DAY 1:

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

Operators in Java

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

Shift Operators

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

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

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

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

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

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

eg : int a=200;

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

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

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

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

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

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

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

eg :
```
int data=100;
int data2=50;
if(data > 60 && data2 < 100)
 System.out.println("test performed...");
```

```
else
 System.out.println("test not performed...");
```

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

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

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

Bitwise Operators : These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.
&, Bitwise AND operator: returns bit by bit AND of input values.
|, Bitwise OR operator: returns bit by bit OR of input values.
^, Bitwise XOR operator: returns bit by bit XOR of input values.
~, Bitwise Complement Operator: This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.

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


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

}

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

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

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

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

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

a = 0000 0000 0000 0000 0000 0000 0011 1100
b = 1111 1111 1111 1111 1111 1111 1100 0100
In Java, negative numbers are stored as 2's complement.

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

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

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

Java Vs C++
Development wise differences
1. Java is platform independent language but c++ is dependent upon operating system.

At compilation time Java Source code(.java) converts into byte code(.class) .The interpreter translates this byte code at run time into native code and gives output.

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

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

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

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

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

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

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

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

Features wise differences.

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

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

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

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

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

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

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

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

Java does not support default arguments.

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

There is no goto statement in Java.

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

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

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

Java is pass-by-value.

Java does not support unsigned integers.

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

Firstly, a copy constructor is nothing more than:

```
public class Blah {
  private int foo;

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

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

```
Blah b2 = b1;
```

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

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

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

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

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

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

Generally Java is more robust than C++ because:

Object handles (references) are automatically initialized to null.
Handles are checked before accessing, and exceptions are thrown in the event of problems.
You cannot access an array out of bounds.
Memory leaks are prevented by automatic garbage collection.
While C++ programmer clearly has more flexibility to create high efficient program, also more chance to encounter error.

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.

What are the rules for naming variables in java?

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

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

JVM has various sub components internally.

1. Class loader sub system: JVM's class loader sub system performs 3 tasks
    a. It loads .class file into memory.
    b. It verifies byte code instructions.
    c. It allots memory required for the program.

2. Run time data area: This is the memory resource used by JVM and it is divided into 5 parts
    a. Method area: Method area stores class code and method code. (metaspace in Java SE 8)
    b. Heap: Objects are created on heap.
    c. Java stacks: Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.
    d. Program counter registers: The program counter registers store memory address of the instruction to be executed by the micro processor.
    e. Native method stacks: The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

3. Native method interface: Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

4. Native method library: holds the native libraries information.

5. Execution engine: Execution engine contains interpreter and JIT compiler, which converts byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler.

JVM architecture (refer to a diagram)
There are mainly three sub systems in the JVM

1. ClassLoader
2. Runtime Memory/Data Areas
3. Execution Engine

1. ClassLoader
This component is responsible for loading the class files to the method area (typcially in RAM) since JVM resides on the RAM and it performs : loading, linking, and initialization.

1.1 Loading

This process usually starts with loading the main class (class with the main()method). ClassLoader reads the .class file and then the JVM stores the following information in the method area.
1. The fully qualified name of the loaded class
2. variable information
3. immediate parent information
4. Type : whether it is a class or interface or enum

Note —Only for the first time, JVM creates an object from a class type object(java.lang.Class) for each loaded java class and stores that object in the heap.

The three main ClassLoaders in JVM,
1. Bootstrap ClassLoader — This is the root class loader and it is the superclass of Extension/Platform ClassLoader. This loads the standard java packages which are inside the rt.jar/jrtfs.jar : java.base module (base file and some other core libraries.)
eg : java.util.ArrayList

2. Extension/Platform ClassLoader — This is the subclass of the Bootstrap ClassLoader and a superclass of Applications ClassLoader. This is responsible for loading classes that are present inside the directory (jre/lib/ext). From JDK 9 onwards , it's replace by Platform class loader . Loads additional classes from other modules eg : java.sql

3. Application ClassLoader — This is the subclass of Extension/Platform ClassLoader and this is responsible for loading the class files from the classpath (classpath can be modified by adding the -classpath command-line option)
eg : tester.TestMe

The four main principles in JVM,
1. Visibility Principle — This principle states that the ClassLoader of a child can see the class loaded by Parent, but a ClassLoader of parent can’t find the class loaded by Child.
2. Uniqueness Principle — This principle states that a class loaded by the parent ClassLoader shouldn’t be loaded by the child again. This ensures that there is no class duplicated.

3. Delegation Hierarchy Principle — This rule states that JVM follows a hierarchy of delegation to choose the class loader for each class loading request. Here, starting from the lowest child level, Application ClassLoader delegates the received class loading request to Extension ClassLoader, and then Extension ClassLoader delegates the request to Bootstrap ClassLoader. If the requested class is found in the Bootstrap path, the class is loaded. Otherwise, the request again transfers back to the Extension ClassLoader level to find the class from the Extension path or custom-specified path. If it also fails, the request comes back to Application ClassLoader to find the class from the System classpath and if Application ClassLoader also fails to load the requested class, then we get the run time exception — ClassNotFoundException.
4. No Unloading Principle — This states that a class cannot be unloaded by the Classloader even though it can load a class.

2. Linking
This process is divided into three main parts .
1. Verification
This phase check the correctness of the .class file.
Byte code verifier will check the following:
1.1 whether it is coming from a valid compiler or not (Because anyone can create their own compiler).
1.2 whether the code has a correct structure and format.

if any of these are missing, JVM will throw a runtime exception called "java.lang.VerifyError" Exception. if not, then the preparation process will take place.

2. Preparation
In this phase, variables memory will be allocated for all static data members and assigned with default values based on the data types.
eg : reference — null
int — 0
boolean— false
eg :
static boolean active=true;
So in this phase, it will check the code and the variable status in boolean type so JVM assigns false to that variable.

3. Resolution
This is the process of replacing the symbolic references with direct references and it is done by searching into the method area to locate the referenced entity.

The JVM does not understand the name that we give to create objects(reference variables) . So the JVM will assign memory location for those objects by replacing their symbolic links with direct links.

3. Initialization
In this phase, the original values will be assigned back to the static variables as mentioned in the code and a static block will be executed(if any). The execution takes place from top to bottom in a class and from parent to child in the class hierarchy.

Important : JVM has a rule saying that the initialization process must be done before a class becomes an active use.

Active use of a class are,
1. using new keyword. (eg: Vehicle car=new Vehicle();).
2. invoking a static method.
3. assigning value to a static field.
4. if a class is an initial class (class with main()method).
5. using a reflection API (Class's newInstance()method).
6. initializing a subclass from the current class.

There are four ways of initializing a class :
using new keyword — this will goes through the initialization process.
using clone(); method — this will get the information from the parent object (source object).
using reflection API (newInstance();) — this will goes through the initialization process.
using IO.ObjectInputStream(); — this will assign initial value from InputStream to all non-transient variable

Runtime Data Area
JVM memory is basically divided into five following parts,

1. Method Area
This is where the class data is stored during the execution of the code and this holds the information of static variables, static methods, static blocks, instance methods, class name, and immediate parent class name(if any). This is a shared resource.

2. Heap Area
This is where the information of all objects(state: non static data members) is stored and it’s a shared resource just like the method area

eg : Book book = new Book(...);

So here, there is an instance of Book is created and it will be loaded into the Heap Area preceded by Book's class information loaded in the method area + creation of Class<Book> instance , in the heap.

Note — there is only one method area and one heap area per JVM.

3. Stack Area
All the local variables, method calls, and partial results of a program (not a native method) are stored in the stack area.

For every thread, a runtime stack will be created. A block of the stack area is known as “Stack Frame” and it holds the local variables of method calls. So whenever the method invocation is completed, the frame will be removed (POP). Since this is a stack, it uses a Last-In-First-Out structure.

4. PC Register (Program Counter Register)
This will hold the thread’s executing information. Each thread has its own PC registers to hold the address of the current executing information and it will be updated with the next execution once the current execution finishes.

5. Native Method Area
This will hold the information about the native methods and these methods are written in a language other than Java, such as C/C++. Just like stack and PC register, a separate native method stack will be created for every new thread.
Take a look at the following diagram,

eg scenario for Thread 1 (T1),

```
M1(){
  M2();
}
-----------------
 M2(){
     M3();
 }
```

When the M1 method is called, the first frame will be created in the T1 thread and from there it will go to method M2 at that time the second frame will be created and from there it will go to method M3 as in the above demo code, so a new frame will be created under M2. Whenever the method exits, the stack frames will be destroyed respectively.

3. Execution Engine

This is where the execution of bytecode (.class) occurs and it executes the bytecode line-by-line. Before running the program, the bytecode should be converted into machine code. l

Mainly, Execution Engine has three main components for executing the Java classes,

Components of Execution Engine
3.1 Interpreter
This is responsible for converting bytecode into machine code. This is slow because of the line-by-line execution even though this interprets the bytecode quickly. The main disadvantage of Interpreter is that when the same method is called multiple times, every time a new interpretation is required and this will reduce the performance of the system. So this is the reason where the JIT compiler will run parallel to the Interpreter.

3.2 JIT Compiler (Just In Time Compiler)
This overcomes the disadvantage of the interpreter. The execution engine first uses the interpreter to execute the bytecode line-by-line and it will use the JIT compiler when it finds some repeated code. (Eg: calling the same method multiple times). At that time JIT compiler compiles the entire bytecode into native code (machine code). These native codes will be stored in the cache. So whenever the repeated method is called, this will provide the native code. Since the execution with the native code is quicker than interpreting the instruction, the performance will be improved.

3.3 Garbage Collector
This will check the heap area whether there are any unreferenced objects and it destroys those objects to reclaim the memory. So it makes space for new objects. This runs in the background and it makes the Java memory efficient.

There are two phases involved in this process,
Mark — In this area, Garbage Collector identifies the unsued objects in the heap area.
Sweep — In here, Garbage Collector removes the objects from the Mark.
This process is done by JVM at regular intervals and it can also be triggered by calling System.gc() method.

3.4 Java Native Interface (JNI)
This is used to interact with the Native(non-java) Method libraries (C/C++) required for the execution. This will allows JVM to call those libraries to overcome the performance constraints and memory management in Java.

3.5 Native Method Libraries
These are the libraries that are written in other programming(non-java) languages such as C and C++ which are required by the Execution Engine. This can be accessed through the JNI and these library collections mostly in the form of .dll or .so file extension.

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)

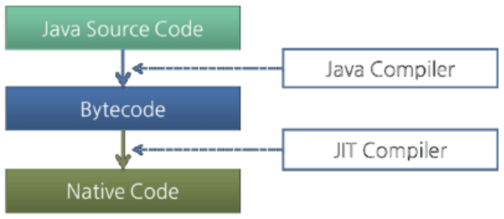
scope -- as per access specifier
lifetime -- class loading --till unloading(@ jvm shut down)
Kept on method area
by def -- inited

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);}



Java Data Types

primitive types (value holding type)

non-numeric

boolean

numeric types

char -- character
UTF 16
un signed

Integral Types

byte (1)   short (2)   int (4)   long (8)

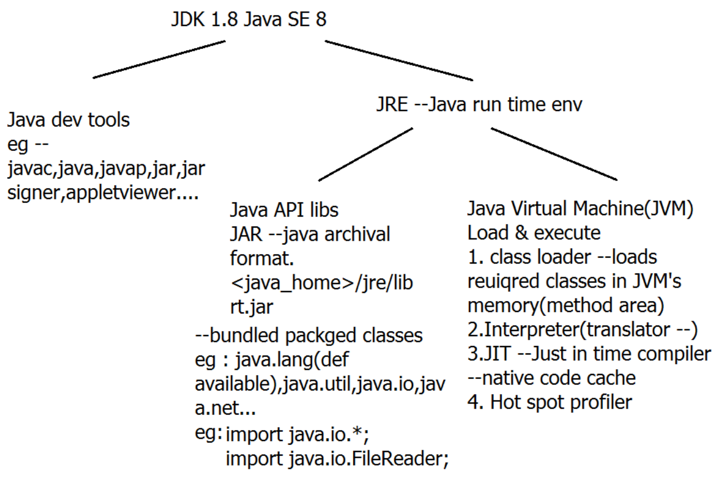
Floating Types

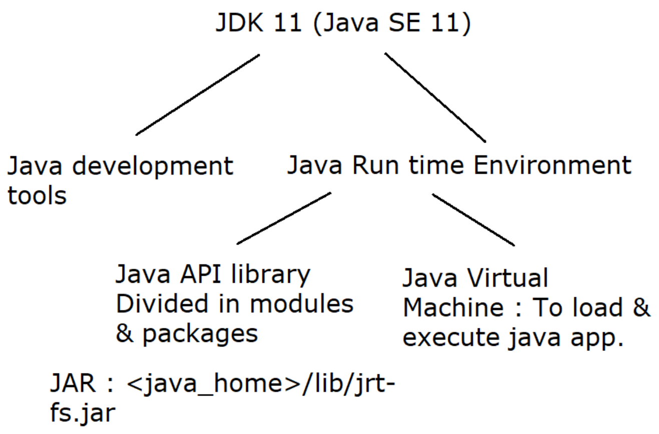
float (4)   double (8)

signed data types
2's complement format
(IEEE 754)

reference types (holding address of the object)

class type   array type   interface type





<u>DAY 2:</u>

Today's topics

JVM Architecture

Enter OOPs -- encapsulation , data hiding

(achieved via class programming)

GC

Packages , classpath & access sepcifiers

Revise

Java Data Types : which 2 categories : primitive type : value holder

non numeric n numeric types

reference types : address of the object dynamically created on the heap

class type ref , array type ref , interface type ref

Types of conversion in prim types

auto promotion(widening) : javac : implicit

eg : long --> float ,int -->long , float --> double

explict narrowing conversion : type casting -- prog

eg : double --> float ,int--> byte , float --> long

solve :

byte b1=10;

int data=b1;//widening : no err

float f=23.45;//javac err
double d1=f;//widening : no err
long l1=454654;//widening : no err
f=l1;//widening : no err

Answer this
What will data type of result of
1. byte & byte : int
2. byte & short : int
3. short & short : int
4. int & long : long
5. long & float : float
6. float & double : double
7. byte/short/int....& double : double

JVM Architecture

1. Class loader : lazy loading policy(on demand) to load required classes in JVM's memory(meta space)
2. Run time data areas
2.1 Method area : metaspace, 1 single copy per JVM , byte code information(class info), static data memebrs
2.2 Heap : 1 single copy per JVM , dynamically created objects(state : non static data members)

2.3 Stacks : 1 per thread.
System threads : main thrd --looks for main(...) & execs sequentially
GC

main thrd's stack : stack frames (1 per method call)
stack frm : method local information : method args , local vars, ret values
main(..) ---> m1() --> m2()

stack : ??

Enter OOP
refer to "regarding class n object"

Encapsulation -- Class Programming (refer to

class,object,state,behaviour
Objective --
Create a class to represent 3D Box
class Box ---tight encapsulation(achieved by making all non static data members:instance vars private)
state -- width,height , depth --- double : private
paramterized constr --3 arg constr.
Instance Methods : for supplying Business logic(B.L)
1. To return Box details in String form (dimensions of Box)
eg : String getBoxDetails()
method declaration --- access specifier(private/default/protected/public) , ret type , name , args
method definition --body

2. To return computed volume of the box.

this --keyword in java
Usages
1. To un-hide instance variable from local variable.
eg : this.width=width;

2. To invoke constructor of the same class , from different constructor.(constr chaining)
eg : this(side,side,side);

Create another class : TestBox --for UI.
Add psvmain(..){...}
Use scanner --to accept box dims from user, create box instance.
display its details & volume.

Pointers vs java references
Similarity -- Pointer & reference --hold an address of the object created on heap.
Difference -- To add robustness to the language ,
pointer arithmatic is not allowed in java.

reference --- holds internal representation of address --

eg :
Box b1=new Box(1,2,3);
b1++;//javac err
b1 += 10; //javac err

String s=new String("hello");
s ++;//javac err
s += 10;//s = s + 10;no javac err

Add more business logic in the Box class

1. Create Cubes
sop("Enter side of a cube");
Box cube=new Box(sc.nextDouble());//javac err

Soln : constructor overloading (constr chaining)

1.5 Create a box with dims = -1,-1 , -1

JVM architecture (refer to a diagram)
There are mainly three sub systems in the JVM
1. ClassLoader
2. Runtime Memory/Data Areas
3. Execution Engine

1. ClassLoader
This component is responsible for loading the class files to the method area (typcially in RAM) since JVM resides on the RAM and it performs : loading, linking, and initialization.

1.1 Loading

This process usually starts with loading the main class (class with the main()method).
ClassLoader reads the .class file and then the JVM stores the following information in the method area.
1. The fully qualified name of the loaded class
2. variable information
3. immediate parent information
4. Type : whether it is a class or interface or enum

Note —Only for the first time, JVM creates an object from a class type object(java.lang.Class) for each loaded java class and stores that object in the heap.

The three main ClassLoaders in JVM,
1. Bootstrap ClassLoader — This is the root class loader and it is the superclass of Extension/Platform ClassLoader. This loads the standard java packages which are inside the rt.jar/jrtfs.jar : java.base module (base file and some other core libraries.)

eg : java.util.ArrayList

2. Extension/Platform ClassLoader — This is the subclass of the Bootstrap ClassLoader and a superclass of Applications ClassLoader. This is responsible for loading classes that are present inside the directory (jre/lib/ext). From JDK 9 onwards , it's replace by Platform class loader . Loads additional classes from other modules eg : java.sql

3. Application ClassLoader — This is the subclass of Extension/Platform ClassLoader and this is responsible for loading the class files from the classpath (classpath can be modified by adding the -classpath command-line option)
eg : tester.TestMe

The four main principles in JVM,
1. Visibility Principle — This principle states that the ClassLoader of a child can see the class loaded by Parent, but a ClassLoader of parent can't find the class loaded by Child.
2. Uniqueness Principle — This principle states that a class loaded by the parent ClassLoader shouldn't be loaded by the child again. This ensures that there is no class duplicated.
3. Delegation Hierarchy Principle — This rule states that JVM follows a hierarchy of delegation to choose the class loader for each class loading request. Here, starting from the lowest child level, Application ClassLoader delegates the received class loading request to Extension ClassLoader, and then Extension ClassLoader delegates the request to Bootstrap ClassLoader. If the requested class is found in the Bootstrap path, the class is loaded. Otherwise, the request again transfers back to the Extension ClassLoader level to find the class from the Extension path or custom-specified path. If it also fails, the request comes back to Application ClassLoader to find the class from the System classpath and if Application ClassLoader also fails to load the requested class, then we get the run time exception — ClassNotFoundException.
4. No Unloading Principle — This states that a class cannot be unloaded by the Classloader even though it can load a class.

2. Linking
This process is divided into three main parts .
1. Verification
This phase check the correctness of the .class file.
Byte code verifier will check the following:
1.1 whether it is coming from a valid compiler or not (Because anyone can create their own compiler).
1.2 whether the code has a correct structure and format.

if any of these are missing, JVM will throw a runtime exception called "java.lang.VerifyError" Exception. if not, then the preparation process will take place.

2. Preparation

In this phase, variables memory will be allocated for all static data members and assigned with default values based on the data types.
eg : reference — null
int — 0
boolean— false
eg :
static boolean active=true;
So in this phase, it will check the code and the variable status in boolean type so JVM assigns false to that variable.

3. Resolution
This is the process of replacing the symbolic references with direct references and it is done by searching into the method area to locate the referenced entity.

The JVM does not understand the name that we give to create objects(reference variables) .
So the JVM will assign memory location for those objects by replacing their symbolic links with direct links.
eg : In TestBox class
Box b1=new Box(1,2,3);
b1.showDims();

3. Initialization
In this phase, the original values will be assigned back to the static variables as mentioned in the code and a static initilizer block will be executed(if any). The execution takes place from top to bottom in a class and from parent to child in the class hierarchy.

Important : JVM has a rule saying that the initialization process must be done before a class becomes an active use.

Active use of a class are,
1. using new keyword. (eg: Vehicle car=new Vehicle();).
2. invoking a static method.
3. assigning value to a static field.
4. if a class is an initial class (class with main()method).
5. using a reflection API (Class's newInstance()method).
6. initializing a subclass from the current class.

There are four ways of initializing a class :
using new keyword — this will goes through the initialization process.
using clone(); method — this will get the information from the parent object (source object).
using reflection API (newInstance();) — this will goes through the initialization process.

using IO.ObjectInputStream(); — this will assign initial value from InputStream to all non-transient variable

Runtime Data Area
JVM memory is basically divided into five following parts,

1. Method Area
This is where the class data is stored during the execution of the code and this holds the information of static variables, static methods, static blocks, instance methods, class name, and immediate parent class name(if any). This is a shared resource.

2. Heap Area
This is where the information of all objects(state: non static data members) is stored and it's a shared resource just like the method area

eg : Book book = new Book(...);
So here, there is an instance of Book is created and it will be loaded into the Heap Area preceded by Book's class information loaded in the method area + creation of Class<Book> instance , in the heap.

Note — there is only one method area and one heap area per JVM.

3. Stack Area
All the local variables, method calls, and partial results of a program (not a native method) are stored in the stack area.

For every thread, a runtime stack will be created. A block of the stack area is known as "Stack Frame" and it holds the local variables of method calls. So whenever the method invocation is completed, the frame will be removed (POP). Since this is a stack, it uses a Last-In-First-Out structure.

4. PC Register (Program Counter Register)
This will hold the thread's executing information. Each thread has its own PC registers to hold the address of the current executing information and it will be updated with the next execution once the current execution finishes.

5. Native Method Area
This will hold the information about the native methods and these methods are written in a language other than Java, such as C/C++. Just like stack and PC register, a separate native method stack will be created for every new thread.

Take a look at the following diagram,

eg scenario for Thread 1 (T1),

```
M1(){
  M2();
}
----------------
 M2(){
     M3();
 }
```

When the M1 method is called, the first frame will be created in the T1 thread and from there it will go to method M2 at that time the second frame will be created and from there it will go to method M3 as in the above demo code, so a new frame will be created under M2. Whenever the method exits, the stack frames will be destroyed respectively.

3. Execution Engine

This is where the execution of bytecode (.class) occurs and it executes the bytecode line-by-line. Before running the program, the bytecode should be converted into machine code. l

Mainly, Execution Engine has three main components for executing the Java classes,

Components of Execution Engine

3.1 Interpreter

This is responsible for converting bytecode into machine code. This is slow because of the line-by-line execution even though this interprets the bytecode quickly. The main disadvantage of Interpreter is that when the same method is called multiple times, every time a new interpretation is required and this will reduce the performance of the system. So this is the reason where the JIT compiler will run parallel to the Interpreter.

3.2 JIT Compiler (Just In Time Compiler)

This overcomes the disadvantage of the interpreter. The execution engine first uses the interpreter to execute the bytecode line-by-line and it will use the JIT compiler when it finds some repeated code. (Eg: calling the same method multiple times). At that time JIT compiler compiles the entire bytecode into native code (machine code). These native codes will be stored in the cache. So whenever the repeated method is called, this will provide the native code. Since the execution with the native code is quicker than interpreting the instruction, the performance will be improved.

3.3 Garbage Collector

This will check the heap area whether there are any unreferenced objects and it destroys those objects to reclaim the memory. So it makes space for new objects. This runs in the background and it makes the Java memory efficient.

There are two phases involved in this process,
Mark — In this area, Garbage Collector identifies the unsued objects in the heap area.
Sweep — In here, Garbage Collector removes the objects from the Mark.
This process is done by JVM at regular intervals and it can also be triggered by calling System.gc() method.

3.4 Java Native Interface (JNI)
This is used to interact with the Native(non-java) Method libraries (C/C++) required for the execution. This will allows JVM to call those libraries to overcome the performance constraints and memory management in Java.

3.5 Native Method Libraries
These are the libraries that are written in other programming(non-java) languages such as C and C++ which are required by the Execution Engine. This can be accessed through the JNI and these library collections mostly in the form of .dll or .so file extension.

What is JIT Compiler?

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

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

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

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

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

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

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

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

Regarding garbage collection

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

Garbage= un -referencable object.

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

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

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

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

Whenever Heap is under pressure (i.e there is no space in heap to create more objects) , JVM activates the GC thread. BUT it refers to Heap only. The classes which are loaded in method area are unloaded : when JVM terminates.

----------------------1st half over----------------

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

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

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

More Details

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

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

How Garbage Collector works

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

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

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

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

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

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

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

Generational Process of Garbage Collection

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

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

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

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

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

Class & Object

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

Class

A class is a user defined blueprint or prototype or template : from which objects are created. It represents the set of properties(DATA) and methods(ACTIONs) 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 state of the new object/s.
7. Fields are variables that provides the state of the class and its objects
8. Methods are used to implement the behavior of the class and its objects.

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

Object

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

An object consists of :

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

Creating an object

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

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

Parameterized constructor is used initialize state of the object.

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

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

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

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

Encapsulation in Java

Encapsulation is defined as the wrapping up of data & code under a single unit. It is the mechanism that binds together code and the data it manipulates.

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

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

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

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

Advantages of Encapsulation:

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

Summary

Encapsulation -- consists of Data hiding + Abstraction

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

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

Day 3:

Today's Topics
Packages & classpath
Acces Specifiers
Enter Eclipse IDE
Arrays
static keyword

Revision

Revise class , object & memory picture

Q 1. What will happen ?

```
Box b1;//as per JVM spec 4/8/16 to store ref type of var : on the stack(local var)
b1=new Box(1,2,3);//BOx class info will be loaded : metspace(method area), instance --on
heap
mem allocated for state : non static data member
//obj : CP -->loaded cls info , width,depth height , color...
s.o.p(b1.getBoxDimensions());//b1---> obj --CP --> method area
s.o.p(b1.getBoxVolume());//6.0
Box b2=b1;//how many objs for GC : 0, b1 n b2 ---> 1st box instance
s.o.p(b2.getBoxDimensions());//1.0 2.0 3.0
b1=null;//b1 is de refed, , how many objs for GC :0
//s.o.p(b1.getBoxDimensions());//run time err : java.lang.NullPointerException
s.o.p(b2.getBoxDimensions());//1.0 2.0 3.0
b2=null;//how many objs for GC : 1
s.o.p(b2.getBoxDimensions());//run time err : java.lang.NullPointerException
```

Q 2. What will happen ?

```
Box b1=new Box(1,2,3);//b1 ---> Box : 1.0,2.0,3.0
b1=new Box(2,3,4);//re assigning the ref, b1 ---> Box : 2.0,3.0,4.0
```

Any objects marked for GC ? YEs : 1 , 1st box obj marked for GC

Q 3. What will happen?

```
Box b1=new Box(10,20,30);//b1 --> Box 10,20,30
b1.testMe();
.....
```

In Box class :

```
void testMe()
{
  Box tmp=new Box(10,20,30);//tmp --> Box : 10,20,30
}//testMe's stack frm popped out --local var "tmp" : not avlble
```

Any objects marked for GC , after testMe method's execution ? : YES : 1 obj created in the method n it's ref not reted to the caller.

Q 4 What will happen?

```
Box b1=new Box(10,20,30);//b1 --> Box 10,20,30
Box b2=b1.testMe();//b2=tmp : copy of refs
.....
```

In Box class :

```
Box testMe()
{
  Box tmp=new Box(10,20,30);//tmp --> Box : 10,20,30
  return tmp;
}
```
Any objects marked for GC , after testMe method's execution ? NO

Confirm understanding of copy of references using Object's hashCode method.
----

1. Packages & access specifiers
refer to "regarding packages"
Objective : Place Box class under package "com.cdac.core" n place TestBox under "com.tester"

2. refer to the diagram "Access specifiers" & "Understanding access specifiers"
Set it up & confirm the table.

3. Enter Eclipse IDE

4. Regarding Arrays

In Java, arrays are full-fledged objects. Like any other objects, arrays are dynamically created & stored on the heap.

Arrays (like any other object) are associated with the class. All arrays of the same dimension and type have the same class. The length of an array does not play any role in establishing the array's class. For example, an array of three ints has the same class as an array of three hundred ints.(Name of the class loaded for this : [I) The length of an array is considered part of its instance data.

eg : The class name for single dimension array of ints is "[I". The class name for a three-dimensional array of bytes is "[[[B". The class name for single dimension array of booleans is "[Z"

Array size(length) is fixed.Implicit super class of array is java.lang.Object.

4.1 Create Array of primitive types

Objective -- Accept no of data samples(of type double) from User(using scanner)
Create suitable array & display it using for-each loop , to confirm default values.

Accept data from User(scanner) & store it in the array.
Display array data using for loop.
Display array data using for-each loop

for-each loop(enhanced for loop)
syntax

Important statement
Works on a copy of array element.

How will you confirm ????????

for-each limitations
1. Can only iterate from 1st elem to last elem , with step size +1
2. Works on a copy of array elems
(can't be used for modifying array elems).

4.2 Array of references : IMPORTANT

Add Box class into "com.cdac.core"
Create a tester class : TestBoxArray : "com.tester"

Objective : Ask user(client) , how many boxes to make ?
Accept Box dims for these boxes.
Store these details suitably.

1. Display using single for-each loop, box dims n volume

JVM has various sub components internally.

1. Class loader sub system: JVM's class loader sub system performs 3 tasks
    a. It loads .class file into memory.
    b. It verifies byte code instructions.
    c. It allots memory required for the program.

2. Run time data area: This is the memory resource used by JVM and it is divided into 5 parts
    a. Method area: Method area stores class code and method code. (metaspace in Java SE 8)
    b. Heap: Objects are created on heap.
    c. Java stacks: Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.

d. Program counter registers: The program counter registers store memory address of the instruction to be executed by the micro processor.

e. Native method stacks: The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

3. Native method interface: Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

4. Native method library: holds the native libraries information.

5. Execution engine: Execution engine contains interpreter and JIT compiler, which covers byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler.

Regarding Packages

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

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

About Packages
1. Creation : package statement has to be placed as the 1st statement in Java source.
eg : package p1; => the classes will be part of package p1.
2.Package names are mapped to folder names.
eg : package p1; class A{....}
A.class must exist in folder p1.
3. For simplicity --- create folder p1 -- under <src> & compile from <src>
From <src>
```
javac -d ..\bin p1\A.java
```

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

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

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

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

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

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

What should be value of classpath ---Must be set to top of packged class hierarchy(eg : bin)
How to set classpath ?
Option 1.
java -cp "D:\ACTS-sep21\day3.1\bin" com.tester.TestBoxFunctionality

OR

Option 2
On cmd prompt : set classpath="D:\sunbeam-sep21\code\day3.1\bin";.;

Option 3
set it from environment variables.
Create a new environment variable :\
CLASSPATH n set its value to : "D:\sunbeam-sep21\code\day3.1\bin";.;

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

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

the first line in the source code file.

3. import and package statements apply to all classes within a source code file.
In other words, there's no way to declare multiple classes in a file and have
them in different packages, or use different imports.

---

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

```
eg -- public class Student {
private int rollNo;..........
public static int idCounter;
public Student(String name.....)
 {
   this.rollNo=++idCounter;
   this.name=name;
 }
}
```

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

Rules -- 1. Can static methods access other static members directly(w/o instance) -- YES
2. Can static methods access other non-static members directly(w/o instance) -- NO

```
eg : class A
{
  private int i;
  private static int j;
  public static void show()
  {

     sop(i);//javac err
     sop(j);//no err
  }
}
```

3. Can non-static methods access other static members directly(w/o instance) -- YES
eg :
In Test class

```
void test1() {test2();}//no error
```

OR
static void test2(){test1();//javac error}

3. static import --- Can directly use all static members from the specified class.
eg --

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

}
```

4. static initializer block
syntax --

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

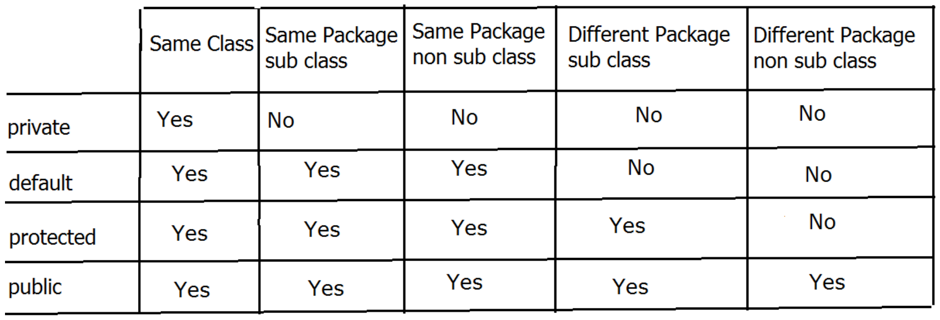
They appear -- within class definition & can access only static members directly.(w/o instance)
A class can have multiple static init blocks(legal BUT not recommended)

Regarding non-static initilizer blocks(instance initilaizer block)
syntax

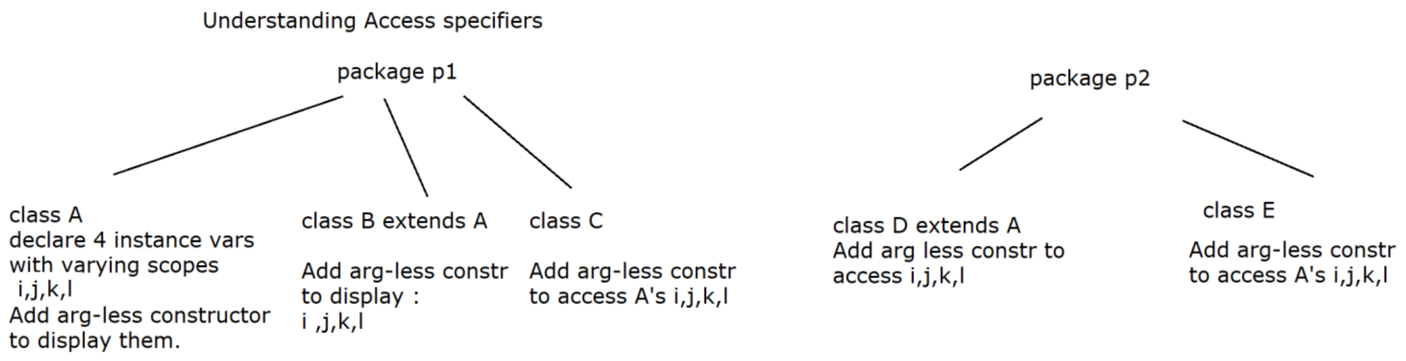
```
{
//will be called per instantiation --- before matching constructor
//Better alternative --- parameterized constructor.
}
```

5. static nested classes ---
eg --

```
class Outer {
// static & non-static members
  static class Nested
  {
    //can access ONLY static members of the outer class DIRECTLY(w/o inst)
  }
}
```



	Same Class	Same Package sub class	Same Package non sub class	Different Package sub class	Different Package non sub class
private	Yes	No	No	No	No
default	Yes	Yes	Yes	No	No
protected	Yes	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes	Yes



class A
declare 4 instance vars
with varying scopes
i,j,k,l
Add arg-less constructor
to display them.

class B extends A
Add arg-less constr
to display :
i ,j,k,l

class C
Add arg-less constr
to access A's i,j,k,l

class D extends A
Add arg less constr to
access i,j,k,l

class E
Add arg-less constr
to access A's i,j,k,l

DAY 4:
Today's Topics

Revise with mem pics : array of references
static keyword
Enter Inheritance & later Polymorphism.

Revise
0. Add isEqual method to Point2D class :a boolean returning method : must return true if n only if both points are having same x,y co-ords or false otherwise.
eg : In TestPoint
Point2D p1=new Point2D(sc.nextInt(),sc.nextInt());//10 20
Point2D p2=new Point2D(sc.nextInt(),sc.nextInt());//10 40
//non static method added in the point class OR with static method : either in Point class or a separate Utils class
sop("Points are "+(p1.isEqual(p2)?"SAME":"DIFFERENT"));

Method in Point2D class

```
public class Point2D {
.....
//isEqual :
public boolean isEqual(Point2D anotherPoint)
{
//this =p1 , anotherPoint=p2
   return this.x == anotherPoint.x && this.y == anotherPoint.y;
}
}
```

1. What will be the memory pic n o/p ?

```
Point2D[] points;
points=new Point2D[3];

int value=10;
for(int i=0;i<points.length;i++) {
 points[i]=new Point2D(value,value+10);
 value++;
}
for(Point2D p : points)
 sop(p.getDetails());
```

2. What will happen ?

```
Point2D[] points=new Point2D[3];
int value=10;
for(Point2D p : points) {
 p=new Point2D(value,value+10);
 value++;
}
for(Point2D p : points)
 sop(p.getDetails()); //NPExc
```

3. Dynamic initialization of array of primitive types

```
int[] ints= {10,20,30,40,50};
```

mem pic pls !

4. Dynamic initialization of array of references

```
Point2D[] points={new Point2D(10,20),new Point2D(50,100),new Point2D(-50,70)};
```

mem pic pls !

5. static keyword
refer to a readme "regarding static"

Q : Emp class
Given : class Emp

```
{
 private int id;
 private String name;
 private double salary;
 private static int noOfEmps;
}
```

If 10 emps are hired(i.e 10 emp objects are created , how many copies of the following will be created ?)

id : 10
name : 10
salary : 10
noOfEmps : 1

Solve this : Add a static method isEqual method to Point2D class :a boolean returning method : must return true if n only if both points are having same x,y co-ords or false otherwise.

```
public static boolean isEqual(Point2D point1,Point2D point2)
{
//point1=p1 , point2=p2
  return point1.x==point2.x && .....;
}
```

How will u invoke above method from Tester ?
p1 , p2
Point2D.isEqual(p1,p2);

6. Understand static initializer block
solve ready code sample

7. Inheritance
refer to "readme inheritance n polymorphism.txt" from today's help.

4.1 What is inheritance n Why
4.2 Real life Examples
4.3 Types
4.4 constr invocation in inheritance hierarchy
super keyword
Person-Student-Faculty scenario (constructor invocation in inheritance hierarchy)
4.5 Show details : via toString : method overriding

Objective : Arrange an event to invite students n faculties

(eg : EventOrganizer app : tester --main / scanner)
Prompt user for event capacity.
Create suitable data structure to hold the participant details
Options
1. Register Student :
2. Register Faculty :
3. Display participant details : for-each
10. Exit

Regarding inheritance

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 -- code 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(behaviour --- method overriding)

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

Fruit -- Apple -- FujiApple

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

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

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

2. Multi level inhertance ---Supported in Java
eg : class A{...} class B extends A{...} class C extends B{...}

3. Hierarchical inheritance ---Supported in Java
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{...} --provided B & C : classes -- compiler err

Why --For simplicity.

(Diamond problem)

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

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

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

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

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

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

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

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

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

Can't add super class of the checked excs.

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() {....}
}
}
```

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

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

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

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

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

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

Can a class be declared as abstract & final ? NO

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

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

final -- keyword in java

Usages

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

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

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

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

Special note on protected

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

Do subclasses inherit private data members from it's superclass?

NO !
Explanation :

As per the java language specification :

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

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

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

eg : Person has data members : private firstName , lastName

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

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

Student IS-A Person

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

So how many slots will you show in Student object ?
CP + 6 slots .

---

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

eg -- public static int idCounter;

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

Rules -- 1. Can static methods access other static members directly(w/o instance) -- YES
2. Can static methods access other non-static members directly(w/o instance) -- NO
eg : class A

```
{
  private int i;
  private static int j;
  public static void show()
  {

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

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

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

They appear -- within class definition & can access only static members directly.(w/o instance)

A class can have multiple static init blocks(legal BUT not recommended)

Regarding non-static initilizer blocks(instance initilaizer block)
syntax

```
{
//will be called per instantiation --- before matching constructor
//Better alternative --- parameterized constructor.
}
```

5. static nested classes ---
eg --

```
class Outer {
// static & non-static members
  static class Nested
  {
    //can access ONLY static members of the outer class DIRECTLY(w/o inst)
  }
}
```

DAY 5:

Today's topics
Continue with inheritance n Polymorphism

Revise
What is inheritance ? IS A , moving from generalization ---> specialization
Why Inheritance : reducing redundancy n increasing re usability
Types : single , multi level , hybrid
Java follows singl root inh hierarchy
Universal super class : java.lang.Object

Inheritance type un supported : multiple inheritance
WHY ? ambiguity

Discuss Diamond problem

constr invocation in inheritance hierarchy : refer to diag.

super --keyword in java

super keyword usages
1. To invoke immediate super cls's MATCHING constr (eg : super(firstName,lastName);) : accessible from sub class constructor

2. To access super cls's accessible members (eg : super.toString())

Important statements
Sub class IS A super class + something more(addtional data members + methods) + something modified(behaviour : methods : method overriding)

java.lang.Object : universal super class of all java classes
public String toString() : what does it return : F.Q className @ hex hashCode
What's the need of overriding toString : to replace "F.Q className @ hex hashCode" by actual state of the object (non static data members=instance vars)

eg :
```
package bank;
public class Account
{
// state : acctNo , name, balance,type : instance vars (=non static data members)
//constr : 4 args
public String toString()
{
//   sop(super);//compile time err
sop(super.toString());//no err : hashCode o/p
  return "Account No "+acctNo"+" balance ="+balance;
}

}
```
In Tester
Account a1=new Account(...);
sop(a1.toString());//what will be o/p ? : uses inherited form from Object : bank.Account@6357567
sop(a1);//what will be o/p n why ? : same o/p as earlier , WHY : PrintStream : println(Object ref) -->
String.valueOf(Object ref) ---> ref==null ? prints null : ref.toString

Is it suitable , if not what is a solution ? : method overriding

What will happen , if you write below code in toString() method of Account class ?
sop(this); : stack overflow

sop(super);
----
1. Revise Person,Student, Faculty , inheritance hierarchy . Complete Faculty class

Objective : Arrange an event to invite students n faculties
(eg : EventOrganizer app : tester --main / scanner)
Prompt user for event capacity.
eg : sop("Enter event capacity");
Create suitable data structure to hold the participant details
Person[] participants=new Person[sc.nextInt()];//50
boolean exit=false;
int counter=0;
while(!exit) {
switch-case
Options
1. Register Student : counter < length
sop("Enter student dtls : fn ln ......");
participants[counter++]=new Student(sc.next(),.........);//javac err : NO ERR : impl conversion done by javac : upcasting

2. Register Faculty :
3. Display participant details : for-each
10. Exit : exit=true

}

1. Upcasting (refer to readme)

IMPORTANT : Javac resolves method binding by type of the reference BUT JVM resolves it by type of the instance it's referring to .

Solve :
Fruit : super class
public String taste()
{
 return "No specific taste";
}
Mango extends Fruit
{
 override : to return "sweet"
}
Orange extends Fruit
{

```
 override : to return "sour"
+ additional func
public void juice()
{
  sop("extracting orange juice");
}
}
Alphonso extends Mango
{
   override : to return"sweet n juicy"
}
Apple extends Fruit
{
   override : to return"sweet n sour"
   //jam
   public void jam()
   {
      sop("creating apple jam...");
   }
}
```

Solve
What will happen ?
Fruit f=new Mango();//no javac err : up casting
f.taste();//javac err : since Javac chks it by type the ref : i.e it will chk if taste() exists in Fruit class
/After adding taste() as a common func, in Fruit class : no javac err. o/p sweet : run time poly.

f=new Orange(...);
f.taste();

f=new Alphonso();
f.taste();

Object o=f;//up casting
sop(o);//hashcode
o.taste();//javac err : since taste() : doesn't exist in Object class
((Alphonso) o).taste();//no javac err ---no runtime err ---> sweet n juicy : downcasting

Solve :

```
Fruit f=new Orange();
f.taste();//sour
((Orange)f).juice();//no javac err, no runtime err , run time poly  : JVM calls : juice : Orange
f=new Apple();//no err
//f.juice();//javac err
((Orange)f).juice();//no javac err , run time err : java.lang.ClassCastException : Apple can't be cast into Orange
```

With simple example : Object , Person , Student,Faculty : refer to Test2.java

2. Event Organizer app
 Suitable data structure : array of references :
Type of array : Person[]

Menu :
1 : Register Student
2. : Register Faculty
3. Display all participant details
100. : Exit

Did it need any downcasting so far ? NO

Objective : Executing sub class specific functionality.
Add a new method "study" in Student class & "teach" in Faculty class
Option 4 : User i/p : seat no
Check if seat no is valid .
In case of valid seat no --- If it's a Student , invoke study method , if it's a Faculty invoke teach method

4. instanceof keyword

Sample Data
a1 b1 2020 java 23456 87
a2 b2 2021 JS 12345 85
a3 b3 10 java,db,react

a4 b4 20 java,.NET

Regarding inheritance

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 -- code 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(behaviour --- method overriding)

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

Fruit -- Apple -- FujiApple

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

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

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{...} --provided B & C : classes -- compiler err

Why --For simplicity.

(Diamond problem)

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

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

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

---

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

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

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

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

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

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

What will be o/p ?
Emp e =new Mgr(...);
e instanceof Mgr -

e instanceof Emp -
e instanceof Object -
e instanceof SalesMgr -
e instanceof Worker -
e=null;
e instanceof Emp/Mgr/SalesMgr/Worker/Object -

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

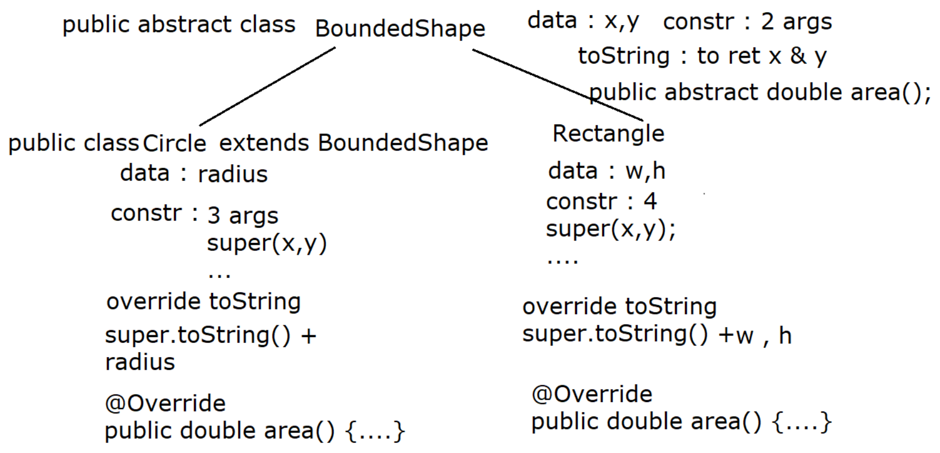
behaviors for the same methods defined in the superclass (and overridden in the subclasses).
A static method is not associated with any instance of a class so the concept is not applicable.

eg : Shapes scenario
Solve this

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

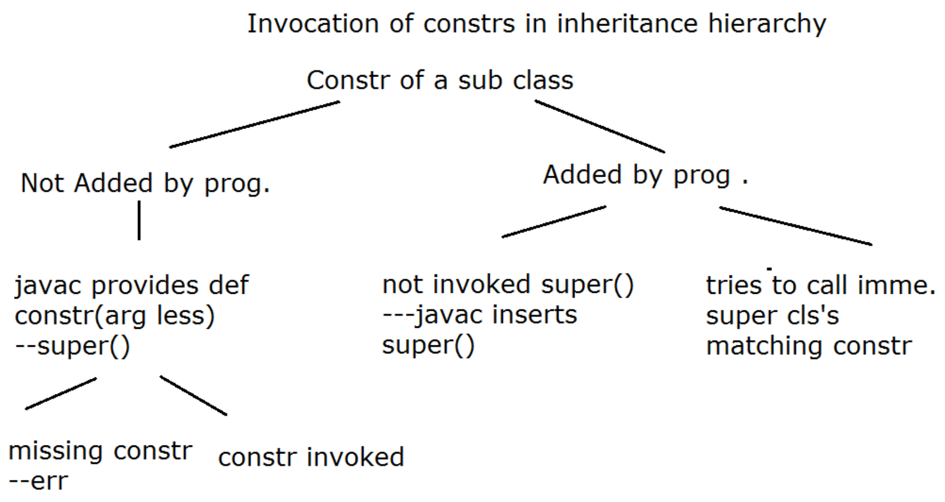
2. Circle -- x,y,radius
Method --public double area()
public String toString()

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


public abstract class BoundedShape
data : x,y   constr : 2 args
toString : to ret x & y
public abstract double area();

public class Circle extends BoundedShape
data : radius
constr : 3 args
super(x,y)
...
override toString
super.toString() +
radius
@Override
public double area() {....}

Rectangle
data : w,h
constr : 4
super(x,y);
....
override toString
super.toString() +w , h
@Override
public double area() {....}


Invocation of constrs in inheritance hierarchy

Constr of a sub class

Not Added by prog.

javac provides def
constr(arg less)
--super()

missing constr
--err

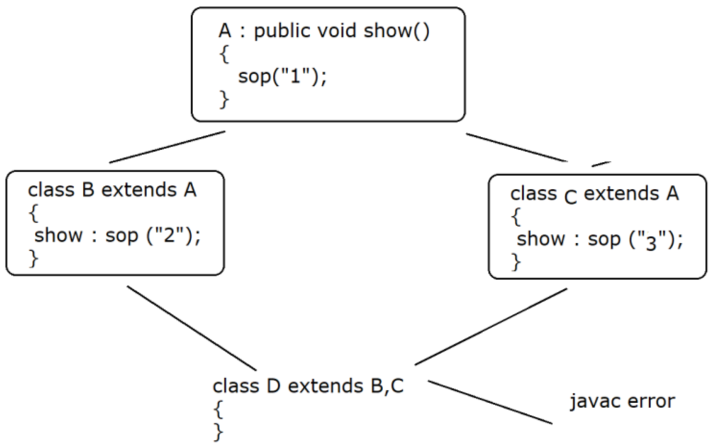
constr invoked

Added by prog .

not invoked super()
---javac inserts
super()

tries to call imme.
super cls's
matching constr

Problem with multiple inheritance : ambiguity!


```
A : public void show()
{
  sop("1");
}

class B extends A
{
 show : sop ("2");
}

class C extends A
{
 show : sop ("3");
}

class D extends B,C
{
}
```
javac error

Polymorphism --changing forms of behaviour

static polymorphism -- detected by javac(early binding) Via method overloading
1.Can exist in same class or in inh hierarchy.
2. same method name,ret type ignored
3. signature - different (no/type/or both)
4. No rules on access specifiers
5. No rules regarding exc handling

Dynamic form of polymorphism (late binding)
1.can exist only in inh hierrachy Overriding method
2.Must have same name,same signature,ret type can be either same or sub type of the super class method ret type(co variance)

4.Overridng form of the method must either same access specifier or wider.

5. Overriding form of the method can't throw any NEW or wider checked excs.

<u>DAY 6:</u>

More details about polymorphism
Abstraction
Partial n complete

Fruit <---Apple,Orange,Mango<---Alphonso
Suppose taste() exists in a Fruit class , it's overridden in all sub classes
Suppose toString() is overridden in Fruit class , also in all sub classes
Solve
What will happen ?
Fruit f=new Mango();//up casting
f.taste();//no javac err , JVM invokes taste on Mango's instance : run time poly
sop(f);//f.toString --> Mango's toString : run time poly

f=new Orange(...);//up casting
f.taste();//no javac err , JVM invokes taste on Orange's instance : run time poly
sop(f);//f.toString -->Orange's toString : run time poly

f=new Alphonso();
f.taste();

sop(f);

Object o=f;//up casting
sop(o);//JVM calls : toString on Alphonso's instance

Solve
Fruit f=new Alphonso();//up casting
f.Alphonso//juicyPulp() is an additional method on Alphonso : javac err
//any soln : down casting : explicitly to be done by prog.
((Alphonso)f).juicyPulp();//no javac err,no run time err, JVM exec method

f=new Orange(...);//no err
((Alphonso)f).juicyPulp();//no javac err, run time err : ClassCastExc : Orange can't be cast in to Alphonso

Is there any guard or any way to perform RTTI checking ? 2 ways : 1. instanceof OR 2. Reflection API

How to avoid class cast exc :
f=new Orange(...);//no err
if(f instanceof Alphonso)
((Alphonso)f).juicyPulp();// no err , method will be called
else
sop("not an Alphonso!!!!!");

o/p : not an Alphonso!!!!!

Important :
When will you need to apply downcasting?
In indirect referencing , i.e when super class ref ---> sub class instance AND it's calling sub class spcific func.

Any probable run time error ?? : java.lang.ClassCastException

Before down casting : ??? : instanceof checking

Answer this

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

What will be o/p ?

Emp e =new Mgr(...);//no err : up casting
e instanceof Mgr - true
e instanceof Emp - true
e instanceof Object - true
e instanceof SalesMgr - false
e instanceof Worker - false
e=null;
e instanceof Emp/Mgr/SalesMgr/Worker/Object -false

Tester :
How will you create a basket of fruits , with 1 apple , 2 alphonso , 1 orange : using dynamic init of array ?(in a single java statment)
eg : Fruit[] basket={new Apple(....),new Alphonso(.....),new Alphonso(.....),new Orange(....)};//5 , 1 array , 4 different fruits

How will you display tastes of all fruits ?
for (Fruit f : basket)
 f.taste();

Enter polymorphism formally : refer to readme inheritance poly.

Refer to method overloading vs method overriding

Identify the rules.
Examples of method overloading : print / println methods

Example of Method overriding
eg : toString,taste

Java Annotation
eg : @Override annotation

5. Enter abstraction
refer : :regarding abstraction"

eg : Shapes scenario
Solve this

1. BoundedShape -- x,y (state) : tight encapsulation
ctor : 2 args

public double area() : only declaration : enter abstract keyword!

```
@Override
public String toString(){...}
```

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

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

eg : In a Tester

```
BoundedShape[] shapes={new Circle(20,40,12.5),new Rectangle(30,40,10,15.7)};
for(BoundedShape s : shapes)
{
  sop(s);//line 1
  sop(s.area());//line 2
}
```

----

a1 b1 2020 java 23456 87
a2 b2 2021 JS 12345 85
a3 b3 10 java,db,react
a4 b4 20 java,.NET

Abstraction is the property with which only the essential details are displayed to the user.
The internal details or the non-essentials details are not displayed to the user.
(Hiding complxities or hiding the implementation details from end user)

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

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

This is what abstraction is.

In Java , abstraction is already achieved using unit of encapsulation : class

When you define methods(functionality) in the class , it's user(Client code using these methods) does not need to know about the actual implementation of the methods, it just needs to know about the invocation.

It can be further achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

When to use abstract classes and abstract methods ?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.(i.e only provide declaration)
Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

eg : BoundedShape & it's method area

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

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

Abstract classes can't be instantiated BUT can create the reference of abstract class type to refer to concrete sub-class instances.
eg : BoundedShape shape=new Rectangle(....);//legal
BoundedShape shape2=new BoundedShape();//javac err : RHS

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

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

Can a class be declared as abstract & final ? NO

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

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

final -- keyword in java

Usages

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

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

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

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

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.

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...

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface.

Abstract class Vs Interface

1) Abstract class can have abstract and non-abstract methods. Interface can have only abstract methods.
2) Abstract class doesn't support multiple inheritance.   Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.   Interface has onlypublic,static and final variables.
4) Abstract class can have static methods, main method and constructor.   Interface can't have static methods, main method or constructor.
5) Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
6) The abstract keyword is used to declare abstract class.The interface keyword is used to declare interface.
7) Example:
```
public abstract class Shape{
public abstract void draw();
}       Example:
public interface Drawable{
void draw();
}
```

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Abstract Class vs. Interface

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

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

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

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

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

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

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

<u>DAY 7 :</u>

Today's Topics
Mem pic to understand run time polymorphism
Revise interfaces
Object class method : equals (reading H.W)
Exception Handling

Query : why overrding form of the method can't reduce visibility ?
eg :

```
A.java
package p1;
public class A
{
 public void show() {...}
}

B.java
 public class B extends A
{
  void show() {...}  //javac err
}
```

Why ?

```
package tester;
public class Test : main
A a1=new A();
a1.show();
a1=new B();
a1.show();
```

Solve this. What will happen ?

```
package p1;
public class MySuperClass {
	static {
		System.out.println(3);
	}
	{
		System.out.println(4);
	}
}
package p1;

public class TestInstanceInitBlock extends MySuperClass {
	private int i;

	public static void main(String[] args) {
		System.out.println("in main");
		TestInstanceInitBlock t1 = new TestInstanceInitBlock();
	//	TestInstanceInitBlock t2 = new TestInstanceInitBlock(10);
	}

	static {
		System.out.println(1);
	}
	{
		System.out.println(5);
	}


	public TestInstanceInitBlock() {
		System.out.println(2);
	}

	public TestInstanceInitBlock(int i) {
		super();
		System.out.println(6);
		this.i = i;
	}
}
```

Revise

What is an interface (i/f) : blue print of a class , consists of specs.(WHAT)

Why ?

1. Complete abstraction
2. multiple inheritance
3. separation : loose coupling (separation between WHAT : specs vs HOW : imple)

Till JDK 1.7 : What does i/f mainly contain ?

data members : public static final

methods : public abstract

Legal Relationships

Can 1 class imple multiple i/fs : YES (implements)

Can 1 i/f extend from multiple i/fs : YES (extends)

Can 1 class extend from multiple super classes : NO

Can 1 i/f be implemented by multiple imple classes ? : YES (implements)

Can imple class access i/f constants directly(w/o using i/fName.memberName) : YES

Can non imple class access i/f constants directly(w/o using i/fName.memberName) : NO

(can access it as : i/fName.memberName)

Objectives
0. Create different type of Printers(ConsolePrinter, FilePrinter ,NetworkPrinter) n access it's common functionality n specific functionality in a single for-each loop.

1. Demo : Create a class implementing multiple i/fs .
Will you face ambiguity issue with extact duplicate behaviour ? : NO
Same Method names with different signature : NO
Same Method names same signature n different ret type : javac err (ambiguity!)
2. Demo : Create a class implementing multiple i/fs .
Will you face ambiguity issue with duplicate data members ? YES
How to resolve : i/fName.memberName
3. Demo : 1 i/f extending multiple super i/fs , n then write imple class
In order to create concrete imple class : MUST imple all inherited abstract behaviour.

4. Marker i/f : Empty (Tag i/f) i/f --NO data members n no methods
eg : Serializable , Cloneable

5. Functional i/f : i/f containing single abstract method (SAM)
Java 8 added annotation : @FunctionalInterface : i/f level annotation

6. Demo : Shapes scenario using i/f n abstract class.
(refer to diag : "abstraction scenario")
----

7. Enter Exception Handling :
What is Exception ? : run time error , detected by JVM (typically by main thread)

Why Exception handling ?
1. To allow the continuation of java app , even in case of run time errors.
2. Reduces the need for checking validation of ret types.
3. Allows a Separation between business logic n error handling

Flow , inheritance hierarchy , checked vs un checked excs
keywords : try, catch , finally , throw , throws , try-with-resources
----
Pending !

Custom exceptions
Objective : Validate speed of running vehicle on a highway
min speed : 30
max speed : 80
Vehicle speed : 20 / 100 /60
This validation err : can not be detected by JVM.

Must be done by the prog : keyword : throw new SpeedOutOfRangeException("too slow / too fast");

Exception Handling

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

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

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

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

Legal syntax
1. try {...} catch (ArithmeticException e){...}
2. try {...} catch (exc1 e){...} catch (exc2 e) {..} ....
3. try {...} catch (NPE e){} catch (AE e) {}catch(Exception e){catch-all}
3.5 try {...} catch (AE e){...} catch (NPE | AOB | 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
Thread class API
public static void sleep(long ms) throws InterruptedException

FileReader API
public FileReader(String fileName) throws FileNotFoundException

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

4.5 Throwable class API

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

5. finally --- keyword in exc handling : represents a block
finally -- block -- finally block ALWAYS survives(except System.exit(0) i.e terminating JVM)
i.e in the presence or absence of excs.
5.1 try{...} catch (Exception e){....} finally {....} : will code continue in normal manner ? YES
5.2 try{.NFE.} catch (NullPointerException e){....} finally {....} --- will code continue in normal -- manner ? -- NO --JVM searches for matching catch ---found --exec catch --code continues.
not found -- aborts code
5.3 try {...} finally {....}

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

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

syntax of try-with-resources

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

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

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

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

```
eg : public class MyException extends Exception{
  public MyException(String mesg)
  {
   super(mesg);
  }
}
public class MyException2 extends RunTimeException{....}
```

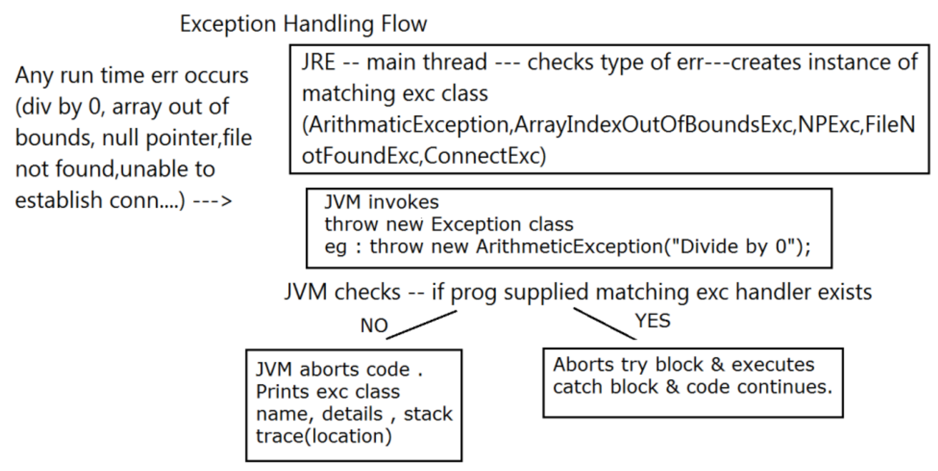
2.CustExc(String msg) : overload the constr : to invoke the super-class constr.
of the form
Exception (String msg)
OR
CustExc(String msg,Throwable rootCause)
public Exception(String message,Throwable cause)

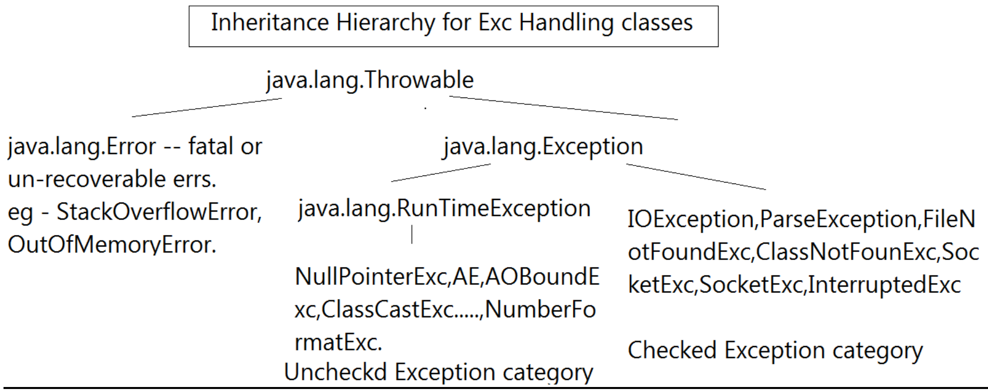
Objective :
Check the speed of vehicle on a freeway
Accept the speed using Scanner : can be speed too low(exc) or too high(exc) or in range

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

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

---


Exception Handling Flow
Any run time err occurs (div by 0, array out of bounds, null pointer,file not found,unable to establish conn....) --->
JRE -- main thread --- checks type of err---creates instance of matching exc class (ArithmaticException,ArrayIndexOutOfBoundsExc,NPExc,FileNotFoundExc,ConnectExc)
JVM invokes
throw new Exception class
eg : throw new ArithmeticException("Divide by 0");
JVM checks -- if prog supplied matching exc handler exists
NO
YES
JVM aborts code . Prints exc class name, details , stack trace(location)
Aborts try block & executes catch block & code continues.


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

DAY 8:

Today's Topics

Custom Exceptions
String Handling
Date Time Handling
Object's equals method
Detailed application using all of above concepts.
Debugging

Revise

What is exception ? --run time error , detected by typically JVM's main thread

Why exception handling ?
1. To continue with the program execution , even after run time errs(eg :invalid inputs,B.L 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 .

Flow
Eg. Trigger : div by 0 ---> JVM chks type of the err --->throw new AE("/ by 0");

JVM chks ---matching exc handler(catch) exists -- NO --supplies a def handler --Aborts : prints -- name , err mesg , stack trace (location details)

YES --try block aborts --JVM invkes MATCHING catch block --- code continues..

syntax of "throw" keyword
throw Throwable instance;
currently used by JVM to raise system excs(eg : AE, AOBExc, NPE,ClassCastExc.....,IOException, SocketExc, SQLExc....)

Inheritance hierarchy of exc handling classes
Throwable <--- Error , Exception
Exception <-----RunTimeException <----un checked excs (eg : AE, AOBExc, NPE,ClassCastExc.....)
Exception <----- IOException, SocketExc, SQLExc , InterruptedException : checked excs

try-catch keywords
checked vs un checked exceptions
who doesn't differentiate between checked vs un checked exceptions --JVM

who differentiates n how ? : javac
Javac forces handling of checked exceptions ---
1.actual handling : try-catch
2. throws keyword

o.w : javac err

2 ways

Actual handling : try-catch
Exc handling delegation : throws

throw vs throws
throw :keyword used to raise the exception(JVM : system/built-in excs , Prog : custom exc), appears in method def.
syntax : throw Throwable instance;

throws : meant for javac , appears in method declaration

```
eg : public void show() throws IOException,InterruptedException
{
 ......
}
```

Meaning : show() : may throw the exc(possibility) , curnt method is NOT handling the exc. so its' caller should handle.

when is adding throws keyword mandatory : unhandled(no try-catch) checked excs
o.w : javac err

finally : always executed(i.e in case of no exc as well as excs/ method rets) except : JVM termination(System.exit(0))
typical use case : cleaning up of non java resources(eg : File handle, socket , db connection, Std in)

OR
try-with-resources

```
eg : try(Scanner sc=new Scanner(System.in);
 Connection cn=....;//DB connection opening
FileInputStream in=....;//opening bin file
)
{
.....
} //in.close,cn.close ,sc.close()
catch(Exception e)
{
 ....
}
```

Can you open ANY Resource(class instance) using try-with-resources block 's header? NO
Only those class instances : whose class has imple . : java.lang.AutoCloseable
i/f method
public void close() throws Exception

Custom exceptions
objective : Accept speed of a vehicle from user (scanner) --highway
min speed : 30
max speed : 80

In case of speed outside the range --Prog will have to detect the err --create instance of user defined exc class --explicitly throw custom exc to the code. The alteration in flow will be managed by JVM.

Steps
1.Create custom exception class : extends Throwable/Error/Exception/RuntimeExc....
eg : public class SpeedOutOfRangeException extends Exception{}
1.1 Add parameterized constr : to init err mesg

2. Create a separate class eg : SpeedUtils --
add a static method for validation the speed

3. Create Tester : UI --scanner --accept the speed n simply call validateSpeed of the SpeedUtils class
----

4. String Handling
4.1 Immutability of strings
4.2 == vs equals
4.3 literal vs non literal string
4.4 API

5. Date Time Handling

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

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

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

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

5. String(String s)

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

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

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

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

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

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

Imp ---

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

Date/Time Handling in Java

API
1. java.util.Date--- represents system date , 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

Arguments --- Date, GregorianCalendar

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

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

var-args ...
variable args syntax.--- Must be last arg in the method args.
Can use primitive type as well as ref types.
Legal ---

```
void doStuff(int... x) {
//B.L
 } // 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

System.out.printf("%n");//legal
System.out.printf(1234);//javac error

// as parameters
void doStuff2(char c, int... x) {..... } // expects first a char,
// then 0 to many ints
eg : ref.doStuff2('a',1,2,3,5);//no javac err
ref.doStuff2();//javac err : 1st arg missing


class Test {
void doStuff3(Animal... animals) {
   for(Animal a : animals)
     sop(a.getName());
} // 0 to many Animals
}
Test ref=new Test();
invocations ---
ref.doStuff3();//no javac err
Animal animals[]={new Cat(),new Dog(),new Horse()};//4 objs
ref.doStuff3(animals);//no javac err
Animal a1=new Horse();
Animal a2=new Cat();
Animal a3=new Dog();
ref.doStuff3(a1,a2,a3);
```

Illegal: javac error

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

DAY 9:

Today's Topics
overriding of equals method
Solving bigger assignment (4)
var-args
Java Enums

1. Solve

```
String s="Hello";//literal string : string cls loaded meta space , vtable : meta space , in heap :
Constant/literal string pool ,literal string gets added in the pool, ref stored in s : stack
s.toUpperCase();//non literal string : "HELLO" : marked GC
s.concat("12345");//non literal string : "Hello12345" : marked GC
sop(s);//Hello
String s1="testing strings";//literal string , pool : 2 strings, s1---> literal string
String s2=new String(s1);//non literal string s2 --> non literal string
sop(s1==s2);//f : ref eq.
sop(s1.equals(s2));//t : content eq
String s3="He"+"llo";//s3 ---> earlier created literal string(Hello) , SCP : Hello,testing strings ,
He,llo,12345
```

String s4="He".concat("llo");//anything gets added newly in the pool : no , new non literal "Hello", s4 --> new non literal string
String s5="hello";//s5 ---> newly created literal string
sop(s==s3);//true
sop(s==s4);//false
sop(s==s5);//false

2. Solve

```
public static void main(String[] args) {
                String s1="hello";
                String s2="hello";
                String s3=new String(s1);
                String s4=s3.intern();
                String s9=new String(s1.toUppercase());
                   String s10=s9.intern();
                String s5="he"+"llo";
                String s6="he".concat("llo");
                System.out.println(s1==s2);
                System.out.println(s1==s3);
                System.out.println(s1==s4);
                System.out.println(s1==s5);
                System.out.println(s1==s6);
                String s7=new String("Hello");//how many string objects are created in this line? :
                String s8=new String("hello");//how many string objects are created in this line? :

        }
```

Revise Date Time Handling
java.util.Date : represents Date n time both , till precision msec.
Constrs :
Date() : Creates date class instance : current date n time
Date(long msec) : Creates date class instance : that represents date n time , of msecs elapsed after Epoch(1st Jan 1970)
Methods : toString , before ,after,compareTo,equals
eg : d1.compareTo(d2) : -1 => d1 before d2
0 => d1.equals (d2)
1 => d1 after d2

Objective : How to accept manufacture date of a Vehicle ?

Steps
1. Create instance of a class : java.text.SimpleDateFormat

2. Parsing (string --> Date) API

3. Formatting API (Date --> string)

Solve Vehicle's assignment based on equals (revise)

Add More validation rules (solve 2nd assignment)

var-args

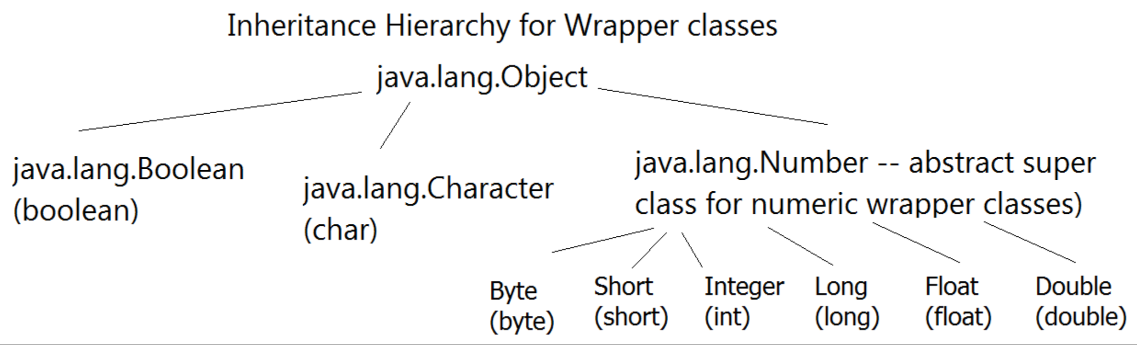
Enum : Vehicle Category
eg : Vehicle HAS-A Category

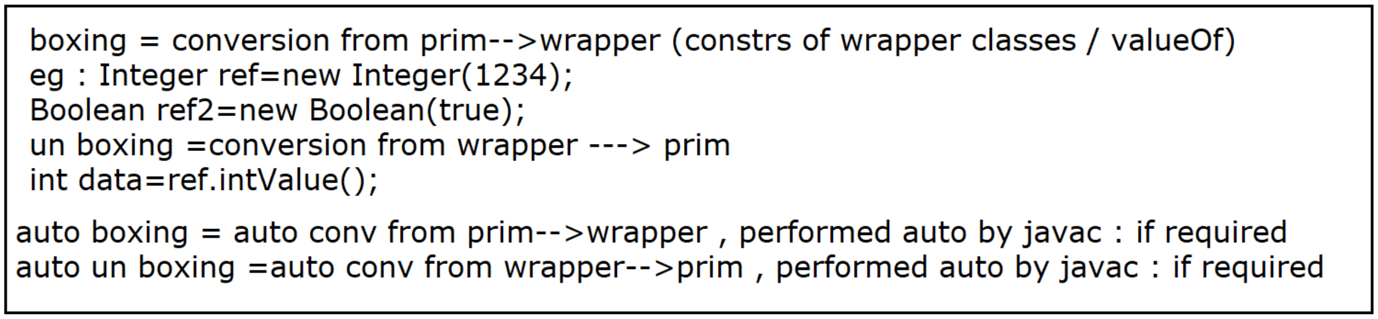
Sampl data for vehicles
12345 red 456789 2-10-2021 petrol
12340 blue 456780 22-10-2021 diesel
12350 white 456790 2-4-2021 ev
12347 black 456789 12-8-2021 petrol

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


Inheritance Hierarchy for Wrapper classes
java.lang.Object
java.lang.Boolean (boolean)
java.lang.Character (char)
java.lang.Number -- abstract super class for numeric wrapper classes)
Byte (byte)
Short (short)
Integer (int)
Long (long)
Float (float)
Double (double)


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

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

A simple usage will look like this:

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

Here EAST, WEST, NORTH and SOUTH are implicitely of type

```
public final static Direction EAST=new Direction("EAST",0) ---super("EAST",0);
public final static Direction WEST=new Direction("WEST",1) ---super("WEST",1);
```

Super class of all enums

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

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

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

enum can be declared within a class or separately.

eg of enum within a class

When declared inside a class, enums are always static by default

```
eg public class TestOuter
 {
  enum Direction
  {
    EAST,
    WEST,
    NORTH,
    SOUTH
  }
 }
```

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

Constructors of enum

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

Important Methods of Enum (implicitly added by javac)

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.

DAY 10:

Today's Topics
More Features of enum
Wrapper classes
HAS-A relationship
Nested classes
Generics

Revise enum
eg :
public enum Color
{RED,GREEN,BLUE}

What is it ? : keyword to represent set of related self typed constants.
It's a special class generated by javac that implicitly extends from --- java.lang.Enum

Access specifier of the Implicitly added constructor :private
which are the args of the ctor : name , ordinal
1st n only statement in the ctor : super(name,ordinal)

Data Members : public static final Color RED,GREEN,BLUE;

Important Methods :
Inherited from Enum : public String name()
public int ordinal()
public String toString()
....
+ javac synthesized methods
1. public static Color[] values()
2. public static Color valueOf(String name) throws IllegalArgumentException : unchked exc
eg : sop("Choose color");
Color c=Color.valueOf(sc.next().toUpperCase());//green => no exc / yellow => exc

static init block : YES
static {
 RED=new Color("RED",0);
GREEN=new Color("GREEN",1);
BLUE=new Color("BLUE",2);
}

Which methods can be overridden ? : toString
inherited version : rets name()

Which methods can not be overridden? : name(), ordinal(),equals(), compareTo()

Which i/fs it implements ? Serializable , Comparable
eg :
```
public enum Color
{RED,GREEN,BLUE}
```
In a Tester class :
```
import static com.app.core.Color.*;
sop(RED.compareTo(BLUE));//what will be the answer ? -1
```

Can you add enum nested within a class or interface ? YES (implicitly it's treated as static member)
eg :
```
class Vehicle
{
  ....
private Color clr;
   enum Color
  {RED,GREEN,BLUE}
}
```

How to access BLUE ? Vehicle.Color.BLUE

Can you add , additional state , constructor , methods to the enum ? : YES
eg : Customer HAS-A CustomerPlan / CustomerType that needs specific reg amount. Can you associate reg amount easily with the plan ? YESS
SILVER : reg amount 500
GOLD : 1000
DIAMOND : 1500
PLATINUM : 2000

Can you add a validation rule , to check if customer has paid correct reg amount for the selected plan ?

1.5 Wrappers

2. Establish HAS-A relation between
Vehicle & DeliveryAddress
(one to one association)

Steps
1. Create a separate class : DeliveryAddress -- city,state,country,zipCode
constr , toString

2. To establish association(containment) between Vehicle & DeliveryAddress
Add new state (instance var) : DeliveryAddress
Don't init it in the constr.(since delivery address should be assigned when customer purchases a vehicle)

3. Add a method to link delivery address to a vehicle.

4. Write a simple tester to test this form of association.
Confirm :In this case , can a DeliveryAddress exist w/o a Vehicle ?
If yes : how will you prevent it ?

Enter Nested classes

5. Add Option in Tester
Purchase Vehicle
i/p : vehicle's chasis no
In case of valid chasis no , if the vehicle is not already sold , accept delivery address details & assign it to the vehicle.
o/p : in case of invalid chasis no or vehicle already sold --throw custom exc
o.w : assign delivery address & give success message.

6. Discuss rules of non static nested class (inner class)

7. wrapper classes

8. Generics

12345 red 45678 2-6-2021 ev
pune mh in 435467

NESTED CLASSES''

1. The inner class(non-static nested) has access to all of the outer class's members, including those marked private , directly(without inst.)
BUT Outer class MUST make an instance of the inner class , to access it's members.

2. To instantiate an inner class, you must have a reference to an instance of the outer class.
syntax :
Instantiating a non-static nested class requires using both the outer inst and nested class names as follows:
BigOuter.Nested n = new BigOuter().new Nested();
3. Such Inner classes can't have static members.(Java SE 8 --allows static final data members)

About method-local inner classes

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

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

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

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

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

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

Nested Classes in Java

non static nested class
--inner class
Inner class can access directly , even private members of outer class.
tight coupling.
Outer.class,Outer$Inner.class

static nested class
Can directly access outer class's static members.

Method local inner class
entire class definition within a method.
Can be used only by the same method, after cls declaration.

-nameless inner class.(name assigned by javac)
entire cls def within single java statement

Types of Nested classes

Non static Nested class --inner class
1. Can't exist w/o outer cls instance.
2. Can access directly outer clas's even private members.

3. Can't contain static members.
JDK 1.8 - static final d.m allowed.

static nested class
--Same as static member of the outer cls.
eg :
```
class A
{
  //static n non static members
  static class B
  {
    //can contain static n n.s members.
  } }
```

Method local inner class
Nested class within a method def.
eg :
```
class A
{
  void show() {
    class B {......}
  }
}
```

Anonymous Inner class
Entire inner class def. is embedded within single java stmt. JDK 1.8 offers a better replacement of this by lambda expression.
eg : W/O inner class
```
public class MyComp imple. Comparator<String>
{
  public int compare(String s1,String s2)
  { return s2.compareTo(s1)}
}
TreeMap<S,User> tm=new TM<>(new MyComp());
tm.putAll(hm);
```
Using ano inner cls syntax.
```
TreeMap<S,User> tm=new TreeMap<>(new Comparator<String>()
{
  public int compare(....)
  {....}
});
tm.putAll(hm);
```

Types of Nested classes in Java -- A class embedded within another class or method or statement .

non static nested class. -- Inner class.
1. Can access private members of the outer class.
2. Can have any access specifier.

3. Cant contain static member, since inner class can be created only with outer class inst.

static nested class
```
class Outer{
....
  static class Nested
  {
    can contain both static & non
static data members.
BUT can access outer's static
members directly(w/o inst)
 Treated as top level static
member.
 }
}
```

Method local inner class
Entire inner class def embedded within method def.
Accessible only to the method --where its defined, after class def.
```
class Outer
{
  void test()
  {
    .....
     class Inner
    {.......}
    Inner in1=new
Inner();
  }
}
```

Anonymous inner class --nameless inner class, embedded within single statement.

Regarding Association

Association is relationship between two separate classes , using object references.
Represents HAS-A
Why : Code reusability

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

Composition and Aggregation are the two forms of association.

eg : refer to association-aggregation-composition.png
Association :
Owner HAS-A Pet -- Owner feeds a Pet & Pet plays with Owner.
Aggregation implies a HAS-A relationship where one entity can exist independently of the other entity.
eg : Class & Student / Bank HAS-A Customer
```
class Bank
{
  private String name;
  private String ifsc;
  private String address;
 //one to many
  private Customer[] customers;

}
class Customer {
 private String name;
 private String address;
 private Date dob;
private Bank myBank;

}
```

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

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

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

---