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 1.40129846432481707e-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

—----------

Refer : <slides>/PPTDay1

WHY Java ?

1. Platform or architecture independent
(Write once run anywhere!)
refer : whiteboard2.png

2. Simple & robust
3. Secure
4. Automatic memory management.
5. Inherent Multi threaded support
6. Object Oriented support -- Encapsulation,Inheritance & polymorphism , abstraction
7. Excellent I/O support

8. Inherent networking support for TCP/IP , UDP/IP programming & for URLs
9. Supports Functional programming
....

Development & Execution of java application.

JDK , JRE & JVM (refer to diagram)

Refer : java-compiler-and-jit-compiler.png
Java Tut sildes

Objective : Create a java appln to display welcome message , with user name (using cmd line arguments) on the console.

Naming conventions in Java
1. class , interfaces , enum names- 1st letter of 1st word must start with upper case & then follow camel case notation.
eg : HelloWorld
2. data members/methods(functions) -- 1st must start with lower case & then follow camel case notation
eg : private int performanceIndex;
method : calculateSalary(....)

3. constants -- all uppercase.
eg : PI

4. Rules on Identifiers

1. Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_), cannot start with a number!
2. Can't use a Java keyword as an identifier.
3. Are Case sensitive

Legal Access specifiers for data members n methods
private : visible within the same class
default(package private) --no access modifier ---visible within same package
protected : accessible withing the same package & accessible to sub classess via inheritance
public : accessible from anywhere.

Legal class level access specifiers -
1. default(scope=current package only)
2. public (scope=accessible form any where)

How to compile ?
Change dir to <day1>/src
javac -d ..\bin HelloWorld.java

Objective --- Write Java application , to say hello to user. User name will be supplied as cmd line argument.
(eg : java SayHello Madhra
O/P Hello , Madhura !)

Objective : accept 2 nums as cmd line args , add them & display the result.

Java API -- java docs
java.lang => pkg name --default
Integer -- class
public static int parseInt(String s) throws NumberFormatException
(In case of un parseable integers.)
eg : int num1=Integer.parseInt(args[0]);

Reference for Java documentation
Java Docs.
Refer to java docs for any explanation of classes/methods..

Basic rules
1. Java compiler doesn't allow accessing of un initialized data members.
2. Files with no public classes(default scoped) can have a name that does not match with any of the classes in the file .
3. A file can have more than one non public class.
4. There can be only one public class per source code file.
5. If there is a public class in a file, the name of the file must match the name
of the public class.
For example, a class declared as public class Example {....}
must be in a source code file named Example.java.
6. Javac doesn't allow accessing of un-inited vars.
eg : int n;
sop(n);

```
Emp e;
sop(e);
 String s="abc";
s++; //javac error -- pointer aritmetic not allowed in java
```

6. Package related rules
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.
7. 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.
8. 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.

Java Data Types
refer to diagram from the slide

1. primitive data types
boolean,char,byte,short,long,float,double

What is unicode ?
refer to readme

2. reference types
class type , array type , interface type,string and enums.
All ref types are inherited with java.lang.object class

Pointers vs java references
 Pointer arithmetic is not allowed in java.
reference --- holds internal representation of address (equivalent to object pointer in c++)

Conversions regarding primitive types

Automatic conversions(widening) ---Automatic promotions
byte--->short--->int---> long--->float--->double
char ---> int

long --->float ---is considered automatic type of conversion(since float data type can hold larger range of values than long data type)

Rules ---
src & destination - must be compatible, typically dest data type must be able to store larger magnitude of values than that of src data type.

1. Any arithmetic operation involving byte,short --- automatically promoted to --int

2. int & long ---> long
3. long & float ---> float
4. byte,short.....& float & double----> double

Narrowing conversion --- forced conversion(type-casting)
eg ---
double ---> int
float --> long
double ---> float

Revise operators & control structures from : PPTDay1

Objective : Create Emp class : fields (instance variables : state) --id(int) , name(String) , salary(double)
tight encapsulation.
Add a paramterized constructor : to init complete state of Emp.
Add method getEmpDetails : to return string form of complete emp details

Steps for attaching scanner , to accept inputs from User.(UI)

What is Scanner ?
A class (java.util.Scanner) that represents text based parser(It has inherent small ~ 1K buffer)
It can parse text data from any source --Console input,Text file , socket, string

Steps

1. import java.util.*; or import java.util.Scanner;
2. create instance of Scanner class
Scanner class constructor
public Scanner (InputStream in)
System.in --- standard input (stdin)

usage -- Scanner sc=new Scanner(System.in);

3. To check data type
public boolean hasNextInt(),
public boolean hasNextByte(),
public boolean hasNextLong()
...
4. To actually read n parse data
public int nextInt() throws InputMismatchException
public double nextDouble() throws InputMismatchException
public String next() throws InputMismatchException

public boolean nextBoolean() throws InputMismatchException
public String nextLine() throws InputMismatchException

5. Before terminating application close scanner.
public void close();

—----------

Long to Float Conversion

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 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.

—----------

Day1

Madhura Mam.

<u>Quick re cap of some of the earlier important topics</u>

Why Java ?
Data Types in Java
Important memory areas in JVM
stack ,heap , method area
Are you absolutely clear with ?
method local variables vs instance variables vs static variables ?
refer to a diagram : "day1-data\day1_help\diagrams\different types of vars.png"
eg : class Emp
{
 id , name , salary , deptId , noOfEmps,address, idGenerator : static
How will you generate emp ids automatically ?
Constructor(ctor) args : name , salary , deptId ,adr
Emp(...)
{
 this.id=++idGenerator;
noOfEmps++;
}

}
Different example :

What will be the memory pic for the following code?
eg : Emp e1=new Emp(101,"Rama Mathur",12345);
Emp e2=e1;
System.out.println(e1);
System.out.println(e2);
e1=null; //Any objects marked for GC ?
System.out.println(e1.getFirstName());
System.out.println(e2.getFirstName());
e2=null; //Any objects marked for GC ?

eg : Consider below code n check if any objects marked for GC ?
Emp e1=new Emp(101,'Rama Mathur',12345);
e1=new Emp(102,"Kirat Potdar",34567);
sop(e1);//it will impliclitly invoke ???
refer : "day1-data\day1_help\diagrams\whiteboard.png"

If Emp class has a private method :
private boolean test()
{...}

What will be legal access specifiers , in overriding this methos in a sub class(Mgr, Worker....)
You CAN NOT override a private method -to achieve run time poly.
BUT you can re declare it!

eg : What will be mem pic ?
double[] data;
data=new double[10];
OR
double[] data=/*new double[]*/ {12.3,1,2,3,4.5};

eg : Given :
Box class : w,d,h
ctor , toString
What will be mem pic ?
Box[] boxes=new Box[2];
How many objects are created in above statement ?

How will you do this ?
Customer wants to place the order for no. of boxes n wants to specify the dimensions @ run time , dynamically ?

sop("Enter no of boxes to make?");

```
Box[] boxes=new Box[sc.nextInt()];//10
for(Box b : boxes) //b=boxes[0],b=boxes[1]....b=boxes[boxes.length-1]
{
 sop("Enter box dims w d h")
 b=new Box(sc.nextDouble(),sc.nextDouble(),sc.nextDouble());//1 2 3 , 2 3 4 3 4 5....
}
//b : loop local var.
for(Box b : boxes)
 sop(b.getWidth());//NPE
```

Soln : replace for-each by for loop

```
for(int i=0;i<boxes.length;i++) //b=boxes[0],b=boxes[1]....b=boxes[boxes.length-1]
{
 sop("Enter box dims w d h")
 boxes[i]=new Box(sc.nextDouble(),sc.nextDouble(),sc.nextDouble());//1 2 3 , 2 3 4 3 4
}
```

Can you use for-each loop to double box widths ? YES

```
for(Box b : boxes)
 b.setWidth(b.getWidth()*2));
```

Summary : You CAN NOT use for-each to modify array elements(since it works on a copy) BUT can be used(in case of array of refs) to modify state of the objects referred.

Customer wants to place order for 3 boxes , with the dimensions hard coded as :
10,20,30
12.5,13.45,20
1,2,3
How will you do this in 1 single java statement ?
Direct init /dynamic init of array : useful testing practice!

```
Box[] boxes=/*new Box[]*/ {new Box(10,20,30),new Box(12.5,13.45,20),new Box(1,2,3)};
```

static keyword usages

Why inheritance ?
Revise important concepts n keywords , types of inheritance

What about multiple inheritance ? : not supprted1
why ? ambiguity issue(diamond problem)

Annotations in Java
It is the info.(metadata/tag) meant for java tools(i.e either javac or JVM)
eg : @Override
Method level annotation
To inform the javac , that following is either overriding form or the implementation form of the method.
So in case of any mistakes , javac will give you error.

Can a class implement multiple interfaces ? : YES

What will happen when a class implements 2 interfaces with dup method declaration ? Any ambiguity issue ?

Constructor invocation in the inheritance hierarchy

Consider Fruits :
Fruit,Mango,Apple,Orange,Alphonso
Can you organize them in inheritance hierarchy ?

Fruit : super class : abstract class

```
public abstract String taste();
/*{
 return "No specific taste";
}*/
```

Mango extends Fruit

```
{

@Override
public String taste()
{
  return "sweet";
}
```

+ additional func

```
public void pulp()
{
  sop("enjoy cool mango pulp!");
}
}
```

```
Alphonso extends Mango
{
@Override
public String taste()
{
  return "sweet n juicy";
}
}


Orange extends Fruit
{
 @Override
public String taste()
{
  return "sour";
}
+ additional func
public void juice()
{
  sop("extracting orange juice");
}
}

Apple extends Fruit
{
   override taste() : to return"sweet n sour"
   //jam
   public void jam()
   {
      sop("creating apple jam...");
   }
}
```

Solve
What will happen ?

1. Fruit f=new Mango(); //no javac err : upcasting
sop(f.taste());// sweet ! : Run tm poly=Dyn method dispatch => JVM decides taste() method to be called on which type of instance @ run time.

f=new Orange(...);//GC : Mango , no javac err : upcasting
sop(f.taste());//sour

f=new Alphonso();//GC : Orange , no javac err : upcasting
f.taste();//sweet n juicy

2. Object o=f;//no javac err : upcasting
sop(o.taste());//javac error!!!!!!!!!!!!!!! : why :
IMPORTANT
Java compiler resolves method binding by the type of reference
JVM resolves method binding by the type of instance/object it's referrring to.

Meaning : IN sop(o.taste());
this stmt : javac check if taste() exists : in java.lang.Object class -- NO
javac err!

BUT o ---> Alphonso instance
soln : downcasting
sop(((Alphonso)o).taste());//no err , o/p : sweet n juicy

3. Solve :

Fruit f=new Orange();
f.taste(); //sour
f.juice();//javac err ! --soln explic

f=new Apple();
f.juice();
(((Orange)f)).juice();//works !

4. Client wants to create a fruit basket , which can contain apples , oranges , alphonsos .
How will you do it ?
array : Fruit[] basket=new Fruit[sc.nextInt()];//1 array : basket of fruits
int counter=0;
Options
1. Add Orange
```
   if(counter < basket.length)
{
    basket[counter++]=new Orange(....); //up casting : indirect ref.
    sop("Orange added to the basket!");
}
 else
```

```
  sop("Basket full!!!!!!!!!!!");
2. Add Alphonso
 if(counter < basket.length)
{
    basket[counter++]=new Alphonso(....); //up casting : indirect ref.
    sop("Alphonso added to the basket!");
}
 else
  sop("Basket full!!!!!!!!!!!");

3. Add Apple
....

4. Display tastes of all the fruits in the basket n if it's Alphonso : pulp ,.../jam /juice
{al1,al2,al3,a1,a2,or1,null...null}
for (Fruit f : basket) //f=basket[0] DIRECT ref
 if(f != null) {
 sop(f.taste());//run tm poly.
if(f instanceof Alphonso)
  ((Alphonso)f).pulp();
else if(.....)
((Apple)f).jam();

}
```

Revise
abstract methods
abstract classes
interfaces n implementation class.

—----------

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 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 Runnable,Comparable{...}
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 init 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 implicitly calls the super class's no-argument constructor

Revise "this" keyword
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.

Java Variables

Method local Variables

scope --local to curnt method.
where -- stack
by def -- un inited

Instance Variables(non - static data members)

scope --as per access specifier.
lifetime -- from obj creation till GC
by def -- inited.
kept on heap

static variables(class vars)

scope -- as per access specifier
lifetime -- class loading --till unloading(@ jvm shut down)
Kept on method area
by def -- inited

def vals --ref --null

javac doesn't allow accessing any un-inited vars.

Javac doesn't support pointer arithmetic
eg : String s="hello";
s++;
Emp e=new Emp(....);
e *= 10;

this -- keyword in java
2 use cases
1. To un hide inst vars from method local vars.
eg : this.height=height;
2. To invoke constr of the SAME class
eg : Box(w,d,h) & Box(double side) {this(side,side,side);}

What will be the memory pic for the following code?

```
eg : Emp e1=new Emp(101,'Rama Mathur',12345);
Emp e2=e1;
System.out.println(e1);//e1.toString()
System.out.println(e2);//e2.toString()
e1=null; //Any objects marked for GC ? NO
//System.out.println(e1.getFirstName());//run time err : NullPointerException
System.out.println(e2.getFirstName());//Rama
e2=null; //Any objects marked for GC ?
```

Student : firstName,lastName,marks,course,gradYear
Faculty : firstName,lastName, expInYears, sme(Subject matter expertise)

Interface in Java

What is interface ?

An interface in java is a blueprint of a class. Typically 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 , with the implementation class

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.

}
```

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, since 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…

—----------

<u>Regarding inheritance</u>

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called 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.

Inheritance is a process from generalization ----> specialization.

It represents : IS A Relationship.

Why -- re usability.

Which is the keyword used for inheritance ? --extends

Summary : Sub class IS-A super class , and something more (additional state + additional methods) and something modified(behavior --- 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.

Types of inheritance
1. Single inheritance ---
eg : class A {...} class B extends A{...}

2. Multi level inhertance
eg : class A{...} class B extends A{...} class C extends B{...}

3. Hierarchical inheritance
When more than one classes inherit a same class then this is called hierarchical inheritance.
eg : class A{..} class B extends A{ ..} class C extends A{...} class D extends A {...}

4. Multiple inhertiance --- NOT supported
class A extends B,C{...} --assuming B & C : classes -- compiler err

Why --For simplicity.

(To avoid : 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. This is known as Diamond Problem.

To avoid this problem , Java does not support multiple inheritance through classes.

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 keywords

this(...) implies invoking constructor from the same class.
super(...) implies invoking constructor from the immeidate super class

1. Only a constr can use this(...) or super(..)
2. It 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.

Enter polymorphism

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

What is method binding ?
Linking a method call to actual method definition

1. static -- compile time polymorphism--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 Test :
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
```

Can you overload static methods ? Yes (eg : Arrays.toString, PrintStream.print/println)

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(since method signature : SAME) --- BUT taken by JRE (JVM)
Achieved via -- method overriding

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

What is it ?
When we declare the same method in child class which is already present in the parent class then this is called method overriding. In this case when we call the method from child class object, the child class version of the method is called.
eg : Fruit class : taste() : "No Specific Taste"
Orange extends Fruit : taste() : "Sour in taste"
Fruit f=new Orange(....);
sop(f.taste());

Important points

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 or static or 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

```
package p1;
class A {
   A getInstance()
        {
              return new A();
```

```
        }
}
package p1;
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.

Annotations

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 --
It is an annotation meant for javac.
It's Method level annotation ,that appears in a sub class
It's Optional BUT recommended.
eg :

```
public class Orange extends Fruit {

@Override
 public void taste(String name) {....} //javac err
}
}
```

Meaning
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(RTTI)

refer : regarding instanceof

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 .

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.

—----------

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.

—----------

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 mem. leaks/holes

JVM creates 2 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 obj whose no. of ref=0)

How to request for GC ?
API of System class
public static void gc()
eg : System.gc();

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.

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;
Box b3=b1;
b1=b2=b3=null; // 1st Box marked for GC
2. re-assigning the reference to another object
eg : Box b1=new Box(10,20,30);
b1=new Box(11,21,31);//1st Box marked for GC

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.

Day 2

Exception Handling

What is exception ?
Run time error , detected by typically Java runtime environment : JRE (Typically by JVM's main thread)

Why exception handling ?

1. To continue with the program execution , even after run time errors occur(eg :invalid inputs,B.L(Business logic) failures,validation failures, file not found, invalid casting....)

2. To separate B.L (try block) from error handling logic(catch)
3. To avoid un necessary checking .

How does JVM perform exception handling ?

Any run time err occurs(eg file not found,accessing element beyond array size,accessing members from null reference, divide by 0 , etc)

1. JRE(JVM 's main thrd) --- creates matching exception class instance(java.io.FileNotFoundException,java.lang.ArrayOutOfBoundsException,NullPointerException,ArithmeticException) n throws it to the code. (i.e raises the exception)

2. JRE checks -- if there exists any exception handling code ? (i.e has programmer provided any exception handler ?)
If No -- JRE provides the default exception handler n aborts java code
It prints details --Fully qualified exception class name,reason behind failure & location details(known as error stack trace)

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

Topics to learn :
Key words in exception handling --- try,catch,finally,throw,throws
Inheritance hierarchy of exception related classes
unchecked vs checked exceptions.
Creating custom exceptions
JDK 1.7 onwards syntax --- try-with-resources(later useful in Java I/O)

Important :
Checked & Unchked exception are detected or occur only in run-time.
JRE/JVM DOES NOT distinguish between them
Compiler(javac) differentiates between them
Javac forces handling of the checked exceptions. upon the prog.(Handling either by supplying matching try-catch block or including it in the throws clause.)

Legal syntax

1. try {...} catch (ArithmeticException e){...}

2. try {...} catch (exc1 e){...} catch (exc2 e) {..} catch (exc3 e) {..} ....

3. try {...} catch (NullPointerException e){} catch (ArithmeticException e) {}catch(Exception e){catch-all}

3.5 try {...} catch (NumberFormatException e){...} catch (NullPointerException | ArithmeticException | ClassCastException 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

DateFormat class API
public Date parse(String source) throws ParseException
Parses text of the given string to produce a date.

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 exception.
Current method is NOT handling it , BUT its caller should handle.
Mandatory to add throws keyword --- only in case of un handled(no try-catch) checked exceptions(i.e classes not extended from RuntimeException).
Use case --used in delegating the exception handling to the caller.

4.5 Throwable class API

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

5. finally --- keyword in exception handling : represents a block

finally block ALWAYS survives(except System.exit(0) i.e terminating JVM)

It will be executed in the presence or absence of exceptions

What is the meaning ?

5.1 try{...} catch (Exception e){....} finally {....} : legal

5.2 try{...} finally {....} catch (Exception e){....} : illegal

5.3 try{....ArithmeticException.....} catch (NullPointerException e){....} finally {....} : legal

5.4 try {...} finally {....} : legal

6. try-with-resources syntax

From Java SE 7 onwards --- Java has introduced java.lang.AutoCloseable -- i/f(interface)

It represents the resources that must be closed -- when they are no longer required.

java.lang.Autocloesable i/f
Method
public void close() throws Exception-- to close the resources.
This will be automatically called by JVM , at the end of try-with-resources block.

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)
{ ......
} //close
catch(Exception e)
{
...
}
```

eg :

```
try(Scanner sc=new Scanner(System.in);
   FileReader fr=new FileReader(fileName))
{
 ..........
} catch -all
```

7. Creating Custom Exception(User defined exception or application exception)
Need :

1.In case of B.L failures (eg : funds transfer : insufficient finds)or Validations(invalid email id) : JVM CAN NOT detect these errors. Programmer will have to create a custom exc class instance n throw it to the code

syntax of throw keyword :
throw Throwable instance;
eg : throw new MinBalanceException("curnt bal will < min bal....");

Steps

1. Create a packaged public class which extends Throwable(not recommended but legal)/Exception(recommended)/Error(not recommended but legal)/RuntimeExc(not recommended but legal)

```
eg : public class MyException extends Exception{
   public MyException(String mesg)
   {
    super(mesg);
   }
}
```

public class MyException2 extends RunTimeException{....} : legal or illegal ? : legal

public class MyException3 extends NullPointerException{....} : legal or illegal ? :
legal
eg : throw new MyException3(....); --legal

2.In the custom exception class , add a parameterized constructor , to invoke the super-class constructor with the error message.

Super class constructor :
Exception (String msg)
OR
public Exception(String message,Throwable cause)

Objective :
Check the speed of vehicle on a freeway or expressway. If the speed is within a range , allow driver to continue
Otherwise , alert the driver , by raising (throwing) custom exception, with suitable message.

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 : Comment on below , legal or illegal ?
throw new NullPointerExc();
throw new InterruptedExc();
throw new Throwable("abc");
throw new BankAccount(...);
throw new AccountOverdrawnException("funds too low...");

---

1. Validate speed of a vehile running on the expressway
min speed 40
max spped 80
In case of speed out of range , throw custom exc to alert the driver.

1. custom exception class
eg :
SpeedOutOfRangeException extends Exception
add parameterized ctor

2. VehicleValidatioRules : class
2.1 Add public static method for speed validation
public static void validateSpeed(int speed)
{
 ......
}

3. TestCustomException
3.1 try(sc)
{
 accept speed --call method
}

Exception Handling

Types of errors

Compilation time error
Run time errors :
-Due to incorrect input of end user or
-Due to incorrect logic of programmer
General errors :

Traditional Error handling
By using if---- else conditional statement we can write error handlers

```
if(condition)
{
}
else
{ if(condition)
{
}
}
```

What is an Exception ?

Once code is compiled and running, it will have to face the real world of erroneous input, inexistent files, hardware failure… Such problems are commonly known as runtime errors, which are likely to cause the program to abort.

It is therefore important to anticipate such problems and handle them correctly, by avoiding loss of data or premature termination, and by notifying the user.

Classes derived from Error are used for serious, system- generated errors, like OutOfMemoryError, that usually cannot be

Object Throwable

Java will only throw objects of a class descended from Throwable

recovered from

Error Exception

...

Classes derived from

RuntimeException ...
Classes derived from

RuntimeException are
used for ordinary system- generated errors, like ArithmeticException

...

Exception are used for
ordinary errors that a program might want to catch and recover from

The Exception Class Hierarchy
Java distinguishes between two types of error conditions :
Errors represents internal errors of the Java run-time system. You won't throw any of those, and you can't do much about them when they happen.

Exceptions represent errors in the Java application program
– written by you.
Because the error is in your program, you are expected to handle the exceptions.
Try to recover, if possible
Minimally, enact a safe and informative shutdown.

The Exception Class Hierarchy

1. RuntimeException : (unchecked exception)
Usually (but not only) errors resulting from programming mistakes,
ArrayIndexOutOfBoundsException, NullPointerException
Do not need to be advertised or caught.
It's your fault : Could have prevented it with an if () clause.

2. Checked exceptions:
File operations
Must be advertised and caught

Some Predefined Exceptions

```
int q = a/b;
```

Try– catch Block
Try block consists of a set of executable statements that can possibly throw exception while executing them.

Try block is followed by one or more catch blocks where exception thrown in the try block is caught.

When nested try blocks are used, inner try block is executed first and any exception thrown in that block is caught in the following catch blocks.

Example

```
public class Test {
public static void main(String[] args) { try {
int i = Integer.parseInt(args[0]); int j = Integer.parseInt(args[1]); System.out.println(i/j);
}
catch (ArithmeticException a) { System.out.println("You're dividing by zero!");
}
}
}
```

Throw From Called Method

The try statement gets a chance to catch exceptions thrown while the try part runs

That includes exceptions thrown by methods called from the try part

Example

```
void f() { try { g();
}
catch (ArithmeticException a) {
…
}
}
```

If g throws an ArithmeticException, that it does not catch, f will get it
In general, the throw and the catch can be separated by any number of method invocations

Multiple catch Parts

To catch more than one kind of exception, a catch part can
specify some general superclass like RuntimeException

But usually, to handle different kinds of exceptions differently, you use multiple catch parts

Available Methods

public void printStackTrace()
java.lang.ArithmeticException: / by zero at DemoClass.calc(DemoClass.java:15)
at DemoClass.almostThere(DemoClass.java:9) at DemoClass.main(DemoClass.java:5)
public String getMessage()
e.g. / by zero
public String toString()
e.g. java.lang.ArithmeticException: / by zero

Example

```
public static void main(String[] args) { try {
int i = Integer.parseInt(args[0]); int j = Integer.parseInt(args[1]); System.out.println(i/j);
}
catch (ArithmeticException a) { System.out.println("You're dividing by zero!");
}
catch (ArrayIndexOutOfBoundsException a) { System.out.println("Requires two parameters.");
}
}
```

Example

```
public static void main(String[] args) { try {
int i = Integer.parseInt(args[0]); int j = Integer.parseInt(args[1]); System.out.println(i/j);
}
catch (ArithmeticException a) { System.out.println("You're dividing by zero!");
}
catch (ArrayIndexOutOfBoundsException a) { System.out.println("Requires two parameters.");
}
catch (RuntimeException a) { System.out.println("Runtime exception.");
}
}
```

Overlapping Catch Parts

If an exception from the try part matches more than one of the catch parts, only the first matching catch part is executed

A common pattern: catch parts for specific cases first, and a more general one at the end

Note that Java does not allow unreachable catch parts, or unreachable code in general

The throw Statement

<throw-statement> ::= throw <expression> ;

Most exceptions are thrown automatically by the language system

Sometimes you want to throw your own
The <expression> is a reference to a throwable object—usually, a new one:

```
throw new NullPointerException();
```

The Throws Clause

A throws clause lists one or more throwable classes separated by commas
This one always throws, but in general, the throws clause means might throw
So any caller of z must catch SomeException, or place it in its own throws clause (propagate exception)

Finally Block

Finally The finally block will execute whether or not an
exception is thrown.

```
class FinallyDemo { static void procA() { try {
System.out.println("inside procA"); throw new RuntimeException("demo");
} finally { System.out.println("procA's finally");
}
}
```

Propagating Exception

```
import java.io.*; class DemoThrow
{                    public static void main(String args[]){ DemoThrow my = new DemoThrow();
}
public DemoThrow() { checkValue();  } public void checkValue (){
try       {
makeCalc(0);
}
catch(ArithmeticException a)
{          System.out.println(a); } catch(IOException io)
```

```
{
System.out.println(io);
}
}
```

Custom Exception

```
class MyException extends Exception
{          MyException(int a)
{ d=a;  }
public String toString(){
return “MyException “+d;       }
}
```

```
class Throwdemo
{ int size,array[]; Throwdemo(int s)
{ size=s; try     {
checkValue();
}
catch(MyException e)
{          System.out.println(e);
}
}
```

```
void checkValue() throws MyException
{
if( v<0)
throw new MyException();
array= new int[3]; for(int i=0;i<3;i++) array[i]=i+1;}
```

```
public static void main(String args[])
{
new Throwdemo(Integer.parseInt(args[0]));
}
}
```

Assignment s

Accept employee details like name,age,email.

If employee age is less than 25, then thow an exception AgeException.

Exception Handling Flow

Inheritance Hierarchy for Exc Handling classes

java.lang.Throwable

java.lang.Error -- fatal or un-recoverable errs.
eg - StackOverflowError, OutOfMemoryError.

java.lang.Exception

java.lang.RunTimeException

NullPointerExc,AE,AOBoundExc,ClassCastExc.....,NumberFormatExc.
Uncheckd Exception category

IOException,ParseException,FileNotFoundExc,ClassNotFounExc,SocketExc,SocketExc,InterruptedExc

Checked Exception category

Throw vs Throws

throws	throw
keyword in java , for exception handling	keyword in java , for exception handling
can appear only in meth declaration.	can only appear as java statement
mandatory for un-handled checked excs	Typically used for raising custom exceptions
Meaning --- Current method IS NOT handling the	Syntax ---
exception BUT its caller should handle.	throw Throwable instance;
throws' keyword allows delegation of exc handling	Legal -- throw new NullPointerException("msg");
to the caller.	throw new Emp(.....); ---Illegal
	Legal -- throw new SocketException("msg",e);
	Legal --throw new InvalidInputException("invalid email");

Day 3

Constant Table
(String Literal Pool)
Heap Memory
Local Variable Table
"someString"
one
two
"someString"

Constant Table
(String Literal Pool)
Heap Memory
Local Variable Table
"someString"
one
two

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 using java docs

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.

Imortant ---

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 , till the msec precision
Constructor
1.Date() --- creates Date class instance representing system date, current date.(till ms precision)
2.Date(long msec) --- creates Date class instance representing date for msec elapsed after epoch(=1st Jan 1970)
eg : Date d1=new Date(100);
Methods --toString,before,after,equals,compareTo

For parsing(string-->Date) & formatting (Date --> String)

1. Create an instance of java.text.SimpleDateFormat (extends DateFormat)
Constr : SimpleDateFormat(String pattern)
pre defined pattern
y --year
MM -- month in digit(1-12)
MMM -- month in abbrevation(Jan,Feb...)
MMMM ---complete month name
d -day

h- Hour
m --minute
s -- second
eg : SimpleDateFormat sdf=new SimpleDateFormat("dd-MM-yyyy , hh:mm:ss");
OR
SimpleDateFormat 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

var-args ... (ellipsis)

variable args syntax : Allows to pass variable no. of arguments to the method

Rules :
It must be last argument in the method argument list
One can use var-args syntax with , primitive type as well as ref types.

Legal usage :

```
void doStuff(int... x) {
 ....
 } // expects from 0 to many ints
```

Usage : ref.doStuff();//no args
int[] ints={1,2,3,4};
ref.doStuff(ints);//array
ref.doStuff(20,34,56);// comma separated list of args

Java API example of var-args
printf method of java.io.PrintStream

System.out.printf("%n");//legal
System.out.printf(1234);//javac error
System.out.printf("Id %d Name %s Salary %8.2f %n",id,name,salary);

More practice :

void doStuff2(char c, int... x) {..... } //=> must supply 1 char n 0 ints or int[] or , sep. ints
eg : ref.doStuff2('a',1,2,3,5);//legal
ref.doStuff2();//illegal : javac err

Can you use var-args syntax , with ref types ?
Yes

```
class Test {
 void doStuff3(Animal... animals) {
   for(Animal a : animals)
     sop(a.getName());
 }
}
```

What will happen ?
In the Tester class's main method :

```
Test ref=new Test();
ref.doStuff3();//legal
Animal animals[]={new Cat(),new Dog(),new Horse()}; //4 objs
ref.doStuff3(animals);//legal
Animal a1=new Horse();
Animal a2=new Cat();
Animal a3=new Dog();
ref.doStuff3(a1,a2,a3);//legal
```

What will happen ?

```
void doStuff4(int x...) {....}  //illegal syntax
void doStuff5(int... x, char... y) {...} //illegal syntax
void doStuff6(String... s, byte b) { } //illegal syntax
```

Java 8 Date n Time related APIs

Java 8 New Features
java.time : new package is introduced

LocalDate : stores only Date (immutable)(yr-mon-day) : inherently thread safe(synchronized).
API

1. public static LocalDate now()
Obtains the current date from the system clock in the default time-zone.
eg : LocalDate today=LocalDate.now();

LocalTime : Time (immutable)(hr:min:sec) : inherently thrd safe.
LocalDateTime : Date n Time : inherently thrd safe.
eg : sop("curnt date "+LocalDate.now());

2. public static LocalDate of(int year,int month,int dayOfMonth)
Obtains an instance of LocalDate from a year, month and day.
eg : LocalDate date=LocalDate.of(sc.nextInt(),sc.nextInt(),sc.nextInt());

3. public static LocalDate parse(CharSequence text) // default date pattern : yyyy-MM-dd
Obtains an instance of LocalDate from a text string such as 2007-02-03.
eg : LocalDate dt=LocalDate.parse(sc.next());//0 based dates
4. Instance Methods :
isBefore,isAfter,isEqual, compareTo
5. How to change default Date pattern(yyyy-MM-dd) ?
Hint : DateTimeFormatter 's ofPattern

6. Computing Age (implying how to compute period between 2 dates)
Lot simpler with Java 8 API
More complex with legacy APIs
Hint : Period : between

7. plus / minus methods

—----------

```
String s="hello";
s.concat("hi");
System.out.println(s);//hello
s+="12345";//s=s+"12345";
System.out.println(s);//hello12345
```

```
String s1="hello";
String s2=new String(s1);
String s3="hello";
String s4="Hello";
System.out.println("s1==s2"+(s1==s2));//f
System.out.println("s1 eq s2"+s1.equals(s2));//t
System.out.println("s1==s3"+(s1==s3));//t
System.out.println("s1 eq s3"+s1.equals(s3));//t
System.out.println("s1==s4"+(s1==s4));//f
System.out.println("s1 eq s4"+s1.equals(s4));//f
System.out.println("s1 ignore case eq
s4"+s1.equalsIgnoreCase(s4));//t
s1=s2=s3=s4=null; //How many objs marked for GC ?1
String s5="hello";
```

—----------

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

```
boxing = conversion from prim-->wrapper (constrs of wrapper classes / valueOf)
eg : Integer ref=new Integer(1234);
Boolean ref2=new Boolean(true);
un boxing =conversion from wrapper ---> prim
int data=ref.intValue();
auto boxing = auto conv from prim-->wrapper , performed auto by javac : if required
auto un boxing =auto conv from wrapper-->prim , performed auto by javac : if required
```

Day 4

What is enum in java ? : Keyword in java

Enumerations (in general) 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 programer's 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 assig----------n 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 implicitly 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)

1. public static Enum[] values() --rets array of enum type of refs.--pointing to singleton objs
2. public static Enum valueOf(String name) throws IllegalArgumentException -- string to enum type converter
values & valueOf methods generated by compiler --so not part of javadocs.
If you 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.

Decompiled code of Enum

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Color.java

package enums;


public final class Color extends Enum
{

    private Color(String s, int i)
    {
```

```
        super(s, i);
    }
    public static Color[] values()
    {
        Color acolor[];
        int i;
        Color acolor1[];
        System.arraycopy(acolor = ENUM$VALUES, 0, acolor1 = new Color[i = acolor.length], 0, i);
        return acolor1;
    }

    public static Color valueOf(String s)
    {
        return (Color)Enum.valueOf(enums/Color, s);
    }

    public static final Color RED;
    public static final Color BLACK;
    public static final Color SILVER;
    public static final Color WHITE;
    public static final Color BLUE;
    private static final Color ENUM$VALUES[];

    static
    {
        RED = new Color("RED", 0);
        BLACK = new Color("BLACK", 1);
        SILVER = new Color("SILVER", 2);
        WHITE = new Color("WHITE", 3);
        BLUE = new Color("BLUE", 4);
        ENUM$VALUES = (new Color[] {
        RED, BLACK, SILVER, WHITE, BLUE
        });
    }
}
```

—----------

Generic syntax ---
Available from Java SE 5 onwards.
Represents Parameterized Types.
Can Create Generic classes, interfaces, methods and constructors.
In Pre-generics(raw) world (till Java SE 4) , similar achieved via Object(java.lang.Object) type of 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;
...
}
T : any ref type.
eg : ArrayList<Emp>

class Pair<T,U> {...}
T ,U ---type --- ref type
eg : HashMap<K,V>
Understand why generics with example.
eg : Create a Holder class , that can hold ANY data type (primitive/ref type)

Non Generic

```java
public class Holder {
//state
    private Object ref;

    public Holder(Object ref) {
        super();
        this.ref = ref;
    }

    public Object getRef() {
        return ref;
    }

}
```

```java
public class Test {
    public static void main(String[] args) {
//create Holder class instance to hold : 1234
        Holder h1=new Holder(1234);//int --> Integer(auto boxing) ---> upcasting
        System.out.println(h1.getClass());//non_generic.Holder
        System.out.println(h1.getRef().getClass());//java.lang.Integer
        int data=(Integer)h1.getRef();//prog : downcasting ,javac : auto un boxin
       //create another Holder class instance to hold : "hello"
        Holder h2=new Holder("hello");//up casting : javac
        System.out.println(h2.getClass());//non_generic.Holder
        System.out.println(h2.getRef().getClass());//java.lang.String
        String s=(String) h2.getRef();
        h1=h2;
        //un comment next line to observe : ClassCastExc
       // data=(Integer)h1.getRef();//run time err : ClassCastExc

    }
}
```

Generics

```java
public class Holder<T> {
//state
    private T ref;

    public Holder(T ref) {
        super();
        this.ref = ref;
    }

    public T getRef() {
        return ref;
    }

}
```

```java
public class Test {
    public static void main(String[] args) {
//create Holder class instance to hold : 1234
        Holder<Integer> h1 = new Holder<Integer>(1234);// int ---> Integer : auto boxing
        int data = h1.getRef();// javac : auto un boxing(Integer --> int)
        // create Holder class instance to hold : "hello"
        Holder<String> h2 = new Holder<String>("hello");// javac : NO CONVERSION!
        String s = h2.getRef();
    //  h1 = h2;// javac error : incompatible : Holder<Integer> is incompatible with
                // Holder<String>!!!!!

    }
}
```

Showroom management system

1. Vehicle
state : chasisNo(string) : Unique ID,
color(enum) , basePrice(double) ,
manufactureDate(LocalDate),company,isAvailable
ctor : 5 args
toString
equals
getters/setters

2.Custom exception class

3. ValidationRules
3.1 Vehicle color must be valid
public static Color parseAndValidateColor(String clr)
{..}

3.2 No dup vehicles should be added in the showroom : Later !
3.3 manufactureDate : must be in current year(eg : 1st Jan 2023 onwards)

```
try(sc) {
  Create Vehicle[] : prompt size
  exit : flag
  counter : boundary chking  --init phase / set up phase of app
   while(!exit)
   {
      show menu
     try-catch all
     switch
     case 1 : add vehicle
     accept i/ps , validate ---errs


    case 2 : display
    case 0 : exit=true
  }

  }
```

Collection Framework

Limitations with Traditional Arrays

Can we change array size dynamically?

Any problems while inserting new element?

While deleting any element from group?

Are the elements of Array automatically in sorted order?

Is searching is easy with array?

Collection

Collection—

A collection is a group of data (object references) handled as a single unit

What do u mean by framework?

Do u remember data structures in c?

Data structure Algorithms

Collection frameworks encapsulates the algorithms

Programmer don't have to worry abut the implementation of the various collections types.

Nevertheless, still collection framework is encourages programmer to extend members of collection to create a specific implementation where one is needed

Collections

Collection framework is in java.util package

How you will implement your ideas

By Extending an abstract class which provides implementation for most of the methods except few methods are left for you as abstract

:

AbstractCollection,

AbstractList

AbstractSet

AbstractMap.

By Using one of the concrete classes supplied by the framework.

The Collection<E> interface

boolean add(E o)

boolean addAll(Collection c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection c)

boolean equals(Object o)

int hashCode()

boolean isEmpty()

Iterator<E> iterator()

boolean remove(Object o)

boolean removeAll(Collection c)

boolean retainAll(Collection c)

int size()

Object[] toArray()

Type[] toArray(Type[] a)

There is no direct

concrete

implementation class

for Collection

interface but exists

for sub interface

This interface supports the most basic and general operations and queries that can be performed on a group of objects.

Iterator

An iterator over a collection.

Iterator takes the place of Enumeration in the Java collections framework.

Iterators allow the caller to remove elements from the underlying collection during the iteration

hasNext():boolean

next(): E

remove():void

List interface

a list is a collection in which duplicate elements are allowed, and where the order is significant

`<a,b,c>`, `<c,a,b>` and `<a,b,b,c>` are different lists

the List interface inherits from Collection, but changes the semantics of some methods, and adds new methods. The list starts its index at 0.

The List interface

Additional methods

- Dealing with position-oriented operations:

```
public void add(int index, E element)

public boolean addAll(int index, Collection c)

public E get(int index)

public int indexOf(Object element)

public int lastIndexOf(Object element)

public Object remove(int index)

public Object set(int index, E element)
```

List implementations

The collections framework contains two concrete implementations to it:

ArrayList and LinkedList.

How to select which class will solve your problem?

ArrayList

ArrayList is recommended when you're adding and removing elements only to the end of the collection satisfies you and when you wish to access elements using their indices.

It is less time-consuming than LinkedList is.

The ArrayList provides a collection backed by an array.

LinkedList

LinkedList is best when add and remove operations happen anywhere, not only at the end.

But LinkedList's added flexibility comes at an added cost -- it results in much slower indexed operations.

Using List

```
import java.util.*;

class ListDemo

{

        public static void main(String args[]) {

           List<Emp> list = new ArrayList<Emp>();

for (int i=0;i<10;i++)

                  list.add(new Emp(i));

          System.out.println("Third Employee from the list: " +

         (list.get(2).getEmpId());

Iterator<Emp> it = list.iterator();

           while(it.hasNext())

              System.out.println("All the Employees: "+

               (it.next().getEmpId());

        }

     }
```

List: Positional access

```
E get(int index);

E set(int index, E element);

void add(int index, E element);

Object remove(int index);
```

boolean addAll(int index, Collection c);

These operations are more efficient with the ArrayList implementation

List: Searching

int indexOf(Object o);

int lastIndexOf(Object o);

Interface List: Iteration

Iterators specific to Lists:

ListIterator<E> listIterator( ) – Places cursor before 1st element

ListIterator<E> listIterator(int index) –Places cursor before specifed index.

starts at the position indicated (0 is first element)

Inherited methods:

boolean hasNext( );

E next( );

void remove( );

Additional methods:

boolean hasPrevious()

E previous()

int previousIndex()

int nextIndex()

List: Iterating backwards

```
boolean hasPrevious( );

E previous( );

int nextIndex( );

int previousIndex( );
```

List: More operations

```
void add(E o);

        Inserts an object at the cursor position

 E set(E o);            // Optional

        Replace the current element; return the old one

E remove(int index); // Optional

        Remove and return the element at that position
```

Queue and Stack

Implement Stack and Queue using Linkedlist

Set

A collection that contains no duplicate elements

Set will not maintain any order for elements

While adding new element it is using Object's equals() and hashCode() method to check , if such element is there or not in set

Set implementations

There are three concrete Set implementations that are part of the Collection Framework:

HashSet, TreeSet, and LinkedHashSet.

How to select a class which will fulfill your requirement?

HashSet and TreeSet, LinkedHashSet

You use HashSet, which maintains its collection in an unordered manner.

If this doesn't suit your needs, you can use TreeSet.

A TreeSet keeps the elements in the collection in sorted order

While HashSet has an undefined order for its elements, LinkedHashSet supports iterating through its elements in the order they were inserted.

Understand that the additional features provided by TreeSet and LinkedHashSet add to the runtime costs.

Map<K,V>

An object that maps keys to values.

A map cannot contain duplicate keys.

each key can map to at most one value.

The Data is stored in pairs of objects: key and value

Every value object has a key object attached to it.

The hash code of key determines where will the pair be stored in the Map.

You can only retrieve a value object through its key.

The Map interface is not extending the Collection interface.

Map consists of Entries or Mappings

Entry or mapping consists of key & value type of references.

Map's Methods

Methods that deal with adding and removing of key-value pairs:

```
public Object put(K key, V value)

public V remove(Object key)

public void putAll(Map<K,V> mapping)

public void clear()
```

Map's Methods

Methods that allow you to query the Map's content:

```
public V get(Object key)

public boolean containsKey(Object key)

public boolean containsValue(Object value)

public int size()

public boolean isEmpty()
```

Map Implementations

Collection framework gives two classes

HashMap and TreeMap

How to select which will fulfill your requirement

HashMap , TreeMap

By default, choose HashMap, it serves the most needs.

TreeMap implementation will maintain the keys of the map in a sorted order.

it's better to simply keep everything in a HashMap while adding, and create a TreeMap at the end:

```
Map <String,String> map = new HashMap<String,String>();

    //   Add and remove elements from unsorted map
    map.put("Foo", "Bar");

    map.put("
    Bar",
    "Foo");
    map.remo
    ve("Foo")
    ;
    map.put("
    Foo",
    "Baz");

        //   Then sort before displaying elements

        //   in sorted order

    map = new TreeMap(map);
```

Historical Implementations

List:

Vector,

Stack

Map:

Hashtable, Properties

What is the Difference?

The main difference between the historical members we discussed and the new ones is that the old ones are synchronized.

-We can ensure about data integrity.

Synchronization is very resource-consuming, so it is preferable not to automatically synchronize every data structure.

Day 5
JAD file for enum

```
package com.app.core;

public final class Color extends Enum
{
```

```
    private Color(String s, int i, int additionalCost)
    {
        super(s, i);
        this.additionalCost = additionalCost;
    }

    public int getAdditionalCost()
    {
        return additionalCost;
    }

    public String toString()
    {
        return (new
StringBuilder(String.valueOf(name().toLowerCase()))).append("@").append(additionalCost).toSt
ring();
    }

    public static Color[] values()
    {
        Color acolor[];
        int i;
        Color acolor1[];
        System.arraycopy(acolor = ENUM$VALUES, 0, acolor1 = new Color[i = acolor.length], 0, i);
        return acolor1;
    }

    public static Color valueOf(String s)
    {
        return (Color)Enum.valueOf(com/app/core/Color, s);
    }

    public static final Color WHITE;
    public static final Color SILVER;
    public static final Color BLACK;
    public static final Color GREY;
    public static final Color RED;
    private int additionalCost;
    private static final Color ENUM$VALUES[];

    static
    {
        WHITE = new Color("WHITE", 0, 1000);
```

```
        SILVER = new Color("SILVER", 1, 2000);
        BLACK = new Color("BLACK", 2, 3000);
        GREY = new Color("GREY", 3, 2500);
        RED = new Color("RED", 4, 3500);
        ENUM$VALUES = (new Color[] {
            WHITE, SILVER, BLACK, GREY, RED
        });
    }
}
```

Collection Readme

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/set/remove

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)
NOTE : Based upon public boolean equals(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/remove 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.

eg : public Emp(int id) { this.id=id;}

2. Using PK , override equals for content equality.

Usage eg : ArrayList<Emp> emps=new AL<>();
emps.add(e1);//id=10
emps.add(e2);//id=20
emps.add(e3);//id=30

int index=emps.indexOf(20);//int ---> Integer --> Object (Integer)
Integer i=new Integer(20); // javac
//internally invokes equals : whose equals --Object | Integer | Emp | NOA
invokes equlas on Integer class
i.equals(e1) ---since it's incomptabile types --rets false
i.equals(e2) ---since it's incomptabile types --rets false
i.equals(e3) ---since it's incomptabile types --rets false
Thus : indexOf rets -1
sop(index);// -1

Solution :
Emp e=new Emp(20);
int index=emps.indexOf(e);
//internally invokes equals : whose equals --Object | Integer | Emp | NOA
invokes equals on Emp class
e.equals(e1) ---it's comptabile types BUT ids are different --rets false
e.equals(e2) --- it's comptabile types --ids are SAME --rets true

Thus : indexOf rets 1
sop(index);// 1

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 of upcasting
? 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.

—----------

Hierarchy of Collection Framework in Java
Iterable
Collection
List
Queue
Set
ArrayList
LinkedList
Vector
Stack
Priority Queue
Dequeue
ArrayDequeue
HashSet
LinkedHashSet
SortedSet
TreeSet
Map
SortedMap
TreeMap
HashTable
LinkedHashMap
HashMap
Interface
Class
Implements
Extends

ArrayList<Integer> list=new AL<>();
list.add(10);
list.add(20);
list.add(30);
list.add(10);
ArrayList<Integer> list2=new AL<>(list);
sop(list);
sop(list2);
for (int i : list)
sop(i);
list
CP
i1
i2
i3
i4
null
ArrayList class
CP 10
CP 20
CP 30
CP 10
Integer class
list2
CP
i11
i12
i13
i14

```java
public static void main(String[] args) {
    // create empty AL to store Integer refs
    ArrayList<Integer> intList = new ArrayList<>();// size=0, capa :10
    System.out.println("Printing empty AL");
    for (int i : intList)
        System.out.println(i);// no iteration : since AL iterates over size
    System.out.println(intList);
    int[] data = { 10, 2, 34, 11, 5, 4, -10, 2, 10, 30, 20, 109, 100 };
    // populate AL with above data
    for (int i : data)// i=data[0].....
        intList.add(i);// append , auto boxing (int ---> Integer)
    System.out.println("populated AL " + intList);
    intList.add(0, 99);
    System.out.println("AL again " + intList);// [99,10,2,34,11,5,4,-10,2,10,30,20,109,100]
    System.out.println(intList.contains(-10));// invokes internally Integer's equals --t
    System.out.println("Removed " + intList.remove(intList.indexOf(-10)));// -10
    //if negative index or out of capacity  index will give index out of bounds
    System.out.println(intList.contains(-10));// f
    System.out.println("AL again " + intList);// [99,10,2,34,11,5,4,2,10,30,20,109,100]
    System.out.println(intList.indexOf(10) + " " + intList.lastIndexOf(107));//first occurence
    //last will be checked in lastindexof
    // double the values of all elems > 50
    for (int i = 0; i < intList.size(); i++)
        if (intList.get(i) > 50)// auto un boxing
            System.out.println("Replaced " + intList.set(i, intList.get(i) * 2));// 99 109 100

    System.out.println("AL again " + intList);// ordered list
    // sort these elems in asc order
    // API : java.util.Collections : public void sort(List<T> list) --uses natural
    // ordering --based java.lang.Comparable<T>
    Collections.sort(intList);
    System.out.println("AL again " + intList);//sorted list
```

Collection Framework Overview

java.lang.Iterable<T> : i/f
Not a part of the collection frmwork.
Method : Iterator<T> iterator();
Implicit Iteration : for-each

java.util.Iterator<T> : generic i/f
Cursor that can traverse ANY collection,in a forward manner
Methods
1. public boolean hasNext()
2. public T next()
3. public void remove()

Collection<E> : sub i/f of Iterable
Does not specify any particular behavior.
No concrete imple. classes
Contains methods for all general purpose operations.
eg : add
,addAll,contains,remove,removeAll,retainAll,size,isEmpty...
.

ListIterator<T> : list specific , sub i/f of Iterator<T> (can be attached to ONLY List imple. classes)
How ?
java.util.List method
1. public ListIterator<T> listIterator()
2. public ListIterator<T> listIterator(int index)

List<E> : sub i/f of Collection
Represents ordered collection.(order is significant)
Supports index based ops
Allows dups
Allows nulls

Set<E> : sub i/f of Collection
Not an ordered collectoin
Does not support index based ops.
Does not allow dups
Allows null

HashSet<E>

LinkedHashSet<E>

SortedSet<E>
sub i/f of Set

TreeSet<E>
: sorted set

Concrete List implementation classes

ArrayList<E>

Inherently
thrd unsafe

AL() : size : 0 ,capa=10
AL(initCapa) : size = 0,capa=initCapa
AL(Collection c) : size=c.size, capa :JVM

LinkedList<E>

Inherently
thrd unsafe

Vector<E>

Inherently thrd safe

```
Internal working
ArrayList<BankAccount> accounts=new AL<>();
accounts.add(new BankAccount(100,.....));//a1
accounts.add(new BankAccount(10,.....));//a2
accounts.add(new BankAccount(50,.....));//a3
sop("Enter a/c no");
BankAccount a=new BankAccount(sc.nextInt());//50
sop(accounts.contains(a));
```

1st
a.equals(a1)
false ——a.equals(a2) ---false

Sorting in JAVA:

Sorting in Java

Natural Ordering(implicit/internal ordering) ---
Sorting criteria will be within UDT or generic type of List.

Steps
1. UDT(generic type of the list) MUST implement
java.lang.Comparable<T>
T --type of the list
eg : public class BankAccount imple. Comparable
<BankAccount>{...}
2. UDT must imple.
public int compareTo(T o)

Must ret
<0 --if this < o sorting criteria-wise
=0 --if this = o sorting criteria-wise
> 0 --if this > o sorting criteria-wise

3. Use Collections class API
public static void sort(List<T> list)
--will auto invoke T's compareTo method , as per internal sorting algo.
Algo --Tim's Sorting algo.

Custom Ordering (explicit/external) --
sorting criteria is kept external to UDT

Steps
1. Create a separate class(or ano inner or lambda expr) which imple. java.util.Comparator<T>
T -- type of object to compare (List's generic type)
eg : public class AccountBalComparator imple Comparator<BankAccount>{...}

2. Must implement
public int compare(T o1,T o2)
Must ret

< 0 --- if o1 < o2 as per sorting criteria
=0 -- if o1=o2 as per sorting criteria
> 0 -- if o1 > o2 as per sorting criteria

3. Use Collections class API
public static void sort(List<T> list,Comparator <T> comp)
This sort(..) will automatically invoke, supplied comparator's compare method.
Implementation note : Uses Tim's sorting algo.

Detailed overview of Collection Framework

java.lang.Iterable<T> : super i/f of Collection i/f.
Important Method
Iterator<T> iterator() : rets the iterator over the underlying collection (can be replaced by implicit for-each loop)
It's NOT a part collection frmwork

java.util.Collection<E> : sub i/f of Iterable
Does not have any concrete implementation classes, since it doesn't impose any specific behavior. Contains general purpose methods : add,addAll,remove,removeAll,retainAll,contains ,size,toArray,isEmpty....

List<E>

Queue<E>

Set<E>

Ordered collection(remembers order of insertion)
Order is significant
Supports index based operations
Allows dups
Allows null refs.(impractical)

List imple. classes

ArrayList<E>

Vector<E>

LinkedList<E>

Data structure : resizable array.
Offers const time performance O(1) : get , set , iterator, listIterator , size
O(n) : add , remove,indexOf,lastIndex

Day6

```java
@Override
public boolean equals(Object o) {
    System.out.println("in vehicle equals");
    if (o instanceof Vehicle)
        return this.chasisNo.equals(((Vehicle) o).chasisNo);
    return false;
}

@Override
public int compareTo(Vehicle anotherVehicle) {
    System.out.println("in vehicle : compare to");
    //sorting criteria : chasis no : String --Comparable<String>
    //i.e has implemented compareTo : <0 , =0 , > 0 : as er uni code values!
    return this.chasisNo.compareTo(anotherVehicle.chasisNo);
```

```java
import java.util.Comparator;

public class VehicleDatePriceComparator implements Comparator<Vehicle>{
    @Override
    public int compare(Vehicle v1,Vehicle v2) {
        System.out.println("in compare -date n price1");
        int retVal=v1.getManufactureDate().compareTo(v2.getManufactureDate());
        if(retVal == 0) {
            //=< dates same --continue to price based comparison
            if(v1.getNetPrice()<v2.getNetPrice())
                return -1;
            if(v1.getNetPrice()==v2.getNetPrice())
                return 0;
            return 1;
        }
        return retVal;
```

```java
public class VehiclePriceComparator implements Comparator<Vehicle> {
    @Override
    public int compare(Vehicle v1, Vehicle v2) {
        System.out.println("in compare : price");
        if (v1.getNetPrice() < v2.getNetPrice())
            return -1;
        if (v1.getNetPrice() == v2.getNetPrice())
            return 0;
        return 1;
```

Day7

Hashing observations (ref : day7.1)
(equal objs ARE producing same hashcode=100) : only MANDATORY part of the contract
obj : detecting dups BUT not ensuring const time perf : O(n)
in emp hashCode
Added true
in emp hashCode
in emp eq
Added true
in emp hashCode
in emp eq
Added false
in emp hashCode
in emp eq
in emp eq
Added true
in emp hashCode
in emp eq

```
in emp eq
in emp eq
Added true
in emp hashCode
in emp eq
in emp eq
in emp eq
in emp eq
Added true
size 5
Emp [id=rnd-001, name=Riya, salary=10000.0]
Emp [id=rnd-003, name=Rakesh, salary=20000.0]
Emp [id=rnd-004, name=Riya, salary=10000.0]
Emp [id=rnd-002, name=Rakesh, salary=20000.0]
Emp [id=rnd-008, name=Riya, salary=10000.0]
-----------------------------------------------------------------------------------------------------------------------
Hashing observations (ref : day7.2)
(equal objs ARE producing same hashcode n un equal objs are producing different hascodes) :
MANDATORY n optional part of the contract is followed
obj : detecting dups and also  ensuring const time perf : O(1)
in emp hashCode
Added true
in emp hashCode
Added true
in emp hashCode
in emp eq
Added false
in emp hashCode
Added true
in emp hashCode
Added true
in emp hashCode
Added true
size 5
Emp [id=rnd-008, name=Riya, salary=10000.0]
Emp [id=rnd-002, name=Rakesh, salary=20000.0]
Emp [id=rnd-001, name=Riya, salary=10000.0]
Emp [id=rnd-004, name=Riya, salary=10000.0]
Emp [id=rnd-003, name=Rakesh, salary=20000.0]
-----------------------------------------------------------------------------------------------------------------------
```

Hashing algo in Java

JVM invokes , hashCode to derive the bucket id
Bucket ID=e1.hashCode() % capa (actual internal formula : and operator)

JVM checks

Empty(null)

elem gets added
add(e) : true

non empty(not null)

JVM invokes equals

false => no dups
elem gets added
add(..) : true

true => dup detected
elem doesn't get added
add(..) : false

hash collision : JVM coming across non empty (equals method invoked) -- chaining(SLL)
JVM performs rehashing , when hashtable can't ensure const time perf.
When
size of D.S > capa * L.F , JVM does re-hashing
Created NEW hash table , with double of the orig. capa, invokes hashing algo , on the earlier nodes.

Contract between overrding of equals & hashCode mandatory for E type of Set & K type of Map

ref1.equals(ref2)

true. ref1.hashCode() MUST be same as ref2.hashCode()

false -- optional BUT recommended. ref1.hashCode() different from ref2.hasCode() Improve performance of Hashinn based data structure.

Contract for overriding equals & hashCode methods--for correct working of HashSet/HashMap/Hashtable (any hashing based data structure)
Equal objects MUST produce SAME hash code.

HOW ?
1. Identify PK(object's identity) data members & override equals.
2. Use the same data members in overriding hashCode

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.

	ArrayList	LinkedList
Structure	ArrayList is an index based data structure where each element is associated with an index.	Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element.
Insertion And Removal	Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted.	Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed.
Retrieval (Searching or getting an element)	Insertion and removal operations in ArrayList are of order O(n). Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based.	Insertion and removal in LinkedList are of order O(1). Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element.
Random Access	Retrieval operation in ArrayList is of order of O(1). ArrayList is of type Random Access. i.e elements can be accessed randomly.	Retrieval operation in LinkedList is of order of O(n). LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element.
Usage	ArrayList can not be used as a Stack or Queue.	LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List.
Memory Occupation	ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it's index.	LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements.
When To Use	If your application does more retrieval than the insertions and deletions, then use ArravList.	If your application does more insertions and deletions than the retrieval, then use LinkedList.

Singly Linked List

Doubly Linked List

Difference between ArrayList and LinkedList in Java

ArrayList and LinkedList both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.

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.

Operations	ArrayList	LinkedList
	Complexity	Complexity
get(int index)	O(1)	O(n/4)
add(E element)	O(1)->O(n)	O(1)
remove(int index)	O(n/2)	O(n/4)
add(int index, E element)	O(n/2)	O(n/4)
Iterator.remove()	O(n/2)	O(1)
ListIterator.add(E element)	O(n/2)	O(1)

---

java.util.LinkedList<E>

Doubly-linked list implementation of the List and Deque interfaces.
It is an ordered collection and supports duplicate elements.
It stores elements in Insertion order.
It supports adding null elements.
It supports index based operations.
Typical use case -- stack or queue.
It does not implement RandomAccess interface.(ArrayList class does!)
So it represents sequential access list.

When we try to access an element from a LinkedList, searching that element starts from the beginning or end of the LinkedList based on whichever is closer to the specified index.(eg : list.get(i))

Structure of LinkedList --refer to diag.

It supports all of List API methods , as seen already in ArrayList.

Java LinkedList methods , inherited from Deque

The following methods are specific to LinkedList class which are inherited from Deque interface:

void addFirst(E e): Inserts the specified element at the beginning of this list.
void addLast(E e): Inserts the specified element at the end of this list.
E getFirst(): Retrieves, but does not remove, the first element of this list. This method differs from peekFirst only in that it throws an exception if this list is empty.
E getLast(): Retrieves, but does not remove, the last element of this list. This method differs from peekLast only in that it throws an exception if this list is empty.
E removeFirst(): Removes and returns the first element from this list.
E removeLast(): Removes and returns the last element from this list.
boolean offerFirst(E e): Inserts the specified element at the front of this list.
boolean offerLast(E e): Inserts the specified element at the end of this list.
E pollFirst(): Retrieves and removes the first element of this list, or returns null if this list is empty.
E pollLast(): Retrieves and removes the last element of this list, or returns null if this list is empty.
E peekFirst(): Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
E peekLast(): Retrieves, but does not remove, the last element of this list, or returns null if this list is empty.

Java LinkedList Usecases

Best Usecase scenario:-

When our frequently used operation is adding or removing elements in the middle of the List, LinkedList is the best class to use.

Why? Because we don't need to do more shifts to add or remove elements at the middle of the list.

Worst Usecase scenario:-

When our frequently used operation is retrieving elements from list, then LinkedList is the worst choice.

Why? Because LinkedList supports only sequential access, does NOT support random access.

NOTE:-
LinkedList implements List, Deque, But it does NOT implement RandomAccess interface.

How to use LinkedList as a queue or stack ?
When a deque is used as a queue its FIFO (First-In-First-Out)Elements are added at the end of the deque and removed from the beginning.
Which method will you use ?????????

When a deque is used as stack its LIFO (Last-In-First-Out) (Preferred to the legacy Stack class.)
When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.
Which method will you use ?????????

Map Handling

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).

More details
Hash collision degrades the performance of HashMap significantly. With this new approach, existing applications can expect performance improvements in case they are using HashMaps having large number of elements by simply upgrading to Java 8.

Hash collisions have negative impact on the lookup time of HashMap. When multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. In case of retrieval, linked list has to be traversed to get the entry. In worst case scenario, when all keys are mapped to the same bucket, the lookup time of HashMap increases from O(1) to O(n).

Java 8 has come with the following improvements/changes of HashMap objects in case of high collisions.

The alternative String hash function added in Java 7 has been removed.
Buckets containing a large number of colliding keys will store their entries in a balanced tree instead of a linked list after certain threshold is reached.
Above changes ensure performance of O(log(n)) in worst case scenarios (hash function is not distributing keys properly) and O(1) with proper hashCode().

How linked list is replaced with binary tree?

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

This JDK 8 change applies only to HashMap, LinkedHashMap and ConcurrentHashMap.

Map Overview (refer to the diagram "regarding Maps")

Map Implementation class
1. HashMap<K,V>
1.1 Constructors
1. HashMap() : size=0, init capa=16, Load Factor =0.75
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();

HashMap Internals
HashMap<I,A> hm=new HM<>();

def init capa=16 & Load Factor =.75

hm.put(k1,v1);

hm.put(k2,v2);

hm.put(k5,v5);

hm.get(k5)

Bucket -- null
Node -- Map.Entry <K,V> + Next Ptr + hash
LinkedList<Node>

JVM invokes k1.hashCode() -- (some internal hash computation) % capacity ---Bucket ID
JVM Checks For Bucket

Empty (null)
Inserts the entry LinkedList --node

put() Rets null

non-empty (non null) hash collision
JVM invokes key's equals(eg : k5.equals (k1))

true --- same Key alrdy exists in the same bucket.
Replaces old value by new val & rets old val ref.
Rets non - null
eg : v1

false -- new entry gets inserted.
k1,v1,hash,next---
k5,v5,hash,null

Rets --NULL

In case of increasing hash collisions --- HM can't ensure const time performance.
If total no of entries(map.size()) > capa * L.F --- Re -hashing takes place. JVM create new HM -- typically twice no of buckets --all entries copied into new HM .

---

put(k,v)

New Entry
--inserts the entry
& rets null

Existing entry(same key
already exists)
--replaces old value by new
value & rets old value.

```
map.put(k1,v1);
map.put(k2,v2);
map.put(k1,v3);
```

```
Set<K> keys=map.keySet();
Collection<V> vals=map.values();
Set<Map.Entry<K,V>>
entries=map.entrySet();
```

putIfAbsent(K,V)

New entry --- inserts the
entry & rets null

Existing entry(same key alrdy
exists) --- doesn't replace old
entry by new.
returns value ref.

Map<K,V>
K --Key Type , V -- Value Type
Map consists of Key & value pairs(refs)=entry(Map.Entry<K,V>)=mapping
Has distinct keys , can contain duplicate values.
Typically in HashMap<K,V> -- key's hashCode , governs where entry has to be inserted in the map.
One key ---> Value

HashMap<K,V>
--un sorted & un ordered Map.
Based upon hashing algo.
Ensures const time perf(O(1)) for -- put,putIfAbsent,get,remove,cotainsKey,size
thread un safe (un synchronized)

LinkedHashMap<K,V>
--un sorted BUT ordered (remembers order of insertion) Map.

SortedMap <K,V> --sub i/f

TreeMap<K,V> -- sorted BUT un ordered Map

1. TreeMap<K,V>() --empty map. Upon population(put/putAll...) --Natural ordering --Key's compareTo invoked.

2. TreeMap<K,V>(Map<? extends K,? extends V) m) --- populated map of type <K,V> from ANY Map(eg :HM,LHM,Ht,TM..) having Key type as K or its sub-type & value type as V or its sub type.

3. TreeMap<K,V>(Comparator<? super K> c) --empty map . Upon population , invokes c's compare.

Limitations
1. Since Map deosn't extend Collection ---can't attach iterator & traverse the map directly.
2. Searching n sorting n removing as per Key based criteria -- directly supported.
For searching/sorting/removing as per value based criteria --involves converting map ---> collection(collection view of a map)

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 ?
Help : API of List /Set i/f : boolean addAll(Collection<? extends E> c)

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);

Generic Sequence
1. Generic Method
What is it ? A Method which has it's own type parameter.
Can it exist in a non generic class? : Yes

Generic method example : Arrays.toList , Collections.sort
java.util.Arrays class
public static <T> List<T> asList(T... a)
Generic method from a non generic class(Arrays)
Type declaration fits between method modifiers n ret type

i/p : T... a : This method can accept : no args or T[] or T t1,T t2....
o/p : FIXED size list
How will you use it for getting a fixed size List<Integer> ?

Another example : Collections class : non generic class
public static <T extends Comparable<? super T>> void sort(List<T> list)
---

2. More about generic syntax : ? , extends ,super
Given :
Emp <---- Mgr <----SalesMgr
Emp <---- Worker <---- TempWorker

What will happen ?

```
Emp e =new Mgr();
e.computeSalary();
e=new TempWorker();
e.computeSalary();
```

```
ArrayList<Emp> l1=new ArrayList<>();//what's the inferred type of the RHS ? :
ArrayList<Mgr> l2=new ArrayList<>();//what's the inferred type of the RHS ? :
l1=l2;
Object o=new Mgr();
ArrayList<Object> l3=new ArrayList<>();//what's the inferred type of the RHS ? :
l3=l2;
l3=l1;
```

What does it imply ?
Does inheritance in generics work in the same manner as learnt earlier ? NO

Wild card in generics comes to the rescue!
Enter "?"
What is it ?
It's a wild card in generic syntax. Can be replaced by ANY type.
Represents any unknown type.

Now what will happen ?

```
ArrayList<?> l1=new ArrayList<>();//what's the implicit type of the RHS ? :
ArrayList<Mgr> l2=new ArrayList<>();
l1=l2;
```

? : is referred to as un bounded wild card.

Hands on
1. Shuffle all the elements of the list randomly.
Collections class Method
public static void shuffle(List<?> list)

2. Collections class
reverse method

3. Write a static method in GenericUtils class : to print elements of ANY List(AL/LL/Vector) of ANY type

1. Add a method printElements to print the elements of ANY List of ANY type.
1.1 Try it with T : parameter Type (lab work!)
1.2 Try it with ?
Any difference ?
Which seems easier to use ?
Test it with tester : AL<Integer> , LinkedList<String>,Vector<Double> ....

2. Add "computeSalary" method in Emp based hierarchy
Modify GenericUtils class .
Add static method which can accept ANY List(AL/LL/Vector) of any type of emps(Emp /Mgr / SalesMgr / Worker/ TempWorker) , computes salary of all of emps supplied via list.

Test it with tester : AL<Mgr> , Vector<HRMgr> , LinkedList<TempWorker>.....
eg :

```
public static void computeSalOfEmpList(List<Emp> emps)
{
  for (Emp e : emps)
  sop(e.computeSalary());
}
```

Tester class

```
AL<Mgr> mgrs=new AL<>();
mgrs.add(m1);mgrs.add(m2);mgrs.add(m3);mgrs.add(m4);
computeSalOfEmpList(mgrs);
```

& compute salary.
Test it from Tester class .
2.1 Try it with T : parameter Type
2.2 Try it with ?
Any difference ?
Which seems easier to use ?

3. Write a method to Find max number from List of any numbers (integer / float / double ...) n return it to the caller.
(Lab exercise)

4. Write a method to Find max element from numbers / strings/ dates

Lab Exercise

5. Write a method that counts the number of elements (stored in Collection) that are greater than a specified element elem.
eg : Count no of strings in a collection of strings > specified string (i.e as per lexicographical ordering)
Count no of doubles in a collection of doubles > specified double value
Count no of dates in a collection of dates > specified date (i.e after a specified date)
...

6. For more details : refer to Collections.copy
For T , ? , extends , super
(Lab exercise)

eg of bounded type in Collection Framework
ArrayList<E> : generic class
public ArrayList(Collection<? extends E> c)
E : Emp
? extends Emp
Meaning : This ctor will create a non-empty AL of Emp type , from AL/LL/Vector/HS/LHS/TS (any Collection imple class) , from any type : Emp or it's sub type
Q : Will you be able populate AL<Mgr> from
1. AL<Emp>
2. HS<Worker>
3. LL<SalesMgr>
4. Vector<Fruit>
5. TS<Object>
6. LHS<HRMgr>
Ans :

Generic syntax :
? : wild card in generic syntax (it can be replaced by ANY type) : un bounded wild card
extends : Represents upper bound
super : Represents lower bound
? extends E : ANY type E or its sub type
? super E : E or its super type
eg : ? extends Emp => Emp or it's sub type(Mgr,Worker.....)
? super Mgr => Mgr or it's super type (Emp , Object)

What will happen ? (javac error or no error?)
1. Vector<Mgr> mgrs=new Vector<>();
mgs.add(m1)....m10
ArrayList<Emp> emps=new AL<>(mgrs);

2. HashSet<HRMgr> hrMgrs=new HS<>();
hrMgrs.add(hrm1)....hrm10
ArrayList<Emp> emps=new AL<>(hrMgrs);
3.
ArrayList<Mgr> mgrs=new AL<>();
mgrs.add(m1).....m10
LinkedList<HRMgr> hrMgrs=new LinkedList<>(mgrs);

Ans this !
ArrayList<Mgr> mgrs=new AL<>();
mgrs.add(m1).....m10
ArrayList<? extends Emp> emp=new AL<>(mgrs);
ArrayList<? extends Object> objs=mgrs;

3.Collections class(Non generic class) : Can contain a generic Method
eg : super keyword in generics : lower bound
Method of Collections class
public static <T> void sort(List<T> list,Comparator<? super T> c)
generic method :
where does type declaration fit ? : It's placed between method modifiers n ret type
1st arg : List<T> list : You can pass List of ANY type (eg : AL/LL/Vector : ??
HS/LHS/TS : ??
HM/LHM/TM : ??) :
Can you pass List of the following types :
T --Customer , Student(extends Person) , BankAccount,Flight, Movie,Person , DacStudent (extends Student)
if above classes have not imple Comparable or Comparator :

2nd arg : Comparator<? super T> c
Instance of the class which imple Comparator
eg : Collections.sort(studentList,comp);//comp : imple Comparator<Student>
 Collections.sort(studentList,comp);//comp : imple Comaprator<Fruit>
Collections.sort(studentList,comp);//comp : imple Comaprator<Person>
Collections.sort(studentList,comp);//comp : imple Comaprator<Object>
Collections.sort(studentList,comp);//comp : imple Comaprator<eDacStudent>

----------Read Later----------
More Details

? : wild card (represents ANY unknown type) => un bounded wildcard

extends => upper bound (type of the upper bound super class / interface)
If your collection(list/set/map) is acting as a producer (of data) i.e while using retrieve operation, use upper bound
eg : ? extends Number => Number or it's subtype (Byte,Short.....Double)
get : type of the data that you get Number or it's subtype, BUT you can't add Integer , if it's collection of Double
Can be accessed using Number type of the ref.
(Producer extends)

super : lower bound
? super T => T or it's super type
Use it whenever your collection is acting as a consumer (data sink) : i.e wheneve you want to add data , to a collection.
(Consumer super)

1. Solve generic questions

2. Solve (Lab work)
Collections class API
public static <T> void copy(List<? super T> dest, List<? extends T> src)

If you consider Emp<---Mgr <---- SalesMgr

Which of the following is valid ?
1. You should be able to copy SalesMgr type of refs into Emp type of the list
2. You should be able to copy SalesMgr type of refs into Mgr type of the list
3. You should be able to copy Emp type of refs into Mgr type of the list
4. You should be able to copy Mgr type of refs into SalesMgr type of the list

It means
If src type of the list for copying is SalesMgr (i.e you have a list of SalesMgrs)
dest type of the list for copy should be ????

If src type of the list for copying is Mgr
dest type of the list for copy should be ????

If src type of the list for copying is Emp

dest type of the list for copy should be ????

If the dest type of the list Mgr
src type of the list should be ???

What is the difference between a wildcard bound and a type parameter bound?

A wildcard can have only one bound, while a type parameter can have several bounds.

A wildcard can have a lower or an upper bound, while there is no such thing as a lower bound for a type parameter.

Wildcard bounds and type parameter bounds are often confused, because they are both called bounds and have in part similar syntax.

Example (of type parameter bound and wildcard bound):

```
class Box< T extends Appendable & Flushable > {
  private T theObject;
  public Box(T arg)              { theObject = arg; }
  public Box(Box< ? extends T > box) { theObject = box.theObject; }
  ...
}
```

Java8 new features

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.
eg Runnable,Comparable,Comparator,Iterable,Consumer,Predicate...
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 a valid functional interface ?
public interface A { double calc(int a,int b);} : YES (SAM : calc)

public interface B extends A {} : YES (SAM : calc)

public interface C extends A { void show();} : NO , 2 abstract : calc,show

public interface D {} : NO : Marker (tag) : empty

```
public interface E extends A {default void show(){}
static void test() {...}
}  : YES  (SAM : calc)
```

4. Lambda Expressions

Objective -- Perform add/subtract/multiply/divide ... operations on 2 double i/ps & return result.

How ?
1. Create a func i/f : to wrap : SAM that represents ANY math op.

refer to readme of lambda expressions.

Day-09

Regarding functional programming
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)

Background

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 or returning the behaviour directly 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 ?

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 : Objective
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);

OR

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().getAsDoubl
e()

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

How do we declare just behaviour in java ?

Refer to Summable interface , implementation & its conversion to lambda.

Using interfaces

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 ?

Applying lambdas to refer to func i/f.
Objectives
1. Collections.sort
2. List -- forEach , removeIf

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.

Sequence Day-09:
Revise generic syntax
1. Generic methods
example

2.
Create GenericUtils class : non generic--static methods
1. Add a method to print any element from any of the following
 Test cases : AL<String> , LL<Integer> , Vector<LocalDate> , HS<Double> , LHS,TS

```
eg : public static void printAllElems(Collection<Object> objs)
{
 for(Object o : objs)
  sop(o);
}
```

Tester
AL<String> strings=new AL<>(Arrays.asList(.....));
strings.add("abcd");
printAllElems(strings);//javac err!

eg : public static void printAllElems(Collection<?> objs)
{
 for(Object o : objs)
  sop(o);
}

Tester
AL<String> strings=new AL<>(Arrays.asList(.....));
strings.add("abcd");
printAllElems(strings);

1.1 Using T : generic method
1.2 Using ? : un bounded wild card

3. Add a method to find sum of salary of emps from any List
Test cases : AL<Mgr> ,LL<Worker> , Vector<SM>

eg : public static double computeSum(List<? extends Emp> empList)
{
 double sum=0;
 for (Emp e : empList)
  sum += e.computeSal();
  return sum;
}

? extends Emp => Emp or any of it's sub types

Tester class
AL<Mgr> mgrs=new AL<>();
mgrs.add(m1);.....
double sum=computeSum(mgrs);
Vector<TempWorker> workers=new Vector<>();
add few workers.....
double sum2=computeSum(workers);

```
AL<String> strings=new AL<>(Arrays.asList(.....));
strings.add("abcd");
double sum3=computeSum(strings);//javac err : since String IS NOT an Emp
```

Solve :
ArrayList(Collection<? extends E> coll)

Can you create populated AL of Fruits(AL<Fruit>) from
1. HS<Orange>
2. Vector<Fruit>
3. LL<String>
4. TS<Mango>
5. AL<Alphonso>
6. LL<Object>
Ans : 1 , 2,4,5

Ref : Map i/f method
public void putAll(Map<? extends K,? extends V> map)

Can you do the bulk addition for populating HM<Integer,BankAccount> from
1. Collection<BankAccount>
2. HM<Integer,BankAccount>
3. TM<Integer,SavingAccount>
4. TM<Integer,LoanAccount>
5. LHM<Integer,Object>
6. TM<Integer,FDAccount>
Ans : 2, 3,4,6

List<? extends Emp> :
legal args : AL/LL/Vector <Mgr>
list.add(new Worker(...));

? : un restricted access
PECS : Producer Extends Consumer Super

extends : upper bound (while using the collection as a Producer , meaning to get data from the collection)

BUT you CAN NOT add any elems to it.
If it was allowed -- it can lead to ClassCastExc

Solution : When you want to treat the collection as a consumer(i.e want to add the elems)
super : keyword in generic syntax
specifies : inclusive lower bound
? super T => T or it's super types

Can you add any elem to such a list(using extends) ?NO
Why ? : to avoid class cast exception

4. Discuss Day 8's generic assignments

5. Add a method to hire a sales mgr to the list
What should be the type of the List ?
AL/LL/Vector <SalesMgr,Mgr,Emp,Object>

6. Add a method to hire a temp worker to the list
What should be the type of the List ?

Today's topics
1. Java 8 new features
default n static methods in i/f
Ref to readme : day9-data\day9_help\java8_new_features.txt

2. Enter Functional prog in Java
Functional i/f
Ref to readme : day9-data\day9_help\java8_new_features.txt

3. Lambda expression
Ref to readme :day9-data\day9_help\regarding lambdas.txt

4. Create your own functional interface n substitute it with lambda expression

Question : Perform ANY arithmetic operation on 2 double operands & return the result
eg --add/multiply/subtract/divide.... ANY such operation !

4.1 Create i/f : SAM => Functional i/f
eg : Computable

4.2 Later : Create another i/f : ComputationUtils : to add a static method
To invoke compute method on 2 double operands n operation

4.3 Create a Tester n use it.
Operations to perform : add/multiply/subtract/divide ....

5. Higher order Functions/methods already existing in Collection frmwork

Solve :

5.1.
Display all products (from a product list)
API : Iterable : forEach
eg : productList.forEach(p -> sop(p));

5.2.
Remove all the products of a specific category
Collection : removeIf
5.3.
Display all products (from a product map)
Ans this
Is Map Iterable ?
Can u attach DIRECTLY a for-each loop to the map ?

Imperative style
```
for(Product p : map.values())
 sop(p);
```
Any Declarative (Functional prog solution ?)
????

----------Pending----------
5.4. Sort the products in the list as per date : custom ordering
How ?

5.5 Use Function literal

5.6 Using above API(higher order funcs) , can you chain multiple operations like sorting & displaying the sorted data ?

OR
Filter the elements n display filtered list.

Objective : Accept product category from user
remove all products from specified category n display remaining list : multiple ops can't be chained n it will modify the original List !

Better Solution :
6. Enter Java 8 Streams (Functional Streams)
6.1 Refer to diagrams from : day9-data\day9_help\streams\diagrams
collections vs streams n streams basics
6.2 Refer to
day9-data\day9_help\streams\readme_streams.txt

The iterators returned by this class's `iterator` and `listIterator` **methods are *fail-fast*: if the 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`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs*.

This class is a member of the Java Collections Framework.

```
case 11:
    Collections.sort(showroom,(Vehicle v1,Vehicle v2)->{
        return ((Double)v1.getNetPrice()).compareTo((Double)v2.getNetPrice());
    }
            );
    return ((Integer)v1.getNetPrice()).compareTo((Integer)v2.getNetPrice());
    break;
case 12:
    // sort the vehicles as per date n price
    Collections.sort(showroom,(Vehicle v1,Vehicle v2)->
    {
        int retnval=v1.getManufactureDate().compareTo(v2.getManufactureDate());
        if(retnval==0)
            return ((Double)v1.getNetPrice()).compareTo((Double)v2.getNetPrice());
        return retnval;
    });
```

Lamda Expression clarity

```
package p4;

@FunctionalInterface
public interface Computable {
    //public abstract : WHAT (specs)
    double anyOperation(double d1, double d2);
}
```

```
package p4;

import static p4.Utils.invokeAnyOperation;

public class Test1 {
    static double invokeAnyOperation(double a, double b, Computable ref) {
        return ref.anyOperation(a, b);
    }

    public static void main(String[] args) {
        //imperative style of prog
        // display addition of 2 numbers
        System.out.println(invokeAnyOperation(10, 20, new Computable() {

            @Override
            public double anyOperation(double d1, double d2) {
                // TODO Auto-generated method stub
                return d1 + d2;
```

```
            }
        }));
        // display subtraction of 2 numbers
        System.out.println(invokeAnyOperation(10, 20, new Computable() {

            @Override
            public double anyOperation(double d1, double d2) {
                // TODO Auto-generated method stub
                return d1 - d2;
            }
        }));

        // replace above by concise ano func : lambda expr
        //declarative style of prog (Functional Prog)
        System.out.println(invokeAnyOperation(10, 20, (a, b) -> a + b));
        //multiply 2 nos using lambda expr
        System.out.println(invokeAnyOperation(10, 20,(d1,d2)-> d1*d2));

    }

}
```

```
package p4;

public interface Utils {
//add a static method , to invoke anyOperation on 2 double operands n ret
result
    // higher order function/method : func or a method which either accepts a
    // behavior or rets the behavior to the caller
    static double invokeAnyOperation(double a, double b, Computable ref) {
        return ref.anyOperation(a, b);
    }
}
```

```
case 8:
                        // API of Collections class : public static void
sort(List<T> list,Comparator<T>
                        // comparator)
                        Collections.sort(showroom, new
VehiclePriceComparator());
                        break;
                    case 9:
                        // sort the vehicles as per date n price
                        Collections.sort(showroom, new
VehicleDatePriceComparator());
                        break;
```

```
                    case 10: // custom ordering using ano innner class
                        Collections.sort(showroom, new Comparator<Vehicle>() {
                            @Override
                            public int compare(Vehicle o1, Vehicle o2) {
                                System.out.println("compare : date -ano inner
cls");
                                return
o1.getManufactureDate().compareTo(o2.getManufactureDate());
                            }
                        });
                        break;

                    case 11:
                        Collections.sort(showroom,(Vehicle v1,Vehicle v2)->{
                             return
((Double)v1.getNetPrice()).compareTo((Double)v2.getNetPrice());
                         }
                                 );
//                      return
((Integer)v1.getNetPrice()).compareTo((Integer)v2.getNetPrice());
                        break;
                    case 12:
                        // sort the vehicles as per date n price
                        Collections.sort(showroom,(Vehicle v1,Vehicle v2)->
                        {
                            int
retnval=v1.getManufactureDate().compareTo(v2.getManufactureDate());
                            if(retnval==0)
                                return
((Double)v1.getNetPrice()).compareTo((Double)v2.getNetPrice());
                            return retnval;
                        });
```

In lambda expression we pass the behavior to function as the argument to the High order function in the class/interface implementing the functional interface so that we are overriding compare by just riding implementing ;logic in lambda expression}
If a method expectsx a interface Instance(Instance of class implementing that interface) and the interface is Functional interface with only one single abstract method, then the lamda expression depiccsts the logic of overriding that single abstract method .

Sort is invokeAnyOperation ,& compare is AnyOperation{we are overriding compare by just riding implementing ;logic in lambda expression}

Functional Streams:

Java 8 Functional streams

A stream represents sequence of elements from a data source .
Streams allow us to operate on the elements from that data source and makes bulk processing convenient and fast.
Streams are wrappers around a data source(eg : array , collection , file) .

A stream does not store data , meaning it is not a data structure. It also never modifies the underlying data source.

API – java.util.stream – supports functional-style operations on streams of elements, such as filter -map-reduce transformations on collections.

Typically Stream will have
source
intermediate operations(0 or multiple)
terminal opeartion (single)

Details

1. How to create stream of ints from array?
java.util.Arrays class method
public static IntStream stream(int[] array)
Returns a sequential IntStream with the specified array as its source.

2. How to create stream(sequential) from Collection ?
Collection i/f method
public Stream<E> stream()

3.How to create stream(parallel) from Collection ?
Collection i/f method
public Stream<E> parallelStream()

IntStream Methods

1. public void forEach(IntConsumer action)
Performs an action for each element of this stream.

Solve :
1.Create int[] ---> IntStream & display its contents.
1.5 Create int[] ---> IntStream --> sort(asc) --> & display its contents.

2. Create AL<Integer> , populate it.
Convert it to seq stream & display elems
2.5 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)
3.5. Create Stream<Integer> from a fixed size list (un sorted) of integers , sort (desc) n display it.

4. list : Display all product names of a particular category , exceeding specific price.
I/P category name & price.
eg : productList.stream()
.filter(p -> p.getCategory()==cat)
.filter(p -> p.getPrice() > price)
.forEach(p -> sop(p.getName()));

5.map : Display all product names of a particular category , exceeding specific price.
map.values() //Collection<Product>
.stream()
.filter(p -> p.getCategory()==cat)
.filter(p -> p.getPrice() > price)
.forEach(p -> sop(p.getName()));

6. Display sum of prices of all products from a chosen category
eg : list.stream() //Stream<Product> : all
.filter(...) //Stream<Product> : filtered
.mapToDouble(p -> p.getPrice() //DoubleStream
.sum(); //terminal

4.5 Prompt user for category n discount.
Apply that discount on all products of specified category n print it.

5. Display sum of all even nos between 1-100 .
Hint : IntStream --range --filter --reduce(sum)

6. Display average of all odd nos between 1-100 .

7.Print avg of odd numbers between 10 & 50 ((if strm is empty print -100 or else print avg)
(filter,average,optional)

8. Display sum of product prices of a specific category.
I/P category name
Products are stored in a map
o/p sum.

9. Display average of product prices of a specific category.
I/P category name
o/p average

10. Sort products of a specific category as per price n display their names.
i/p category name

Exam objectives
Important Facts of Java 8 Streams:
1. Streams are implicitely closed , after terminal operation (i.e they can't be re used after terminal operation)
Otherwise throws --IllegalStateException (reason :stream has already been operated upon or closed)
Where as , collections are re-usable.

2. Streams follow , vertical execution order.

3. Streams support lazy evaluation (meaning none of the intermediate operations are performed , until its closed by terminal operation)

Collections Vs Java 8 Streams

1. Collections(Set/List/Map) actually stores the elements (i.e references)	1. Doesn't store all the elements (represents only sequence of operations to be performed on elements
2. Support add/remove (API supports)	2. Stream API doesn't support add/remove operations.
3. Eagerly evaluated	3. Lazily evaluated.
4. Supports external iteration (using for-each/iterators)	4. Supports internal iteration
5. Don't support easy way for concurrent handling	5. Easy way for parallel processing
6. Can be iterated over multiple times.	6. Can't be iterated over multiple times ,can be processed only once.
7. Don't support aggregate functionality (eg : match,filter count,limit,reduce,flatMap ..)	7. Support functional features like match,filter....

- **Stream Source**
  - Streams can be created from Collections, Lists, Sets, ints, longs, doubles, arrays, lines of a file
- Stream operations are either intermediate or terminal.
  - **Intermediate operations** such as filter, map or sort return a stream so we can chain multiple intermediate operations.
  - **Terminal operations** such as forEach, collect or reduce are either void or return a non-stream result.

Read data from a text file till EOF(end of file) using buffer
(chaining of I/O streams) n display file contents .

```
try(sc ; BufferedReader br=new BR(new FR(sc.next()))
{...}
```

Method of BufferedReader
public String readLine() throws IOException
Rets : line of text from the buffer, null at EOF

```
BufferedReader br=new BufferedReader(new FileReader(fileName));
//Method of BR : public String readLine() throws IOException
Rets : line by line data from a buffer.
End of stream (File) : rets null
while((s=br.readLine()) != null) sop (s);
```

File copy (using buffers) : text file User i/p : src n dest file names

Chaining of streams
PrintWriter pw=new PrintWriter(new FileWriter(destFileName));
Methods of PW
same as PrintStream
print / println /printf

---

java.io Overview (java.io)
Any run time err inread/write --java.io.IOException
java.io API --blocked API. Any read/write operation causes the blocking of invoker thread.
till the operation is complete.

Binary data streams
For read/write operation ,
data xfer unit is byte/byte[]

Char data streams
D.T.U --- char / char[]

InputStream	OutputStream	Reader	Writer
abstract super classes			
Bin i/p stream	Bin o/p stream	char i/p strm	char o/p strm
FileInputStream	FileOutputStream	FileReader	FileWriter
Node streams for handling files -- un buffered.			
PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Node strms for handling data pipes between thrds unbuffered			
BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
			PrintWriter
ObjectInputStream	ObjectOutputStream		

Day10
BufferefWriter vs PrintWriter

BufferedWriter vs PrintWriter
common --char o/p strms , buffered strms(have buffering capability.

BufferedWriter(Writer w, int size) can specify size of buffer.	PrintWriter(Write w) -- can't specify size of buffer.(def size is supplied)
API(methods) --limited write methods only	API -- write ... + print,println,printf
can only wrap char o/p strms (doesn't support the conversion char --bin)	can wrap either char or bin strms PrintWriter(Write w) PrintWriter(OutputStream out)
does't support auto flushing	Supports auto flushing(flush on new line) PrintWriter(Writer w,boolean autoFlush) PrintWriter(OutputStream out,boolean autoFlush)

Typical devices

Scanner vs Buffered Reader

Scanner vs BufferedReader
Common Points -- Both represent buffered character input streams.

Scanner
1. Smaller inherent buffer (~ 1K)

2. Inherently thread un safe

3. Does have parsing support (eg : nextInt,nextFloat....)

BufferedReader -- a subclass of Writer
1. Larger inherent buffer (~ 8k)

2. Inherently thread safe

3. No parsing API

Scanner can be combined with BR , to achieve both of the advantages.
eg : Scanner sc=new Scanner(br.readLine()); & then parse....

Types of Streams in java.io

Node Streams (Device Handling streams)
eg : FileInputStream, PipedInputStream, Socket Streams

Buffering Streams -- used typically in larger data volumes . Can be added for reading as well as writing .
JA , instead of reading data directly from a node stream, data can be read into a buffer & then into JA.
Similarly , JA can dump data into a buffer , when buffer becomes full, it can flush out the contents to device (node stream)

eg : BufferedInputStream,BufferedOutputStream,PrintWriter.

Conversion/Filtering streams
Instead of reading data in chunks of bytes or chars --formatting strms can convert the data into primitive types or java objects.
eg : DataInputStream DataOutputStream, ObjectOutputStream,ObjectInputStream

---

I/O Streams
I/O Fundamentals

The basic units of Input/Output in Java are Streams A stream can be thought of as a flow of data from a source to a sink Source stream initiates the flow of data (also called as InputStream) Sink stream terminates the flow of data (also called OutputStream) Sources and sinks are node streams

Types of node streams

Types of node streams are:

- Files
- Memory
- Pipes (primarily used between threads and processes)

Data within streams

Java supports two types of streams

- Character streams
- Byte streams

Input and output of character data are handled by readers and writers

Input and output of byte data are handled by input streams and output streams

- The term stream refers to byte stream
- The terms readers and writers refer to character data

Data within streams

Byte Streams Character Streams

Source Streams **InputStream Reader** Sink Streams

OutputStream Writer

InputStream Methods

Three basic read methods

- int read ()
- int read (byte[] buffer)
- int read (byte[] buffer, int offset, int length)

Other methods

- void close ()
- int available () //bytes that can be immediately read ▪ skip

(long n)

- boolean markSupported () //mark and reset are operational ▪ void

mark (int readLimit) //marks the point and allocate buffer ▪ void reset ()

//returns stream to the marked point

OutputStream Methods

Three basic read methods

- void write (int c)
- void write (byte[] buffer)

- void write (byte[] buffer, int offset, int length)

Other methods

- void close ()

- void flush () //forces writes to happen

You should close output streams when you are finished with them

DemoFile.java

```
1 import java.io.*;
2 public class DemoFile
3 {
4 public static void main(String[] args) {
5 try
6 {
      // create a inputstream object to read from a file
7 FileInputStream in = new
FileInputStream(args[0]); 8 // create a
outputstream object to write into a file 9
FileOutputStream out = new
FileOutputStream(args[1]); 10 int v;
11 while ((v = in.read()) != -1)
12 out.write(v);
13
14 in.close();
15 out.close();
16 }
17 catch (IOException io)
18 {
19 System.err.println("**An IO problem occurred " + io);
20 }
21 }
22 }
```

Reader Methods

Three basic read methods

- void read (int c)
- void read (char[] cbuffer)
- void read (char[] cbuffer, int offset, int

length) Other methods

- void close ()
- boolean ready ()
- skip (long n)
- boolean markSupported ()
- void mark (int readAheadLimit)
- void reset ()

Writer Methods

Three basic read methods

- void write (int c)
- void write (char[] cbuffer)
- void write (char[] cbuffer, int offset, int

length) ▪ void write (String string)

- void write (String string, int offset, int

length) Other methods

- void close ()

- void flush ()

Node Streams

Java has three basic types

- File streams
- Memory streams
- Pipe streams

Possible to create new node stream classes but you have to deal with native function calls

Node Streams

Type	Character Streams	Byte Streams
File	**FileReader FileWriter**	**FileInputStream FileOutputStream**
Memory: Array	**CharArrayReader CharArrayWriter**	**ByteArrayInputStream ByteArrayOutputStream**
Memory: String	**StringReader StringWriter**	
Pipe	**PipedReader PipedWriter**	**PipedInputStream PipedOutputStream**

Reader/Writer Example

```
import java.io.*;
public class TestNodeStream {
        public static void main (String[] args) {
                try { FileReader input = new FileReader (args[0]);
                        FileWriter output = new FileWriter (args[1]);
```

```
                    char[] buffer = new char [128];
                    int charsRead;
                    charsRead = input.read (buffer); //Read first buffer
                    while (charsRead != -1) {
                    output.write (buffer, 0, charsRead); //Write buffer to output file
                    charsRead = input.read (buffer); //Read the next
                    buffer }
                    input.close ();
                    output.close ();
                }catch (IOException ioe) {
                    ioe.printStackTrace (); }
        }
```

Buffered Reader/Writer Example

```
import java.io.*;
public class TestBufferedStream {
        public static void main (String[] args) {
                try {
                FileReader input = new FileReader (args[0]);
                BufferedReader bufInput = new BufferedReader (input);
        //Chaining of file reader object
                        FileWriter output = new FileWriter (args[1]);
                        BufferedWriter bufOutput = new BufferedWriter (output);
                        String line;
                        line = bufInput.readLine (); //Read the first line
                        while (line != null) {
                                bufOutput.write (line, 0, line.length);
                                bufOutput.newLine ();
                                line = bufInput.readLine (); //Read the next line
                        }
                        input.close (); output.close ();
                }catch (IOException ioe) {
                        ioe.printStackTrace ();
                } } }
```

I/O Stream Chaining

Illustrated in the previous example

A program rarely uses a single stream object

It chains a series of streams to process data

Data Source Program FileInputStreamBufferedInputStream **DataInputStream**

Program Data Sink

FileOutputStream

BufferedOutputStream **DataOutputStream**

Processing Streams

Perform some sort of conversion on another stream

Also known as filter streams

A filter input stream is created with a connection to an existing input stream

When you try to read from filer input stream object, it supplies you with characters that came from the original stream

Conversions then may be performed to convert raw data into a more usable form

Types of Processing Streams Type Character Streams Byte Streams

Buffering **BufferedReader BufferedWriter** **BufferedInputStream BufferedOutputStream**

Filtering ***FilterReader FilterWriter*** ***FilterInputStream FilterOutputStream***

Converting between bytes & characters **InputStreamReader** **OutputStreamWriter**

Object Serialization **ObjectInputStream ObjectOutputStream**

Data Conversion **DataInputStream DataOutputStream**

Counting **LineNumberReader LineNumberInputStream** Peeking ahead **PushbackReader**

PushbackInputStream Printing **PrintWriter PrintStream**

Creating a Random Access File

Often you would want to read data within a file without reading the file from the beginning to end

Java provides a RandomAccessFile class for handling this type of input/output

Two options for creating

- With file name myRAFile = new RandomAccessFile (String name, String mode);
- With file object myRAFile = new RandomAccessFle (File file, String more);
- Examples
- RandomAccessFile myRAFile = new RandomAccessFile ("stock.dbf", "rw");

Random Access Files

RandomAccessFile objects expect to read and write information in the same manner as data input and data output objects

You have access to all of read () and write () operations found in the DataInputStream and DataOutputStream classes

long getFilePointer () **//Current location of file pointer** void seek (long pos) **//Set file pointer to a specified position** long length () **//Length of the file**

Console Input output

For User interaction with application we required console I/O. Writing to the standard output, the console that launched the application, is achieved using **`System.out`**

`System.out` is of type **`PrintStream`**,

so it allows writing to the standard output a variety if data types using either

`println()` or `print()`:

- `void println(char)`
- `void println(String)`
- `void println(float)`
- `void println(int)`

Reading From the Standard Input

`System.in`, which is an `InputStream` object, accepts user keyboard information and transfers it to the application.

`InputStream` reads binary data, so some bridging must be done to convert it to buffered character data.

InputStreamReader in = new InputStreamReader(System.in); BufferedReader bufin = new BufferedReader(in);

ReadKeyboard.java

```
1 import java.io.*;
2 class DemoKeyboard
3 {
4 public static void main(String args[]) { 5
InputStreamReader in = new
6 InputStreamReader(System.in); 7
BufferedReader bufin = new
8 BufferedReader(in);
9 while(true)
10 {
11 try {
12 System.out.println(bufin.readLine()); 13 }
14 catch(IOException e) {
15 e.printStackTrace();
16 }
17 }
18 }
19 }
```

Serialization

Ability to read or write a Java object to a stream Addition since JDK 1.1

Saving an object to persistent storage is called persistence Java provides a Serializable interface

Serializable has no methods and is a marker interface

Serialization – Object Graphs

When an object is serialized, only data of the object are preserved

If a data variable is an object, data members of the object are also serialized if that object's class is serializable

The tree of object's data, including these sub-objects constitutes an object graph

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"

Writing Objects to Streams

```
import java.io.*;
import java.util.Date;
public class SerializeDate {
        public SerializeDate () {
                Date d = new Date ();
                        try { FileOutputStream f = new FileOutputStream
                            ("date.ser"); ObjectOutputStream s = new
                                        ObjectOutputStream (f);
                            s.writeObject (d);
                            s.close ();
                }catch (IOExceptions ioe) {
                            e.printStackTrace ();
                }
        }
```

```
public static void main (String[] args) {
        SerializeDate s = new SerializeDate ();
} }
```

Transient Members

There are two reasons for declaring a data member as transient: The member is an object that does not implement **`Serializable`**. If We don't wish to write the data member value into the stream. salary will not be a part of serialization process

```
public class Emp implements Serializable {
        transient int salary;
}
```

The `File` Class

The **`File`** class provides a variety of methods which will give more information about a file

Create **`File`** objects and you can check following properties of a file whether a File object can be read and written to.

whether a **`File`** object exists.
Get file (and path) names.
Get a list of file names inside a directory.
Create a Directory.
Rename a file.
last modification date.
Delete the **`File`** object.

```
File f1 = new File("Demo.txt");
```

File Class methods

boolean delete()

isDirectory()

exists() –

listFiles()

mkdir() – Creates a directory according to this File object. renameTo() - Renames the file denoted by this object. getName()

getParent()

getPath()

length

list()

Assignment

Create a class Book having details like bcode, bname , price

Create object of book class and write into a file

Read same object from the file and Display it on console

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.

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 class 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 ---
int readInt() throws IOException;
double readDouble() throws IOException;
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 to JVM , to allow or to not allow serialization n de-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 serialVersionUID, 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.

Copy text file (uisng buffering)

Triggers for sending data from PW --> next node
1. buf full or 2. public void flush() --sends the buf contents to next node

3. public void close() or 4. auto flush => flush on new line
For this PrintWriter(Writer w , boolean autoFlush)

Chain of I/O streams for serialization

---

chain of streams for de-serialization

What are the pre requisites for de-serialization ?
1. state : from bin file
2. behavior (.class files)

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:

eg : list.forEach(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)

list.forEach(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
    }
}
```

Blocking Triggers Vs Un blocking triggers

1. Blocked on sleep (eg : Thread.sleep(ms))	1.1 sleep over 1.2 interrupt (InterruptedException)
2. Blocked in i/o eg : System.in.read()	2 Data available or write operation complete
3. Blocked on join join() join(long ms)	3.1 specified thread completes exec(dead) 3.2 interrupt 3.3 tmout elapsed.
4. Blocked on monitor	4 Monitor being free(lock released)
5. Blocked on wait Object class API public void wait() throws IE public void wait(long ms) throws IE	5.1 notify 5.2 notifyAll 5.3 interrupt 5.4 tmout elapsed

---

Multi Tasking --- To inc effective utilization of Processor & resources.
(to ensure that blocking task DOESn't affect other runnable tasks)

Processs level --- swapping unit = entire process	**Thread level -- finer control. Process can be sub-dived into parallel , concurrent runnable task --- individual task represented by a thrd.**
Can't be controlled by JRE/JVM	**Can be controlled partially**
Context switching --heavy weight	**Context switching --light weight --since all thrds belonging to same process SHARE common adr space.**

---

java.lang.Runnable i/f
SAM : public void run() : prog MUST imple this method in either approach

```
public class MyRunnableTask implements Runnable
{
  //imple run() : to supply B.L
}
```

Tester :
Thread t1 =new Thread(new MyRunnableTask(),"one");

java.lang.Thread : class implements Runnable
run() : BLANK B.L

```
public class MyThread extends Thread
{
 //MyThread IS a Thread
  //invoke super(name) n start it
//MUST override run() : to add B.L
}
```

Objective : testing concurrent exec of multiple thrds

system thrd : main : parent thread
Will be used for creating child thrds
main(...)
create child thrds : by calling the constr n test concurrency

"one" "two" "three" "four"

JVM starts 2 system threads

main

searches for
psvm(String[] args)
n begin it's execution

GC

Objective : Integration of collection+ I/O +Threads
Create a multi threaded java app to store Student data sorted as per dob in 1 text file (done by 1 thread) n sorted as per subject in another text file (done by another thread)

main thread

1.get sample data from CollectionUtils in a HashMap
2. Create thrds , pass the SAME HM to both of thrds n start the child thrds
3. wait for child thrds to finish exec
4. main over

HashMap : students
fileName

dob_sorter

```
run()
{
 hm-->collection -->strm->
sorted as per dob--> write it in
text file
}
```

+ constr : HM,fileName

subject_sorter

```
run()
{
hm-->collection -->strm->
sorted as per subject-->
write it in another text file
}
```

+ constr : HM , fileName

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.

What are different ways to supply Runnable instance?
1. Create a separate class that implements Runnable
OR
2. Anonymous inner class
OR
3. lambda expression

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 --Blocking API
 Forces the invoker thread to release processor & monitor & wait outside the monitor(i.e releases the lock)

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.

Multithreading pdf in day 12 code help folder

Multithreading

Objective

In this session, you will be able to:

- Define a thread
- Create threads in java using two approaches
- Describe the life cycle of a thread
- Synchronize threads

Thread

- A thread is a single sequential flow of control within a program. •
lightweight process
- execution context

Threads in Java

• There are two techniques for creating a thread: ▪

Subclassing *Thread* and Overriding run ▪

Implementing the *Runnable* Interface

The *Runnable* Interface

• The *Runnable* interface should be implemented by any class whose instances are intended to be executed by a thread.

• The class must define a method of no arguments called run.

• public void **run()**

▪ When an object implementing interface *Runnable* is used to create a thread,

starting the thread causes the object's run method to be called in that separately executing thread.

The Thread Class

- **Thread()**

  - Allocates a new Thread object.

- **Thread(Runnable target)**

  - Allocates a new Thread object.

- **Thread(Runnable target,String name)**

  - Allocates a new Thread object.

- **Thread(String name)**

  - Allocates a new Thread object.

Subclassing Thread

```
class SimpleThread extends Thread {
 public SimpleThread(String str) {
 super(str);
 }
 public void run() {
 for (int i = 0; i < 10; i++) {
 System.out.println(i + " " + getName());  try {
 sleep((long)(Math.random() * 1000));  } catch
(InterruptedException e) {}
 }
 System.out.println("DONE! " + getName());  }
}
```

Implementing the Runnable Interface

```
  class SimpleThread implements Runnable {
        public void run() {
              Thread t = Thread.currentThread();
              for (int i = 0; i < 10; i++) {
```

```
                System.out.println(i + " " + t.getName() );
                try {
                        Thread.sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + t.getName());
    }
}
```

[redacted]

```
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;
public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
clockThread = new Thread(this, "Clock");
                clockThread.start();
        }
    }//start
    public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
                repaint();
try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){}
        }//while
}//run
```

Implementing the Runnable Interface

```
public void paint(Graphics g) {
    // get the time and convert it to a date
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        // format it and display it
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        g.drawString(dateFormatter.format(date), 50, 50);
}//paint
```

```
        // overrides Applet's stop method
        public void stop() {
                clockThread = null;
        }//stop
}// Clock
```

The Life Cycle of a Thread

- Creating a Thread
- Starting a Thread
- Making a Thread Not Runnable
- Stopping a Thread

The Life Cycle of a Thread

- Creating a Thread:

```
public void start() {
        if (clockThread == null) {
                clockThread = new Thread(this, "Clock");
                clockThread.start();
        }
}
```

- After the statement containing new is executed, clockThread is in the *New Thread* state; no system resources are allocated for it yet.

- When a thread is in this state, we can only call the start method on it.

The Life Cycle of a Thread

- Starting a Thread :

```
public void start() {
        if (clockThread == null) {
                clockThread = new Thread(this, "Clock");
```

```
clockThread.start();}}
```

- The start method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's run method.

- After the start method returns, the thread is said to be "running". In reality a thread that has been started is actually in the Runnable state.

- At any given time, a "running" thread actually may be waiting for its turn in the CPU.

The Life Cycle of a Thread

- The run( ) method :

```
public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
                repaint();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){}
        }}
```

- Clock's run method loops while the condition

  clockThread = = myThread is true.

- Within the loop, the applet repaints (which further calls paint) itself and then tells the thread to sleep for one second.

The Life Cycle of a Thread

- Making a Thread Not Runnable:

- A thread becomes Not Runnable when one of these events occurs: ▪ Its **sleep( )** method is invoked.

- The thread calls the **wait( )** method to wait for a specific condition to be satisifed.

- The thread is blocking on I/O.

The Life Cycle of a Thread • The clockThread in the Clock applet becomes *Not Runnable* when the run method calls *sleep* on the current thread:

```
public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
                repaint();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){}
        }
}
```

The Life Cycle of a Thread

- Stopping a Thread :
- A program doesn't stop a thread, rather, a thread arranges for its own death by having a run method that terminates naturally. For example, the while loop in this run method is a finite loop-- it will iterate 100 times and then exit:

```
public void run() {
int i = 0;
                while (i < 100) {
i++;
                        System.out.println("i = " + i);
                } }
```

- A thread with this run method dies naturally when the loop completes and the run method exits.

The Life Cycle of a Thread

- Stopping a Thread :
  - Stopping the Clock applet thread

```
public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
                repaint();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){}
        }}
```

- The exit condition for this run method is the exit condition for the while loop because there is no code after the while loop:

while (clockThread = = my Thread)

The Life Cycle of a Thread

- Stopping a Thread :
  - Stopping the Clock applet thread

- When we leave the page, the application in which the applet is running calls the applet's stop method. This method then sets the clockThread to null, thereby telling the main loop in the run method to terminate: public void stop() {// applets' stop method

clockThread = null; }

- If we revisit the page, the start method is called again and the clock starts up again with a new thread.

The Life Cycle of a Thread

- The **isAlive()** Method :
  - The **isAlive()** method returns true if the thread has been started and not stopped.
  - If the **isAlive()** method returns false, you know that the thread either is a New Thread or is dead.
  - If the **isAlive()** method returns true, you know that the thread is either Runnable or Not Runnable.

Thread Priority

• The Java runtime supports fixed priority preemptive scheduling.

• When a Java thread is created, it inherits its priority from the thread that created it. Which can be changed later using the *setPriority* method.

• Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY.

Thread Priority

• The runtime system chooses the runnable thread with the highest priority for execution.

• If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.

• The chosen thread will run until one of the following conditions is true: ▪ A higher priority thread becomes runnable.

▪ It yields, or its run method exits.

▪ On systems that support time-slicing, its time allotment has expired.

Controlling Threads

• It's an art of moving threads from one state to another state.

• It is achieved by triggering state transitions. • The various pathways out of the running state can be : ▪ Yielding

▪ Suspending and then resuming

▪ Sleeping and then waking up

▪ Blocking and then continuing

▪ Waiting and then being notified

Yielding

- A call to yield method causes the currently executing thread to move to the ready state, if the scheduler is willing to run any other thread in place of the yielding thread.

ready

scheduled yield()

running

- A thread that has yielded goes into ready state.
- The yield method is a static method of the Thread class.

Suspending

- It's a mechanism that allows any arbitrary thread to make another thread un-runnable for an infinite period of time.
- The suspended thread becomes runnable when some other thread resumes it
- Deadlock prone, because the control comes from outside the thread. • The exact effect of suspend and resume is much better implemented using wait and notify.

Sleeping

- A sleeping thread passes time without doing anything and without using the CPU.
- A call to the sleep method requests the currently executing thread to cease it's execution for a specified period of time.
  - public static void sleep(long ms)

    throws InterruptedException
  - public static void sleep(long ms, int ns)

    throws InterruptedException
- Both sleep and yield work on currently executing thread.

Blocking

- Methods that perform input or output have to wait for some occurrence in the outside world before they can proceed, this behavior is known as blocking.

```
try {
        Socket s = new Socket("Devil",5555);
        InputStream is = s.getInputStream();
        int b = is.read();
}
catch(IOException e) {
// Handle the exception}
```

Blocking

- A thread can also become blocked if it fails to acquire the lock for a monitor if it issues a **wait( )** call.

- The **wait( )** method puts an executing thread into the waiting state, and the **notify( )** and **notifyAll( )** put them back to ready state.

Thread Synchronization

```
public class Consumer extends Thread {
        private MailBox mailBoxObj;
        public Consumer(Mailbox box) {
                this.mailBoxObj = box;
        }
        public void run() {
                while(true) {
                        if(mailBoxObj.request) {
                        System.out.println(mailBoxObj.msg);
                                mailBoxObj.request = false;
                        }
                try {
                        sleep(50);
                } catch(InterruptedException e){}
        }
```

Better Solution

```
class MailBox{
        private boolean request;
        private String msg;
        public synchronized void storeMessage( String s){
                request = true;
                msg = s;
        }
        public synchronized String retrieveMessage(){
                request = false;
                return msg;
        }
}
```

The Complete Solution

```
public synchronized String retrieveMessage(){  while(request == false){ //
                No message to retrieve try{
                        wait();
        } catch (InterruptedException e) {}
                }
                request = false;
                notify();
                return str;
}
```

Summary

In this session, you learnt to:

- Define a thread
- Create threads using two approaches •

Describe the life cycle of a thread • Synchronize threads

Day 13

User thread vs Daemon threads

User Threads Vs Daemon Threads

Created by default (i.e by using any of the Thread class constr)	Are created using Thread class method public void setDaemon(boolean flag) ---flag =true-- indicates creation of daemon thread.
	eg -- Thread t1=new Thread(new MyTask(),"abc");
	t1.setDaemon(true);t1.start();

User threads , if running , prevent JVM from termination	Daemon threads, DON'T prevent JVM from termination (when all user threads have teminated) Usage -- typically as service threads. eg -- Garbage collector

NOTE : Java appln terminates only when all user threads have terminated.

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
BUT a single lock CAN NOT be shared across multiple threads.

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.

—----------

Need of synchronization (sharing of common resource/s across multiple threads)

Day 14

ORM Overview (Object Relational Mapping)

Object World	Relational Database World
POJO classes	DB Tables
POJO properties	Table Columns
POJOs	Table rows
Unique ID Property	Primary Key

JDBC Steps

Add JDBC driver in run time class path.(In IDE add ext jar)

1. Load Type IV JDBC driver in JVM's method area.(meta space)
API of java.lang.Class<T>
public static <T> T forName(String clsName) throws ClassNotFoundException
2. Get a FIXED connection from underlying DB.
API of java.sql.DriverManager class
public static Connection getConnection(String dbURL,String userName,String password)
throws SQLException
dbURL -- DB sepecific for Type IV JDBC driver
syntax : jdbc:mysql://localhost:3306/test

java.sql.Connection

public Statement createStatement()
throws SQLException

Use case -- for static n non -repeatative queries
static --NO IN , OUT , IN OUT params

public PreparedStatement prepareStatement(String sql) throws SE
Use case -- dynamic queries -- supports IN params.
Best for repeatative queries.
Represents --pre parsed n pre-compiled statement object

public CallableStatement prepareCall(String invocationSyntax) throws SE
Use case --- For executing stored procedures/funs
Supports -- IN , OUT, IN OUT params.

Execution Methods

A

public ResultSet executeQuery(String sql) throws SE

public ResultSet executeQuery() throws SE

Use case -- for select queries (for ResultSet returning queries)

public int executeUpdate(String sql) throws SE

public int executeUpdate() throws SE

Use Case --- can be used for
DDL ---create table/alter table/create procedure/create function....
DML --insert/update/delete
Rets -- no of rows updated

public boolean execute() throws SE

public boolean execute(String sql) throws SE

Use case -- for executing stored procedure /function OR in case of unknows results.
In case of unknown results --
It rets true -- if query rets ResultSet
If false -- it can either mean update count or end of results.

ResultSet processing
---java.sql.ResultSet -- i/f
Object representation of selceted rows n columns.
API of ResultSet
1. public boolean next() throws SE

2. How to read row data ?
public Type getType(int colPos) throws SE

public Type getType(String colName) throws SE

Type --- JDBC data type / generic SQL type

public Type getType(String colName) throws SE

Type --- JDBC data type / generic SQL type

Native DB types JDBC Data types
char/varchar/varchar2 --- string
number(n) ---- int/long
number(m,n) --- float/double
date --- java.sql.Date
time --- java.sql.Time
timestamp -- java.sql.TimeStamp
clob -- java.sql.Clob
blob -- java.sql.Blob

Before terminating -- close RST, ST/PST/CST
Close DB connection (or use try-with-resources)

A

Set IN Param/s
API of PreparedStatement
public void setType(int paramPosition,Type val) throws SE

Type -- JDBC data type
eg :

Refer pdf 14.1 help folder jdbc.pdf

Correct sequencing logic for producer-consumer scenario , with wait n notify

```
public class Producer implements Runnable {
    private EmpUtils u;
    private boolean exit;
    public Producer(EmpUtils u) {
        this.u = u;
    }
    @Override
    public void run() {
        System.out.println("started thrd  -- "
                + Thread.currentThread().getName());
        try {
            int i = 1;
            while (!exit) {
                u.writeData(new Emp(i, "aa" + i, "HRD", i * 100));
                i++;
                Thread.sleep(20);
            }
        } catch (Exception e) {
            System.out.println("err in " + Thread.currentThread().getName()
                    + "  " + e);
        }
        System.out.println("thrd over -- " + Thread.currentThread().getName());

    }

    public void stop() {
        exit = true;
    }
```

```java
public class Consumer implements Runnable {
    private EmpUtils u;
    private boolean exit;
    public Consumer(EmpUtils u) {
        this.u=u;
    }

    @Override
    public void run() {
        System.out.println("started thrd  -- "
                + Thread.currentThread().getName());
        try {

            while (!exit) {
            //  System.out.println("Read Data "+u.readData());
                u.readData();
                Thread.sleep(400);
            }
        } catch (Exception e) {
            System.out.println("err in " + Thread.currentThread().getName()
                    + "  " + e);
        }
        System.out.println("thrd over -- " + Thread.currentThread().getName());

    }
    public void stop()
    {
        exit=true;
    }
```

```java
public class EmpUtils {
    private Emp e;
    private boolean dataReady;

    public synchronized void writeData(Emp ref) throws Exception {
        System.out.println("w entered --- " + Thread.currentThread().getName());// producer
        while (dataReady)
            this.wait();//p : Blocked on wait : outside the monitor
        //dataReady : false
        // produce data
        e = ref;
        System.out.println("Write Data " + e);
        System.out.println("w exited --- " + Thread.currentThread().getName());// producer
        dataReady=true;
        this.notify();
    }

    public synchronized Emp readData() throws Exception {
        System.out.println("r entered --- " + Thread.currentThread().getName());
        while(!dataReady)
            this.wait();
        //dataReady : true
        // consume data
        System.out.println("Read  Data " + e);

        System.out.println("r exited --- " + Thread.currentThread().getName());
        dataReady=false;
        this.notify();
        return e;
    }
```

```java
public static void main(String[] args) throws Exception {
    EmpUtils u = new EmpUtils();
    String[] nms = { "Producer", "Consumer" };
    // produce task ---thrd --strt
    Producer p1 = new Producer(u);// thrd doesn't exist
    // Thread(Runnable inst,String nm)
    Thread t1 = new Thread(p1, nms[0]);// new
    t1.start();// rdy
    Thread.sleep(10);
    Consumer c1 = new Consumer(u);
    // Thread(Runnable inst,String nm)
    Thread t2 = new Thread(c1, nms[1]);// consumer thrd---new
    t2.start();// rdy pool
    // Thread.sleep(5);
    System.out.println("Press enter to exit");
    System.in.read();//main thrd Blocked on i/p
    p1.stop();
    c1.stop();
    t1.join(1000);//main blocked on join for t1 : max 1 sec
    t2.join(1500);//main blocked on join for t2 : max 1.5 sec
    if(t1.isAlive())
        t1.interrupt();//main sending interrupt  to t1 --t1 : InterruptedExc -- run() : dead
    if(t2.isAlive())
        t2.interrupt();//main sending interrupt  to t2 --t2 : InterruptedExc -- run() : dead
    t1.join();
    t2.join();
    System.out.println("main over...");
}
```

Day 15

Reflection Readme
Why ?

Java Reflection provides ability to inspect and modify the runtime behavior of application.

What is Java reflection ?
Java API for (java.lang.Class<?> , java.lang.reflect)
1. Inspecting classes,interfaces,fields, methods w/o knowing their names at compile time
2. Instantiating new objects
3. Invoking private methods and get/set field values .
4. Dynamic method invocation

Usage

It's of little use in normal programming but it's the backbone for most of the Java, J2EE frameworks,IDEs,debuggers.

Examples

1. JUnit uses reflection to parse @Test annotation to get the test methods and then invoke it.
2. Spring Framework uses it for dependency injection
3. Web servers use it to forward the request to correct module by parsing their web.xml files and request URI.
4. Eclipse auto completion of method names
5. Struts
6. Hibernate .....

They all use java reflection because all these frameworks have no knowledge and access of user defined classes, interfaces, their methods etc.

Drawbacks

1. Poor Performance Since java reflection resolve the types dynamically, it involves processing like scanning the classpath to find the class to load, causing slow performance.
2.
Security Restrictions Reflection requires runtime permissions that might not be available for system running under security manager. This can cause you application to fail at runtime because of security manager.
3. Security Issues Using reflection we can access part of code that we are not supposed to access, for example we can access private fields of a class and change its value. This can be a serious security threat and cause your application to behave abnormally.
4. High Maintenance Reflection code is hard to understand and debug, also any issues with the code cant be found at compile time because the classes might not be available, making it less flexible and hard to maintain.

Entry Point for all reflection -- java.lang.Class<?> -- represents loaded class information in method area.

Important :
JVM instantiates an immutable instance of java.lang.Class , per every loaded class in method area , which provides methods to examine the runtime properties of the object and create new objects, invoke its method and get/set object fields.

How to get access to loaded class(java.lang.Class) ?
1.Through static variable class
eg : Emp.class ---loaded class info --Class<Emp>
eg : Student.class => Class<Student>
2.Using getClass() method of Object
public final Class<?> getClass()
3.Method of java.lang.Class
public static Class<?> forName(String fullyQualifiedName) throws ClassNotFoundException

code eg : TestReflection.java

For primitive types and arrays, we can use static variable class. Wrapper classes provide another static variable TYPE to get the class.

eg :

```
// Get Class using reflection
Class<?> c = MyClass.class;
c = new MyClass(5).getClass();
try {
	// below method is used most of the times in frameworks like JUnit
	//Spring dependency injection, Tomcat web container
	//Eclipse auto completion of method names, hibernate, Struts2 etc.
	//because MyClass is not available at compile time
	concreteClass = Class.forName("com.app.core.MyClass");
} catch (ClassNotFoundException e) {
	e.printStackTrace();
}
System.out.println(concreteClass.getCanonicalName());
//for primitive types, wrapper classes and arrays
Class<?> booleanClass = boolean.class;
System.out.println(booleanClass.getCanonicalName()); // prints boolean

Class<?> cDouble = Double.TYPE;
System.out.println(cDouble.getCanonicalName()); // prints double

Class<?> cDoubleArray = Class.forName("[D");
System.out.println(cDoubleArray.getCanonicalName()); //prints double[]

Class<?> twoDStringArray = String[][].class;
System.out.println(twoDStringArray.getCanonicalName()); // prints java.lang.String[][]
```

Important Methods of java.lang.Class , for inspection

code eg : Reflection1.java

1. To find fully qualified class name

public String getName()

1.5 To find if its class or interface

public boolean isInterface()

2. To find class modifiers

public int getModifiers();

2.1 java.lang.reflect.Modifier class API
public static boolean isPublic(int m)
public static boolean isAbstract(int m)
puiblic static boolean isFinal(int m)

3. To find super class
public Class<? super T> getSuperclass()

Returns the Class representing the superclass of the entity.
For Object class, an interface, a primitive type, or void, ---rets null.
For an array class --Object class

4. To find interfaces
public Class<?>[] getInterfaces()

Determines the interfaces implemented by the class

5. To get accessible fields
public Field[] getFields() throws SecurityException

Returns an array containing Field objects reflecting all the accessible public fields of the class or interface .

6. To get all fields , including private fields
public Field[] getDeclaredFields() throws SecurityException

Returns an array of Field objects reflecting all the fields(public, protected, default and private fields, but NO inherited fields.)

7. To get all accessible constructors of the class
public Constructor<?>[] getConstructors() throws SecurityException

Returns an array containing Constructor objects reflecting all the public constructors of the class

7.1 To find out constr args
java.lang.reflect.Constructor class API

public Class<?>[] getParameterTypes()

Returns an array of Class objects that represent the formal parameter types, in declaration order.

8. To get all visible methods of the class
public Method[] getMethods() throws SecurityException

Returns an array containing Method objects reflecting all the visible methods(including inherited methods) of the class/interface.

Method class API
8.1 public String getName() --rets method name
8.2 public Class<?> getReturnType() -- rets a class representing Method's ret type.
8.3 public Class<?>[] getParameterTypes()
Returns an array of Class objects that represent the formal parameter types, in declaration order

9. To get all (including private but not inherited) methods of the class.

public Method[] getDeclaredMethods() throws SecurityException
& then use similar API of Method class as mentioned above.

Class inspection part ends here.....

1. How to create an instance of the class using default constr?
Class API
public T newInstance() throws InstantiationException

2. How to create an instance of the class using parameterized constr?
2.1 Get the required constr.
Class API
public Constructor<T> getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException

2.2 Constructor class API
public T newInstance(Object... initargs) throws InstantiationException

3. How to invoke private method of the class dynamically ?
3.1 Get desired method object
Class API
public Method getDeclaredMethod(String name,Class<?>... parameterTypes) throws NoSuchMethodException,SecurityException
3.2 Method class API
public void setAccessible(boolean flag)throws SecurityException

3.3 Invoke the method dynamically
API of Method class
public Object invoke(Object invocationObj,Object... methodArgs)throws IllegalAccessException,IllegalArgumentException, InvocationTargetException
Meaning --

Invokes the underlying method represented by this Method object, on the specified object with the specified parameters.
For static methods --1st arg can be null(its ignored anyway!)

—----------

Sequence Day15

1. Revise JDBC steps

2. Continue with Layered application, to complete CRUD n stored procedure

Layers involved
Tester(for UI n Managing DAO layer)
DAO layer (i/f n implementation class)
DB Utils
POJO/Model/Entity
DB layer

Dev steps
1.Create new java project with JDBC JAR
2.Table : users
3.pojos. User --obj representation of db data
4. DBUtils
4.1 public static Connection openConnection() throws SE
4.2 public static void closeConnection() throws SE
5. DAO layer
5.1 i/f : dao.UserDao
method declarations for CRUD ops
eg : Display user details for all users , under specific role , with reg amount > specified amount
i/p : role , amount
 List<User> findUsersByRoleAndAmount(String role,double amount) throws SE
5.2 DAO imple class
dao.UserDaoImpl
1. import pkg : java.sql
2. state : cn , psts(PreparedStatement)/csts(CallableStatement)
3. def ctor : called by tester --exactly once ---init phase of app
3.1 get db cn (calling DBUtils methods)
3.2 create PSTs/ CSTs
4. add a method to clean up DB resources ---called by tester --exactly once ---destroy phase of app
close PSTs/ CSTs
close cn. (DB Utils method)
5. CRUD method implementation/s
List<User>: findUsersByRoleAndAmount(String role,double amount) throws SE
empty List : users,
set IN params --exec query --RST --
while(rst.next())
 users.add(new User(.....));//ORM
ret users;
....

6. Tester
 main
try(sc) {
  create dao instance

```
  while(!exit)
  {
    try {
     switch-case ---invoke CRUD methods of the dao layer
      exit : clen up dao
    } catch-all -- sc.nextLine();
  }
} catch-all
```

Next Objective :
Un subscribe user.
i/p : user id

1. UserDao i/f
String unsubscribeUser(int userId) throws SE,custom exc

2. UserDaoImpl

Next Objective :
sign up : register new user

Next Objective :
change password
email , password, new password

Steps for invoking n executing a stored proc / function
0. Create stored proc/func in DB
1. Create CST(CallableStaement) from Connection
API of Connection i/f
CST prepareCall(String invocationSyntax) throws SE
invocationSyntax for a proc
"{call proc_name(?,?,.....?)}"

invocationSyntax for a func
"{?=call func_name(?,?,.....?)}"

{} => esc sequence meant for JDBC driver , to translate proc/func invocation as per native DB syntax.
2. Register OUT / IN OUT parameter before executing the stored proc/func
(can be done exactly once)
(Meaning : inform the JVM about JDBC data types of the OUT or IN OUT params)
API of CST
public void registerOutParameter(int paramIndex,int jdbcType) throws SQLException
For 2nd arg : jdbc type (Use java.sql.Types class constants)

3. Set IN params : must be called per request
Inherited from PST
public void setType(int paramIndex,Type val) throws SQLException

4. Execute stored proc/func
API of CST
public boolean execute() throws SE
ret value is not used here.

5. Get results stored in OUT / IN OUT params
API of CST
public Type getType(int paramIndex) throws SE

3. Discuss voting assignment(Optional lab work)

4. Reflection overview
Just supplied the introduction n overview with simple code example
Refer : to it's readme n readymade code samples
eclipse project : test_reflection

----------PENDING----------

5. Cloning in Java
—----------