraction.

- Technically, the variables or data of a class is hidden from any other class and can be accessed only through any member function of it's own class in which it ~~de~~ is declared.

- Encapsulation can be achieved by declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

* Advantages of Data Encapsulation:

Ⓐ Data Hiding:

The user will ~~no~~ have no idea about the inner implementation of the class.

She/He will only passing the values to a setter ~~and~~ method and variables are getting initialized with that value.

Ⓑ Increased flexibility:

We can make the variables of the class read-only or write-only depending on our requirement.

Ⓒ Reusability:

Encapsulation also improves the re-usability and is easy to change with new requirements.

Ⓓ Testing code is easy:

Encapsulated code is easy to test for unit testing.

Java Bean class is the example of fully encapsulated class.

④ Polymorphism :

The word polymorphism means having many forms.

Polymorphism allows us to perform a single action in diffrent ways.

Technically, Polymorphism allows you to define one interface and have multiple implementations.

It is popularly represented by the butterfly, which morphs from larva to pupa to imago.

* Types of polymorphism :

Ⓐ Compile-time polymorphism :

- It is also known as static polymorphism. or early binding.

Compile-time polymorphism is a polymorphism that is resolved during compilation process.

- Overloading of methods is called through the reference variable of a class.
- Compile-time polymorphism is achieved by method overloading and operator overloading.

* Method overloading :

Method overloading occurs when a class has many methods with the same name but diffrent parameters.

Ⓐ Method overloading by changing the number of parameters :

In this type, Method overloading is done by overloading methods in the function call with a varied number of parameters.

```
show(String a)
show(String a, String b)
```

Ⓑ Method overloading by changing Datatype of parameter :

In this type, Method overloading is done by overloading methods in the function call with a diffrent types of parameters. (Datatypes)

```
show(float a, float b)
show(int a, int b)
```

When a function is called, the compiler looks at the data type of input parameters and decides how to resolve the method call.

© By changing the sequence of parameters:

In this type, overloading is dependant on the sequence of the parameters.

```
show(int a , float b)
show(float a, int b)
```

* Invalid cases of method overloading:

Method overloading does not allow changing the return type of method (function). It occurs ambiguity.

```
int sum (int, int);
String sum (int, int);
```

Because the arguments are matching, the code above will not compile. Both methods have the same amount of data types and the same sequence of data types in the parameters.

* Operator Overloading:

An operator is said to be overloaded if it can be used to perform more than one function. Operator overloading is an overloading method in which an existing operator is given a new meaning.

Note: In Java, the '+' operator is overloaded. Java, on the other hand, does not allow for user-defined operator overloading.

To add integers, the + operator can be employed as an airthmetic addition operator. It can also be used to join strings together.

* Advantages of compile-time polymorphism:

- It improves code clearity and allows for the use of single name for similar procedures.
- It has a faster execution time since it discovered early in the compilation process.

Note: The only disadvantage of compile-time polymorphism is that it doesn't include inheritance.

(B) Runtime polymorphism or Dynamic Method Dispatch:

It is a process in which a function call to the overridden method is resolved at Runtime.

- This type of polymorphism is achieved by method overriding.

- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclass) of that method is to be executed based upon the type of the object being refered to at the time the call occurs.
  Thus, this determination is made at runtime.

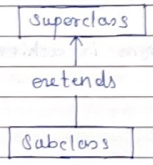
- At runtime, it depends on the type of the object being referred to (not the type of reference variable) that determines which version of an overriden method will be executed.

* Upcasting:

  A superclass reference variable can refer to a subclass object.
  This is also known as upcasting.

- Java uses this fact to resolve calls to overridden methods at runtime.

Superclass obj = new Subclass



Note: In Java, we can override methods only, not the variables (data members), so runtime polymorphism cannot be achieved by data members.

* Advantages of Dynamic Method Dispatch:

- Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
- It allows a class to specify methods that will be common to all of it's derivaties, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allow subclasses to add it's specific methods subclasses to define the specific implementations of some.

* Method Overriding:

  In any ~~prog~~ oop language,
  Overriding is a feature that allows a subclass or child class to provide a specific implementations of a method that is already provided by one of it's super classes or parent-classes.

When a method in a subclass has the same name, same parameters or signature, and same return type as a method in it's super-class, then the method in the subclass is said to override the method in the superclass.

* Rules for method overriding:

① Overriding and Access-Modifiers:
The access-modifier for an overriding method can allow more, but not less, access than the overridden method.
For example, a protected instance method in the super class can be made public, but not private, in the subclass. Doing so genrate compile-time error.

② Final methods can not be overridden.

③ Static methods can not be overridden:
When you define a static method with same signature as a static method in base class, it is known as method hiding.

④ private methods can not be overridden:
private methods cannot be overridden as they are bonded during compile-time.

Therefore we can't even override private methods in a subclass.

⑤ The overriding method must have same return type (or subtype):
From Java 5.0 onwards it is possible to have diffrent return type for an overriding method in child class. but child's return type should be sub-type of parent's return type.
This phenomena is called as "covarient return type"

⑥ Invoking overridden method from sub-class:
We can call parent class method in overriding method using 'super' keyword.

⑦ Overriding and Constructor:
We can not override constructor as parent and child class can never have constructor with same name.

Note: Constructor name must alway be same as class name.

⑧ Overriding & Exception-Handling:
ⓐ If superclass overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.
ⓑ If the superclass overridden method does not throws an exception, subclass's overridding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error.
There is no issue if subclass overriden method is not throwing any exception.

⑨ Overridding and Abstract Method:

Abstract methods in an interface or Abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

⑩ Synchronized / strictfp Method:

It's possible that a synchronized / strictfp method can override a non synchronized / strictfp one or vice-versa.

Note: ① In C++, we need 'virtual' keyword to achieve overridding or Run-time polymorphism.

In Java, methods are virtual by default.

② We can have multilevel method-overridding.

* Why method overridding ?

→ It allows a genral class to specify methods that will be common to all of it's derivaties while allowing subclasses to define the specific implementations of some or all of those methods.

Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

* Data Types in Java:

* Statically typed language:

where each variable and expression type is already known at compile time.

Once a variable declared to be of a certain data type, it cannot hold values of other data types.

ex. C, C++, Java

* Dynamically typed language:

These languages can receive diffrent data types over time

ex. Python, Ruby.

Note: Java is a statically typed and also a strongly typed language

Ⓐ Primitive Data Type:

In Java, the primitive data types are the predefined data types of Java.

Java has 8 primitive data types

① byte ⑤ float
② short ⑥ double
③ int ⑦ char
④ long ⑧ boolean

(b) Non-primitive or Reference or object Data types:

In Java, the user defined data types are called as Non-primitive or reference data types.

- The reference data types will contain a memory address of variable values because the reference types won't store the variable value directly in the memory.
  ex: Strings, arrays, classes, interfaces. etc.

- When an reference variables will be stored, the variable will be stored in the 'stack' and the original object will be stored in the 'heap.'

- In object data type, although two copies will be created but they both will point to the same variable in the heap.

- Hence, changes made to any variable will reflect the change in both the variables.

* Wrapper classes:

A wrapper class is a class whose ~~wraps~~ object wraps or contain primitive data types. When we create an object ~~of~~ to a wrapper class, it contains a field and in this field, we can store primitive data types.

In other words, we can wrap a primitive value into a wrapper class object.

* Need of wrapper classes:

- They convert primitive data types into objects are needed if we wish to modify the arguments ~~passing~~ passed into a method.
  Because primitive types are passed by value.

- The classes in java.util packages handles by only objects and hence wrapper classes help in this case.

- Data structures in the collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

- An object is needed to support synchronization in multithreading.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

In Java, primitive data types are treated differently so do there comes introduction of wrapper classes.
Where two components play a role.

① Autoboxing:

Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.

for example,
Converting int to Integer class.

The Java compiler applies autoboxing when a primitive value is:
- Passed as a parameter to a method that expects an object of corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

② Unboxing:

Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.

for example,
Conversion of Integer to int.

The Java compiler applies to unbox when an object of wrapper class is:
- Passed as a parameter to a method that expects a value of the corresponding primitive type.
- Assigned to a variable of the corresponding primitive type.

```
char ch = 'a';
Character a = ch;  // Autoboxing

Character ch = 'a';
char a = ch;  // unboxing
```

* Advantages:
- Autoboxing and Unboxing lets developers write cleaner code, making it easier to read.
- The technique lets us use primitive types and wrapper class objects interchangeably and we do not need to perform any typecasting explicitly.

* Type Conversion:

When you assign a value of one data type to another, the two datatype if compatatible, then Java will perform the conversion automatically known as Automatic Type Conversion.
If not, they need to be cast or converted explicitly.
for example, Assigning an int value to a long variable.

① Widening or Automatic Type Conversion :

Widening conversion takes place when two data types are automatically converted.

This happens when :
- The two data types are compatible.
- When we assign a value of a smaller data type to a bigger data type.

Byte → Short → int → long → float → double

② Narrowing or Explicit Conversion:

If we want to assign a value of a larger data type to a smaller dtype we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, the target type specifies the desired type to convert the specified value to.

double → float → long → int → short → byte.

Note: Typecasting is the assessment of the value of one primitive data type to another type.

* class Type Casting :

Ⓐ Upcasting :

Upcasting is a casting a subtype to a super-type in an upward direction to the inheritance tree.

Upcasting is the typecasting of a child object to a parent object.

- A sub-class object is reffered by a super-class reference variable.
- One can relate with dynamic polymorphism.

Implicit casting means class typecasting ~~can be~~ done by compiler without cast syntax.

Explicit Casting means class typecasting done by the programmer with cast syntax.

Ⓑ Downcasting :

Downcasting refers to the procedure when sub-class type refers to the object of the parent class is known as downcasting. (parent object → Child object)

- If it is performed directly compiler gives an error as ClassCastException is thrown at runtime.

- It is only achievable with the use of instanceof operator.

- The object which is already upcast, that object only can be performed downcast.

* Rules to perform class type casting:

① Classes must be "IS-A-Relationship"

② An object must have the property of a class in which it is going to cast.

* Instanceof keyword:

Instanceof is a keyword that is used for checking if a reference variable is containing a given type of object reference or not.

The instanceof keyword checks whether an object is an instance of a specific class or an interface.

Note: Implementation of the instanceof operator returns a boolean if the object parameter (which can be a expression) is an instance of a class type.

It returns true or false. In java, we do not have 0 and 1 boolean return type.

* Packages in Java:

Package in Java is a mechanism to encapsulate a group of classes, sub ~~classes~~ packages and interfaces.

Used for:
- Preventing naming conflicts.
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier.
- Providing controlled access:
  protected and default have packages level access control.
- Packages can be considered as data encapsulation.

* Types of packages:

(A) Built-In Packages:

These packages consist of a large number of classes which are a part of Java API.

Commonly use packages (Built in)

① java.lang:
contains language support classes. classes which defines primitive data types, math operations.

② java.io:
contains classes for supporting input/output operations.

③ java.util:
contains utility classes which implement data structures like Linked List, Dictonary and support for date/time operations.

④ java.applet:
contains classes for creating Applets.

⑤ java.awt:
contains classes for implementing the components for graphical user interfaces.

⑥ java.net:
contains classes for supporting networking operations.

⑧ User-defined Packages:

These are the packages that are defined by the user.

Note: Every class is part of some packages.

We can access public classes in another (named) package using
package-name.class-name.

* Strings in Java:

String in Java are Objects that are backed internally by a char array.

Since arrays are immutable (cannot grow), Strings are immutable as well.
Whenever a change to a String is made, an entirely new string is created.

String str = "Avengers";

Memory allotment of String:
Whenever a string Object is created as a literal, the object will be created in String constant pool.
This allows JVM to optimize the initialization of String literal.

- The string can also be declared using new operator. i.e. dynamically allocated.
- In case of String dynamically allocated they are assigned to a new memory location in heap.
- This string will ~~be~~ not be added to string constant pool.
- If you want to store this string in the constant pool then you will need to "intern" it.

String str = new String("Endgame");
String internString = str.intern();

* Interfaces and classes in strings in Java:

① charBuffer:

This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of charSequence.

An example of such usage is the regular-expression package java.util.regex.

② StringBuffer:

StringBuffer is a peer class of String that provides much of the functionality of strings.

- StringBuffer represents growable & writable charcter sequences.

```
StringBuffer s = new StringBuffer("Endgame");
```

③ StringBuilder:

The StringBuilder in Java represents a mutable sequence of character.

- Alternative to string class.

```
StringBuilder str = new StringBuilder();
str.append("Avenger");
```

④ StringJoiner:

StringJoiner is a class in java.util package which is used to construct a sequence of characters (strings) separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix.

Note: It is preferred to use String literals as it allows JVM to optimize memory allocation.

All object in Java are stored in heap. The reference variable is to the object stored in the stack area or they can be contained in other objects which puts them in the heap area also.

⑤ String class:

String is a character of sequence.

There are two ways of creating string.

① String Literal:

```
String s = "Avengers Endgame";
```

② Using new keyword:

```
String s = new string ("Avengers Endgame");
```

* Constructors:

A constructor in Java is a special method that is used to initilize objects.

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created.

Note: Every time an object is created using the new() keyword, at ~~last~~ least one constructor is called.

It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

- Constructors must have the same name as the class within which it is defined.
- Constructors do not return any type.
- Constructors ~~only~~ are called only once at the time of object creation.

* When is a constructor called?

→ Each time an object is created using a new() keyword, at least one constructor (it could be default also) is invoked to assign initial values to the data members of the same class.

* Rules for writing constructors:

- Constructor of a class must have the same name as the class name which it resides.
- A constructor in Java, can not be abstract, final, static or synchronized.
- Access modifiers can be used in constructor declaration to control it's access i.e. which other class can call the constructor.

* Types of constructors in Java:

(A) No-Argument Constructor: (Default Constructor)

A constructor that has no parameter is known as the default constructor.

If we don't define constructor in a class, then the compiler creates default constructor (with no arguments) for the class.

Note: Default constructor provides the default values to the object like 0, null, etc depending on the type.

(B) Parameterized Constructor:

A constructor that has parameters is known as parameterized constructor.

We use it when to initialize fields of the class with our own values.

* Does constructor return any value?

→ There are no "return value" statements in the constructor, but the constructor returns the current class instance.

We can write 'return' inside a constructor.

Note: Just like methods, we can overload constructors for creating objects in different ways.

Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters, and order of the parameters.

* Copy Constructor:

Like c++, Java also supports copy constructor. But unlike c++, Java doesn't create a default copy constructor if you don't write you own.

A copy constructor in Java class is a constructor that creates an object using another object of the same ~~class~~ Java class.

* Constructor chaining:

Constructor chaining is the process of calling one constructor from another construstor with respect to the current object.

* Constructor chaining can be done in two ways:

(A) Within same class:

It can be done using this() keyword for constructors in same class.

(B) From base class:

By using super() keyword to call constructor from the base class.

Constructor chaining occurs through inheritance. A subclass constructors task is to call super-class's constructor first. This ensures that creation of subclasses's object start with the initialization of the data members of the super class.

There could be any numbers of classes in the inheritance chain. Every constructor calls up the chain till class at the top is reached.

* Why do we need constructor chaining?

→ This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a ~~spe~~ seprate constructor for each task and make their chain which makes the program more readable.