Java Vs C++

Development wise differences

1. Java is platform independent language but c++ is dependent upon operating system.
At compilation time Java Source code(.java) converts into byte code(.class) .The interpreter translates this byte code at run time into native code and gives output.

2. Java uses both a compiler and interpreter, while C++ only uses a compiler

Syntactical differences

1. There is no final semi-colon at the end of the class definition.
2. Functions are called as methods.
3. main method is a member of class
& has a fixed form
public static void main(String[] args) -- argument is an array of String. This array contains the command-line arguments.
4. main method must be inside some class (there can be more than one main function -- there can even be one in every class)
5. Like the C++ << operator,

To write to standard output, you can use either of the following:

```
System.out.println( ... )
System.out.print( ... )
```

The former prints the given expression followed by a newline, while the latter just prints the given expression.

These functions can be used to print values of any type. eg :

```
System.out.print("hello");   // print a String
System.out.print(16);       // print an integer
System.out.print(5.5 * .2);  // print a floating-point number
```

The + operator can be useful when printing. It is overloaded to work on Strings as follows:
If either operand is a String, it
converts the other operand to a String (if necessary)
creates a new String by concatenating both operands .

Features wise differences.

1. C++ supports pointers whereas Java does not support pointer arithmetic. It supports Restricted pointers.
Java references (Restricted pointers) can't be arithmatically modified.

2. C++ supports operator overloading , multiple inheritance but java does not.

3. C++ is nearer to hardware than Java.

4. Everything (except fundamental or primitive types) is an object in Java (Single root hierarchy as everything gets derived from java.lang.Object).

Java is similar to C++ but it doesn't have the complicated aspects of C++, such as pointers, templates, unions, operator overloading, structures, etc. Java also does not support conditional compilation (#ifdef/#ifndef type).

Thread support is built into Java but not in C++. C++11, the most recent iteration of the C++ programming language, does have Thread support though.

Internet support is built into Java, but not in C++. On the other hand, C++ has support for socket programming which can be used.

Java does not support header files and library files. Java uses import to include different classes and methods.

Java does not support default arguments.

There is no scope resolution operator :: in Java. It has . using which we can qualify classes with the namespace they came from.

There is no goto statement in Java.

Because of the lack of destructors in Java, exception and auto garbage collector handling is different than C++.

Java has method overloading, but no operator overloading unlike C++.

The String class does use the + and += operators to concatenate strings and String expressions use automatic type conversion,

Java is pass-by-value.

Java does not support unsigned integers.

Why java doesn't support c++ copy constructor?
Java does. They're just not called implicitly like they are in C++ .

Firstly, a copy constructor is nothing more than:

```
public class Blah {
  private int foo;
```

```
  public Blah() { } // public no-args constructor
  public Blah(Blah b) { foo = b.foo; }  // copy constructor
}
```

Now C++ will implicitly call the copy constructor with a statement like this:

```
Blah b2 = b1;
```

Cloning/copying in that instance simply makes no sense in Java because all b1 and b2 are references and not value objects like they are in C++. In C++ that statement makes a copy of the object's state. In Java it simply copies the reference. The object's state is not copied so implicitly calling the copy constructor makes no sense.

All stand-alone C++ programs require a function named main and can have numerous other functions. Java does not have stand alone functions, all functions (called methods) are members of a class. All classes in Java ultimately inherit from the Object class, while it is possible to create inheritance trees that are completely unrelated to one another in C++. In this sense , Java is a pure Object oriented language, while C++ is a mixture of Object oriented and structure language.

The interface keyword in Java is used to create the equivalence of an abstract base class containing only method declarations and constants. No variable data members or method definitions are allowed(true till Java 8) . C++ does not support interface concept. Java does not support multiple inheritance. To some extent, the interface feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.(death of a diamond)

Java is running on a Virtual Machine, which can recollect unused memory to the operating system, so Java does not destructor. Unlike C++, Java cannot access pointers to do memory operation directly. This leads to a whole host of subtle and extremely important differences between Java and C++.

Furthermore, the C++ compiler does not check whether all local variables are initialized before they are read. It is quite easy to forget initializing a variable in C++. The value of the variable is then the random bit pattern that happened to be in the memory location that the local variable occupies.

Java does not have global functions and global data. Static in Java is just like global in C++, can be accessed through class name directly, and shared by all instances of the class. For C++, static data members must be defined out side of class definition, because they don't belong to any specific instance of the class.

Generally Java is more robust than C++ because:

Object handles (references) are automatically initialized to null.
Handles are checked before accessing, and exceptions are thrown in the event of problems.
You cannot access an array out of bounds.
Memory leaks are prevented by automatic garbage collection.

While C++ programmer clearly has more flexibility to create high efficient program, also more chance to encounter error.

1. byte: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). --- -2^7 ---- 2^7-1
2. short: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).
-2^15 ---- 2^15-1
3.int: The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).
4. long: The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive).

5.float: The float data type is a single-precision 32-bit IEEE 754 floating point.
Covers a range from 1.4012984632481707e-45 to 3.40282346638528860e+38 (positive or negative).
BE careful -- in assigning integer to float & vice versa.

6. double : 8 bytes IEEE 754. Covers a range from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative).

7. boolean
Typically 1-bit(as per underlying JVM specification) May take on the values true and false only.
true and false are defined constants of the language. Booleans may not be cast into any other type of variable nor may any other variable be cast into a boolean.

8. char -- unsigned char. --- UTF 16
range 0----65535
Operators in Java

Arithmetic Operators
Unary Operators
Assignment Operator
Relational Operators
Logical Operators
Ternary Operator
Bitwise Operators
Shift Operators

Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.
* : Multiplication
/ : Division
% : Modulo
+ : Addition

– : Subtraction

Unary Operators: Unary operators need only one operand. They are used to increment, decrement or negate a value.
– :Unary minus, used for negating the values.
eg : int a=20;
int b=-a;

++ :Increment operator, used for incrementing the value by 1. There are two varieties of increment operator.
Post-Increment : Value is first used for computing the result and then incremented.
Pre-Increment : Value is incremented first and then result is computed.
eg : int n1=10;
int n2=n1++;
System.out.println(n2+" "+n1);
What will be output ?

— : Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operator.
Post-decrement : Value is first used for computing the result and then decremented.
Pre-Decrement : Value is decremented first and then result is computed.

! : Logical not operator, used for inverting a boolean value.
eg :
boolean jobDone=true;
boolean flag=!jobDone;
System.out.println(flag);

Assignment Operator : '=' Assignment operator is used to assign a value to any variable. It has a right to left associativity.

eg : int a=200;

In many cases assignment operator can be combined with other operators to build a shorter version of statement called Compound Statement.

eg : int a=100;
a += 10;
System.out.println(a);

+=, for adding left operand with right operand and then assigning it to variable on the left.
-=, for subtracting left operand with right operand and then assigning it to variable on the left.
*=, for multiplying left operand with right operand and then assigning it to variable on the left.
/=, for dividing left operand with right operand and then assigning it to variable on the left.
%=, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

Relational Operators : These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are used in looping statements and conditional if else statements.

==, Equal to : returns true if left hand side is equal to right hand side.
!=, Not Equal to : returns true if left hand side is not equal to right hand side.
<, less than : returns true if left hand side is less than right hand side.
<=, less than or equal to : returns true if left hand side is less than or equal to right hand side.
>, Greater than : returns true if left hand side is greater than right hand side.
>=, Greater than or equal to: returns true if left hand side is greater than or equal to right hand side.

Logical Operators : These operators are used to perform “logical AND” and “logical OR” operation, i.e. the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for making a decision.

Conditional operators are-
&&, Logical AND : returns true when both conditions are true.
||, Logical OR : returns true if at least one condition is true.

eg :
```
int data=100;
int data2=50;
if(data > 60 && data2 < 100)
 System.out.println("test performed...");
else
 System.out.println("test not performed...");
```

Ternary operator : Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. General format is-

condition ? if true : if false
The above statement means that if the condition evaluates to true, then execute the statements after the ‘?’ else execute the statements after the ‘:’.

eg :
```
int data=100;
System.out.println(data>100?"Yes":"No");
```

Bitwise Operators : These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.
&, Bitwise AND operator: returns bit by bit AND of input values.

|, Bitwise OR operator: returns bit by bit OR of input values.
^, Bitwise XOR operator: returns bit by bit XOR of input values.
~, Bitwise Complement Operator: This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.

eg :

```
String binary[] = {
                    "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
                    "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
                   };
                   int a = 3; // 0 + 2 + 1 or 0011 in binary
                   int b = 6; // 4 + 2 + 0 or 0110 in binary
                   int c = a | b;
                   int d = a & b;
                   int e = a ^ b;


                   System.out.println("      a = " + binary[a]);
                   System.out.println("      b = " + binary[b]);
                   System.out.println("    a|b = " + binary[c]);
                   System.out.println("    a&b = " + binary[d]);
                   System.out.println("    a^b = " + binary[e]);

                  }
```

Shift Operators : These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.

<<, Left shift operator: shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
eg :

```
int a = 25;
System.out.println(a<<4); //25 * 16 = 400
a=-25;
System.out.println(a<<4);//-25 * 16 = -400
```

Signed right shift operator
The signed right shift operator '>>' uses the sign bit to fill the trailing positions. For example, if the number is positive then 0 will be used to fill the trailing positions and if the number is negative then 1 will be used to fill the trailing positions.

Assume if a = 60 and b = -60; now in binary format, they will be as follows −

a = 0000 0000 0000 0000 0000 0000 0011 1100

b = 1111 1111 1111 1111 1111 1111 1100 0100
In Java, negative numbers are stored as 2's complement.

Thus a >> 1 = 0000 0000 0000 0000 0000 0000 0001 1110
And b >> 1 = 1111 1111 1111 1111 1111 1111 1110 0010

Unsigned right shift operator
The unsigned right shift operator '>>' do not use the sign bit to fill the trailing positions. It always fills the trailing positions by 0s.

Thus a >>> 1 = 0000 0000 0000 0000 0000 0000 0001 1110
And b >>> 1 = 0111 1111 1111 1111 1111 1111 1110 0010

eg : D:\ACTS-2020\java11\test2\src\operators\Tester.java

The range of values that can be represented by a float or double is much larger than the range that can be represented by a long. Although one might lose significant digits when converting from a long to a float, it is still a "widening" operation because the range is wider.

From the Java Language Specification, §5.1.2:

A widening conversion of an int or a long value to float, or of a long value to double, may result in loss of precision - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode (§4.2.4).
Note that a double can exactly represent every possible int value.
What are the rules for naming variables in java?

Answer:
All variable names must begin with a letter of the alphabet, an underscore (_), or a dollar sign ($). Can't begin with a digit. The rest of the characters may be any of those previously mentioned plus the digits 0-9.

The convention is to always use a (lower case) letter of the alphabet. The dollar sign and the underscore are discouraged.
What is Unicode ?

The Unicode-characters are universal characters encoding standard. It represents way different characters can be represented in different documents like text file, web pages etc.

It is the industry standard designed to consistently and uniquely encode characters used in written languages throughout the world.

The Unicode standard uses hexadecimal to express a character.
For example the value 0x0041 represents A.

The ASCII character set contained limited number of characters. It doesn't have Japanese characters , can't support Devnagari scripts.

The idea behind Unicode was to create a single character set that included every reasonable character in all writing systems in the world.

The Unicode standard was initially designed using 16 bits to encode characters because the primary machines were 16-bit PCs. When the specification for the Java language was created, the Unicode standard was accepted and the char primitive was defined as a 16-bit data type, with characters in the hexadecimal range from 0x0000 to 0xFFFF.
What is JIT Compiler?

The Just In Time Compiler (JIT) concept and more generally adaptive optimization is well known concept in many languages besides Java (.Net, Lua, JRuby).

In order to explain what is JIT Compiler I want to start with a definition of compiler concept. According to wikipedia compiler is "a computer program that transforms the source language into another computer language (the target language)".

We are all familiar with static java compiler (javac) that compiles human readable .java files to a byte code that can be interpreted by JVM - .class files. Then what does JIT compile? The answer will given a moment later after explanation of what is "Just in Time".

According to most researches, 80% of execution time is spent in executing 20% of code. That would be great if there was a way to determine those 20% of code and to optimize them. That's exactly what JIT does - during runtime it gathers statistics, finds the "hot" code compiles it from JVM interpreted bytecode (that is stored in .class files) to a native code that is executed directly by Operating System and heavily optimizes it. Smallest compilation unit is single method. Compilation and statistics gathering is done in parallel to program execution by special threads. During statistics gathering the compiler makes hypotheses about code function and as the time passes tries to prove or to disprove them. If the hypothesis is dis-proven the code is deoptimized and recompiled again.

The name "Hotspot" of Sun (Oracle) JVM is chosen because of the ability of this Virtual Machine to find "hot" spots in code.

What optimizations does JIT?
Let's look closely at more optimizations done by JIT.

Inline methods - instead of calling method on an instance of the object it copies the method to caller code. The hot methods should be located as close to the caller as possible to prevent any overhead.
Eliminate locks if monitor is not reachable from other threads
Replace interface with direct method calls for method implemented only once to eliminate calling of virtual functions overhead
Join adjacent synchronized blocks on the same object
Eliminate dead code
Drop memory write for non-volatile variables
Remove prechecking NullPointerException and IndexOutOfBoundsException

When the Java VM invokes a Java method, it uses an invoker method as specified in the method block of the loaded class object. The Java VM has several invoker methods, for example, a different invoker is used if the method is synchronized or if it is a native method.The JIT compiler uses its own invoker. Sun production releases check the method access bit for value ACC_MACHINE_COMPILED to notify the interpreter that the code for this method has already been compiled and stored in the loaded class. JIT compiler compiles the method block into native code for this method and stores that in the code block for that method. Once the code has been compiled the ACC_MACHINE_COMPILED bit, which is used on the Sun platform, is set.
Regarding garbage collection

Garbage Collection is a process to identify and delete the objects from Heap memory which are not in use. GC frees the space after removing unreferenced objects.

Garbage= un -referencable object.

Automatic Gargabe Collection --- to avoid memory. leaks/holes

JVM creates 2 system thrds --- main thrd(to exec main() sequentially) -- foreground thrd
G.C --- daemon thrd ---background thrd --- JVM activates it periodically(only if required)
--- GC releases the memory occupied by un-referenced objects allocated on the heap(the object whose no. of ref=0)

How to request for GC ?
API of System class
public static void gc()
eg : System.gc();//it's simply a REQUEST to JVM , for running GC htread.

Object class API
protected void finalize() throws Throwable
Automatically called by the garbage collector on an object before garbage collection of the object takes place.
----------1st half over----------

Releasing of non- Java resources(eg - closing of DB connection, closing file handles,closing socket connections) is NOT done automatically by GC

Triggers for marking the object for GC(candidate for GC)

1. Nullifying all valid refs.
eg : Box b1=new Box(1,2,3);
Box b2=b1;
b1=b2=null;//Box obj is marked for GC
2. re-assigning the reference to another object
eg : Box b1=new Box(10,20,30);
b1=new Box(2,3,4);
3. Object created within a method & its ref NOT returned to the caller.

4. Island of isolation

More Details

Garbage Collection is a process to identify and delete the objects from Heap memory which are not in use. GC frees the space after removing unreferenced objects.

The event in which Garbage Collectors are doing their job is called “Stop the world” event which means all of your application threads are put on hold until the garbage is collected.

How Garbage Collector works

The basic process of Hotspot JVM Garbage collector completes in two phases:

1. Marking
This phase is called marking phase in which GC identifies which objects are in use or which are not. All objects are scanned in the marking phase to make this determination.

2. Deletion
In Deletion phase, the marked object is deleted and the memory is released. Deletion of the unreferenced objects can be done in two ways:

2.1 Normal Deletion: In this phase, all unused objects will be removed and memory allocator has pointers to free space where a new object can be allocated.
OR
2.2 Deletion and Compaction: As you see in normal deletion there are free blocks between referenced objects.
To further improve performance, in addition to deleting unreferenced objects, remaining referenced object will be compact.

Why Heap divided into Generations
It is a time consuming process to scan all of the objects from a whole heap and further mark and compact them.
The list of the object grows gradually which leads to longer garbage collection time as more and more objects are allocated with time.

In General Applications most of the objects are short-lived. Fewer and fewer objects remain allocated over time.

That’s why to enhance the performance of the JVM, Heap is broken up into smaller parts called generations and JVM performs GC in these generations when
the memory is about to fill up.

Generational Process of Garbage Collection

1. New objects are allocated in Eden Space of Young Generation. Both Survivor Spaces are empty in starting.

2. A minor garbage collection will trigger once the Eden space fills up.
Referenced objects are moved to the S0 survivor space and Eden Space will be cleared and all unreferenced objects will be deleted.
3. It will happen again to Eden space when next time GC will be triggered. But, in this case, all referenced objects are moved to S1 survivor space. In addition, objects from the last minor GC on the S0 survivor space have their age incremented and get moved to S1. Now both Eden and S0 will be cleared, and this process will repeat every time when GC is triggered. On every GC triggered, survivor spaces will be switched and object's age will be incremented.
4. Once the objects reach a certain age threshold, they are promoted from young generation to old generation. So, this is how objects promotion takes place.
5. The major GC will be triggered once the old generation completely fills up.

Available Garbage collectors in Hotspot JVM
1. Serial Garbage Collector: Serial GC designed for the single-threaded environments. It uses just a single thread to collect garbage.
It is best suited for simple command-line programs. Though it can be used on multiprocessors for applications with small data sets.

2. Parallel Garbage Collector: Unlike Serial GC it uses multiple threads for garbage collection.
It is a default collector of JVM and it is also called the Throughput garbage collector.

3. CMS(concurrent mark & sweep) Garbage Collector: CMS uses multiple threads at the same time to scan the heap memory and mark in the available for eviction
and then sweep the marked instances.

4. G1 Garbage Collector: G1 Garbage collector is also called the Garbage First. It is available since Java 7 and its long-term goal is to replace the CMS collector.
The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Class & Object

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

Class

A class is a user defined blueprint or prototype or template : from which objects are created.
It represents the set of properties or methods that are common to all objects of one type.

Class declaration includes

1. Access specifiers : A class can be public or has default access
2. Class name: The name should begin with a capital letter & then follow camel case convention
3. Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends.
(Implicit super class of all java classes is java.lang.Object)

4. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements.
A class can implement more than one interface.
eg : public class Emp extends Person implements Artist,Player{...}
5. Body: The class body surrounded by braces, { }.

6. Constructors are used for initializing new objects.
7. Fields are variables that provides the state of the class and its objects
8. Methods are used to implement the behavior of the class and its objects.

eg : Student,Employee,Flight,PurchaseOrder, Shape ,BankAccount.....

Object

It is a basic unit of Object Oriented Programming and represents the real life entities.
A typical Java program creates many objects, which interact by invoking methods.

An object consists of :

State : It is represented by attributes of an object. (properties of an object) / instance variables(non static)
Behavior : It is represented by methods of an object (actions upon data)
Identity : It gives a unique identity to an object and enables one object to interact with other objects.
eg : Emp id / Student PRN / Invoice No

Creating an object

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
The new operator also invokes the class constructor.

Constructor -- is a special method having
same name as the class name
no explicit return type
may be parameterized or parameter less.

Parameterized constructor is used initialize state of the object.

If a class does not explicitly declare any constr , the Java compiler automatically provides a no-argument constructor,called the default constructor.

This default constructor implicitely calls the super class's no-argument constructor

Revise "this" keyword
this => current object reference
Usages of this
1. To unhide , instance variables from method local variables.(to resolve the conflict)
eg : this.name=name;

2. To invoke the constructor , from another overloaded constructor in the same class.(constructor chaining , to avoid duplication)

Encapsulation in Java

Encapsulation is defined as the wrapping up of data & code under a single unit. It is the mechanism that binds together code and the data it manipulates.
It's is a protective shield that prevents the data from being accessed by the code outside this shield.

The variables or data of a class is hidden from any other class and can be accessed only through any member function/method of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Tight Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods as its accessors.

Advantages of Encapsulation:

1. Data Hiding (security)
2. Increased Flexibility: We can make the variables of the class as read-only or write only or r/w.
3. Reusability: Encapsulation also improves the re-usability and easy to change with new requirements.
4. Testing code is easy

Summary

Encapsulation -- consists of Data hiding + Abstraction

Information hiding -- achieved by private data members & supplying public accessors.

Abstraction -- achieved by supplying an interface to the Client (customer) .Highlighting only WHAT is to be done & not highlighting HOW it's internally implemented.

Regarding inheritance

In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy. The classes in the lower hierarchy inherit all the variables (attributes) and methods (dynamic behaviors) from the higher hierarchies.

A class in the lower hierarchy is called a subclass (or derived, child, extended class). A class in the upper hierarchy is called a superclass (or base, parent class).

By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, redundancy can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. Re usability is maximum.

A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). It can use the inherited methods and variables as they are. It may also override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same name.

Summary : Sub class IS-A super class , and something more (additional state + additional methods) and something modified(behaviour --- method overriding)

eg :
Person,Student,Faculty
Emp,Manager,SalesManager,HRManager,Worker,TempWorker
Shape, Circle,Rectangle,Cyllinder,Cuboid
BankAccount ,LoanAccount,HomeLoanAccount,VehicleLoanAccount
Student,GradStudent,PostGradStudent

Fruit -- Apple -- FujiApple

A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.

It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

Regarding inheritance

In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy. The classes in the lower hierarchy inherit all the variables (attributes) and methods (dynamic behaviors) from the higher hierarchies.

A class in the lower hierarchy is called a subclass (or derived, child, extended class). A class in the upper hierarchy is called a superclass (or base, parent class).

By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, redundancy can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. Re usability is maximum.

A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). It can use the inherited methods and variables as they are. It may also

override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same name.

Summary : Sub class IS-A super class , and something more (additional state + additional methods) and something modified(behaviour --- method overriding)

eg :
Person,Student,Faculty
Emp,Manager,SalesManager,HRManager,Worker,TempWorker
Shape, Circle,Rectangle,Cyllinder,Cuboid
LoanAccount,HomeLoanAccount,VehicleLoanAccount,
Student,GradStudent,PostGradStudent

Fruit -- Apple -- FujiApple

A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.

It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

Inheritance --- generalization ----> specialization.

IS A Relationship.

Why -- code re usability.

super class ---base class
sub class --derived class

keyword --extends

Types of inheritance
1. single inheritance ---
class A{...} class B extends A{...}
2. multi level inhertance
class A{...} class B extends A{...} class C extends B{...}
3. multiple inhertiance --- NOT supported
class A extends B,C{...} -- compiler err

Why --For simplicity.

(Diamond problem)

We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method. BUT which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Constructor invocations in inheritance hierarchy -- single & multi level.

eg -- Based on class A -- super class & B its sub class.
Further extend it by class C as a sub-class of B.
Check constructor invocation.

super keyword usage
1. To access super class's visible members
eg : class A
{
void show(){sop("in A's show");}
}
class B extends A {
//overriding form /sub class version
void show(){sop("in B's show");
super.show();
}
}
eg : B b1=new B();
b1.show();

2. To invoke immediate super class's matching constructor --- accessible only from sub class constructor.(super(...))

eg : Organize following in suitable class hierarchy(under "inh" package)
Person -- firstName,lastName
Student --firstName,lastName,grad year,course,fees,marks
Faculty -- firstName,lastName,yrs of experience , sme

Confirm invocation of constructors & super.

Regarding this & super

1. Only a constr can use this() or super()
2. Has to be 1st statement in the constructor
3. Any constructor can never have both ie. this() & super()
4. super & this (w/o brackets) are used to access (visible) members of super class or the same class.

eg :
Simple example 1

1. Fruit,Apple,Orange,Cherry
Add taste() method to display its taste.

2. Create FruitUtils class. (later!)
Add static method , addFruit to add a fruit to the Fruit Basket.

3. Write a Tester to create basket of fruits.
(populate basket based upon user's choice)
Menu
1. Add Apple
2. Add Orange
3. Add Cherry
4. Display taste of all fruits in the basket.
5. Exit : terminate the application.

Example 2

1. Shape -- x,y
Method --public double area()
public String toString()

2. Circle -- x,y,radius
Method --public double area()
public String toString()

3. Rectangle -- x,y,w,h
Method --public double area()
public String toString()

4. Square-- x,y,side
Method --public double area()
public String toString()

5. Create a ShapeFactory class
Add a method(generateShape) to return randomly generated shape.

6. Create a Tester . Invoke ShapeFactory's generateShape() method , in a for-loop
to display details & area of each shape.

Polymorphism ---one functionality --multiple (changing) forms

1. static -- compile time --early binding ---resolved by javac.

Achieved via method overloading

rules -- can be in same class or in sub classes.
same method name
signature -- different (no/type/both)
ret type --- ignored by compiler.

eg --- void test(int i,int j){...}
void test(int i) {..}
void test(double i){..}
void test(int i,double j,boolean flag){..}
int test(int a,int b){...}

RULE -- when javac doesn't find exact match --tries to resolve it by the closest arg type(just wider than the specified arg)

solve --- EasyOver.java
(More interesting examples after boxing & var-args)

2. Dynamic polymorphism --- late binding --- dynamic method dispatch ---resolved by JRE.
Dynamic method dispatch -- which form of method to send for execution ---This decision can't be taken by javac --- BUT taken by JRE
Achieved via -- method overriding

Method Overriding --- Means of achieving run-time polymorphism

NO "virtual" keyword in java.

All java methods can be overridden : if they are not marked as private,static,final

Super-class form of method - --- overridden method

sub-class form --- overriding form of the method

Rules : to be followed by overriding method in a sub-class

1. same method name, same signature, ret type must be same or its sub-type(co-variance)
eg of co-variance

```
class A {
   A getInstance()
         {
             return new A();
```

```
        }
}

class B extends A
{
   B getInstance()
        {
            return new B();
        }
}
```

2. scope---must be same or wider.

3. Will be discussed in exeception handling.
Can not add in its throws clause any new or broader checked exceptions.
BUT can add any new unchecked excs.
Can add any subset or sub-class of checked excs.

```
class A
{
  void show() throws IOExc
  {...}
}
class B extends A
{
  void show() throws Exc
  {...}
}
```

Can't add super class of the checked excs.

example of run time polymorphism -- Car & its sub classes.

From JDK 1.5 onwards : Annoations are available --- metadata meant for Compiler or JRE.(Java tools)

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML.

eg @Override,@Deprecated,@SuppressWarnings,@FunctionalInterface

@Override --
Annotation meant for javac.
Optional BUT recommended.
eg :

```
public class Orange extends Fruit {
@Override
```

```
 public void taste() {....}
}
```

While overriding the method --- if u want to inform the compiler that : following is the overriding form of the method use :
@Override
method declaration

Run time polymorphism or Dynamic method dispatch in detail

Super -class ref. can directly refer to sub-class object(direct=w/o type casting) as its the example of up-casting(similar to widening auto. conversion) .
When such a super class ref is used to invoke the overriding method: which form of the method to send for execution : this decision is taken by JRE & not by compiler. In such case --- overriding form of the method(sub-class version) will be dispatched for exec.

Super -class ref. can directly refer to sub-class inst BUT it can only access the members declared in super-class -- directly.

eg : A ref=new B(); ref.show() ---> this will invoke the sub-class: overriding form of the show () method

---

Applying inheritance & polymorphism
java.lang.Object --- Universal super class of all java classes including arrays.

Object class method
public String toString() ---Rets string representation of object.
Returns --- Fully qualified class Name @ hash code
hash code --internal memory representation.(hash code is mainly used in hashing based data structures -- will be done in Collection framework)

Why override toString?
To replace hash code version by actual details of any object.

eg -- Use it in sub classes. (override toString to display Account or Point2D or Emp details)

Object class method
public boolean equals(Object o)
Returns true --- If 'this' (invoker ref) & o ---refers to the same object(i.e reference equality) i.e this==o , otherwise returns false.

Need of overriding equals method ?
To replace reference equality by content identity equality , based upon prim key criteria.

eg : In Car scenario

(Primary key -- int registration no)

---

instanceof -- keyword in java --used for testing run time type information.

It is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false.

For null --instanceof returns false.
For sub-class object --instanceof super class -- rets true
For super-class object --instanceof sub class -- rets false

eg ---
Emp e =new Mgr(...);
e instanceof Mgr --true
e instanceof Emp --true
e instanceof Object --true
e instance of SalesMgr -- false
e instanceof Worker -- false

Solve

```
Fruit f=new Fruit();
f.taste();
f.pulp();
((Mango)f).pulp();
f=new Orange();
f.taste();
((Mango)f).pulp();
if(f instanceof Mango)
 ((Mango)f).pulp();
else
 sop("Invalid fruit....");
if(f instanceof Object)
 ((Mango)f).pulp();
else
 sop("Invalid fruit....");
```

---

abstract : keyword in Java
abstract methods ---methods only with declaration & no definition
eg : public abstract double calNetsalry();

Any time a class has one or multilple abstract methods ---- class must be declared as abstract class.

eg. public abstract class Emp {....}

Abstract classes can't be instantiated BUT can create the ref. of abstract class type to refer to concrete sub-class instances.
Emp e1=new Emp(...);//illegal
Emp e1=new Mgr(....);//legal

Abstract classes CAN HAVE concrete(non-abstract) methods.

Abstract classes MUST provide constructor/s to init its own private data members.

Can a class be decalred as abstract & final ? NO

Can an abstract class be crerated with 100% concrete functionality?
Yes
eg --- Event adapter classes

Use "abstract" keyword in Emp , Mgr ,Worker hierarchy & test it
final -- keyword in java

Usages

1 final data member(primitive types) - constant.
eg -- public final int data=123;

2. final methods ---can't be overridden.
usage eg public final void show{.....}
eg -- Object class -- wait , notify ,notifyAll

3. final class --- can't be sub-classed(or extended) -- i.e stopping inheritance hierarchy.
eg -- String ,StringBuffer,StringBuilder

4. final reference -- references can't be re-assigned.
eg -- final Emp e=new Mgr(.....);
e=new Worker(.....);//compiler err

Special note on protected

Protected members act as default scope within the same package.
BUT outside pkg -- a sub-class can access it through inheritance(i.e just inherits it directly) & CAN'T be accessed by creating super class instance.
static --- keyword in java
Usages

1. static data members --- Memory allocated only once @ class loading time --- not saved on object heap --- but in special memory area -- method area (meta space) . -- shared across all objects of the same class.
Initialized to their default values(eg --double --0.0,char -0, boolean -false,ref -null)
How to refer ? -- className.memberName

eg -- public static int idCounter;

2. static methods --- Can be accessed w/o instantiation. (ClassName.methodName(....))
Can't access 'this' or 'super' from within static method.

Rules -- 1. Can static methods access other static members directly(w/o instance) -- YES
2. Can static methods access other non-static members directly(w/o instance) -- NO
eg : class A
{
 private int i;
 private static int j;
 public static void show()
 {

   sop(i);//javac err
   sop(j);//no err
 }
}
3. Can non-static methods access other static members directly(w/o instance) -- YES
eg :
In Test class
void test1() {test2();}//no error
OR
static void test2(){test1();//javac error}

3. static import --- Can directly use all static members from the specified class.
eg --
//can access directly , ALL static members of the System class
import static java.lang.System.*;
import static java.lang.Math.*;
import java.util.Scanner;
main(...)
{
  out.println(.....);
  Scanner sc=new Scanner(in);
  sqrt(12.34);
  gc();
  exit(0);

}

4. static initializer block
syntax --
static {
// block gets called only once @ class loading time , by JVM's classlaoder
// usage --1. to init all static data members
//& can add functionality -which HAS to be called precisely once.
Use case : singleton pattern , J2EE for loading hibernate/spring... frmwork.
}

They appear -- within class definition & can access only static members directly.(w/o instance)
A class can have multiple static init blocks(legal BUT not recommended)

Regarding non-static initilizer blocks(instance initilaizer block)
syntax
{
//will be called per instantiation --- before matching constructor
//Better alternative --- parameterized constructor.
}

5. static nested classes ---
eg --
class Outer {
// static & non-static members
  static class Nested
  {
    //can access ONLY static members of the outer class DIRECTLY(w/o inst)
  }
}

Regarding Packages

What is a package ?
Collection of functionally similar classes & interfaces.

Creating user defined packages
Need ?
1. To group functionally similar classes together.
2. Avoids name space collision (allows duplicate class names in different packages)
3. Finer control over access specifiers.

About Packages
1. Creation : package statement has to be placed as the 1st statement in Java source.
eg : package p1; => the classes will be part of package p1.
2.Package names are mapped to folder names.
eg : package p1.p2; class A{....}
A.class must exist in folder p1\p2

3. For simplicity --- create folder p1\p2 -- under <src> & compile from <src>
From <src>
javac -d ..\bin p1\p2\A.java

-> javac will auto. create the sub-folder <p1>\<p2> under the <bin> folder & place A.class within <p1>\<p2>

NOTE : Its not mandatory to create java sources(.java) under package named folder. BUT its mandatory to store packged compiled classes(.class) under package named folders

Earlier half is just maintained as convenience(eg --- javac can then detect auto. dependencies & compile classes).

3.5 How to launch / run packaged java classes?
cd <bin>
java FullyQualifiedClassName
java p1.p2.A

4. To run the pkged classes from any folder : you must set Java specific environment variable : classpath
set classpath=g:\dac1\day2\bin;

classpath= Java only environment variable
Used mainly by JRE's classloader : to locate & load the classes.
Classloader will try to locate the classes from current folder, if not found --- will refer to classpath entries : to resolve & load Java classes.

What should be value of classpath? ---Must be set to top of packged class hierarchy(eg : bin)
set classpath=d:\dac\day2\bin;.;(cmd line invocation)
OR better still
set it from environment variables.

Rules
1. If the class is part of a package, the package statement must be the first line in the source code file, before any import statements that may be present.

2. If there are import statements, they must go between the package statement
(if there is one) and the class declaration. If there isn't a package statement,
then the import statement(s) must be the first line(s) in the source code file.
If there are no package or import statements, the class declaration must be
the first line in the source code file.

3. import and package statements apply to all classes within a source code file.

In other words, there's no way to declare multiple classes in a file and have them in different packages, or use different imports.

NOTE : Setting classpath on all platforms
Refer :
https://www.javacodestuffs.com/2020/09/how-to-set-classpath-in-java-windows.html#:~:text=%20How%20to%20set%20classpath%20in%20Java%20-,is%20one%20way%20to%20tell%20applications%2C...%20More%20
OR
https://www.edureka.co/blog/set-Java-classpath/

Inheritance --- generalization ----> specialization.

IS A Relationship.

Why -- code/state re usability.

super class ---base class
sub class --derived class

keyword --extends

Types of inheritance
1. single inheritance ---Supported in Java
class A{...} class B extends A{...}
2. multi level inhertance
class A{...} class B extends A{...} class C extends B{...}
Supported in java
3. multiple inhertiance --- NOT supported
class A extends B,C{...} -- compiler err

Why --For simplicity.

(Diamond problem)

We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method. BUT which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Constructor invocations in inheritance hierarchy -- single & multi level.

eg -- Based on class A -- super class & B its sub class.
Further extend it by class C as a sub-class of B.

super keyword usage
1. To access super class's visible members(data members n methods)
eg : p1 : package
class A { void show(){sop("in A's show");}}

package p1 :
class B extends A {
  //overriding form /sub class version
 void show(){sop("in B's show");
   super.show();
}
}
eg : B b1=new B();
b1.show();
2. To invoke immediate super class's matching constructor --- accessible only from sub class constructor.(super(...))

eg : Organize following in suitable class hierarchy(under "inheritance" package) : tight encapsulation
Person -- firstName,lastName
Student --firstName,lastName,grad year,course,fees,marks
Faculty -- firstName,lastName,yrs of experience , sme

Confirm invocation of constructors & super.

Regarding this & super

1. Only a constr can use this(...) or super(..)
2. Has to be 1st statement in the constructor
3. Any constructor can never have both ie. this() & super()
4. super & this (w/o brackets) are used to access (visible) members of super class or the same class.

eg :
1.Simple example to understand inheritance n polymorphism

1.1 Fruit : name
Add a parametrized constr , to accept name of the fruit.
Add taste() method to display its taste.
eg : public void taste() : "no specific taste"

1.2 Apple : extends Fruit
parametrized constr ---super(name);
override : taste
method definition : sweet n sour in taste

1.3 Similarly : add Orange n Mango

parametrized constr ---super(name);
Add taste() method to display its taste.
Orange : Sour in taste
Mango : sweet in taste

1.5 Write a simple tester : to understand upcasting n run time polymorphism.

2. Another example
Write a Tester to create basket of fruits.
(populate basket based on user choice)

1. Fruit,Apple,Orange,Cherry
Add taste() method to display its taste.

2. Create FruitUtils class.
Add static method , addFruit to add a fruit to the Fruit Basket.

3. Write a Tester to create basket of fruits.
(populate basket based upon user's choice)
Menu
1. Add Apple
2. Add Orange
3. Add Cherry
4. Display taste of all fruits in the basket.
5. Exit : terminate the application.

Menu
1. Add Apple
2. Add Orange
3. Add Mango
4. Display taste of all fruits in the basket (for-each)
5 : Exit : terminate the application.

2. Create FruitUtils class.(later)
Add static method , addFruit to add a fruit to the Fruit Basket.

Example 2 (Lab work)

1. Shape -- x,y
Method --public double area()
public String toString()

2. Circle -- x,y,radius
Method --public double area()
public String toString()

3. Rectangle -- x,y,w,h
Method --public double area()
public String toString()

4. Square-- x,y,side
Method --public double area()
public String toString()

5. Create a ShapeFactory class
Add a method(generateShape) to return randomly generated shape.

6. Create a Tester . Invoke ShapeFactory's generateShape() method , in a for-loop
to display details & area of each shape.

Polymorphism ---one functionality --multiple (changing) forms
1. static -- compile time --early binding ---resolved by javac.

Achieved via method overloading

rules -- can be in same class or in sub classes.
same method name
signature -- different (number/type/both)
ret type --- ignored by compiler.

eg --- void test(int i,int j){...}
void test(int i) {..}
void test(double i){..}
void test(int i,double j,boolean flag){..}
int test(int a,int b){...}

RULE -- when javac doesn't find exact match --tries to resolve it by the closest arg type(just wider than the specified arg)

solve --- EasyOver.java
(More interesting examples after boxing & var-args)

2. Dynamic polymorphism (run time polymorphism) --- late binding --- dynamic method dispatch --- resolved by JRE.

Dynamic method dispatch -- which form of method to send for execution ---This decision can't be taken by javac --- BUT taken by JRE
Achieved via -- method overriding

Method Overriding --- Means of achieving run-time polymorphism

NO "virtual" keyword in java.

All java methods can be overridden : if they are not marked as private,static,final

Super-class form of method - --- overridden method

sub-class form --- overriding form of the method

Rules : to be followed by overriding method in a sub-class

1. same method name, same signature, ret type must be same or its sub-type(co-variance)
eg of co-variance

```
class A {
   A getInstance()
          {
              return new A();
          }
}

class B extends A
{
   B getInstance()
          {
              return new B();
          }
}
```

2. scope---must be same or wider.

3. Will be discussed in exeception handling.
Can not add in its throws clause any new or broader checked exceptions.
BUT can add any new unchecked excs.
Can add any subset or sub-class of checked excs.

```
class A
{
  void show() throws IOExc
  {...}
}
class B extends A
{
  void show() throws Exc
  {...}
```

}
Can't add super class of the checked excs.

example of run time polymorphism -- Car & its sub classes.

From JDK 1.5 onwards : Annoations are available --- metadata meant for Compiler or JRE.(Java tools)

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML.

eg @Override,@Deprecated,@SuppressWarnings,@FunctionalInterface

@Override --
Annotation meant for javac.
Method level annotation
Optional BUT recommended.
eg : Fruit <----- Orange
public class Orange extends Fruit {
@Override
 public void taste() {....}

}

While overriding the method in a sub class -- if you want to inform the compiler that : following is the overriding form of the method use :
@Override
method declaration {...}

Run time polymorphism or Dynamic method dispatch in detail

Super -class ref. can directly refer to sub-class object(direct=w/o type casting) as its the example of up-casting(similar to widening auto. conversion) .
When such a super class ref is used to invoke the overriding method: which form of the method to send for execution : this decision is taken by JRE & not by compiler. In such case --- overriding form of the method(sub-class version) will be dispatched for exec.

Super -class ref. can directly refer to sub-class inst BUT it can only access the members declared in super-class -- directly.

eg : A ref=new B(); ref.show() ---> this will invoke the sub-class: overriding form of the show () method
---
Applying inheritance & polymorphism

java.lang.Object --- Universal super class of all java classes including arrays.

Object class method
public String toString() ---Rets string representation of object.
Returns --- Fully qualified class Name @ hash code
hash code --internal memory representation.(hash code is mainly used in hashing based data structures -- will be done in Collection framework)

Why override toString?
To replace hash code version by actual details of any object.

Objective -- Use it in sub classes. (override toString to display Account or Point2D or Emp details or Student / Faculty)

Object class method
public boolean equals(Object o)
Returns true --- If 'this' (invoker ref) & o ---refers to the same object(i.e reference equality) i.e this==o , otherwise returns false.

Need of overriding equals method ?
To replace reference equality by content identity equality , based upon prim key criteria.

eg : In Car scenario
(Primary key -- int registration no)

Objective : use it for understanding downcasting n instanceof keyword

---
instanceof -- keyword in java --used for testing run time type information.

refer : regarding instanceof

Solve
```
Fruit f=new Fruit();
f.taste();
f.pulp();
((Mango)f).pulp();
f=new Orange();
f.taste();
((Mango)f).pulp();
if(f instanceof Mango)
 ((Mango)f).pulp();
else
 sop("Invalid fruit....");
if(f instanceof Object)
```

```
 ((Mango)f).pulp();
else
 sop("Invalid fruit....");
```

---

abstract : keyword in Java
abstract methods ---methods only with declaration & no definition
eg : public abstract double calNetsalry();
private abstract double calNetsalry();//javac error

Any time a class has one or multilple abstract methods ---- class must be declared as abstract class.
eg. public abstract class Emp {....}

Abstract classes can't be instantiated BUT can create the ref. of abstract class type to refer to concrete sub-class instances.
Emp e1=new Emp(...);//illegal : RHS
Emp e1=new Mgr(....);//legal : provided Mgr class is concrete

Abstract classes CAN HAVE concrete(non-abstract) methods.

Abstract classes MUST provide constructor/s to init its own private data members.(to create concrete sub class instance)
eg : Emp : empId, dept...: private
Mgr extends Emp : to init empId, dept ... : MUST supply a constr in Emp class.

Can a class be decalred as abstract & final ? NO

Can an abstract class be crerated with 100% concrete functionality?
Yes
eg --- Event adapter classes / HttpServlet

Use "abstract" keyword in Emp , Mgr ,Worker hierarchy & test it

final -- keyword in java

Usages

1 final data member(primitive types) - constant.
eg -- public final int data=123;

2. final methods ---can't be overridden.
usage eg public final void show{.....}
eg -- Object class -- wait , notify ,notifyAll

3. final class --- can't be sub-classed(or extended) -- i.e stopping inheritance hierarchy.

eg -- String ,StringBuffer,StringBuilder
eg : public class MyString extends String {...} //javac err

4. final reference -- references can't be re-assigned.
eg --final Emp e=new Mgr(.....);//up casting
e=new Worker(.....);//compiler err

Special note on protected

Protected members act as default scope within the same package.
BUT outside pkg -- a sub-class can access it through inheritance(i.e just inherits it directly) & CAN'T be accessed by creating super class instance.
Do subclasses inherit private data members from it's superclass?

NO !
Explanation :

As per the java language specification :

Members of a class that are declared private are not inherited by subclasses of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

BUT what we mean here by inheritance is , are private members accessible in a subclass ?
That answer is NO

BUT , sub class instance DOES CONTAIN private fields of their superclasses .

eg : Person has data members : private firstName , lastName

Student extends Person
It has ADDITIONAL data members :private gradYear,course,fees,marks

Answer this !
Can you access firstName & lastName from Student class ? NO

Student IS-A Person

So when you create an instance of a Student : firstName n lastName will be present in Student object , mem allocated in heap.

So how many slots will you show in Student object ?
CP + 6 slots .
static --- keyword in java
Usages
1. static data members --- Memory allocated only once @ class loading time --- not saved on object heap --- but in special memory area -- method area (meta space) .

-- shared across all objects of the same class.
Initialized to their default values(eg --double --0.0,char -0, boolean -false,ref -null)
How to refer ? -- className.memberName

eg -- public static int idCounter;

2. static methods --- Can be accessed w/o instantiation. (ClassName.methodName(....))
Can't access 'this' or 'super' from within static method.

Rules -- 1. Can static methods access other static members directly(w/o instance) -- YES
2. Can static methods access other non-static members directly(w/o instance) -- NO
eg : class A
{
  private int i;
  private static int j;
  public static void show()
  {

    sop(i);//javac err
    sop(j);//no err
  }
}
3. Can non-static methods access other static members directly(w/o instance) -- YES
eg :
In Test class
void test1() {test2();}//no error
OR
static void test2(){test1();//javac error}

3. static import --- Can directly use all static members from the specified class.
eg --
//can access directly , ALL static members of the System class
import static java.lang.System.*;
import static java.lang.Math.*;
import java.util.Scanner;
main(...)
{
  out.println(.....);
  Scanner sc=new Scanner(in);
  sqrt(12.34);
  gc();
  exit(0);

}

4. static initializer block

syntax --
static {
// block gets called only once @ class loading time , by JVM's classlaoder
// usage --1. to init all static data members
//& can add functionality -which HAS to be called precisely once.
Use case : singleton pattern , J2EE for loading hibernate/spring... frmwork.
}

They appear -- within class definition & can access only static members directly.(w/o instance)
A class can have multiple static init blocks(legal BUT not recommended)

Regarding non-static initilizer blocks(instance initilaizer block)
syntax
{
//will be called per instantiation --- before matching constructor
//Better alternative --- parameterized constructor.
}

5. static nested classes ---
eg --
class Outer {
// static & non-static members
  static class Nested
  {
    //can access ONLY static members of the outer class DIRECTLY(w/o inst)
  }
}

Upcasting

The most important aspect of inheritance is the relationship expressed between the new class and the base class. This relationship can be summarized by saying,
The new class "IS A" type of the existing class.
eg : Student is of Person type or Faculty is of Person type.

This description is not just a fancy way of explaining inheritance—it’s supported directly by the language.

Meaning :
Can we say ?
Person p=new Student(....);//YES --upcasting
sop(p);//dynamic method dispatch

As another example, consider a base class called Fruit that represents any fruit, and a derived class called Mango.

Because inheritance means that all of the methods in the base class are also available in the derived class,
any message you can send to the base class can also be sent to the derived class. If the Fruit class has a taste( ) method, so will Mango.
This means we can accurately say that a Mango object is also a type of Fruit.

Regarding inheritance

In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy.
The classes in the lower hierarchy inherit all the variables (attributes/state) and methods (dynamic behaviors) from the higher hierarchies.

A class in the lower hierarchy is called a subclass (or derived, child, extended class). A class in the upper hierarchy is called a superclass (or base, parent class).

By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, redundancy can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. Re usability is maximum.

A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). It can use the inherited methods and variables as they are. It may also override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same name.

Summary : Sub class IS-A super class , and something more (additional state + additional methods) and something modified(behaviour --- method overriding)

eg :
Person,Student,Faculty
Emp,Manager,SalesManager,HRManager,Worker,TempWorker,PermanentWorker
Shape, Circle,Rectangle,Cyllinder,Cuboid
BankAccount ,LoanAccount,HomeLoanAccount,VehicleLoanAccount,
Student,GradStudent,PostGradStudent

Fruit -- Apple -- FujiApple

A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.

It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

Inheritance --- generalization ----> specialization.

IS A Relationship.

Why -- code re usability.

super class ---base class
sub class --derived class

keyword --extends

Types of inheritance
1. single inheritance ---
class A{...} class B extends A{...}
2. multi level inhertance
class A{...} class B extends A{...} class C extends B{...}
3. multiple inhertiance --- NOT supported
class A extends B,C{...} -- compiler err

Why --For simplicity.

(Diamond problem)

We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method. BUT which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Constructor invocations in inheritance hierarchy -- single & multi level.

eg -- Based on class A -- super class & B its sub class.
Further extend it by class C as a sub-class of B.

super keyword usage
1. To access super class's visible members(data members n methods)
eg : p1 : package
class A { void show(){sop("in A's show");}}

package p1 :
class B extends A {
  //overriding form /sub class version
 void show(){sop("in B's show");
   super.show();
}

}
eg : B b1=new B();
b1.show();
2. To invoke immediate super class's matching constructor --- accessible only from sub class constructor.(super(...))

eg : Organize following in suitable class hierarchy(under "inheritance" package) : tight encapsulation
Person -- firstName,lastName
Student --firstName,lastName,grad year,course,fees,marks
Faculty -- firstName,lastName,yrs of experience , sme

Confirm invocation of constructors & super.

Regarding this & super

1. Only a constr can use this(...) or super(..)
2. Has to be 1st statement in the constructor
3. Any constructor can never have both ie. this() & super()
4. super & this (w/o brackets) are used to access (visible) members of super class or the same class.

eg :
1.Simple example to understand inheritance n polymorphism

1.1 Fruit : name
Add a parametrized constr , to accept name of the fruit.
Add taste() method to display its taste.
eg : public void taste() : "no specific taste"

1.2 Apple : extends Fruit
parametrized constr ---super(name);
override : taste
method definition : sweet n sour in taste

1.3 Similarly : add Orange n Mango
parametrized constr ---super(name);
Add taste() method to display its taste.
Orange : Sour in taste
Mango : sweet in taste

1.5 Write a simple tester : to understand upcasting n run time polymorphism.

2. Another example
Write a Tester to create basket of fruits.
(populate basket based on user choice)

1. Fruit,Apple,Orange,Cherry
Add taste() method to display its taste.

2. Create FruitUtils class.
Add static method , addFruit to add a fruit to the Fruit Basket.

3. Write a Tester to create basket of fruits.
(populate basket based upon user's choice)
Menu
1. Add Apple
2. Add Orange
3. Add Cherry
4. Display taste of all fruits in the basket.
5. Exit : terminate the application.

Menu
1. Add Apple
2. Add Orange
3. Add Mango
4. Display taste of all fruits in the basket (for-each)
5 : Exit : terminate the application.

2. Create FruitUtils class.(later)
Add static method , addFruit to add a fruit to the Fruit Basket.

Example 2 (Lab work)

1. Shape -- x,y
Method --public double area()
public String toString()

2. Circle -- x,y,radius
Method --public double area()
public String toString()

3. Rectangle -- x,y,w,h
Method --public double area()
public String toString()

4. Square-- x,y,side
Method --public double area()
public String toString()

5. Create a ShapeFactory class

Add a method(generateShape) to return randomly generated shape.

6. Create a Tester . Invoke ShapeFactory's generateShape() method , in a for-loop
to display details & area of each shape.

Polymorphism ---one functionality --multiple (changing) forms
1. static -- compile time --early binding ---resolved by javac.

Achieved via method overloading

rules -- can be in same class or in sub classes.
same method name
signature -- different (no of arguments/type of args/both)
ret type --- ignored by compiler.

eg --- In class TestMethodOverloading :
void test(int i,int j){...}
void test(int i) {..}
void test(double i){..}
void test(int i,double j,boolean flag){..}
int test(int a,int b){...}   //javac error
---
2. Dynamic(run time) polymorphism --- late binding --- dynamic method dispatch ---resolved by JRE.

Dynamic method dispatch -- which form of method to send for execution ---This decision can't be taken by javac --- BUT taken by JRE
Achieved via -- method overriding

Method Overriding --- Means of achieving run-time polymorphism

NO "virtual" keyword in java. (i.e all methods are implicitly virtual)

All java methods can be overridden : if they are not marked as private,static,final

Super-class form of method - --- overridden method

sub-class form --- overriding form of the method

Rules : to be followed by overriding method in a sub-class

1. same method name, same signature, ret type must be same or its sub-type(co-variance)
eg of co-variance
```
class A {
   public A getInstance()
           {
```

```
            return new A();
        }
}

class B extends A
{
  public B getInstance()
        {
            return new B();
        }
}
```

2. scope---must be same or wider.

3. Will be discussed in exeception handling.
Can not add in its throws clause any new or broader checked exceptions.
BUT can add any new unchecked excs.
Can add any subset or sub-class of checked excs.

```
class A
{
 void show() throws IOExc
 {...}
}
class B extends A
{
 void show() throws Exc
 {...}
}
```

Can't add super class of the checked excs.

example of run time polymorphism -- Car & its sub classes.

From JDK 1.5 onwards : Annoations are available --- metadata meant for Compiler or JRE.(Java tools)

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML.

eg @Override,@Deprecated,@SuppressWarnings,@FunctionalInterface

@Override --
Annotation meant for javac.
Method level annotation ,appearing in a sub class
Optional BUT recommended.
eg :

```
public class Orange extends Fruit {
@Override
 public void taste() {....}
}
}
```

While overriding the method --- if you want to inform the compiler that : following is the overriding form of the method use :
@Override
method declaration {...}

Run time polymorphism or Dynamic method dispatch in detail

Super -class ref. can directly refer to sub-class object(direct=w/o type casting) as its the example of up-casting(similar to widening auto. conversion) .

When such a super class ref is used to invoke the overriding method: which form of the method to send for execution : this decision is taken by JRE & not by compiler. In such case --- overriding form of the method(sub-class version) will be dispatched for exec.

Javac resolves the method binding by the type of the reference & JVM resolves the method binding by type of the object it's referring to.

Super -class ref. can directly refer to sub-class inst BUT it can only access the members declared in super-class -- directly.

eg : A ref=new B(); ref.show() ---> this will invoke the sub-class: overriding form of the show () method

---

Applying inheritance & polymorphism

Important statement
Java compiler resolves method binding by type of the reference & JVM resolves it by the type of the obejct, reference is referring to.

java.lang.Object --- Universal super class of all java classes including arrays.

Object class method
public String toString() ---Rets string representation of object.
Returns --- Fully qualified class Name @ hash code
hash code --internal memory representation.(hash code is mainly used in hashing based data structures -- will be done in Collection framework)

Why override toString?

To replace hash code version by actual details of any object.

Objective -- Use it in sub classes. (override toString to display Account or Point2D or Emp details or Student / Faculty)

Object class method
public boolean equals(Object o)
Returns true --- If 'this' (invoker ref) & o ---refers to the same object(i.e reference equality) i.e this==o , otherwise returns false.

Need of overriding equals method ?
To replace reference equality by content identity equality , based upon prim key criteria.

eg : In Car scenario
(Primary key -- int registration no)

Objective : use it for understanding downcasting n instanceof keyword

---
instanceof -- keyword in java --used for testing run time type information.

refer : regarding instanceof

Solve
```
Fruit f=new Fruit();
f.taste();
f.pulp();
((Mango)f).pulp();
f=new Orange();
f.taste();
((Mango)f).pulp();
if(f instanceof Mango)
 ((Mango)f).pulp();
else
 sop("Invalid fruit....");
if(f instanceof Object)
 ((Mango)f).pulp();
else
 sop("Invalid fruit....");
```

---
abstract : keyword in Java
abstract methods ---methods only with declaration & no definition
eg : public abstract double calNetsalry();
private/static/final abstract double calNetsalry();//javac error

Any time a class has one or multilple abstract methods ---- class must be declared as abstract class.
eg. public abstract class Emp {....}

Abstract classes can't be instantiated BUT can create the reference of abstract class type to refer to concrete sub-class instances.
Emp e1=new Emp(...);//illegal : RHS
OR
Emp e1=new Mgr(....);//legal : provided Mgr class is concrete
OR in case if Emp <---Mgr <----SalesMgr,HRMgr...(Given : Emp n Mgr : abstract , SalesMgr,HRMgr : concrete)
Emp e1=new Mgr(....);//javac err
OR
e1=new SalesMgr(...);//no err

Abstract classes CAN HAVE concrete(non-abstract) methods.

Abstract classes MUST provide constructor/s to init its own private data members.(for creating concrete sub class instance)
eg : Emp : empId, dept...: private
Mgr extends Emp : to init empId, dept ... : MUST supply a constr in Emp class.

Can a class be declared as abstract & final ? NO

Can an abstract class be crerated with 100% concrete functionality?
Yes
eg --- Event adapter classes / HttpServlet

Use "abstract" keyword in Emp , Mgr ,Worker hierarchy & test it

final -- keyword in java

Usages

1 final data member(primitive types) - constant.
eg -- public final int DATA=123;

2. final methods ---can't be overridden.
usage eg public final void show{.....}
This show() method CAN NOT be overridden by any of the sub classes
eg -- Object class -- wait , notify ,notifyAll

3. final class --- can't be sub-classed(or extended) -- i.e stopping inheritance hierarchy.
eg -- String ,StringBuffer,StringBuilder
eg : public class MyString extends String {...} //javac err

4. final reference -- references can't be re-assigned.
eg -- final Emp e=new Mgr(.....);//up casting
e=new Worker(.....);//compiler err

Special note on protected

Protected members act as default scope within the same package.
BUT outside pkg -- a sub-class can access it through inheritance(i.e just inherits it directly) & CAN'T be accessed by creating super class instance.
Abstraction is the property with which only the essential details are displayed to the user.
The internal details or the non-essentials details are not displayed to the user.
(Hiding complexities or hiding the implementation details from end user)

Eg: An ATM is considered as just money rendering machine rather than its internal complex details

Consider a real-life example of a person walking to an ATM She only knows how to withdraw / deposit money. But as the end user , she does not really need to know about how ATM connects with the underlying bank to inform about this transaction ...

This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

When to use abstract classes and abstract methods ?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.(i.e only provide declaration)
Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
eg : BoundedShape & it's method area
instanceof -- keyword in java --used for testing run time type information.

It is used to test whether the object is an instance of the specified type (class or subclass or interface).

Meaning
In "a instanceof B", the expression returns true if the reference to which a points is an instance of class B, a subclass of B (directly or indirectly), or a class that implements the B interface (directly or indirectly).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false.

For null --instanceof returns false.

For sub-class object --instanceof super class -- rets true
For super-class object --instanceof sub class -- rets false

eg ---Object <----Emp <---Mgr <---SalesMgr
Emp <--- Worker

What will be o/p ?
Emp e =new Mgr(...);
e instanceof Mgr --true
e instanceof Emp --true
e instanceof Object --true
e instanceof SalesMgr -- false
e instanceof Worker --false
Emp e=null;
e instanceof Emp/Mgr/SalesMgr/Worker --- false

Why static methods can't be overridden in java ?

Method overriding is a way to achieve dynamic method dispatch(i.e run time polymorphism) Meaning which behaviour to choose or which method to choose for execution --this decision is taken at the run time depending upon type of the object by the JVM(late binding). Since it depends upon the type of the object , for static methods this concept is not applicable. (since they are not associated with any object)

Overriding depends on having an instance of a class. The point of polymorphism is that you can subclass a class and the objects implementing those subclasses will have different behaviors for the same methods defined in the superclass (and overridden in the subclasses). A static method is not associated with any instance of a class so the concept is not applicable.
Exception Handling

Regarding Exception Handling in java.....
Any run time err occurs(eg file not found,accessing out of array size,accessing func from null ref, divide by 0)
---JRE(main thrd) --- creates matching exc class instance(java.io.FileNotFoundException,java.lang.ArrayOutOfBoundsExc,NullPointerExc,ArithmeticExc)
--- JRE checks -- if prog has proivided exc handling code ?
--- NO -- JRE aborts java code(by supplying def handler) & prints details --F.Q exc class name,reason behind failure & location details(err stack trace

--- YES (try---catch) JRE execs exc handling block & continues with the rest of the code.

syntax(key words) --- try,catch,finally,throw,throws
Inheritance hierarchy of exc classes
unchecked vs checked excs.
Creating custom excs
JDK 1.7 syntax --- try-with-resources(in I/O or device prog)

Checked & Unchked exception are detected or occur only in run-time.
JRE/JVM DOES NOT distinguish between them (un handled : w/o try-catch chked or un checked exc will cause aborting of code)
Compiler(javac) differentiates between them
Javac forces handling of the checked exc. upon the prog.(Handling by supplying matching try-catch block or including it in the throws clause.)

Legal syntax
1. try {...} catch (exc1 e){...}
2. try {...} catch (exc1 e){...} catch (exc2 e) {..} ....
3. try {...} catch (NPE e){} catch (AE e) {}catch(Exception e){catch-all}
3.5 3. try {...} catch (AE e){...} catch (NPE | AOB e) {...}catch(Exception e){catch-all}
4. throws syntax ---
method declaration throws comma separated list of exc classes.
eg : Integer class API
public static int parseInt(String s) throws NumberFormatException
Thread class API
public static void sleep(long ms) throws InterruptedException

FileReader API
public FileReader(String fileName) throws FileNotFoundException

throws --- keyword meant for javac
Meaning -- Method MAY raise specified exc.
Current method is NOT handling it , BUT its caller should handle.
Mandatory--- only in case of un handled(no try-catch) chked excs(not extended from RuntimeException).
Use case --used in delegating the exception to caller.

4.5 Throwable class API

1. public String toString() -- rets Name of exc class & reason.(detailed err mesg)
2. public String getMessage() -- rets error mesg of exception
3. public void printStackTrace() --- Displays name of exc class, reason, location dtls.

5. finally --- keyword in exc handling

finally -- block -- finally block ALWAYS survives(except System.exit(0) i.e terminating JVM)
i.e in the presence or absence of excs.
5.1 try{...} catch (Exception e){....} finally {....}
5.2 try{...} catch (NullPointerException e){....} finally {....}
5.3 try {...} finally {....}

try-with-resources
From Java SE 7 onwards --- Java has introduced java.lang.AutoCloseable -- i/f
It represents --- resources that must be closed -- when no longer required.
Autocloesable i/f method
public void close() throws Exception-- closing resources.

Java I/O classes(eg : BufferedReader,PrintWriter.....),Scanner -- have already implemented this i/f -- to automatically close resource when no longer required.

syntax of try-with-resources
try (//can open one or multiple AutoCloseable resources)
{ .....
} catch(Exception e)
{
}
eg :
try(Scanner sc=new Scanner(System.in);
  FileReader fr=new FR(....))
{
 .....
} catch -all

Creating Custom Exc(User defined exception or application exc)
Need :

1. Validations : In case of validation failures : Prog will have to throw custom exc class instance
2. B.L failures (eg : funds transfer : insufficient finds) : Prog will have to throw custom exc class instance

1. Create a pkged public class which extends Throwable(not reco but legal)/Exception(recommended)/Error(not reco but legal)/RuntimeExc(not reco but legal)
eg : public class MyException extends Exception{
  public MyException(String mesg)
  {
   super(mesg);
  }
}
public class MyException2 extends RunTimeException{....}

2.CustExc(String msg) : overload the constr : to invoke the super-class constr.
of the form
Exception (String msg)
OR
CustExc(String msg,Throwable rootCause)
public Exception(String message,Throwable cause)

Objective :
Check the speed of vehicle on a freeway
Accept the speed using Scanner : can be speed too low(exc) or too high(exc) or in range

keyword -- throw --for throwing exception.
JVM uses it to throw built-in exceptions(eg : NullPointerExc , IOException etc) & prog uses it throw custom exception(user defined excs) in case of B.L or validation failures.

syntax :
throw Throwable instance;
eg :
throw new NullPointerExc();// no javac err
throw new InterruptedExc();// no javac err
throw new Throwable("abc");// no javac err
throw new Account(...);//javac err (provided it doesn't extend from Throwable hierarchy)
throw new AccountOverdrawnException("funds too low...");//proper usage

Interface in Java

What is interface ?

An interface in java is a blueprint of a class. It has public static final data members and public n abstract methods only.

The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface (not method body)(true till JDK 1.7) .
It is used to achieve full abstraction and multiple inheritance in Java.

Java Interface also represents IS-A relationship.

It cannot be instantiated just like abstract class.

Why java interfaces?

1. It is used to achieve full abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

(Interfaces allow complete separation between WHAT(specification or a contract) is to be done Vs HOW (implementation details) it's to be done

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

---

syntax of interface

default(no modifier)/public interface NameOfInterface extends comma separated list of super interfaces

```
{
  //data members --- public static final : added implicitly by javac
  int DATA=100;
  //methods -- public abstract : added implicitly by javac
  double calc(double d1,double d2);

}
```

Implementing class syntax

default(no modifier)/public class NameOfClass extends SuperCls implements comma separated list of interfaces {

```
  //Mandatory for implementation class to be non-abstract(concrete): MUST define/implement all abstract methods inherited from all i/fs.(interface)

}
```

eg : public class Circle extends Shape implements Computable,Runnable {...}

1. Relationship between classes and interfaces

A class inherits from another class(extends), an interface extends another interfaces(extends) but a class implements an interface.

2. Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

eg :

```
Multiple inheritance in java
interface Printable{
void print();
}

interface Showable{
void show();
}

class A implements Printable,Showable{
```

```
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A obj = new A();
obj.print();
obj.show();
 }
}
```

Question

Multiple inheritance is not supported through class in java but it is possible by interface, why?

Multiple inheritance is not supported in case of class, sincee it can create an ambiguity.
But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestTnterface1 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestTnterface1 obj = new TestTnterface1();
obj.print();
 }
}
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1,
so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
```

```
void show();
}
class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 }
}
```

Q) What is marker or tagged interface?

An interface that has no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc.
They are used to provide some essential information to the JVM(Run time marker) so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface.

```
eg :
interface printable{
 void print();
 interface MessagePrintable{
   void msg();
 }
}
```

Q . What is a functional i/f
An interface containing sing abstract methods (SAM)
eg : Comparator , Runnable , Consumer...

Abstract Class vs. Interface

Java provides and supports the creation of abstract classes and interfaces.
Both implementations share some common features, but they differ in the following features:

1. All methods in an interface are implicitly abstract. On the other hand, an abstract class may contain both abstract and non-abstract methods.

2.A class may implement a number of Interfaces, but can extend only one abstract class.

3.
In order for a class to implement an interface, it must implement all its declared methods.
However, a class may not implement all declared methods of an abstract class. Though, in this case, the sub-class must also be declared as abstract.
Abstract classes can implement interfaces without even providing the implementation of interface methods.

4.
Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.

5.
Members of a Java interface are public by default. A member of an abstract class can either be private, protected or public.

6.
An interface is absolutely abstract and cannot be instantiated, doesn't support a constructor.
An abstract class also cannot be instantiated BUT can contain a constructor to be used while creating concrete(non abstract) sub class instance.

String class API
Important String class constructors
1.String(byte[] bytes) --- byte[] ----> String converter

2.String(char[] chars) --- char[] ---> String converter

3.String (byte[] bytes,int offset,int len) ---byte[] ----> String converter from the specified offset , specified len no of bytes will be converted.
eg . String s=new String(bytes,3,4); --- String will contain bytes[3]----bytes[6]

4. String(char[] ch,int offset,int len)

5. String(String s)

String class methods --- to go through
charAt,compareTo,contains,copyValueOf,format,valueOf,getBytes,toCharArray,toLowerCase,indexOf,lastIndexOf,split,replace,startsWith,endsWith,length,intern

1.
boolean equals(Object o) ---- ret true iff 2 strings are having same contents (case sensitive)

About equals()

super class def. --- java.lang.Object
public boolean equals(Object o)
Rets true iff both refs(this & o) are equal i.e referring to the same object.

Sub-class developers MUST override equals for content-wise(depending on Object's state) comparison.

2. concat,charAt,indexOf,lastIndexOf,toUpperCase,toLowerCase,format,split

printf & Formatter class
Refer to java.util.Formatter class for formatting conversion details.

Imp ---

Formatting details
%c -- character
%b -- boolean
%h -- hex value of hashcode of obj ref.
%s -- string
%d -- int
%f,%g -- float/double
%x -- hex value
%n -- line separator
%tD -- Date
%tT -- Time
%tc -- Time stamp(date & Time)
%td-%1$tb-%1$tY -- can be applied to GC or Date.

Date/Time Handling in Java

API
1. java.util.Date--- represents system date.
Constructor
1.Date() --- creates Date class instance representing system date.
2.Date(long msec) --- creates Date class instance representing date for msec elapsed after epoch(=1st Jan 1970)

For parsing & formatting
1. Create an instance of java.text.SimpleDateFormat
Constr : SimpleDateFormat(String pattern)
pre defined pattern
y --yr
MM -- month in digit(1-12)
MMM -- month in abbrevation
MMMM ---complete month name

h- Hour
m --minute
s -- second
eg : SDF sdf=new SimpleDateFormat("dd-MM-yyyy");

2. Parsing (use inherited API) string ----> Date
public Date parse(String s) throws ParseException

3. Formatting
public String format(Date d)

2. java.util.GregorianCalendar
month range --- 0-11
GregorianCalendar(int yr,int mon,int date);
GregorianCalendar(int yr,int mon,int date,int hr,int min,int sec);

2.5 How to find out current year ?
GregorianCalendar class API (inherited from Calendar class)
public int get(String fieldName)
eg : gc.get(Calendar.YEAR);

3. Date/Time formatting via printf
%tc -- for complete timestamp(date & time)
%tD -- for date
%tT -- time

Arguments --- Date, GregorianCalendar

static import syntax ---
eg -- import static java.util.Calendar.*;
or import static java.lang.System.*;

in such src - u can access directly static members of Calendar class or from 2nd statement u can directly use out.println("testing static imports!");

var-args
variable args syntax.--- Must be last arg in the method args.
Can use primitive type or ref types.
Legal ---
void doStuff(int... x) {

} // expects from 0 to many ints
Usage : ref.doStuff();

```
int[] ints={1,2,3,4};
ref.doStuff(ints);
ref.doStuff(20,34,56);

System.out.printf("%n");
// as parameters
void doStuff2(char c, int... x) { } // expects first a char,
// then 0 to many ints
void doStuff3(Animal... animal) { } // 0 to many Animals
invocations ---
ref.doStuff3();
Animal[] animals={new Cat(),new Dog(),new Horse()};
ref.doStuff3(animals);
ref.doStuff3(a1,a2,a3);

Illegal:
void doStuff4(int x...) { } // bad syntax
void doStuff5(int... x, char... y) { } // too many var-args
void doStuff6(String... s, byte b) { } // var-arg must be last


Regarding wrapper classes
1. What's need of wrapper classes?
---1. to be able to add primitive types to growable collection(growable data structure eg --
LinkedList)
--- 2.  wrapper classes contain useful api(eg --- parseInt,parseFloat....,isDigit,isWhiteSpace...)
2. What are wrappers? --- Class  equivalent for primitive types
-- Inheritance hierarchy
java.lang.Object --- Character (char)
java.lang.Object --- Boolean
Object -- Number -- Byte,Short,Integer,Long,Float,Double
3. Constrs & methods --- for boxing & unboxing
boxing= conversion from prim type to the wrapper type(class type)
un-boxing = conversion from wrapper type to the prim type
eg
Integer(int data) --- boxing
Integer i1=new Integer(100);
//un-boxing
int data=i1.intValue();


Integer i1=100;//no err from JDK 1.5
sop(i1);
int data=1234;
i1++;//Integer--->int(auto unboxing), inc ,auto box
Object o=123.45;//auto-boxing(double--->Double)--up casted to Object
Number n1=true;//auto-box----X(up casted) to Number
Object o2=false;//auto box -- up casting
```

Double d1=1234;//auto boxing (int --->Integer) ---X--Double

4. JDK 1.5 onwards --- boxing &unboxing performed automatically by java compiler,when required. --- auto-boxing , auto-unboxing,
5. examples

What is enum in java ?

Enumerations are generally a set of related constants.

They have been in other programming languages like C++ from beginning. BUT more powerful in Java.

Supported in Java since JDK 1.5 release.

Enumeration in java is supported by keyword enum. enums are a special type of class that always extends java.lang.Enum.

It's a combination of class & interface features.

Why ?

1. Helps to define constants.
2. Adds type safety to constants.

eg interface MovieConstants
{

int AGE_MINOR=16;
int AGE_MIN = 10;
int AGE_MAX=70;

int TKT_COST_SILVER =100;
int TKT_COST_GOLD =200;
int TKT_COST_PLATINUM =300;

}

If by mistake application uses TKT_COST to compare ages of user , what will happen ?

Both being int type neither javac or jvm can realise err , but you will get wrong results.

It should not be allowed --as they represent different types ---AGE type & TKT_COST type.

3. You can't iterate over all constant values from i/f but with enums you can.
4 . Consider this

```
eg interface Menu
{
  String SOUP="Tomato soup";
  String DOSA="Mysore Dosa";
  String RICE="Fried rice";
}
```

Can you assign any price along with menu? ---Not easily !
But with enums you can.

A simple usage will look like this:

```
public/default enum Direction {
  EAST,
  WEST,
  NORTH,
  SOUTH      //optionally can end with ";"
 }
```

Here EAST, WEST, NORTH and SOUTH are implicitely of type
public final static Direction EAST=new Direction("EAST",0) ---super("EAST",0);
public final static Direction WEST=new Direction("WEST",1) ---super("WEST",1);

Super class of all enums

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

ie. they are comparable and serializable implicitly.
All enum types in java are singleton by default.
So, you can compare enum types using '==' operator also.

Since enums extends java.lang.Enum, so they can not extend any other class because java does not support multiple inheritance . But, enums can implement any number of interfaces.

enum can be declared within a class or separately.

eg of enum within a class

When declared inside a class, enums are always static by default
eg public class TestOuter
 {

```
  enum Direction
  {
    EAST,
    WEST,
    NORTH,
    SOUTH
  }
 }
```

To access a direction -- use TestOuter.Direction.NORTH.

Constructors of enum

By default, you don't have to supply constructor definition.
Javac implicitely calls super class constructor , Enum(String name,int ordinal)

Important Methods of Enum (implicitly added by javac)

Enum[] values() --rets array of enum type of refs.--pointing to singleton objs
Enum valueOf(String name) throws IllegalArgumentException -- string to enum type converter
values & valueOf methods generated by compiler --so not part of javadocs.
If u pass a different name (eg -- ABC) to valueOf ---throws IllegalArgumentException

Inherited from Suerpclass Enum

String name() --rets name of constant in string form
int ordinal() --rets index of the const as it appears in enum.--starts with 0
public String toString() : overridden to return name of the enum constant.

You can supply your own constructor/s to initialize the state(data member of enum types.

```
enum Direction {
  // Enum types
  EAST(0), WEST(180), NORTH(90), SOUTH(270);

  // Constructor
  private Direction(final int angle) {
    this.angle = angle;
  }

  // Internal state
  private int angle;

  public int getAngle() {
    return angle;
  }
}
```

BUT u can't instantiate enums using these constructors , since they are implicitely private.

You can override toString BUT you can't override equals since it's declared as final method in enum.
1. The inner class(non-static nested) has access to all of the outer class's members, including those marked private , directly(without inst.)
BUT Outer class MUST make an instance of the inner class , to access it's members.
2. To instantiate an inner class, you must have a reference to an instance of the outer class.
syntax :
Instantiating a non-static nested class requires using both the outer inst and nested class names as follows:
BigOuter.Nested n = new BigOuter().new Nested();
3. Such Inner classes can't have static members.(Java SE 8 --allows static final data members)

About method-local inner classes

1.A method-local inner class is defined within a method of the enclosing class.
2.For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.
3. A method-local inner class cannot use variables declared within the method
(including parameters) unless those variables are marked final or effectively final.

static nested classes
1.A static nested class is not an inner class, it's a top-level nested class.
2. You don't need an instance of the
outer class to instantiate a static nested class.

4.It cannot access non-static members of the outer class directly BUT can access static members of the outer class.

5. It can contain both static & non-static members.

6. JVM will not load any class's static init block -- until u actually refer to something from that class. (Lazy loading) This is true for static nested classes too.

7. Instantiating a static nested class requires using Outer class name and instance of nested class names as follows:
Outer.Nested n = new Outer.new Nested();

Regarding Association

Association is relationship between two separate classes ,typically using object references.

Represents HAS-A
Why : Code reusability

Association can be one-to-one, one-to-many, many-to-one, many-to-many.

Composition and Aggregation are the two forms of association.

eg : refer to association-aggregation-composition.png
Association :
Owner HAS-A Pet -- Owner feeds a Pet & Pet plays with Owner.
Aggregation implies a HAS-A relationship where the child can exist independently of the parent.
eg : Class & Student / Bank HAS-A Customer

```
class Bank
{
  private String name;
  private String ifsc;
  private String address;
 //one to many
  private Customer[] customers;

}
class Customer {...
 +
//many to one
  private Bank myBank;
}
```

Composition (Part Of or Belongs To)
Pet HAS-A Tail
It implies a relationship where the child cannot exist independent of the parent.
eg : Human HAS-A Lungs / Car HAS-A Engine / Person HAS-A Address
(when parent is deleted , typically child cant't exist on its own)
eg :

```
class Person
{
   private String firstName,lastName;
   private Date dob;
   private String uid;
    private Address adr;
   class Address
   {
     private String street,city,state,country;
     ....
   }

//setter / method
}
```

Aggregation is a weaker form of HAS-A relationship than Composition

1. The inner class(non-static nested) has access to all of the outer class's members, including those marked private , directly(without inst.)
BUT Outer class MUST make an instance of the inner class , to access it's members.
2. To instantiate an inner class, you must have a reference to an instance of the outer class.
syntax :
Instantiating a non-static nested class requires using both the outer inst and nested class names as follows:
BigOuter.Nested n = new BigOuter().new Nested();
3. Such Inner classes can't have static members.(Java SE 8 --allows static final data members)

About method-local inner classes

1.A method-local inner class is defined within a method of the enclosing class.
2.For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.
3. A method-local inner class cannot use variables declared within the method
(including parameters) unless those variables are marked final or effectively final.

static nested classes
1.A static nested class is not an inner class, it's a top-level nested class.
2. You don't need an instance of the
outer class to instantiate a static nested class.

4.It cannot access non-static members of the outer class directly BUT can access static members of the outer class.

5. It can contain both static & non-static members.

6. JVM will not load any class's static init block -- until u actually refer to something from that class. (Lazy loading) This is true for static nested classes too.

7. Instantiating a static nested class requires using Outer class name and instance of nested class names as follows:
Outer.Nested n = new Outer.new Nested();

Generic syntax ---
Available from Java SE 5 onwards.
Represents Parameterized Types.
Can Create Generic classes, interfaces, methods and constructors.
In Pre-generics world , similar achieved via Object class reference.

Syntax -- similar to c++ templates (angle brackets)

eg : ArrayList<Emp> , HashMap<Integer,Account> .....
1. Syntax is different than C++ --for nested collections only.
2. NO code bloat issues unlike c++;

Advantages
Adds Type Safety to the code @ compile time

Meaning :
1. Can add type safe code where type-mismatch errors(i.e ClassCastExceptions) are detected at compile time.
2. No need of explicit type casting, as all casts are automatic and implicit.

A generic class means that the class declaration includes a type parameter.
eg --- class MyGeneric<T>
{
 private T ref;
}

class MyGeneric<T,U> {...}
T ,U ---type --- ref type
eg : ArrayList<Emp>

Understand why generics with example.
eg : Create a Holder class , that can hold ANY data type (primitive/ref type)

List<E> features
1. List represents ordered collection --- order is significant(It remembers the order of insertion)
2. Allows null references
3. Allows duplicates
4. Supports index based operation

java.util.ArrayList<E> -- E -- type of ref.
1. ArrayList<E> -- constructor
API
ArrayList() -- default constructor. -- creates EMPTY array list object , with init capacity=10,size=0;
eg ---ArrayList<Integer> l1=new ArrayList<>();

1.5 1. ArrayList<E> -- constructor
API
public ArrayList(int capacity) -- -- creates EMPTY array list object , with init capacity=capacity,size=0;
eg ---ArrayList<Integer> l1=new ArrayList<>(100);
l1.add(1);.....l1.add(100);
l1.add(101);//capa=150 --as per JVM spec.

2. add methods
boolean add(E e) --- append
void add(int index,E e) --- insert
void addAll(Collection<E> e) -- bulk append operation
eg : l1 --- AL<Emp>
l1.addAll(.....);
AL,LL,Vector --- legal
HS,TS,LHS --legal
HM,LHM,TM --illegal --javac error

2.5 Retrieve elem from list
E get(int index)
index ranges from ---0 ---(size-1)
java.lang.IndexOutOfBoundsException

3. display list contents using --- toString

4. Attaching Iterator
Collection<E> interface method -- implemented by ArrayList
Iterator<E> iterator()
---places iterator BEFORE 1st element ref.
Iterator<E> i/f methods
boolean hasNext() -- rets true if there exists next element, false otherwise.

E next() --- returns the element next to iterator position

void remove() -- removes last returned element from iterator.

Limitation --- type forward only & can start from 1st elem only.

Regarding exceptions with Iterator/List
1. java.util.NoSuchElementException -- thrown whenever trying to access the elem beyond the size of list via Iterator/ListIterator
2. java.lang.IllegalStateException --- thrown whenever trying to remove elem before calling next().
3. java.util.ConcurrentModificationException-- thrown typically --- when trying to use same iterator/list iterator --after structrually modifying list(eg add/remove methods of list)
Above describes fail-fast behaviour of the Iterator/ListIterator

Exception while accessing element by index.

4. java.lang.IndexOutOfBoundsException -- thrown typically -- while trying to access elem beyond size(0---size-1) --via get

6. Attaching for-each = attaching implicit iterator.

Attaching ListIterator ---scrollable iterator or to beign iteration from a specific element -- List ONLY or list specific iterator.
ListIterator<E> listItearator() --places LI before 1st element
ListIterator<E> listItearator(int index) --places LI before specified index.

4. search for a particular element in list
boolean contains(Object o)

5. searching for 1st occurrence
use -- indexOf
int indexOf(Object o)
rets index of 1st occurrence of specified elem. Rets -1 if elem not found.
searching for last occurrence
use -- lastIndexOf
int lastIndexOf(Object o)
rets index of last occurrence of specified elem. Rets -1 if elem not found.

5.5
E set(int index,E e)
Replaces old elem at spepcified index by new elem.
Returns old elem

6. remove methods
E remove(int index) ---removes elem at specified index & returns removed elem.

boolean remove(Object o) --- removes element specified by argument , rets true -- if elem is removed or false if elem cant be removed.

Objectives in Integer list
0. Create ArrayList of integers & populate it.
1. check if element exists in the list.
2. disp index of 1st occurance of the elem
3. double values in the list --if elem val > 20
4. remove elem at the specified index
5. remove by elem. -- rets true /false.

NOTE :
For searching or removing based upon primary key , in List Implementation classes --- All search methods (contains,indexOf,lastIndexOf,remove(Object o)) -- based upon equals method(of type of List eg --Account/Customer/Emp....)
For correct working
1. Identify prim key & create overloaded constr using PK.

2. Using PK , override equals for content equality.

Objective --- Create simple List(ArrayList) of Account & test complete API
1.1
Create Empty Arraylist of Accounts
1.2 Accept a/c info from user till user types "stop" & populate AL.
1.2.1 -- Display AL content using for-each
1.3 Accept account id & display summary or error mesg
1.4 Accept src id , dest id & funds transfer.
1.5 Accept acct id & remove a/c --
1.6 Apply interest on all saving a/cs
1.7 Sort accounts as per asc a/c ids.
1.8 Sort accounts as per desc a/c ids.
1.9 Sort a/cs as per creation date -- w/o touching UDT
2.0 Sort a/cs as per bal

Sorting --- For sorting elements as per Natural(implicit i.e criteria defined within UDT class definition) ordering or Custom(explicit i.e criteria defined outside UDT , in a separate class or anonymus iner class)

Steps for Natural ordering
Natural Ordering is specified in generic i/f
java.lang.Comparable<T>
T -- UDT , class type of the object to be compared.
eg -- Emp,Account , Customer
I/f method
int compareTo(T o)
Steps
1. UDT must implement Comparable<T>
eg : public class Account implements Comparable<Account>
2. Must override method
public int compareTo(T o)
{
use sorting criteria to ret
< 0 if this < o,
=0 if this = o
> 0 if this > o

}
3.Use java.util.Collections class API
Method
public static void sort(List<T> l1)

l1 -- List of type T.

sort method internally invokes compareTo method(prog supplied) of UDT & using advanced sorting algorithm , sort the list elems.

Limitation of natural Ordering
Can supply only 1 criteria at given time & that too is embedded within UDT class definition
Instead keep sorting criteria external --using Custom ordering

Typically use -- Natural ordering in consistence with equals method.

Alternative is Custom Ordering(external ordering)
I/f used is --- java.util.Comparator<T>
T -- type of object to be compared.

Steps
1. Create a separate class (eg. AccountBalComparator) which implements Comparator<T>
eg
public class AccountBalComparator implements Comparator<Account>{...}

2.Implement(override) i/f method -- to supply comparison criteria.

int compare(T o1,T o2)
Must return

< 0 if o1<o2
=0 if o1=o2
> 0 if o1 > o2
3. Invoke Collections class method for actual sorting.
public static void sort(List<T> l1,Comparator<T> c)
parameters
l1 --- List to be sorted(since List is i/f --- any of its implementation class inst. can be passed)
c - instance of the class which has implemented compare method.(or implemented Comparator)

Internally sort method invokes compare method from the supplied Comparator class instance.

More on generic syntax
Constructor of ArrayList(Collection<? extends E> c)
? -- wild card in generic syntax (denotes any unknown type)
--Added for supporting inheritance in generics.'

extends -- keyword in generics, to specify upper bound
? extends E -- E or sub type
Complete meaning --- Can create new populated ArrayList of type E , from ANY Collection(ArrayList,LinkedList,Vector,HashSet,LinkedhashSet,TreeSet) of type E or its sub type.

```
ArrayList<Emp> l1=new ArrayList<>();
l1.add(new Emp(1,"aa",1000);
l1.add(new Emp(2,"ab",2000);
ArrayList<Emp> l2=new ArrayList<>(l1);
sop(l2.size());
----------
 HashSet<Emp> hs=new HashSet<>();
hs.add(new Emp(1,"aa",1000);
hs.add(new Emp(2,"ab",2000);
l2=new ArrayList<>(hs);
----
Vector<Mgr> v1=new Vector<>();
v1.add(new Mgr(....));
v1.add(new Mgr(....));
ArrayList<Emp> l2=new ArrayList<Mgr>(v1);
AL<Mgr> mgrs=new AL<>(hs);
```

Map API
HashMap<K,V> --
1. un-sorted(not sorted as per Natural ordering or custom ordering based criteria) & un-ordered(doesn't remember order of insertion) map implementation class.
2. No duplicate keys.
3. Guarantees constant time performance --- via 2 attributes --initial cpacity & load factor.
4. Allows null key reference(once).
5. Inherently thrd unsafe.

HashMap constrs
1. HashMap<K,V>() --- creates empty map , init capa = 16 & load factor .75
2. HashMap<K,V>(int capa) --- creates empty map , init capa specified & load factor .75
3.HashMap<K,V>(int capa,float loadFactor) --- creates empty map , init capa & load factor specified

4. HashMap constrcutor for creating populated map
HashMap(Map <? extends K,? extends V> m)
? -- wild card in generics, represents unknown type
extends -- represents upper bound
? extends K --- K or its sub type
? extends V -- V or its sub type.
Complete meaning -- Creates populated HM<K,V> from ANY map(ie. any Map imple class) of type K or its sub type & V or its sub type.
eg : Suppose Emp <---- Mgr

```
HM<Integer,Emp> hm=new HM<>();
hm.put(1,e1);
hm.put(2,m1);
HM<Integer,Emp> hm2=new HM<>(hm);
sop(hm2);
LHM<Integer,Emp> lhm=new LHM<>(hm);//legal
HM<Integer,Mgr> hm3=new HM<Integer,Emp>(hm);//javac error
```

TM<Integer,Mgr> hm4=new TM<>();
hm4.put.....
HM<Integer,Emp> hm5=new HM<>(hm4);

HM(Map<? extends K,? extends V>map)

put,get,size,isEmpty,containsKey,containValue,remove

Objective : Create AccountMap
Identify key & value type
create empty unsorted map(HashMap<K,V>) & populate the same

Disp all entries of HM ---can use only toString
1.get acct summary --- i/p --id o/p --- err / dtls
2.Withdraw --- specify Account id & Amt ---- o/p : update acct dtls if acct exists o.w err msg or exc
3.funds transfer ---
i/p sid,dest id, amt

4.remove --- account
i/p id
5.Apply interest on on saving type of a/cs.
or
display all accts created after date.

Attach for-each to map & observe.
Sort the map as per : asc order of accts Ids.
Sort the map as per : desc order of accts Ids
Sort the accts as per : balance

If map sorting involves key based sorting criteria --- can be sorted by converting into TreeMap
Constructors of TreeMap
1. TreeMap() -- Creates empty map , based upon natural ordering of keys

2. TreeMap(Map<? extends K,? extends V> map)
Creates populated map , based upon natural ordering of keys
3. TreeMap(Comparator<? super K> c)
Regarding generic syntax & its usage in TreeMap constructor.
<? super K>
? --- wild card --- any unknown type
super --- gives lower bound
K --- key type
? super K --- Any type which is either K or its super type.

TreeMap(Comparator<? super K> c) --- creates new empty TreeMap, which will sort its element as per custom ordering(i.e will invoke compare(...) of Key type)

<? extends K>
? -- any type or wild card
extends -- specifies upper bound
K -- key type
? extends K --- Any type as Key type or its sub type.

same meaning for <? extends V>
TreeMap(Map<? extends K,? extends V> m)

disp acct ids of all accounts ---impossible directly....(will be done by Collection view of map @ the end)
Apply interest to all saving type a/cs

difficult directly ---so get a collection view of the map & sort the same.

Limitations on Maps
1. Maps can be sorted as per key's criteria alone.
2. can't attach iterators/for-each(till JDK 1.7)/for
3 Maps can be searched as per key's criteria alone.

To fix --- get a collection view of a map (i.e convert map to collection)

API of Map i/f

1. To get set of keys asso. with a Map
Set<K> keySet();
2. To get collection of values from a map
Collection<V> values();
3. To get set of Entries(key & val pair) ---
entrySet
Set<Map.Entry> entrySet()
Methods of Map.Entry

K getKey()
V getValue()

7. conversion from collection to array
Object[] toArray() -- non generic version --- rets array of objects

T[] toArray(T[] type)
T = type of collection .
Rets array of actual type.

8. sorting lists --- Natural ordering creiteria
Using java.util.Collections --- collection utility class.
static void sort(List<E> l1) ---sorts specified list as per natural sorting criteria.

List<E> features
1. List represents ordered collection --- order is significant(It remembers the order of insertion)
2. Allows null references
3. Allows duplicates
4. Supports index based operation

java.util.ArrayList<E> -- E -- type of ref.
1. ArrayList<E> -- constructor
API
ArrayList() -- default constructor. -- creates EMPTY array list object , with init capacity=10,size=0;
eg ---ArrayList<Integer> l1=new ArrayList<>();

1.5 1. ArrayList<E> -- constructor
API
public ArrayList(int capacity) -- -- creates EMPTY array list object , with init capacity=capacity,size=0;
eg ---ArrayList<Integer> l1=new ArrayList<>(100);
l1.add(1);.....l1.add(100);
l1.add(101);//capa=150 --as per JVM spec.

2. add methods
boolean add(E e) --- append
void add(int index,E e) --- insert
void addAll(Collection<E> e) -- bulk append operation
eg : l1 --- AL<Emp>
l1.addAll(.....);
AL,LL,Vector --- legal
HS,TS,LHS --legal
HM,LHM,TM --illegal --javac error

2.5 Retrieve elem from list
E get(int index)
index ranges from ---0 ---(size-1)
java.lang.IndexOutOfBoundsException

3. display list contents using --- toString

4. Attaching Iterator
Collection<E> interface method -- implemented by ArrayList
Iterator<E> iterator()
---places iterator BEFORE 1st element ref.
Iterator<E> i/f methods
boolean hasNext() -- rets true if there exists next element, false otherwise.

E next() --- returns the element next to iterator position

void remove() -- removes last returned element from iterator.

Limitation --- type forward only & can start from 1st elem only.

Regarding exceptions with Iterator/List
1. java.util.NoSuchElementException -- thrown whenever trying to access the elem beyond the size of list via Iterator/ListIterator
2. java.lang.IllegalStateException --- thrown whenever trying to remove elem before calling next().
3. java.util.ConcurrentModificationException-- thrown typically --- when trying to use same iterator/list iterator --after structrually modifying list(eg add/remove methods of list)
Above describes fail-fast behaviour of the Iterator/ListIterator

Exception while accessing element by index.

4. java.lang.IndexOutOfBoundsException -- thrown typically -- while trying to access elem beyond size(0---size-1) --via get

6. Attaching for-each = attaching implicit iterator.

Attaching ListIterator ---scrollable iterator or to beign iteration from a specific element -- List ONLY or list specific iterator.
ListIterator<E> listItearator() --places LI before 1st element
ListIterator<E> listItearator(int index) --places LI before specified index.

4. search for a particular element in list
boolean contains(Object o)

5. searching for 1st occurrence
use -- indexOf
int indexOf(Object o)
rets index of 1st occurrence of specified elem. Rets -1 if elem not found.
searching for last occurrence
use -- lastIndexOf
int lastIndexOf(Object o)
rets index of last occurrence of specified elem. Rets -1 if elem not found.

5.5
E set(int index,E e)
Replaces old elem at spepcified index by new elem.
Returns old elem

6. remove methods
E remove(int index) ---removes elem at specified index & returns removed elem.

boolean remove(Object o) --- removes element specified by argument , rets true -- if elem is removed or false if elem cant be removed.

Objectives in Integer list
0. Create ArrayList of integers & populate it.
1. check if element exists in the list.
2. disp index of 1st occurance of the elem
3. double values in the list --if elem val > 20
4. remove elem at the specified index
5. remove by elem. -- rets true /false.

NOTE :
For searching or removing based upon primary key , in List Implementation classes --- All search methods (contains,indexOf,lastIndexOf,remove(Object o)) -- based upon equals method(of type of List eg --Account/Customer/Emp....)
For correct working
1. Identify prim key & create overloaded constr using PK.

2. Using PK , override equals for content equality.

Objective --- Create simple List(ArrayList) of Account & test complete API
1.1
Create Empty Arraylist of Accounts
1.2 Accept a/c info from user till user types "stop" & populate AL.
1.2.1 -- Display AL content using for-each
1.3 Accept account id & display summary or error mesg
1.4 Accept src id , dest id & funds transfer.
1.5 Accept acct id & remove a/c --
1.6 Apply interest on all saving a/cs
1.7 Sort accounts as per asc a/c ids.
1.8 Sort accounts as per desc a/c ids.
1.9 Sort a/cs as per creation date -- w/o touching UDT
2.0 Sort a/cs as per bal

Sorting --- For sorting elements as per Natural(implicit i.e criteria defined within UDT class definition) ordering or Custom(explicit i.e criteria defined outside UDT , in a separate class or anonymus iner class)

Steps for Natural ordering
Natural Ordering is specified in generic i/f
java.lang.Comparable<T>
T -- UDT , class type of the object to be compared.
eg -- Emp,Account , Customer
I/f method
int compareTo(T o)
Steps
1. UDT must implement Comparable<T>
eg : public class Account implements Comparable<Account>
2. Must override method
public int compareTo(T o)
{
use sorting criteria to ret
< 0 if this < o,
=0 if this = o
> 0 if this > o

}
3.Use java.util.Collections class API
Method
public static void sort(List<T> l1)

l1 -- List of type T.

sort method internally invokes compareTo method(prog supplied) of UDT & using advanced sorting algorithm , sort the list elems.

Limitation of natural Ordering
Can supply only 1 criteria at given time & that too is embedded within UDT class definition
Instead keep sorting criteria external --using Custom ordering

Typically use -- Natural ordering in consistence with equals method.

Alternative is Custom Ordering(external ordering)
I/f used is --- java.util.Comparator<T>
T -- type of object to be compared.

Steps
1. Create a separate class (eg. AccountBalComparator) which implements Comparator<T>
eg
public class AccountBalComparator implements Comparator<Account>{...}

2.Implement(override) i/f method -- to supply comparison criteria.

int compare(T o1,T o2)
Must return

< 0 if o1<o2
=0 if o1=o2
> 0 if o1 > o2
3. Invoke Collections class method for actual sorting.
public static void sort(List<T> l1,Comparator<T> c)
parameters
l1 --- List to be sorted(since List is i/f --- any of its implementation class inst. can be passed)
c - instance of the class which has implemented compare method.(or implemented Comparator)

Internally sort method invokes compare method from the supplied Comparator class instance.

More on generic syntax
Constructor of ArrayList(Collection<? extends E> c)
? -- wild card in generic syntax (denotes any unknown type)
--Added for supporting inheritance in generics.'

extends -- keyword in generics, to specify upper bound
? extends E -- E or sub type
Complete meaning --- Can create new populated ArrayList of type E , from ANY Collection(ArrayList,LinkedList,Vector,HashSet,LinkedhashSet,TreeSet) of type E or its sub type.

```
ArrayList<Emp> l1=new ArrayList<>();
l1.add(new Emp(1,"aa",1000);
l1.add(new Emp(2,"ab",2000);
ArrayList<Emp> l2=new ArrayList<>(l1);
sop(l2.size());
----------
 HashSet<Emp> hs=new HashSet<>();
hs.add(new Emp(1,"aa",1000);
hs.add(new Emp(2,"ab",2000);
l2=new ArrayList<>(hs);
----
Vector<Mgr> v1=new Vector<>();
v1.add(new Mgr(....));
v1.add(new Mgr(....));
ArrayList<Emp> l2=new ArrayList<Mgr>(v1);
AL<Mgr> mgrs=new AL<>(hs);
```

Map API
HashMap<K,V> --
1. un-sorted(not sorted as per Natural ordering or custom ordering based criteria) & un-ordered(doesn't remember order of insertion) map implementation class.
2. No duplicate keys.
3. Guarantees constant time performance --- via 2 attributes --initial cpacity & load factor.
4. Allows null key reference(once).
5. Inherently thrd unsafe.

HashMap constrs
1. HashMap<K,V>() --- creates empty map , init capa = 16 & load factor .75
2. HashMap<K,V>(int capa) --- creates empty map , init capa specified & load factor .75
3.HashMap<K,V>(int capa,float loadFactor) --- creates empty map , init capa & load factor specified

4. HashMap constrcutor for creating populated map
HashMap(Map <? extends K,? extends V> m)
? -- wild card in generics, represents unknown type
extends -- represents upper bound
? extends K --- K or its sub type
? extends V -- V or its sub type.
Complete meaning -- Creates populated HM<K,V> from ANY map(ie. any Map imple class) of type K or its sub type & V or its sub type.
eg : Suppose Emp <---- Mgr

```
HM<Integer,Emp> hm=new HM<>();
hm.put(1,e1);
hm.put(2,m1);
HM<Integer,Emp> hm2=new HM<>(hm);
sop(hm2);
LHM<Integer,Emp> lhm=new LHM<>(hm);//legal
HM<Integer,Mgr> hm3=new HM<Integer,Emp>(hm);//javac error
```

TM<Integer,Mgr> hm4=new TM<>();
hm4.put.....
HM<Integer,Emp> hm5=new HM<>(hm4);

HM(Map<? extends K,? extends V>map)

put,get,size,isEmpty,containsKey,containValue,remove

Objective : Create AccountMap
Identify key & value type
create empty unsorted map(HashMap<K,V>) & populate the same

Disp all entries of HM ---can use only toString
1.get acct summary --- i/p --id o/p --- err / dtls
2.Withdraw --- specify Account id & Amt ---- o/p : update acct dtls if acct exists o.w err msg or exc
3.funds transfer ---
i/p sid,dest id, amt

4.remove --- account
i/p id
5.Apply interest on on saving type of a/cs.
or
display all accts created after date.

Attach for-each to map & observe.
Sort the map as per : asc order of accts Ids.
Sort the map as per : desc order of accts Ids
Sort the accts as per : balance

If map sorting involves key based sorting criteria --- can be sorted by converting into TreeMap
Constructors of TreeMap
1. TreeMap() -- Creates empty map , based upon natural ordering of keys

2. TreeMap(Map<? extends K,? extends V> map)
Creates populated map , based upon natural ordering of keys
3. TreeMap(Comparator<? super K> c)
Regarding generic syntax & its usage in TreeMap constructor.
<? super K>
? --- wild card --- any unknown type
super --- gives lower bound
K --- key type
? super K --- Any type which is either K or its super type.

TreeMap(Comparator<? super K> c) --- creates new empty TreeMap, which will sort its element as per custom ordering(i.e will invoke compare(...) of Key type)

<? extends K>
? -- any type or wild card
extends -- specifies upper bound
K -- key type
? extends K --- Any type as Key type or its sub type.

same meaning for <? extends V>
TreeMap(Map<? extends K,? extends V> m)

disp acct ids of all accounts ---impossible directly....(will be done by Collection view of map @ the end)
Apply interest to all saving type a/cs

difficult directly ---so get a collection view of the map & sort the same.

Limitations on Maps
1. Maps can be sorted as per key's criteria alone.
2. can't attach iterators/for-each(till JDK 1.7)/for
3 Maps can be searched as per key's criteria alone.

To fix --- get a collection view of a map (i.e convert map to collection)

API of Map i/f

1. To get set of keys asso. with a Map
Set<K> keySet();
2. To get collection of values from a map
Collection<V> values();
3. To get set of Entries(key & val pair) ---
entrySet
Set<Map.Entry> entrySet()
Methods of Map.Entry

K getKey()
V getValue()

7. conversion from collection to array
Object[] toArray() -- non generic version --- rets array of objects

T[] toArray(T[] type)
T = type of collection .
Rets array of actual type.

8. sorting lists --- Natural ordering creiteria
Using java.util.Collections --- collection utility class.
static void sort(List<E> l1) ---sorts specified list as per natural sorting criteria.

Difference between ArrayList and LinkedList in Java

Underlying data structure
ArrayList and LinkedList both implements List interface and their methods and results are almost identical.

But ArrayList is a resizable array implementation , where as LinkedList is doubly-linked list implementation of List i/f. LinkelList also implements Deque i/f.

ArrayList Vs LinkedList

1) Search: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly
which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which

requires the traversal through all the elements for searching an element.

2) Deletion: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance:
O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList's each element maintains two pointers (addresses) which points to the both neighbor elements in the list.
Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed.
While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) Inserts Performance: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes
hence the memory consumption is high in LinkedList comparatively.

There are few similarities between these classes which are as follows:

Both ArrayList and LinkedList are implementation of List interface.
They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order
in which the elements got inserted into the List.
Both these classes are non-synchronized and can be made synchronized explicitly by using Collections.synchronizedList method.
The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created,
in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException).

When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)).
Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get(index) method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and
remove operations and more search operations requirement, ArrayList would be your best bet.
Generics Question

1. Given
public class Fruit{}
public class Melon extends Fruit{}
public class WaterMelon extends Melon{}

Which of the following, will be legal ?

1.List<? extends Fruit> fruits=new ArrayList<Fruit>();
2.List<? extends Fruit> fruits=new ArrayList<Melon>();
3.List<? extends Fruit> fruits=new LinkedList<WaterMelon>();
4.List<? extends Fruit> fruits=new Vector<Object>();
5.List<? super Melon> melons=new ArrayList<Fruit>();
6.List<? super Melon> melons=new LinkedList<>();
7.List<? super Melon> melons=new LinkedList<WaterMelon>();

2.
Given
public class Fruit{}
public class Melon extends Fruit{}
public class WaterMelon extends Melon{}

Which of the following, will be legal ?

1. List<Melon> melons=new ArrayList<>();
2. HashSet<Fruit> fruits=new HashSet<>();
3. LinkedList<WaterMelon> wMelons=new LinkedList<>();
4. melons.addAll(fruits);
5. melons.addAll(wMelons);
6. fruits.addAll(melons);
7. fruits.addAll(wMelons);
8. wMelons.addAll(fruits);
9. wMelons.addAll(melons);

Regarding Hashing based Data structures....(eg : HashSet,HashTable,HashMap)

Steps for Creating HashSet

1. Type class in HashSet must override : hashCode & equals method both in consistent manner.

Object class API
public int hashCode() --- rets int : which represents internal addr where obj is sitting on the heap(typically -- specific to JVM internals)

public boolean equals(Object ref) -- Object class rets true : iff 2 refs are referring to the same copy.

2. Rule to observe while overriding these methods
If 2 refs are equal via equals method then their hashCode values must be same.
eg : If ref1.equals(ref2) ---> true then ref1.hashCode() = ref2.hashCode()

Converse may not be mandatory.(i.e if ref1.equals(ref2) = false then its not mandatory that ref1.hashCode() != ref2.hashCode() : but recommended for better working of hashing based D.S)

String class , Wrapper classes , Date related classes have already folowed this contract.

Questions :
1. How does hashing based data structure ensure constant time performance?

If no of elements(size) > capacity * load factor --- re-hashing takes place ---
New data structure is created --(hashtable) -- with approx double the original capacity --- HS takes all earlier entries from orig set & places them in newly created D.S -- using hashCode & equals. -- ensures lesser hash collisions.

2. Why there is a guarantee that a duplicate ref can't exist in yet another bucket ?

Answer is thanks to the contract between overriding of hashCode & equals methods

If two elements are the same (via equals() returns true when you compare them), their hashCode() method must return the same number. If element type violate this, then elems that are equal might be stored in different buckets, and the hashset would not be able to find elements (because it's going to look in the same bucket).

If two elements are different(i.e equals method rets false) , then it doesn't matter if their hash codes are the same or not. They will be stored in the same bucket if their hash codes are the same, and in this case, the hashset will use equals() to tell them apart.

List<E> features
1. List represents ordered collection --- order is significant(It remembers the order of insertion)
2. Allows null references
3. Allows duplicates
4. Supports index based operation

java.util.ArrayList<E> -- E -- type of ref.
1. ArrayList<E> -- constructor
API
ArrayList() -- default constructor. -- creates EMPTY array list object , with init capacity=10,size=0;
eg ---ArrayList<Integer> l1=new ArrayList<>();

1.5 1. ArrayList<E> -- constructor
API
public ArrayList(int capacity) -- -- creates EMPTY array list object , with init capacity=capacity,size=0;
eg ---ArrayList<Integer> l1=new ArrayList<>(100);
l1.add(1);.....l1.add(100);
l1.add(101);//capa=150 --as per JVM spec.

2. add methods
boolean add(E e) --- append
void add(int index,E e) --- insert
void addAll(Collection<E> e) -- bulk append operation
eg : l1 --- AL<Emp>
l1.addAll(.....);
AL,LL,Vector --- legal
HS,TS,LHS --legal
HM,LHM,TM --illegal --javac error

2.5 Retrieve elem from list
E get(int index)
index ranges from ---0 ---(size-1)
java.lang.IndexOutOfBoundsException

3. display list contents using --- toString

4. Attaching Iterator
Collection<E> interface method -- implemented by ArrayList
Iterator<E> iterator()
---places iterator BEFORE 1st element ref.
Iterator<E> i/f methods
boolean hasNext() -- rets true if there exists next element, false otherwise.

E next() --- returns the element next to iterator position

void remove() -- removes last returned element from iterator.

Limitation --- type forward only & can start from 1st elem only.

Regarding exceptions with Iterator/List
1. java.util.NoSuchElementException -- thrown whenever trying to access the elem beyond the size of list via Iterator/ListIterator
2. java.lang.IllegalStateException --- thrown whenever trying to remove elem before calling next().
3. java.util.ConcurrentModificationException-- thrown typically --- when trying to use same iterator/list iterator --after structrually modifying list(eg add/remove methods of list)
Above describes fail-fast behaviour of the Iterator/ListIterator

Exception while accessing element by index.

4. java.lang.IndexOutOfBoundsException -- thrown typically -- while trying to access elem beyond size(0---size-1) --via get

6. Attaching for-each = attaching implicit iterator.

Attaching ListIterator ---scrollable iterator or to beign iteration from a specific element -- List ONLY or list specific iterator.
ListIterator<E> listItearator() --places LI before 1st element
ListIterator<E> listItearator(int index) --places LI before specified index.

4. search for a particular element in list
boolean contains(Object o)

5. searching for 1st occurrence
use -- indexOf
int indexOf(Object o)
rets index of 1st occurrence of specified elem. Rets -1 if elem not found.
searching for last occurrence
use -- lastIndexOf
int lastIndexOf(Object o)
rets index of last occurrence of specified elem. Rets -1 if elem not found.

5.5
E set(int index,E e)
Replaces old elem at spepcified index by new elem.
Returns old elem

6. remove methods
E remove(int index) ---removes elem at specified index & returns removed elem.

boolean remove(Object o) --- removes element specified by argument , rets true -- if elem is removed or false if elem cant be removed.

Objectives in Integer list
0. Create ArrayList of integers & populate it.
1. check if element exists in the list.
2. disp index of 1st occurance of the elem
3. double values in the list --if elem val > 20
4. remove elem at the specified index
5. remove by elem. -- rets true /false.

NOTE :
For searching or removing based upon primary key , in List Implementation classes --- All search methods (contains,indexOf,lastIndexOf,remove(Object o)) -- based upon equals method(of type of List eg --Account/Customer/Emp....)
For correct working
1. Identify prim key & create overloaded constr using PK.
2. Using PK , override equals for content equality.

Objective --- Create simple List(ArrayList) of Account & test complete API
1.1
Create Empty Arraylist of Accounts
1.2 Accept a/c info from user till user types "stop" & populate AL.
1.2.1 -- Display AL content using for-each
1.3 Accept account id & display summary or error mesg
1.4 Accept src id , dest id & funds transfer.
1.5 Accept acct id & remove a/c --
1.6 Apply interest on all saving a/cs
1.7 Sort accounts as per asc a/c ids.
1.8 Sort accounts as per desc a/c ids.
1.9 Sort a/cs as per creation date -- w/o touching UDT
2.0 Sort a/cs as per bal

Sorting --- For sorting elements as per Natural(implicit i.e criteria defined within UDT class definition) ordering or Custom(explicit i.e criteria defined outside UDT , in a separate class or anonymus iner class)

Steps for Natural ordering
Natural Ordering is specified in generic i/f
java.lang.Comparable<T>
T -- UDT , class type of the object to be compared.
eg -- Emp,Account , Customer
I/f method
int compareTo(T o)
Steps
1. UDT must implement Comparable<T>
eg : public class Account implements Comparable<Account>
2. Must override method
public int compareTo(T o)
{
use sorting criteria to ret
< 0 if this < o,
=0 if this = o
> 0 if this > o

}
3.Use java.util.Collections class API
Method
public static void sort(List<T> l1)
l1 -- List of type T.

sort method internally invokes compareTo method(prog supplied) of UDT & using advanced sorting algorithm , sort the list elems.

Limitation of natural Ordering
Can supply only 1 criteria at given time & that too is embedded within UDT class definition
Instead keep sorting criteria external --using Custom ordering

Typically use -- Natural ordering in consistence with equals method.

Alternative is Custom Ordering(external ordering)
I/f used is --- java.util.Comparator<T>
T -- type of object to be compared.

Steps
1. Create a separate class (eg. AccountBalComparator) which implements Comparator<T>
eg
public class AccountBalComparator implements Comparator<Account>{...}

2.Implement(override) i/f method -- to supply comparison criteria.

int compare(T o1,T o2)
Must return

< 0 if o1<o2
=0 if o1=o2
> 0 if o1 > o2
3. Invoke Collections class method for actual sorting.
public static void sort(List<T> l1,Comparator<T> c)
parameters
l1 --- List to be sorted(since List is i/f --- any of its implementation class inst. can be passed)
c - instance of the class which has implemented compare method.(or implemented Comparator)

Internally sort method invokes compare method from the supplied Comparator class instance.

More on generic syntax
Constructor of ArrayList(Collection<? extends E> c)
? -- wild card in generic syntax (denotes any unknown type)
--Added for supporting inheritance in generics.'

extends -- keyword in generics, to specify upper bound
? extends E -- E or sub type
Complete meaning --- Can create new populated ArrayList of type E , from ANY Collection(ArrayList,LinkedList,Vector,HashSet,LinkedhashSet,TreeSet) of type E or its sub type.
ArrayList<Emp> l1=new ArrayList<>();
l1.add(new Emp(1,"aa",1000);

```
l1.add(new Emp(2,"ab",2000);
ArrayList<Emp> l2=new ArrayList<>(l1);
sop(l2.size());
----------
 HashSet<Emp> hs=new HashSet<>();
hs.add(new Emp(1,"aa",1000);
hs.add(new Emp(2,"ab",2000);
l2=new ArrayList<>(hs);
----
Vector<Mgr> v1=new Vector<>();
v1.add(new Mgr(....));
v1.add(new Mgr(....));
ArrayList<Emp> l2=new ArrayList<Mgr>(v1);
AL<Mgr> mgrs=new AL<>(hs);
```

Map API
HashMap<K,V> --
1. un-sorted(not sorted as per Natural ordering or custom ordering based criteria) & un-ordered(doesn't remember order of insertion) map implementation class.
2. No duplicate keys.
3. Guarantees constant time performance --- via 2 attributes --initial cpacity & load factor.
4. Allows null key reference(once).
5. Inherently thrd unsafe.

HashMap constrs
1. HashMap<K,V>() --- creates empty map , init capa = 16 & load factor .75
2. HashMap<K,V>(int capa) --- creates empty map , init capa specified & load factor .75
3.HashMap<K,V>(int capa,float loadFactor) --- creates empty map , init capa & load factor specified

4. HashMap constrcutor for creating populated map
HashMap(Map <? extends K,? extends V> m)
? -- wild card in generics, represents unknown type
extends -- represents upper bound
? extends K --- K or its sub type
? extends V -- V or its sub type.
Complete meaning -- Creates populated HM<K,V> from ANY map(ie. any Map imple class) of type K or its sub type & V or its sub type.
eg : Suppose Emp <---- Mgr

```
HM<Integer,Emp> hm=new HM<>();
hm.put(1,e1);
hm.put(2,m1);
HM<Integer,Emp> hm2=new HM<>(hm);
sop(hm2);
LHM<Integer,Emp> lhm=new LHM<>(hm);//legal
HM<Integer,Mgr> hm3=new HM<Integer,Emp>(hm);//javac error

TM<Integer,Mgr> hm4=new TM<>();
```

hm4.put.....
HM<Integer,Emp> hm5=new HM<>(hm4);

HM(Map<? extends K,? extends V>map)

put,get,size,isEmpty,containsKey,containValue,remove

Objective : Create AccountMap
Identify key & value type
create empty unsorted map(HashMap<K,V>) & populate the same

Disp all entries of HM ---can use only toString
1.get acct summary --- i/p --id o/p --- err / dtls
2.Withdraw --- specify Account id & Amt ---- o/p : update acct dtls if acct exists o.w err msg or exc
3.funds transfer ---
i/p sid,dest id, amt

4.remove --- account
i/p id
5.Apply interest on on saving type of a/cs.
or
display all accts created after date.

Attach for-each to map & observe.
Sort the map as per : asc order of accts Ids.
Sort the map as per : desc order of accts Ids
Sort the accts as per : balance

If map sorting involves key based sorting criteria --- can be sorted by converting into TreeMap
Constructors of TreeMap
1. TreeMap() -- Creates empty map , based upon natural ordering of keys
2. TreeMap(Map<? extends K,? extends V> map)
Creates populated map , based upon natural ordering of keys

3. TreeMap(Comparator<? super K> c)
Regarding generic syntax & its usage in TreeMap constructor.
<? super K>
? --- wild card --- any unknown type
super --- gives lower bound
K --- key type
? super K --- Any type which is either K or its super type.

TreeMap(Comparator<? super K> c) --- creates new empty TreeMap, which will sort its element as per custom ordering(i.e will invoke compare(...) of Key type)

<? extends K>
? -- any type or wild card
extends -- specifies upper bound
K -- key type
? extends K --- Any type as Key type or its sub type.

same meaning for <? extends V>
TreeMap(Map<? extends K,? extends V> m)

disp acct ids of all accounts ---impossible directly....(will be done by Collection view of map @ the end)
Apply interest to all saving type a/cs

difficult directly ---so get a collection view of the map & sort the same.

Limitations on Maps
1. Maps can be sorted as per key's criteria alone.
2. can't attach iterators/for-each(till JDK 1.7)/for
3 Maps can be searched as per key's criteria alone.

To fix --- get a collection view of a map (i.e convert map to collection)

API of Map i/f

1. To get set of keys asso. with a Map
Set<K> keySet();
2. To get collection of values from a map
Collection<V> values();
3. To get set of Entries(key & val pair) ---
entrySet
Set<Map.Entry> entrySet()
Methods of Map.Entry
K getKey()
V getValue()

7. conversion from collection to array
Object[] toArray() -- non generic version --- rets array of objects

T[] toArray(T[] type)
T = type of collection .
Rets array of actual type.

8. sorting lists --- Natural ordering creiteria
Using java.util.Collections --- collection utility class.
static void sort(List<E> l1) ---sorts specified list as per natural sorting criteria.
Regarding Hashing based Data structures....(eg : HashSet,HashTable,HashMap)

Steps for Creating HashSet

1. Type class in HashSet must override : hashCode & equals method both in consistent manner.

Object class API
public int hashCode() --- rets int : which represents internal addr where obj is sitting on the heap(typically -- specific to JVM internals)

public boolean equals(Object ref) -- Object class rets true : iff 2 refs are referring to the same copy.

2. Rule to observe while overriding these methods
If 2 refs are equal via equals method then their hashCode values must be same.
eg : If ref1.equals(ref2) ---> true then ref1.hashCode() = ref2.hashCode()
Converse may not be mandatory.(i.e if ref1.equals(ref2) = false then its not mandatory that ref1.hashCode() != ref2.hashCode() :
but recommended for better working of hashing based D.S)

String class , Wrapper classes , Date related classes have already folowed this contract.

Questions :
1. How does hashing based data structure ensure constant time performance?

If no of elements(size) > capacity * load factor --- re-hashing takes place ---
New data structure is created --(hashtable) -- with approx double the original capacity ---
HS takes all earlier entries from orig set & places them in newly created D.S -- using hashCode & equals. -- ensures lesser hash collisions.

2. Why there is a guarantee that a duplicate ref can't exist in yet another bucket ?

Answer is thanks to the contract between overriding of hashCode & equals methods

If two elements are the same (via equals() returns true when you compare them), their hashCode() method must return the same number.
If element type violate this, then elems that are equal might be stored in different buckets,
and the hashset would not be able to find elements (because it's going to look in the same bucket).

If two elements are different(i.e equals method rets false) , then it doesn't matter if their hash codes are the same or not.
They will be stored in the same bucket if their hash codes are the same, and in this case, the hashset will use equals() to tell them apart.
Regarding Hashing based Data structures....(eg : HashSet,HashTable,HashMap)

Steps for Creating HashSet

1. Type class in HashSet must override : hashCode & equals method both in consistent manner.

Object class API
public int hashCode() --- rets int : which represents internal addr where obj is sitting on the heap(typically -- specific to JVM internals)

public boolean equals(Object ref) -- Object class rets true : iff 2 refs are referring to the same copy.

2. Rule to observe while overriding these methods
If 2 refs are equal via equals method then their hashCode values must be same.
eg : If ref1.equals(ref2) ---> true then ref1.hashCode() = ref2.hashCode()
Converse may not be mandatory.(i.e if ref1.equals(ref2) = false then its not mandatory that ref1.hashCode() != ref2.hashCode() :
but recommended for better working of hashing based D.S)

String class , Wrapper classes , Date related classes have already folowed this contract.

Questions :
1. How does hashing based data structure ensure constant time performance?

If no of elements(size) > capacity * load factor --- re-hashing takes place ---
New data structure is created --(hashtable) -- with approx double the original capacity --- HS takes all earlier entries from orig set &
places them in newly created D.S -- using hashCode & equals. -- ensures lesser hash collisions.

2. Why there is a guarantee that a duplicate ref can't exist in yet another bucket ?

Answer is thanks to the contract between overriding of hashCode & equals methods

If two elements are the same (via equals() returns true when you compare them), their hashCode() method must return the same number.
If element type violate this, then elems that are equal might be stored in different buckets, and the hashset would not be able to find elements
(because it's going to look in the same bucket).

If two elements are different(i.e equals method rets false) , then it doesn't matter if their hash codes are the same or not.
They will be stored in the same bucket if their hash codes are the same, and in this case, the hashset will use equals() to tell them apart.

How HashMap internally works in Java

Hash Map is one of the most used collection. It doesn't extend from Collection i/f.
BUT collection view of a map can be obtained using keySet,values or entrySet()

Internal Implementation

HashMap works on the principal of hashing.

What is hashing ?
Hashing means using some function or algorithm to map object data to some representative integer value.

Map.Entry interface ---static nested interface of Map i/f

This interface represents a map entry (key-value pair).

HashMap in Java stores both key and value object ref , in bucket, as an object of Entry class which implements this nested interface Map.Entry.

hashCode() -HashMap provides put(key, value) for storing and get(key) method for retrieving Values from HashMap.

When put() method is used to store (Key, Value) pair, HashMap implementation calls hashcode on Key object to calculate a hash that is used to
find a bucket where Entry object will be stored.

When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.

equals() - equals() method is used to compare objects for equality. In case of HashMap key object is used for comparison, also using equals()
method Map knows how to handle hashing collision (hashing collision means more than one key having the same hash value, thus assigned to the same bucket.
In that case objects are stored in a linked list (growable --singly linked)

Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0,
table[1] as bucket1 and so on.

HashMap uses equals() method to see if the key is equal to any of the already inserted keys (Recall that there may be more than one entry in the same bucket).
Note that, with in a bucket key-value pair entries (Entry objects) are stored in a linked-list . In case hash is same, but equals() returns false (which essentially means more than one key having the same hash or hash collision) Entry objects are stored, with in the same bucket, in a linked-list.

In short , there are three scenarios in case of put() -

Using hashCode() method, hash value will be calculated. Using that hash it will be ascertained, in which bucket particular entry will be stored.
equals() method is used to find if such a key already exists in that bucket, if no then a new node is created with the map entry and stored within the same bucket.
A linked-list is used to store those nodes.
If equals() method returns true, which means that the key already exists in the bucket. In that case, the new value will overwrite the old value for the matched key.

How get() methods works internally

As we already know how Entry objects are stored in a bucket and what happens in the case of Hash Collision it is easy to understand what happens when key object
is passed in the get method of the HashMap to retrieve a value.

Using the key again hash value will be calculated to determine the bucket where that Entry object is stored, in case there are more than one Entry object with in
the same bucket stored as a linked-list equals() method will be used to find out the correct key. As soon as the matching key is found get() method will return
the value object stored in the Entry object.

In case of null Key

Since HashMap also allows null, though there can only be one null key in HashMap. While storing the Entry object HashMap implementation checks if the key is null,

in case key is null, it always map to bucket 0 as hash is not calculated for null keys.

HashMap changes in Java 8

Though HashMap implementation provides constant time performance O(1) for get() and put() method but that is in the ideal case when the Hash function distributes
the objects evenly among the buckets.

But the performance may worsen in the case hashCode() used is not proper and there are lots of hash collisions. In case of hash collision entry objects are stored
as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a
worst case scenario the complexity becomes O(n).

To address this issue in Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with
storing Entry objects in linked list but after the number of items in a bucket becomes larger than a certain threshold, the bucket will change from using a linked
list to a balanced tree, this will improve the worst case performance from O(n) to O(log n).
Map Overview (refer to the diagram "regarding Maps")

Map Implementation class
1. HashMap<K,V>
1.1 Constructors
1. HashMap()
2. HashMap(int initCapa)
3. HashMap(int initCapa,float loadFactor)
4. public HashMap(Map<? extends K,? extends V> m)
Meaning : Creates populated HashMap of type K,V from any Map (AL/LL/Vector : javac err
HS/LHS/TS : javac err
HM/LHM/TM : no error) having generic type of K or its sub type & V or its sub type

Steps n API

0. Create new empy map to store account details
HashMap<Integer,BankAccount> hm=new HM<>();//size=0,capa=16,L.F=0.75

0.5 Create new account
Map i/f API
1. public V put(K key,V value)
Meaning : It will insert the new entry into map.If key already exists : it will replace old value by new value.
Rets : null in case of new entry or old value ref. in case of existing entry.
eg : sop(map.put(k1,v1));//null
sop(map.put(k2,v2));//null
sop(map.put(k3,v3));//null
sop(map.put(k1,v4));//v1

//which entries : k1:v4,k2:v2 ,k3:v3

2. public V putIfAbsent(K key,V value)
eg : sop(map.putIfAbsent(k1,v1));//null
sop(map.putIfAbsent(k2,v2));//null
sop(map.putIfAbsent(k3,v3));//null
sop(map.putIfAbsent(k1,v4));//v1
//which values(entries) : k1:v1 k2:v2 k3:v3

3. public void putAll(Map<? extends K,? extends V> m)
eg : map1.putAll(map2);
Meaning : It will copy all entries from map2 ----> map1
(put : replace)

4. public V get(Object key)
Rets value type of ref if key is found else rets null.
eg : map.get(k2) : v2
map.get(k10) : null

5. boolean containsKey(Object key)
Returns true if this map contains a mapping for the specified key , otherwise false;
eg : map.containsKey(k1) ---true

6. boolean containsValue(Object value)
Returns true if this map maps one or more keys to the specified value.
eg : map : k1:v1 k2:v2 k3:v3
map.containsValue(v3) ---- true

containsKey : O(1)
containsValue : O(n)

7. public V remove(Object K)
Tries to remove the entry(=mapping=key n value pair) if key is found --rets existing value ref.
Rets null if key is not found.;
eg : map : k1:v1 k2:v2 k3:v3
sop(map.remove(k2));//v2
sop(map);//k1:v1 k3:v3
sop(map.remove(k20));//null
sop(map);//k1:v1 k3:v3

How to overcome limitations of Map (can't iterate over map , can't search/sort/remove by any value based criteria)

Solution : Convert the map into its Collection view

1. How to extract key type refs from a Map ?
public Set<K> keySet()

eg : HM<Integer,BankAccount> hm=new HM<>();
add some a/cs
Set<Integer> keys =hm.keySet();//O(n)

2. How to get value type of references from a Map ?
public Collection<V> values();
eg : HM<Integer,BankAccount> hm=new HM<>();
added some a/cs ....
Collection<BankAccount> accts =hm.values();//O(n)

3. How to get key-value pair(entry) of references from a Map ?
Map : i/f
Nested i/f : Map.Entry<K,V> : Entry in a Map
public Set<Map.Entry<K,V>> entrySet();

4. Method of Map.Entry i/f
public K getKey()
public V getValue();

What is a Stream?

A sequence of elements from a source that supports data processing operations.

• Sequence of elements - Like a collection, a stream provides an interface to a sequenced set of values of a specific type.

• Source - Streams refer to collections, arrays, or I/O resources.

• Data processing operations - Supports common operations from functional programming languages.
• e.g. filter, map, reduce, find, match, sort etc

They have nothing to do with java.io -- InputStream or outputstream

The Streams also support Pipelining and Internal Iterations. The Java 8 Streams are designed in such a way that most of its stream operations returns Streams only. This help us creating chain of various stream operations. This is called as pipelining. The pipelined operations looks similar to a sql query.(or Hibernate Query API)

Concurrency is IMPORTANT. But it comes with a learning curve.
So , Java 8 goes one more step ahead and has developed a Streams API which allows us to use multi cores easily.

Parallel processing = divide a larger task into smaller sub tasks (forking), then processing the sub tasks in parallel and then combining the results together to get the final output (joining).

Java 8 Streams API provides a similar mechanism to work with Java Collections.

The Java 8 Streams concept is based on converting Collections to a Stream (or arrays to a stream) , processing the elements in parallel and then gathering the resulting elements into a Collection.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless. What does that mean?

A function is non-interfering when it does not modify the underlying data source of the stream, e.g.

```
List<String> myList =Arrays.asList("a1", "a2", "b1", "c2", "c1");
myList.stream().filter(s -> s.startsWith("c")).map(String::toUpperCase) .sorted()
  .forEach(System.out::println);
```

In the above example no lambda expression does modify myList by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic,
e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

API
The starting point is java.util.stream.Stream i/f
Different ways of creating streams
1. Can be created of any type of Collection (Collection, List, Set):

java.util.Collection<E> API

1.1 default Stream<E> stream()
1.2 public default Stream<E> parallelStream()

NOTE that Java 8 streams can't be reused, will raise IllegalStateException

2. Stream of Array
How to create stream from an array?

Arrays class API
public static <T> Stream<T> stream(T[] array)
Returns a sequential Stream with the specified array as its source.

3. Can be attached to Map ,via entrySet method.
Refer to CreateStreams.java

4. To create streams out of three primitive types: int, long and double.
As Stream<T> is a generic interface , can't support primitives.
So IntStream, LongStream, DoubleStream are added.
API of java.util.stream.IntStream
4.1 static IntStream of(int... values)
Returns a sequential ordered stream whose elements are the specified values.

4.2 static IntStream range(int startInclusive,int endExclusive)
Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

4.3 static IntStream rangeClosed(int startInclusive,int endInclusive)
Returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1.

5. To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed – the source, intermediate operation(s) and a terminal operation.

6.java.util.stream.Stream<T> i/f API

6.1 Stream<T> skip(long n)
Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream(stateful intermediate operation)

6.2 map
<R> Stream<R> map(Function<? super T,? extends R> mapper)
Returns a stream consisting of the results of applying the given function to the elements of this stream(intermediate stateless operation)

mapToInt
IntStream mapToInt(ToIntFunction<? super T> mapper)
Returns an IntStream consisting of the results of applying the given function to the elements of this stream.

6.3 filter
Stream<T> filter(Predicate<? super T> predicate)
Returns a stream consisting of the elements of this stream that match the given predicate.(intermediate stateless operation)

ref : StreamAPI1.java
---
7. Confirm lazyness of streams.
Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.
ref : LazyStreams.java

8. Reduce operation
Readymade methods of IntStream
count(), max(), min(), sum(),average()

9. Customized reduce operation

ref : ReduceStream.java

10 collect

Reduction of a stream can also be executed by another terminal operation – the collect() method.
eg : StreamCollect.java
Good examples in java.util.stream.Collectors -api docs.

Details ---

1. Streams are functional programming design pattern for processing sequence of elements sequentially or in parallel.(a.k.a Monad in functional programming)

2. Stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements

3. Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations .
Terminal operations are either void or return a non-stream result.

4. They can't be reused.

5. Collections vs Streams:
Collections are in-memory data structures which hold elements within it. Each element in the collection is computed before it actually becomes a part of that collection. On the other hand Streams are fixed data structures which computes the elements on-demand basis.

The Java 8 Streams -- lazily constructed Collections, where the values are computed when user demands for it.
Actual Collections behave absolutely opposite to it and they are set of eagerly computed values (no matter if the user demands for a particular value or not).

Functional programming In java

Functional Programming (FP) is one of the type of programming pattern that helps the process of building application by using of higher order functions, avoiding shared state, mutable data

Functional programming vs OOP

Declarative vs Imperative :

Functional programming is a declarative pattern, meaning that the program logic is expressed without explicitly describing the flow control.Imperative programs spend lines of code describing the specific steps used to achieve the desired results — the flow control

Declarative programs remove the flow control process, and instead spend lines of code describing the data flow.

In Functional programming

Functions are treated as a first class citizens.
Meaning : You can
1.1 define anonymous functions
1.2 assign a function to a variable (function literal)
1.3 pass function as a parameter
1.4 return function as a return value

Why FP ?

1. To write more readable , maintainable , clean & concise code.
2. To use APIs easily n effectively.
3. To enable parallel processing

OOP uses imperative style of programming (where you will have to specify what's to be done & how --both) .
FP uses declarative style of programming (where you will just have to specify what's to be done

2. Functional interfaces
An interface which has exactly single abstract method(SAM) is called functional interface.

eg Runnable,Comparable,Comparator,Iterable,Consumer,Predicate,Supplier,Function...
Java SE 8 has introduced a new package for functional i/f
java.util.function

New annotation introduced -- @FunctionalInterface
(since Java SE 8)

Functional i/f references can be substituted by lambda expressions, method references, or constructor references.

Solve -- Is following valid functional interface ?
public interface A { double calc(int a,int b);} :
public interface B extends A {} :
public interface C extends A { void show();} :
public interface D {} -- Marker / empty / tag i/f :
public interface E extends A {default void show(){}} --

13. Addition of "default" keyword to add default method implementation , in interfaces.

Java 8 enables us to add non-abstract method implementations to interfaces by using the default keyword. This feature is also known as Extension Methods.

Why default keyword ?
1. To maintain backward compatibility with earlier Java SE versions

2. To avoid implementing new functionality in all implementation classes.

eg : Java added in Iterable<T> interface
default void forEach(Consumer<? super T> action) -- as a default method implementation

eg :

```
interface Formula {
   double calculate(double a);//javac adds implicit keywords : public n abstract
           //javac adds implicit keyword public
   default double sqrt(double a,double b) {
      return Math.sqrt(a+b);
   }
}
```

Q : If you write an implementation class MyFormula

```
public class MyFormula imple Formula
{
 .....
}
```

Which methods have to be implemented to avoid javac err?
1. calculate
2. sqrt
3. both
4. neither

Q : Can MyFormula class override the def. of sqrt ?

1 Display all elements of ArrayList
forEach
2. Create AL of integers
remove all odd numbers.

3. Create AL of emps
Remove underperforming employees (performance index < 7)
Dsiplay the list

4. Enter Java 8 Streams
1.Create int[] ---> IntStream & display its contents.
2. Create AL<Integer> , populate it.
Convert it to seq stream & display elems
Convert it to parallel stream & display elems
3. Create stream of ints between 1-100 & display all even elements.
(Hint : IntStream methods --range,filter,forEach)

4. Display all emp names from a particular dept , joined after specific date
(stream,filter,forEach)

5. Display sum of all even nos between 1-100 .
(stream , filter ,sum)
6. Display sum of salaries of all emps from a specific dept

7. Create a supplier of random numbers
eg :

```
Supplier<Double> randomValue = () -> Math.random();

// Print the random value using get()
System.out.println(randomValue.get());
```

8. Create a supplier of LocalDate & Time

eg :

```
Supplier<LocalDateTime> s = () -> LocalDateTime.now();
LocalDateTime time = s.get();

System.out.println(time);

Supplier<String> s1 = () -> dtf.format(LocalDateTime.now());
String time2 = s1.get();

System.out.println(time2);
```

1. Addition of "default" keyword to add default method implementation , in interfaces.

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as Extension Methods.

Why default keyword ?
1. To maintain backward compatibility with earlier Java SE versions
2. To avoid implementing new functionality in all implementation classes.
eg : Java added in Iterable<T> interface
default void forEach(Consumer<? super T> action) -- as a default method implementation

eg :

```
interface Formula {
   double calculate(double a);//public n abstract
         //public
   default double sqrt(double a,double b) {
      return Math.sqrt(a+b);
   }
}
```

In case of ambiguity or to refer to def imple. from i/f -- use InterfaceName.super.methodName(...) syntax

2 Can add static methods in java interfaces --- It's a better alternative to writing static library methods in helper class(eg --Arrays or Collections)

Such static methods can't be overridden in implementation class.
BUT can be re-declared.

They have to be invoked using interface name , even in implementation or non implementation classes.(otherwise compiler error)

3. Functional interfaces ---An interface which has exactly single abstract method(SAM) is called functional interface. (were present earlier)
eg Runnable,Comparable,Comparator,Iterable,Consumer,Predicate...
New annotation introduced -- @FunctionalInterface
(since Java SE 8 : a new package --java.util.function --functional interfaces)

Functional i/f references can be substituted by lambda expressions, method references, or constructor references.

Solve -- Is following valid functional interface ?
public interface A { double calc(int a,int b);} : YES (contains SAM)
public interface B extends A {} : YES (inherits SAM)
public interface C extends A { void show();} : NO (2 abstract methods)
public interface D {} --NO (marker / tag i/f)
public interface E extends A {default void show(){}
static boolean test(int data) {...}
} --YES (SAM)

4. Refer to readme of lambda expressions.

5. Date/Time APIs
Java 8 Date/Time related APIs
Java 8 New Features
java.time : new package is introduced
LocalDate : Date (immutable)(yr-mon-day) : inherently thrd safe.
API
1. public static LocalDate now()
Obtains the current date from the system clock in the default time-zone.
eg :
LocalTime : Time (immutable)(hr-min-sec) : inherently thrd safe.
LocalDateTime : Date n Time : inherently thrd safe.
eg : sop("curnt date "+now());
2. public static LocalDate of(int year,int month,int dayOfMonth)
Obtains an instance of LocalDate from a year, month and day.
eg : ?????
3. public static LocalDate parse(CharSequence text)
Obtains an instance of LocalDate from a text string such as 2007-12-03.
eg : LocalDate dt=parse(sc.next());//0 based dates

4. Methods :
isBefore,isAfter,isEqual
5. Can you change default DateTime format ? : YES
How : use java.time.format.DateTimeFormatter

Regarding functional programming

What is functional programming ?

Functional programming is the way of writing s/w applications that uses only pure fuctions & immutable values.

Main concepts of FP are
1. Pure functions & side effects
2. Referential transperency
3. First class functions & higher order funcs.
4. Anonymous functions
5. Immutability
6. Recursion & tail recursion
7. Statements
8. Strict & Lazy evaluations
9. Pattern Matching
10 Closures

Why Functional Programming paradigm
Elegance and simplicity
Easier decomposition of problems
Code more closely tied to the problem domain

Through these , one can achieve :

Straightforward unit testing
Easier debugging
Simple concurrency
Lambda expressions

It's derived from lambda caculus.

It was a big change in calculus world , which gave tremedous ease in maths

Now the same concept is being used in programming languages.
1st language to use lambda
LISP
c , c++ , c# , scala , javascript , python

Finally in java also(Java SE 8 onwards)

Why lamda expressions?

Java is an object-oriented language. With the exception of primitive data types, everything in Java is an object. Even an array is an Object. Every class creates instances that are objects. There is no way of defining just a function / method which stays in Java all by itself. There is no way of passing a method as argument or returning a method body for that instance.
i.e passing /returning the behaviour was not possible till java 8.

It was slightly possible using anonymous inner classes --but that still required us to write a class !

What is lambda expression ?

Concise anonymous function which can be passed around
It has
1. list of params
2. body
3. return type.(optional)

Lambda expressions in Java is usually written using syntax (argument) -> (body). For example:

(type1 arg1, type2 arg2...) -> { body }

Following are some examples of Lambda expressions.

1.(int a, int b) -> { return a + b; }
OR can be reduced to
(int a, int b) -> a + b
OR further can be reduced to
(a,b) -> a+b

2. () -> System.out.println("Hello World");

3. s -> System.out.println(s)

4. () -> 42

5. () -> 3.1415

Above is just a syntax of lambda . But how to use them ?
Answer is ---You can use lambda expressions as targets of functional i/f reference.

Why lambdas --
Easy way of passing a behaviour.
Till Java SE 7 , there was no way of passing a method as argument or returning a method body for that instance.
To enable this style of functional programming , lambdas are introduced.

How to pass a behaviour ? : Using lambda expression

What is a functional programming paradigm ?
A language where below features are supported.

Functions are treated as a first class citizens.
Meaning : You can
1.1 define anonymous functions
1.2 assign a function to a variable (function literal)
1.3 pass function as a parameter
1.4 return function as a return value

Why FP ?

1. To write more readable , maintainable , clean & concise code.
2. To use APIs easily n effectively.
3. To enable parallel processing

OOP uses imperative style of programming (where you will have to specify what's to be done & how --both) .
FP uses declarative style of programming (where you will just have to specify what's to be done

eg :
Find out the average salary of emp from the specified dept.

How will you do it in imperative manner?
eg : List<Emp> l1=new AL<>();
l1.add(..);.....
String dept=sc.next();
double total=0;
int num=0;
for(Emp e : l1)
 if(e.getDept().equals(dept)) {
  total += e.getSal();
  num++;
 }

sop(total/num);
Vs
How to do it in declarative style ?
eg : List<Emp> l1=new AL<>();
l1.add(..);.....

l1.stream().filter(e->e.getDept().equals(dept)).mapToDouble(Emp::getSal).average().getAsDouble()

Immediate Objectives

1. Create your own functional interface n use it in lambda expression
eg : Perform ANY arithmetic operation on 2 double values & return the result
eg --add/multiply/subtract/divide....

Explore Existing higher order functions

2. Iterable : forEach

3. Collection : removeIf(Predicate<? super T> filter)
func method : public boolean test(T o)

Objective : remove details of accts having balance < specified bal
eg : list.removeIf(new Predicate()
{
  public boolean test(BankAccount a)
  {
    return a.getBalance() < specifiedBalance;
  }
}
);
OR
list.removeIf(a -> a.getBalance() < specifiedBalance);

4. Map : forEach

5. Sorting : custom ordering
Objective : sort list of account as per creation date n balance
Collections.sort(list,(a1,a2) -> {.....});

Objective --
1. Perform ANY operation on 2 double values & return the result
eg --add/multiply/subtract/divide....

2. Convert from ANY src type to ANY dest type
eg : String ---> length
String ----> upper case string

celcius ---> fahrenheit (f=c*1.8+32)
Student ---> GPA
number ---> square root

1. Create generic interface Converter<F,T> to specify single abstract method --convert , from F ---> T

2. Create a Tester class (with main method)
Add a static method to test the converter.

I/P -- 1. conversion source type(From)
2. conversion behaviour

O/P -- conversion result.(To)

```
public static <F,T> testConverter(F from, Converter<F,T> c)
{
  return c.convert(from);
}
```

3. main(..) invokes this static method for testing Converter.
But what will be 2nd argument ?

Exam objective :

Main Differences between Lambda Expression and Anonymous class

1. One key difference between using Anonymous class and Lambda expression is the use of "this" keyword.

For anonymous class ‘this’ keyword resolves to anonymous class, whereas for lambda expression ‘this’ keyword resolves to enclosing class where lambda is written.

2. Another difference between lambda expression and anonymous class is in the way these two are compiled.
Java compiler compiles lambda expressions and convert them into private method of the class.
I/O handling

Desc of FileInputStream --- java.io.FileInputStream
bin i/p stream connected to file device(bin/char) -- to read data.

Desc of FileOutputStream --- java.io.FileOutputStream
bin o/p stream connected to file device(bin/char) -- to write data.

Desc of FileReader--- java.io.FileReader
char i/p stream connected to file device(char) -- to read data.

Desc of FileWriter--- java.io.FileWriter
char o/p stream connected to file device(char) -- to write data.

Objective --- Read data from text file in buffered manner.

1. java.io.FileReader(String fileName) throws FileNotFoundException
--- Stream class to represent unbuffered char data reading from a text file.
Has methods -- to read data using char/char[]
eg -- public int read() throws IOException

public int read(char[] data) throws IOException
Usage eg-- char[] data=new char[100];
int no= fin.read(data);

public int read(char[] data,int offset,int noOfChars) throws IOException
Usage eg-- char[] data=new char[100];
int no= fin.read(data,10,15);
eg -- 12 chars available
no=12;data[10]----data[21]

1.5 FileReader(File f) throws FileNotFoundException
java.io.File -- class represents path to file or a folder.

2. Improved version -- Buffered data read .
For char oriented streams--- java.io.BufferedReader(Reader r)

API of BR ---
String readLine() --- reads data from a buffer in line by line manner-- & rets null at end of Stream condition.

Objective -- Replace JDK 1.6 try-catch-finally BY JDK 1.7 try-with-resources syntax.
Meaning --- From Java SE 7 onwards --- Introduced java.lang.AutoCloseable -- i/f
It represents --- resources that must be closed -- when no longer required.
i/f method
public void close() throws Exception-- closing resources.
java.io --- classes -- have implemented this i/f -- to auto close resource when no longer required.
syntax of try-with-resources
try (//open one or multiple AutoCloseable resources)
{ .....
} catch(Exception e)
{
}

Objective ---To confirm device independence of Java I/O --- replace File device by Console
i.e --- Read data from console i/p --- in buffered manner till 'stop' & echo back it on the console.

required stream classes --- BR(ISR(System.in))
Alternative is --- use Scanner class.

Adv. of Scanner over above chain ----- contains ready-made parsing methods(eg --- nextInt,nextDouble.....)
But Scanner is not Buffered Stream
Can combine both approaches.(new Scanner(br.readLine())

Objective --- Combine scanner & buffered reader api --- to avail buffering + parsing api. --- BufferedReader provides buffering BUT no simple parsing API. -- supplies br.readLine only

Scanner -- Can be attached to file directly
Constr -- Scanner(File f)
BUT no buffering .

How to use both?
Create BR br=new BR(new FR(...));
while ((s=br.readLine())!=null)
 {
   //scanner can be attached to string ---Scanner(String s)
Scanner sc=new Scanner(s);
// parse data using Scanner API --next,nextInt,nextBoolean
}

Overloaded constructor of FileReader(File f)

java.io.File ---- class represents path to file / folder
Regarding java.io.File -----
Does not follow stream class hierarchy, extends Object directly.
File class --- represents abstract path which can refer to file or folder.
Usage --- 1. To access/check file/folder attributes(exists,file or folder,read/w/exec permisssions,path,parent folder,create new empty file,create tmp files & delete them auto upon termination,mkdir,mkdirs,rename,move,size,last modified ,if folder---list entries from folder,filter entries)
Constructor ---
File (String path) ---
eg --- File f1=new File("abc.dat");
if (f1.exists() && f1.isFile() && f1.canRead())
...attach FileInputStream or FileReader

File (String path) ---
File class API --- boolean exists(),boolean isFile() , boolean canRead()

Objective --- Text File copy operation --- in buffered manner.
For writing data to text file using Buffered streams

java.io.PrintWriter --- char oriented buffered o/p stream --- which can wrap any device.(Binary o/p stream or Char o/p stream)
Constructors---
PrintWriter(Writer w) --- no auto flushing,no conversion, only buffering
PrintWriter(Writer w, boolean flushOnNewLine)--- automatically flush buffer contents on to the writer stream --upon new line
PrintWriter(OutputStream w) --- can wrap binary o/p stream -- buffering +conversion(char-->binary),no auto-flush option
PrintWriter(OutputStream w , boolean flushOnNewLine) ---
API Methods----print/println/printf same as in PrintStream class(same type as System.out)

Stream class which represents --- Char o/p stream connected to Text file. --- java.io.FileWriter
Constructor
FileWriter(String fileName) throws IOException -- new file will be created & data will be written in char format.
FileWriter(String fileName,boolean append) --- if append is true , data will be appended to existing text file.

Collection & I/O
Objective ---
Items Inventory
Item -- code(String-Prim key),desc,category,quantity,price,shipmentDate
constr,toString.

Create suitable collection of Items(HashMap) --- sort map as per desc item code ,& store sorted item dtls in 1 text file .
NOTE : individual item rec MUST be written on separate line.

Sort items as shipment Date & store sorted dtls in another file . Before exiting ensure closing of data strms .
(buffered manner)

Objective -- Restore collection of items created in above requirement ---in form of HashMap . -- buffering is optional.

Objective --- using Binary file streams.
Classes --- FileInputStream -- unbuffered bin i/p stream connected to bin file device.
FileOutputStream --unbuffered bin o/p stream connected to bin file device.

But these classes --- dont provide buffering & have only read() write() methods in units of bytes/byte[]

API of InputStream class
1. int read() throws IOException
Will try to read 1 byte from the stream.
Data un-available method blocks.
Returns byte--->int to caller.
eg -- int data=System.in.read();
2. int read(byte[] bytes) throws IOException
Will try to read data from underlying stream.
Data un-available -- method blocks.
Rets actual no of bytes read.

eg :
byte[] bytes=new byte[100];
int no=System.in.read(bytes);
no data --method blocks.
10 bytes available -no =10;bytes[0]-----bytes[9]
110 bytes available -- no=100;bytes[0]....bytes[99]

3. int read(byte[] bytes,int offset,int maxNoOfBytes) throws IOException
Will try to read data from underlying stream.
Data un-available -- method blocks.
Rets actual no of bytes read.
eg :
byte[] bytes=new byte[100];
int no=System.in.read(bytes,10,15);
no data --BLOCKS
5 bytes available --no=5;bytes[10].....bytes[14]
110 bytes available -- no=15;bytes[10]..bytes[24]

4. int available() throws IOException
Returns no of available bytes in the stream
no data ---DOESN't block -- rets 0.

Important API of OutputStream
1. public void write(int byte) throws IOException
2. public void write(byte[] bytes) throws IOException
3. public void write(byte[] bytes,int offset,int maxNo) throws IOException
bytes[offset].....bytes[offset+maxNo-1] -- written out to stream
4. void flush() throws IOException

5. void close() throws IOException

Using BIS(BufferedInputStream) -- enables buffering BUT doesn't provide any advanced API(ie. read(), read(byte[]), read(byte[] b,int off,int len) . Same is true with BOS.(BufferedOutputStream)

Objective ---
Create Customer/Account based collection. Sort if reqd.
Store Sorted collection to bin file in buffered manner --
& re-store the same.

Use advanced streams in such cases ---
Mixed Data streams
java.io.DataOutputStream ---implements DataOutput i/f
(converter stream) prim types / string ---> binary
Constructor -- DataOutputStream (OutputStream out)
API ---
public void writeInt(int i) throws IOExc
public void writeChar(char i) throws IOExc
public void writeFloat,writeDouble.....
For Strings
public void writeUTF(String s) throws IOExc ---uses Modified UTF 8 convention
or
public void writeChars(String s) throws IOExc --- uses UTF16 convention

eg : Items Inventory
Item -- code(String-Prim key),desc,category,quantity,price,shipmentDate
constr,toString.

Objective ---
Customer data is already stored in bin file.
Read customer data from Bin file --- in buffered manner & upload the same in HM .display customer details.

Stream class --- java.io.DataInputStream -- implements DataInput
Conversion stream(converts from bin ---> prim type or String)

Constructor
DataInputStream(InputStream in)
API Methods
public int readInt() throws IOException

public double readDouble() throws IOException
public char readChar() throws IOException
public String readUTF() throws IOException(must be used with writeUTF)
public String readChars() throws IOException(must be used with writeChars)

Most Advanced streams ---
Binary streams which can read/write data from/to binary stream in units of Object/Collection of Object refs (i.e Data Transfer Unit = Object/Collection of Objects)

Stream Class for writing Objects to bin. stream
java.io.ObjectOutputStream implements DataOutput,ObjectOutput

Description --- ObjectOutputStream class performs serialization.
serialization= extracting state of object & converting it in binary form.

state of object = non-static & non-transient data members
Constructor
ObjectOutputStream(OutputStream out)
out--- dest Binary o/p stream --- where serialized data stream has to be sent.

API methods ---
public void writeInt(int i) throws IOExc
public void writeChar(char i) throws IOExc
public void writeFloat,writeDouble.....
For Strings
public void writeUTF(String s) throws IOExc ---uses Modified UTF 8 convention

public void writeObject(Object o) throws IOException,NotSerializableException

De-serialization---- conversion or re-construction of Java objs from bin stream.
java.io.ObjectInputStream --- performs de-serialization.--- implements DataInput,ObjectInput
Constructor --- ObjectInputStream(InputStream in)
API methods ---
readInt,readShort,readUTF,readChars..... +
public Object readObject() throws IOException

Objective --- attach OIS to the bin file using FIS & display customer data.

Objective :
Confirming concepts of serialization & de-serialization
Emp -- int id, String name,double salary,Address adr;
Address -- String state,city,street.

Objective -- Understanding Set & its implementation classes
HashSet -- based upon hashing algorithm

More involved scenario.
(store customer info & Items to be purchased)
Data members - int no,Customer info, AL<Item>, Date creationDate

Method References in Java 8

Method reference is a shorthand notation of a lambda expression to call a method.
Can all lambda expressions be concised into method reference ? NO

eg :
If your lambda expression is like this:

s -> System.out.println(s)
then you can replace it with a method reference like this:
(since we are directly calling an existing method in a lambda expression , we can refer to the method itself)

System.out::println
The :: operator is used in method reference to separate the class or object from the method name

Four types of method references
1. Method reference to an instance method of an object – object::instanceMethod
2. Method reference to a static method of a class – Class::staticMethod
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod
4. Method reference to a constructor – Class::new

1. Method reference to an instance method of an object

```
@FunctionalInterface
interface MyInterface{
   void display();
}
public class Example {
   public void myMethod(){
         System.out.println("Instance Method");
   }
   public static void main(String[] args) {
```

```
        Example obj = new Example();
        // Method reference using the object of the class
        MyInterface ref = obj::myMethod;
        // Calling the method of functional interface
        ref.display();
    }
}
```

2. Method reference to a static method of a class

```
import java.util.function.BiFunction;
class Multiplication{
  public static int multiply(int a, int b){
        return a*b;
  }
}
public class Example {
  public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;
        int pr = product.apply(11, 5);
        System.out.println("Product of given number is: "+pr);
  }
}
```

3. Method reference to an instance method of an arbitrary object of a particular type

```
import java.util.Arrays;
public class Example {

  public static void main(String[] args) {
        String[] stringArray = { "aa", "bb", "cc", "dd", "ee"};
        /* Method reference to an instance method of an arbitrary
         * object of a particular type
         */
//Arrays.sort(stringArray, (s1,s2)->s1.compareTo(s2));
        Arrays.sort(stringArray, String::compareTo);
        Arrays.stream(stringArray).forEach(System.out::println);
  }
}
```

4. Method reference to a constructor

```
@FunctionalInterface
interface MyInterface{
   Hello display(String say);
}
class Hello{
   public Hello(String say){
      System.out.print(say);
   }
```

```
}
public class Example {
   public static void main(String[] args) {
          //Method reference to a constructor
      MyInterface ref = Hello::new;
      ref.display("Hello World!");
   }
}
```

4.1

```
@FunctionalInterface
interface MyFunctionalInterface {

   Student getStudent();
}
class Student {

   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Following example uses constructor reference.

```
public class ConstructorReferenceDemo {

   public static void main(String[] args) {

      MyFunctionalInterface ref = Student::new;

      Supplier<Student> s1 = Student::new;// Supplier Example
      Supplier<Student> s2 = () -> new Student();// equals to above line

      System.out.println(ref.getStudent());//Student class toString() call
      System.out.println(s1.get());//Student class toString() call
   }
}
```

What is the need of ObjectInputStream & ObjectOutputStream ?

To achieve Persistence.

Persistence=Saving the state of the java object in permanent manner.

In the absence of Object streams, if you want to persist(save in permanent manner) state of objects or application data in binary manner --- prog has to convert all data to binary & then only it can be written to streams.

Object streams supply ready made functionality for the same.

Stream Class for writing Objects to bin. stream
java.io.ObjectOutputStream implements DataOutput,ObjectOutput

Description --- ObjectOutputStream class performs serialization.
serialization= extracting state of object & converting it in binary form.
(Details --Serialization literally refers to arranging something in a sequence. It is a process in Java where the state of an object is transformed into a stream of bits. The transformation maintains a sequence in accordance to the metadata supplied)

state of object = non-static & non-transient data members
Constructor
ObjectOutputStream(OutputStream out)
out--- dest Binary o/p stream --- where serialized data stream has to be sent.
API methods ---
public void writeInt(int i) throws IOExc
public void writeChar(char i) throws IOExc
public void writeFloat,writeDouble.....
For Strings
public void writeUTF(String s) throws IOExc ---uses Modified UTF 8 convention
+
public void writeObject(Object o) throws IOException,NotSerializableException

De-serialization---- conversion or re-construction of Java objs from bin stream.
java.io.ObjectInputStream --- performs de-serialization.--- implements DataInput,ObjectInput
Constructor --- ObjectInputStream(InputStream in)
API methods ---
readInt,readShort,readUTF,readChars..... +
public Object readObject() throws IOException,ClassNotFoundException,InvalidClassException

Serialization/De-serialization

Ability to write or read a Java object to/from a binary stream

Supported since JDK 1.1

Saving an object to persistent storage(current example -- bin file later can be replaced by DB or sockets) is called persistence

Java provides a java.io.Serializable interface for checking serializability of java classes.(object)

Meaning --- At the time of serialization(writeObject) or de-serialization(readObject) --- JVM checks if the concerned object is Serializable(i.e has it implemented Serializable i/f) --if yes then only proceeds , otherwise throws Exception ---java.io.NotSerializableException

Serializable i/f has no methods / data members and is a marker(tag) interface. Its role is to provide a run time marker for serialization.

Details
What actually gets serialized?
When an object is serialized, mainly state of the object(=non-static & non-transient data members) are preserved.

If a data member is an object(ref) , data members of the object are also serialized if that objects class is serializable

eg : If Product class HAS - A reference of ShippingAddress

The tree of objects data, including these sub-objects constitutes an object graph

eg : HM<String,Product> hm .....
out.writeObject(hm);
HM -- String --Product (id,name,price,qty,category +shippingDetails)

If a serializable object contains reference to non-serializable element, the entire serialization fails

If the object graph contains a non-serializable object reference, the object can still be serialized if the non-serializable reference is marked transient

Details --- transient is a keyword in java.
Can be applied to data member.(primitive as well as ref types)
transient implies ---skip from serialization.(meant for JVM)
During de-serialization ---transient(or even static) members are initialized to def values.
Usage -- To persist --partial state of the serializable object
---

If super-class is serializable, then sub-class is automatically serializable.

If super-class is NOT serializable --- super class must have a default constructor (otherwise InvalidClassException is thrown by JVM during de serilaization)
sub-class developer has to explicitely write the state of super-class.

What happens during deserialization?(in.readObject())

When an object is deserialized, the JVM tries to bring the object back to life by making a new object on the heap that has the same state the serialized object had at the time it was serialized.

1. (Class.forName("com.app.core.Account")--class loading purpose,
1.5 Matches incoming Serial version UID with the computed one
If matches --continues to steps 2.
Otherwise --- InvalidClassException is thrown.
2. If JVM comes across any non serializable super class , having no def constr --- InvalidClassExc is thrown.
Otherwise continues
Class.newInstance() or similar reflection API -- EMPTY/BLANK object is created on heap.
3. setting state of the object from bin stream)
The static/transient variables, which come back have either null (for object references) or as default primitive values.

4. Constructor of serializable class does not get called during de-serialization.
why ?

Think -- what is the need of constructor?
Constructor initializes the object variables with either default values or values which is assigned inside constructor. BUT we want to initialize the state of the object from binary stream.

What are pre-requisites for de-serialization?
Byte codes (.class file) for entire object graph to be de-serialized + Bin data stream containing state.

Details

Java Deserializing process says, "For serializable objects, the no-arg constructor for the first non-serializable supertype is run."

It means during deserialization process, JVM checks the inheritance class hierarchy of instance in process.

It checks, if the Class instance in process has implemented Serializable interface, If yes, then JVM will check Parent class(If any) of the instance to see if Parent has implemented Serializable interface, If yes, then JVM continues to check all its Parents upwards till it encounters a class which doesn't implement Serializable interface. If all inheritance hierarchy of instance has implemented Serializable interface as one shown above then JVM will end up at default extended Object class which doesn't implemented Serializable interface. So it will invoke a default constructor of Object class.

If in between searching the super classes, any class is found non-serializable then its default constructor will be used . If any super class of instance to be de-serialized is non-serializable and also does not have a default constructor then the java.io.InvalidClassException is thrown by JVM.

So till now we got the instance located in memory using one of superclasss default constructor. Note that after this no constructor will be called for any class. After executing super class constructor, JVM

read the byte stream and use instances meta data to set type information and other meta information of instance.

After the blank instance is created, JVM first set its static fields and then invokes the default readObject() method (if its not overridden, otherwise overridden method will be called) internally which is responsible for setting the values from byte stream to blank instance. After the readObject() method is completed, the deserialization process is done and you are ready to work with new deserialized instance.

What is serialversion UID?

The serialVersionUID is a universal version identifier for a Serializable class. Deserialization uses this number to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, then an InvalidClassException is thrown.

How to generate ?
serialver F.Q class name(for a class that imple. Serializable)
eg : serialver java.util.HashMap

Details
Each time an object is serialized, the object (including every object in its graph) is 'stamped' with a version ID number for the object's class. The ID is called the serialVersionUlD, and it's computed based on information about the class structure(class constructors,implemented interfaces,data members).

As an object is being deserialized, if the class has changed since the object was serialized, the class could have a different serialVersionUID, and deserialization will fail.(java.lang.InvalidClassException).

Since different java compilers or even different versions of java compilers can produce different serial version UID , its always recommended , that a programmer should add UID even in the 1st version of class & modify it whenever the class is modified substantially.

examples of incompatible changes
-- Deleting fields , Moving classes up or down the hierarchy ,changing a non-static field to static or a non-transient field to transient ,changing the declared type of a primitive field ....

examples of compatible changes
adding fields,adding classes,adding Serializable,modifying access specifer of the field....

Serialization format overview
Contents of serialized binary stream
It has all the information about the instance which was serialized by serialization process.
This information includes
class’s meta data
type information of instance fields
values of instance fields as well.

This same information is needed when object is re-constructed back to a new object instance. While deserializing an object, the JVM reads its class metadata from the stream of bytes which specifies whether the class of an object implements either ‘Serializable’ or ‘Externalizable’ interface.
Detailed format
1. Magic no.
2. Serialization format version no.
3. Class desc -- class name,serial version uid,desc of data members(
4. State of the object.(non static & non transient data members)

Limitations
1. Java technology only
2. Difficult to maintain in case of changing class format
3. May lead to security leaks.

Important facts of serialization n deserialization
1.
Transient and static fields are ignored in serialization. After de-serialization transient fields and non-final static fields will be inited to default values. Final static fields still have values since they are part of the class data.
2.
ObjectOutputStream.writeObject(obj) and ObjectInputStream.readObject() are used in serialization and de-serialization.
3.
During serialization, you need to handle IOException; during de-serialization, you need to handle IOException and ClassNotFoundException. So the de-serializaed class type must be in the classpath.
4.
Uninitialized non-serializable, non-transient instance fields are tolerated. When adding "private Address adr; no error during serialization.

But , private Address adr = new Address(); will cause exception:
Exception in thread "main" java.io.NotSerializableException: com.app.core.Address

5. Serialization and de-serialization can be used for copying and cloning objects. It is slower than regular clone, but can produce a deep copy very easily.

6. If you need to serialize a Serializable class Employee, but one of its super classes is not Serializable, can Employee class still be serialized and de-serialized?

The answer is yes, provided that the non-serializable super-class has a no-arg constructor, which is invoked at de-serialization to initialize that super-class.

What will be the state of data members?
Sub class (serializable) data members will have the restored state & super class(non serializable) data members will have def inited state

7. You must be careful while modifying a class implementing java.io.Serializable. If class does not contain a serialVersionUID field, its serialVersionUID will be automatically generated by the compiler(using serialver tool). Different compilers, or different versions of the same compiler, will generate potentially different values.

Computation of serialVersionUID is based on not only fields, but also on other aspect of the class like implements clause, constructors, etc. So the best practice is to explicitly declare a serialVersionUID field to maintain backward compatibility. If you need to modify the serializable class substantially and expect it to be incompatible with previous versions, then you need to increment serialVersionUID to avoid mixing different versions.

8. Important differences between Serializable and Externalizable

8.1
If you implement Serializable interface , automatically state of the object gets seriliazed. BUT if u implement Externalizable i/f -- you have to explicitly mention which fields you want to serialize.

8.2
Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().

8.3
Default Serialization process will take place for classes implementing Serializable interface.
Programmer defined Serialization process for classes implementing Externalizable interface.

8.4
Serializable i/f uses java reflection to re construct object during de-serialization and does not require no-arg constructor. But Externalizable requires public no-arg constructor.
Singleton
In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time, in a particular JVM.
It is a creational design pattern which talks about the creation of an object.

After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

eg : Device drivers, Cache , DB connection

How To design a singleton class?

Private constructor to restrict instantiation of the class from other classes.
Private static variable of the same class that is the only instance of the class.
Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

Lazy initialization

Mark constructor as private.
Write a static method that has return type object of this singleton class.

Eager initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

Mark constructor as private.
Create static init block to instantiate a singleton

Factory design pattern

It is a creational design pattern which talks about the creation of an object. The factory design pattern says that define an interface (A java interface or an abstract class) and let the subclasses decide which object to instantiate. The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses. It is one of the best ways to create an object where object creation logic is hidden to the client.

Implementation:
1. Define a factory method inside an interface.
2. Let the subclass implements the above factory method and decide which object to create.
In Java constructors are not polymorphic, but by allowing subclass to create an object, we are adding polymorphic behavior to the instantiation. i.e we are trying to achieve Pseudo polymorphism by letting the subclass to decide what to create, and so this Factory method is also called as Virtual constructor.

eg : Shape Scenario

Method overriding n exception handling
Overriding form of the method(in sub class) can't add any NEW or BROADER checked excpetions
Confirm with examples.
eg :

```
1.  package p1;
class A
{
  void show()
  {
    sop("1");
  }
}

class B extends A
{
  @Override
  void show() throws InterruptedException //javac error : can't add any NEW checked excpetions
```

```
 {
   sop("2");
 }
}

In Tester
A ref=new B();
ref.show();


2.  package p1;
class A
{
 void show() throws IOException
 {
   sop("1");
 }
}

class B extends A
{
 @Override
 void show() //no javac error
 {
   sop("2");
 }
}

3.  package p1;
class A
{
 void show() throws IOException
 {
   sop("1");
 }
}

class B extends A
{
 @Override
 void show() throws FileNotFoundException// no javac error : FileNotFoundException IS A
IOException
 {
   sop("2");
 }
}
4.  package p1;
class A
```

```
{
  void show() throws IOException
  {
    sop("1");
  }
}

class B extends A
{
  @Override
  void show() throws Exception //javac err : Exception is super cls : can't add any BROADER chked excs.
  {
    sop("2");
  }
}
```

Race condition

The situation where two or more threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions.

Critical Section
A code section that leads to race conditions is called a critical section.
eg : Joint Bank Account : shared resource
updateBalance n checkBalance

Preventing Race Conditions
To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Synchronization = Applying thread safety or applying locks

When is synchronization required ?
It's required iff multiple threads are sharing SAME common resource(eg : any collection,data file , db table ,socket , reservation...) & one thread is accessing & other one is modifying the resource.

How to apply synchronization in java ?
keyword -- synchronized.
Can appear as method modifier or at block level.

To avoid race condition / guard critical section , you apply synchronization.

Using synchronized keyword , a lock is applied at the object level.(i.e instance of the shared resource eg : JointAccount)

Important statements

1. lock/monitor can be associated with any java object.
2. When does thrd need to acquire the lock (=enter the monitor)?-- if its invoking either synchronized methods or code from synchronized blocks
3. Can single thrd acquire multiple locks -- YES
4. Blocking trigger
unable to acquire lock(enter monitor) : Blocked on monitor/lock
Un blocked -- lock released / monitor free.(synchronized method rets or synchronized block over)
5. If a thread invokes sleep(or invokes join,yield,notify) or encounters context switching , it holds any locks it has—it doesn't release them.

What's the need of synchronized blocks?

1. Instead of writing long synchronized methods (n thus reducing the performance due to larger extent of the lock) , identify critical section & guard it using synchronized block.

2. While using inherently thread un safe API(StringBuilder, ArrayList,LinkedList,HS,LHS,HM...) in multi thrded environment : you can still apply thread safety : using synchronized blocks.

synchronized block syntax

synchronized(shared resource ref.)
{
Access the methods of shared resource in mutually exclusive manner.
}

1. Only methods (or blocks) can be synchronized, not variables or classes.
2. Each object has just one lock.
3. Not all methods in a class need to be synchronized. A class can have both synchronized and non-synchronized methods.
4. If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method,only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In otherwords, once a thread acquires the lock on an object, no other thread can enter ANY of the synchronized methods in that class (for that object).

5. If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods. If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
6. If a thread goes to sleep(or invokes join,yield,notify) or encounters context switching , it holds any locks it has—it doesn't release them.
7. A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires

a lock and then attempts to call a synchronized method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other synchronized methods on the same object, using the lock the thread already has.

eg :

```
class A {
private B b1;
 synchrnized void test()
 {
   ...
  b1.testMe();
 }
}
class B
{

 synchronized void testMe()
 {
  //some B.L
 }
}
```

Similar can be achieved using nested synchronized blocks.

8. You can synchronize a block of code rather than a method.

When to use synched blocks?

Because synchronization does hurt concurrency, you don't want to synchronize any more code than is necessary to protect your data. So if the scope of a method is more than needed, you can reduce the scope of the synchronized part to something less than a full method—to just a block. OR when u are using Thread un-safe(un-sunchronized eg -- StringBuilder or HashMap or HashSet) classes in your appln.

Regarding static & non -static synchronized

1. Threads calling non-static synchronized methods in the same class will only block each other if they're invoked using the same instance. That's because they each lock on "this" instance, and if they're called using two different instances, they get two locks, which do not interfere with each other.
2. Threads calling static synchronized methods in the same class will always block each other—they all lock on the same Class instance.

3. A static synchronized method and a non-static synchronized method will not block each other, ever. The static method locks on a Class instance(java.lang.Class<?>) while the non-static method locks on the "this" instance—these actions do not interfere with each other at all.

A race condition is a special condition that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition.

Race condition means that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a critical section Java code example that may fail if executed by multiple threads simultaneously:

```
public class Counter {

   protected long count = 0;

   public void add(long value){
      this.count = this.count + value;
   }
}
```

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system(scheduler) switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the Java virtual machine. Rather it is executed as a set of smaller instructions, similar to this:

Read this.count from memory into PC register.
Add value to PC register.
Write register to memory.
Observe what happens with the following mixed execution of threads A and B:

```
    this.count = 0;
```

A: Reads this.count into a register (0)
B: Reads this.count into a register (0)
B: Adds value 2 to register
B: Writes register value (2) back to memory. this.count now equals 2
A: Adds value 3 to register
A: Writes register value (3) back to memory. this.count now equals 3

The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

In the execution sequence example listed above, both threads read the value 0 from memory. Then they add their individual values, 2 and 3, to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B.

Race Conditions in Critical Sections
The code in the add() method in the example earlier contains a critical section. When multiple threads execute this critical section, race conditions occur.

More formally, the situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

Preventing Race Conditions
To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.
What happens when you call start on NEW Thread ?

It internally invokes a native method(not written in java) start0()

Its invocation will --

1. cause a new native thread-of-execution to be created (by native OS)
2. cause the run method to be invoked on that thread.
Thread related API

Starting point
1. java.lang.Runnable --functional i/f
SAM (single abstract method) -- public void run()
Prog MUST override run() -- to supply thread exec. logic.

2. java.lang.Thread --class -- imple . Runnable
It has imple. run() -- blank manner.

3. Constrs of Thread class in "extends" scenario
3.1 Thread() -- Creates a new un-named thrd.
3.2 Thread(String name) -- Creates a new named thrd.

4. Constrs of Thread class in "implements" scneario
4.1 Thread(Runnable instance) -- Creates a new un-named thrd.
4.2 Thread(Runnable instance,String name) -- Creates a new named thrd.

Methods of Thread class

1. public void start() -- To cause transition from NEW -- RUNNABLE
throws IllegalThreadStateException -- if thrd is alrdy runnable or dead.
2. public static void yield() -- Requests the underlying native scheduler to release CPU & enters rdy pool.
Use case -- co operative multi tasking(to allow lesser prio thrds to access processor)

3. public void setName(String nm)
4. public String getName()
5. Priority scale -- 1---10
Thread class consts --MIN_PRIO=1 , MAX_PRIO=10 , NORM_PRIO =5
public void setPriority(int prio)
6. public static Thread currentThread() -- rets invoker(current) thrd ref.
7. public String toString() -- Overrides Object class method , to ret
Thread name,priority,name of thrd grp.
8.public static void sleep(long ms) throws InterruptedException
Blocks invoker thread till specified msecs.
9. public void join() throws InterruptedException
Blocking method(API)
--Causes the invoker thread to block till specified thread gets over.
eg : t1 & t2
t1's run()
{
 .....
 t2.join();//who is waiting for whom for which purpose ? : t1 is waiting for t2 : to complete exec
....
}

t2's run()
{
  //some B.L :read data from file
}
join method can be used effectively to avoid orphan threads
main has to wait for child thrds to complete exec
How ?
In main(..)
t1.join();
t2.join();

10 public void join(long ms) throws InterruptedException
eg : In main method
t1.join(10000);//main is waiting for t1 to finish exec: upto max 10 sec
//t1 gets over after 2 secs : main un blocks
//If t1 doesn't get over within 10 secs : main will be blocked for 10 sec n auto un block.

--Causes the invoker thread to block till specified thread gets over OR tmout elapsed
11. public void interrupt() -- interrupts(un blocks) the threads blocked on ---sleep/join/wait

Methods of Object class (Use Case : Inter thread communication)

1. public final void wait() throws InterruptedException,IllegalMonitorStateException
Meaning -- Forces the invoker thread to release processor & monitor & wait outside .

Trigger for InterruptedException
Some other thread sends interrupt signal to the waiting thread.
Trigger for IllegalMonitorStateException
If the invoker thread is not an owner of the monitor
(i.e if its invoking neither a synched method nor a block)

2. public final void notify() throws IllegalMonitorStateException
Meaning -- Un blocks (wakes up) exactly 1 thread , which has invoked wait on the same object's monitor.

May raise IllegalMonitorStateException --if the current thread is not the owner of a lock.

3. public final void notifyAll() throws IllegalMonitorStateException
Un blocks ALL waiting threads , on the same object's monitor.