Eclipse me **Java ->OCA Preparation -> OcaVideo1** dekho

1. == is an operator that does internal overloading which is used for both the primitive and reference variables.
2. For Primitive type it checks the values of the variable but for the reference variables it calls the equals() of the object class which checks the reference of those variables.
3. So in the case of collections it is not sufficient to check only the value of the variable so we override both the hashCode() and equals() to check both the val and reference.
4. Boolean is in the bit format whereas all the other types are in the byte format so it is impossible to type cast it. **Line no. :16**
5. A reference variable if not initialized then it holds a null value. **Line no. :3 and 25**
6. Inheritance is a **IS-A** and interface is a **HAS-A.**
7. List is an Interface which is implemented by the ArrayList and the LinkedList.
8. While giving command line arguments the space is used as delimiter and not the (,)
9. If whitespaces are given before the string, then all the whitespaces are ignored and the first string is taken as the first argument
10. Call by value doesn’t reflect back the changes in the main() whereas call be reference reflects the changes back in the main().
11. The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation.

**Ex:** In ur code if u declared the instance variables as public then no person will use getters or setters as they can directly access the variables.

1. **2 Common reasons for Inheritance:** i) To promote code reuse ii)To use polymorphism
2. **The problem in code reusability** is when u make changes in one place and u need to track down all the places that need to be changed.
3. In Inheritance, if u have a method that has the parameter of type Superclass then that method can be invoked even by using the instance of its subclass. And then that method can access any of the members of the superclass without being concerned about the subclass’ object being passed to it.
4. The IS-A relationship defines the inheritance either with a class or with the interface
5. HAS-A relationships are based on usage, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B.
6. There are a few key things you should know about references:

■ A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).

■ A reference is a variable, so it can be reassigned to other objects (unless the reference is declared final).

■ A reference variable's type determines the methods that can be invoked on the object the variable is referencing.

■ A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—**it can refer to any** subtype **of the declared type!**

■ A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that implements the interface.

1. **Deadly Diamond of Death** is the problem that arises when multiple inheritance occurs. Where a diamond is formed when A is extended by B & C and is further extended by D.
2. If u give **Parent obj=new Child()** then the obj can access only those methods that are available in the parent or which are overridden in the child from the parent.
3. **Here the obj has a reference of Parent but it is an object of Child.**
4. You can’t reduce the visibility of a method while overriding.
5. The subclass is guaranteed to be able to do everything the superclass can do.
6. **Rules for overriding a method are as follows:** i) The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. ii) The access level CAN be less restrictive than that of the overridden method. iii) Only public and protected can be overridden. Iv) The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception. V) The Exceptions used in overriding method must not be greater than the exception of overridden method. v) An overriding method doesn't have to declare any exceptions that it will never throw, regardless of what the overridden method declares.

**NOTE:** If u override a class that throws exception but the overriding method doesn’t then the object with reference of parent but instance of child will show compilation error.

1. **Rules for OverLoading:** ■Overloaded methods MUST change the argument list.

■ Overloaded methods CAN change the return type.

■ Overloaded methods CAN change the access modifier.

■ Overloaded methods CAN declare new or broader checked exceptions.

■ A method can be overloaded in the same class or in a subclass.

1. If u have a method of superclass without an exception but the sub class has the same method with different arguments and with Exception then the method shows compile time error because of the error. The different types of arguments doesn’t create a mess.
2. U can access a not-static method or variable using an object of that class.
3. **which overridden version of the method to call (in other words, from which class in the inheritance tree) is decided at runtime based on object type, but which overloaded version of the method to call is based on the reference type of the argument passed at compile time**
4. If u have a method with the parameter type as the parent class but while invoking the methods if I call that method with the object of child class type yet it calls the method with the object of the parent type only.
5. **Parent obj=new Child();** this can be used only when the child class overrides one of the methods of the parent class.
6. For creating an object of a class we need to call the constructor of that class and all its superclasses. This process runs unless there is a explicit call to a parameterized constructor of the same class.
7. Constructors are called whenever we use the keyword **“new”.**
8. Even an abstract class has a constructor.
9. Constructors never have a return type and should always match with the name of the class.
10. If there is only one constructor that too with params then the obj can be instantiated only by passing parameters along with the obj. Here a default constructor won’t work.
11. An object is given explicit values (ie while declaring itself).
12. **Rules for Constructors:**

* Constructors can use all the access modifiers including private.(Only the code within the class can instantiate. The class can declare a static method or variable to access the obj)
* If u create a parameterized constructor without a default one then the JVM won’t provide u with a default constructor.
* Every Constructor’s first statement should be either a call to the overloaded constructor(this) or a call to its superclass(super). It can also be inserted by the compiler.
* We can’t access any of the variables or methods without the superclass’ constructor being called
* Only static variables and methods can be accessed as part of the call to super() or this(). (Example: super(Animal.NAME) is OK, because NAME is declared as a static variable.)
* Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
* Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
* A constructor can be called only from within a constructor or by instantiating an obj.

1. **Rules for a default constructor:** i)It has the same access modifier as the class. ii) It has no arguments. Iii) It includes a call to the no argument super constructor.
2. There are 3 places in a java prg. where an operation can be performed: Methods, Constructors and Initialization blocks (ie. static{} which gets invoked whenever the class is first loaded in the memory.)
3. **Initialization Block:** The **Static Initialization Block** is a block which doesn’t have a name nor a return type. It gets executed as soon as the prg. is first loaded in the memory. If there exists more than one Static block then it gets executed in the order it appears in the prg. **Instance Initialization Block** is a block which gets executed after each instance is created. It gets called right before all the calls to the super class’ constructor are called. It usually consists data that can be used by the constructors being called.
4. If there exist more than one Instance initialization blocks in a class then it gets executed in the top down approach(Order in which it is typed).
5. **Static :** Variables and methods marked static belong to the class, rather than to any particular instance.
6. For accessing a non-static method the JVM needs an object to be instantiated first. Static method doesn’t need an object to get invoked by the JVM.
7. **Difference between redefining and overriding:**

**Redefining** is similar to the overriding but the only difference lies that it can be done only for the static methods. **Overriding** can be done for any of the methods present in the parent class and its sub class.

**Two minutes Drill**

1. Encapsulation helps hide implementation behind an interface (or API).
2. Encapsulated code has two features:

❑ Instance variables are kept protected (usually with the private modifier).

❑ Getter and setter methods provide access to instance variables.

1. **IS-A** refers to inheritance using wither of the 2 keywords- extends, implements.
2. **HAS-A** means an instance of one class has a reference to an instance of same or other class.
3. A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
4. A single object can be referred to by reference variables of many different types—as long as they are the same type or a supertype of the object.
5. The overriding method must have the same return type, except that, as of Java 5, the return type can be a subclass, and this is known as a **covariant return.**
6. The overriding method may throw fewer or narrower checked exceptions, or any unchecked exception. **(Exceptions chote hone chahiye aur access specifiers bade)**
7. Final methods can’t be overridden.
8. Only inherited methods may be overridden, and remember that private methods are not inherited.
9. Methods from a superclass can be overloaded in a subclass.
10. Polymorphism applies to overriding, not to overloading.
11. Object type (not the reference variable's type) determines which overridden method is used at runtime.
12. Reference type determines which overloaded method will be used at compile time.
13. Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
14. For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
15. Nothing can be returned from a void, but you can return nothing. (return ;)
16. Methods with an object reference return type can return a subtype.
17. Methods with an interface return type can return any implementer.
18. Constructors can use any access modifier (even private!).
19. The default constructor is a no-arg constructor with a no-arg call to super().
20. Abstract classes can have constructors which are invoked when its concrete class’ object is created. Whereas an Interface cannot have a constructor.
21. Constructors are never inherited; thus they cannot be overridden.
22. All static members belong to the class, not to any instance. (ie a static member is class specific and not instance specific)
23. A static method can't access an instance variable directly.
24. An abstract class need not implement any or all methods of the interface.
25. **If there is an inheritance tree going to be executed then all the static blocks of all the classes will be initialized in a top-down approach then the static methods are executed. The instance blocks are called before the constructor is called though they are placed below the constructor being called. Multiple instance blocks are called in top-down order and all the instance blocks are called at each instantiation.**
26. Polymorphism doesn’t apply on the static methods.
27. Polymorphism is only for the methods and not for the variables. (ie. If u have same method and same variable in both the superclass and the subclass then while creating an instance of subclass with a reference of the superclass then the variable of the superclass and the method of the subclass will be invoked.)

**Chapter 3:**

1. Instance variables and objects live on the heap whereas local variables live on the stack.
2. The reference variables are also stored on stack as it is considered as a local variable. **Ex:** Superclass obj;
3. **Remember that a reference variable isn't an object; it's a way to get to an object.**
4. Method placed outside the main() live on the stack till it completes execution. As soon as the execution completes, both the method and the parameters passed are popped from stack.
5. **Integer literals can be of 4 types:** i) Decimal, ii) Octal, iii) Hexadecimal, iv) Binary.
6. Literals are the values that could be stored in a variable. **Ex:** 7
7. Numeric literals can be declared using an Underscore. Bu the rule says that an underscore can’t be either in the beginning or at the end. ( numbers ke beech me (,) use karte hai na waise hi isko bhi use karna chahiye to improve the readability.)
8. Underscores ca be included in any of the numeric values but in float or double it can’t be written next to the decimal point.
9. **Decimal Literal:** int length = 343;
10. **Binary Literal:** Binary literals can use only the digits 0 and 1. Binary literals must start with either 0B or 0b, as shown: int b1 = 0B101010; // set b1 to binary 101010 (decimal 42)

int b2 = 0b00011; // set b2 to binary 11 (decimal 3)

1. **Octal Literal:** int six = 06; // Equal to decimal 6

int seven = 07; // Equal to decimal 7

int eight = 010; // Equal to decimal 8

int nine = 011; // Equal to decimal 9

1. **Hexadecimal Literal:** Java will accept uppercase or lowercase letters for the extra digits. Hexadecimal literals must start with either 0X or 0x, as shown: int x = 0X0001;

int y = 0x7fffffff; int z = 0xDeadCafe;

1. The numeric literals can be given either with an int or with a double. But if it is double the value is given with the suffix L or l
2. **Floating point literals** are by default double(64 bits) so if u want the floating point literal to be of float type(32 bits) the suffix the value with an f or F.

**Note:** If you don't do this, the compiler will complain about a possible loss of precision, because you're trying to fit a number into a (potentially) less precise "container." The F suffix gives you a way to tell the compiler, "Hey, I know what I'm doing, and I'll take the risk, thank you very much."

float f = 23.467890; // Compiler error, possible loss of precision

float g = 49837849.029847F; // OK; has the suffix "F"

1. **Boolean Literal:** int x = 1; if (x) { } // Compiler error!
2. **Character Literal:** You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with \u as follows: char letterN = '\u004E'; // The letter 'N'

**Note:** Remember, characters are just 16-bit unsigned integers under the hood. That means you can assign a number literal, assuming it will fit into the unsigned 16-bit range (0 to 65535). For example, the following are all legal:

char a = 0x892; // hexadecimal literal

char b = 982; // int literal

char c = (char)70000; // The cast is required; 70000 is out of char range

char d = (char) -98; // Ridiculous, but legal

1. Strings are not primitives but are considered to be literals. The other non-primitive literal representation is array.
2. **Variables** are just the bit holders (either primitive holder or obj holder), with a designated type (int, double, button,string[]). The bits represent the values present in that variable.

**Ex for primitive:** A byte with a value of 6, for example, means that the bit pattern in the variable (the byte holder) is 00000110, representing the 8 bits.

1. An object holder holds a value (JVM specific val) that points to some object in the heap or null.
2. **A literal integer is always an int, but more importantly, the result of an expression involving anything int-sized or smaller is always an int.**

**Ex:** add two bytes together and you'll get an int—even if those two bytes are tiny, int\*short=int, short/byte=int

1. **Casting** lets you convert primitive values from one type to another.
2. An **Implicit cast** happens when you're doing a widening conversion—in other words, putting a smaller thing (say, a byte) into a bigger container (such as an int).
3. **Note:** Yaad rakh ki agar int ko float me karna ho to nahi hoga without explicit cast but ulta karna ho to easy hai. Similar case is for long and double.
4. Every floating point literal is implicitly 64 bits and not float. (If u write **float f=23.2;** then by default it is assumed to be a double(64 bits) and not float(32 bits) unless u either explicitly cast it or append an f or F at the end of the literal.

**Note:** If we give **byte b=(byte) 128**;(127 is the highest value because a byte is of 8 bits out of which the left most digit is the sign bit and the remaining 7 digits if are 1 then it can have value only upto 127 (**01111111)** so when we take 128, it becomes an int(32 bits) so it is 00000000000000000000000010000000 but as we cast it it turns to 10000000 so here the sign bit is 1 so it becomes –ve and for calculating a –ve number we take 2’s compliment so flipping the digits give us 01111111 then adding a one gives us 1000000 which is 128 and then the sign bit says that the value should be negative so -128 is displayed.

1. If u don’t want to explicitly cast a byte then u go use shorthand operators (byte b=127; b+=2;)

**Variable Scope:**

1. class Layout { // class

static int s = 343; // static variable

int x; // instance variable

{ x = 7; int x2 = 5; } // initialization block

Layout() { x += 8; int x3 = 6;} // constructor

void doStuff(int a) { // method

int y = 0; // local variable

for(int z = 0; z < 4; z++) { // 'for' code block

y += z + x;

}

}

}

As with variables in all Java programs, the variables in this program (s, x, x2, x3,y, and z) all have a scope:

■ s is a static variable.

■ x is an instance variable.

■ y and a are local variables (sometimes called a "method local" variable).

■ z is a block variable.

■ x2 is an init block variable, a flavor of local variable.

■ x3 is a constructor variable, a flavor of local variable.

1. there are four basic scopes:

■ **Static variables** have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the Java Virtual Machine (JVM).

■ **Instance variables** are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.

**■ Local variables** are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive and still be "out of scope."

**■ Block variables** live only as long as the code block is executing.

1. If u invoke a method from some other method then the called method can’t use the local variables of the calling method.
2. Similarly the block variables(within a block such as a loops, conditional statements) can be used only within the block and not out of the block.
3. Local variables are sometimes called stack, temporary, automatic, or method variables. Although you can leave a local variable uninitialized, the compiler complains if you try to use a local variable before initializing it with a value.
4. **Any variable can be left uninitialized but No variable can be used without initializing except the instance variables which are defined inside the class and not any other method, constructor or initialization block because instance variables are assigned default values if not being assigned by the user.**
5. Instance variables are initialized to a default value each time a new instance is created, although they may be given an explicit value after the object's superconstructors have completed. **(Object banate hi sare instance variables reinitialize hote hai default values se for sare constructors call hote hai and uske baad instance variables reinitialize hote hai explicit values se)**
6. Any Operation on an object that has a default value (ie. Null) will throw a runtime exception. So to avoid such an exception we can first check whether it is null or not and then perform the opr.
7. An array object(is declared as an instance variable, as local variables has to be initialized) unless it is initialized holds a null value. And trying to access it’s array elements will also result in an exception. But if the array obj is locally declared but the elements are not initialized then the array elements will hold a default primitive val if not explicitly assigned.
8. **The compiler can't always tell whether a local variable has been initialized before use.**

**Ex:** IfI declare int x; if(y>0){x=10;} sysout(x); It will show a compilation error.

1. Any Object with a null value can’t have a dot operator on it but can be printed using sysout();.
2. If u have created an obj(A a=new A(5);) and u pass the obj to another reference(A b=a;) then any changes made to any of the obj will change the value in the other obj as well. (Ie if 2 references point to the same instance then the value changed using one instance will reflect the other one)
3. But this is not the case for a String obj because String obj are immutable.

**Passing Variables:**

1. When you pass an object variable into a method, you must keep in mind that you're passing the object reference, and not the actual object itself.
2. More importantly, you must remember that you aren't even passing the actual reference variable, but rather a copy of the reference variable(bit patterns and not the value or obj).
3. Both the caller and the called method will now have identical copies of the reference; thus, both will refer to the same exact (not a copy) object on the heap. So pt 36) is valid here also.
4. Java never uses **PASS-BY-REFERENCE** though u r passing a reference variable because. It is because everytime we pass an obj or a primitive variable, we are just passing the bit pattern of the value that represents the value(**Ex:** If I pass a=3; then the method being invoked will get 00000011) So again point no. 36 is applicable here.
5. The bottom line on pass-by-value: The called method can't change the caller's variable, although for object reference variables, the called method can change the object the variable referred to.

**Ex:** void bar() {

Foo f = new Foo();

doStuff(f);

}

void doStuff(Foo g) {

g.setName("Boo");

g = new Foo();

}

Re-assigning g does not reassign f! At the end of the bar() method, two Foo objects have been created: one referenced by the local variable f and one referenced by the local (argument) variable g. (f var change nahi ho raha hai kyuki uska bit pattern g me aa gaya and g new obj ko point kar raha hai).

1. Pg 197 (247 out of 1094) me ek prg. hai usme static size var hai aur ek local size var hai to jo update ho raha hai wo local wala hai and not the static one isliye wapas main() me se call karte time original value hi print hoti hai and not the local one.
2. Pg 198 (248 out of 1094) ke prg. me method ko call karte time wo obj pass kar raha hai so called method me us obj ke instance var ki value 99 me change kar diya fir ek new obj bana jisme 420 value store hui hai. Toh main() me last me jab wapas obj.var ko call karte hai to 99 hi aaega and not 28 kyuki mene us obj ke reference se value change kar di aur 420 isliye nahi aaega kyuki wo naya obj bana tha.

**Garbage Collector:**

1. Phrases such as "garbage collection" or "memory management" are collectively called as Object’s lifecycle.
2. In programming languages such as C, C++ when we write certain prg. that obtains the data, and stores it in a temporary mem. to process it and then store in the DB, so in such cases any fault in manual garbage collection can lead to loss small part of mem(Mem. Leaks). So if the same prg. runs in an iteration then a large part of mem is inaccessible leading the prg. to crash. So in such situations manual garbage collection can double the effort of writing a prg.
3. Java’s garbage collector provides an automatic sol. For mem. Mgmt. But the downside of this is that we can’t completely control when it run’s and when doesn’t.
4. **The heap is that part of memory where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process.**
5. The purpose of a garbage collector is deleting any objects that are no longer reachable by the Java program running.
6. Every Java prg. is in a constant cycle of creating the obj that it requires and then discarding them when not. So a GC looks for those discarded obj and deletes them from the mem. it also might never delete it
7. **When does the GC run?** The GC runs under the control of the JVM. So the JVM decides when to run the GC. We can just req. the JVM to run the GC but no guarantees. The JVM runs the GC when it senses that the mem. Is running low.
8. **How does the GC work?** It might use a **Mark and Sweep algo.**, or a **Reference Counting algo.**  but nothing is guaranteed because the GC can use any algo. for any type of prg.
9. **An object is eligible for garbage collection when no live thread can access it.** If our Java program has a reference variable that refers to an object, and that reference variable is available to a live thread, then that object is considered reachable.
10. The GC removes objects from memory when they are not used. However, if you maintain too many live objects (objects referenced from other live objects), the system can run out of memory.
11. GC cannot ensure that there is enough memory, it ensures that the memory that is available will be managed as efficiently as possible.
12. **How to make Objects eligible for GC:**
    1. **Nulling a Reference:** If there are no reachable references, it doesn't matter what happens to the object. The **first way** to remove a reference to an object is to set the reference variable that refers to the object to null
    2. **Reassigning a Reference Variable:** We can also decouple a reference variable(say var1) from an object (say obj1) by setting the reference variable(var1) to refer to some other object(say obj2).
    3. GC applies even on the local variables within a method. But the variables that are returned aren’t applicable for GC as their reference is passed back to the obj or method that invoked it, but the variables that were declared inside the method and aren’t returned can be GC’ed. **Ex:** Pg 203(253 of 1094)
    4. **Isolating a Reference:** There is another way to make an obj eligible for the Gc even if they have a valid reference alive. This scenario is called **Islands of Isolation.**

A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it can usually discover any such islands of objects and remove them. As you can imagine, such islands can become quite large, theoretically containing hundreds of objects. **Pg 204-205 (255 of 1094)**

* 1. **Forcing GC:** Though it is impossible to force the JVM for GC, however it is possible only to suggest to the JVM that it perform garbage collection.

**NOTE:** GC Routines are a part of Runtime Class which is a special class that has a single obj(singleton) for each main prg. The runtime obj provides a mechanism to communicate with the VM directly.

**Runtime.getRuntime();** will give u the singleton obj which can be used to call gc();

Or the same can be done using a System class instead of a Runtime class (System.gc())

**Point e) and its corresponding note and para are not in the syllabus**

1. **The only thing you can guarantee is that if you are running very low on memory, the garbage collector will run before it throws an OutOfMemoryException.**
2. Java provides a mechanism that lets you run some code just before your object is deleted by the garbage collector. This code is located in a method named finalize() that all classes inherit from class Object. But this method doesn’t guarantee to run because finalize() is called only before the GC is run.
3. **Points to remember:**

■ For any given object, finalize() will be called only once (at most) by the garbage collector.

■ Calling finalize() can actually result in saving an object from deletion.

1. **Note:** Finalize() can have any code that we have in a method like passing a reference to some other obj making the obj ineligible for GC. Now if the flow goes back to the normal code and comes down sequentially then the GC will not run the finalize() for the same obj again(As per the 1st point) and the obj will be deleted even if we didn’t want to.

**Two Minutes Drill:**

1. Static var live as long as the class lives.
2. Instance var live as long as their obj lives.
3. Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
4. Block variables (for example, in a for or an if) live until the block completes.
5. Floating-point numbers are implicitly doubles (64 bits).
6. Narrowing a primitive truncates the high order bits.(right side)
7. Compound assignments (such as +=) perform an automatic cast.

**Chapter 4:**

1. Java operators can’t be overloaded. However, a few exceptional operators that come overloaded:
   1. The + operator can be used to add two numeric primitives together or to perform a concatenation operation if either operand is a String.
   2. The &, |, and ^ operators can all be used in two different ways, although on this version of the exam, their bit-twiddling capabilities won't be tested.
2. **Compound Assignment Operator (short hand operator):** +=, -=, \*=, /=
3. The \* and / are higher precedence than + and -.
4. the expression on the right side of the = will always be evaluated first.
5. **Relational Operator:**  <, <=, >, >=, ==, and !=
6. Relational operators always result in a boolean (true or false) value
7. **Relational Operators are used to compare an int with a char or a float and vice versa.**
8. == and != are also called as equality operators.
9. All the relational operators return a Boolean value after the comparison of the operands.
10. These primitive types and object reference can’t be compared with any other types like String(Incompatible types.
11. There are four different types of things that can be tested:

■ Numbers

■ Characters

■ Boolean primitives

■ Object reference variables

1. Only for Boolean we can even assign the values within the condition of looping or conditional stmts. But that will be an infinite loop as we have assigned it there only so it will obviously be true.
2. **Remember, the == operator is looking at the bits in the variable, so for reference variables, this means that if the bits in both reference variables are identical, then both refer to the same object.**
3. **== and equals() words the same way for the objects of a class but for String objects it works in different way.** For String’s equals() the values are case sensitive.
4. **Comparing enums:** Enums can be compared for equality using both == and equals().
5. **Instanceof comparison:** The instanceof operator is used for object reference variables only, and you can use it to check whether an object is of a particular type(class or interface).
6. You can test an object reference against its own class type or any of its superclasses.
7. Instanceof() displays compilation error only when we use it between 2 different hierarchies.
8. Arrays are also objects so it is an instance of the object class.
9. If u initialize and array of class type the array obj will be true if it is compared with the parent class with a ([]) or directly with an obj.
10. String Concatenation operator can give two different outputs based on the parenthesis. Ex: a=”fdfs”+4+3; **o/p:** fdfs43 Ex: a=”fdfs”+(4+3); **o/p:** fdfs7
11. **If either operand is a String, the + operator becomes a String concatenation operator. If both operands are numbers, the + operator is the addition operator.**
12. The conditional operator is a *ternary* operator (it has *three* operands) and is used to evaluate boolean expressions, much like an if statement, except instead of executing a block of code if the test is true, a conditional operator will assign a value to a variable. In other words, the goal of the conditional operator is to decide which of two values to assign to a variable.

**Syntax:** x = (boolean expression) ? value to assign if true : value to assign if false

1. You can even nest conditional operators into one statement :
   1. String status = (numOfPets<4)?"Pet count OK":(sizeOfYard > 8)? "Pet limit on the edge":"too many pets";

**NOT IN SYLLABUS:**

1. **Logical Operators:** **Bitwise:** &, |, ^, **Short-Circuit LO:** &&, and || **!**
2. **Bitwise operators** compare two variables bit-by-bit and return a variable
3. **Short-Circuit Operators** does comparison in short way like if we use && with two Boolean expressions then if the 1st expression is a false then it doesn’t even bother to compare the next expression. And in || if one expression is true it doesn’t check the 2nd expression.
4. **The || and && operators work only with boolean operands. The exam may try to fool you by using integers with these operators: Ex:**if (5 && 6) { }
5. **Logical Operators(Not Short-circuit):** &, |
6. Not short-circuit operators aren't the short-circuit operators, they evaluate both sides of expr.
7. **Logical Operators:** ^(xor), !(Boolean invert)
8. The ^ (exclusive-OR) operator evaluates only boolean values. For an exclusive-OR (^) expression to be true, EXACTLY one operand must be true.

**Two Minutes Drill:**

1. When comparing characters, Java uses the Unicode value of the character as the numerical value.
2. In Equality Operators four types of things can be tested: numbers, characters, booleans, and reference variables.
3. When comparing reference variables, == returns true only if both references refer to the same object.
4. instanceof is for reference variables only.
5. Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
6. Prefix operators (for example, ++x and --x) run before the value is used in the expression.
7. Postfix operators (for example, x++ and x--) run after the value is used in the expression
8. Variables marked final cannot be incremented or decremented.
9. Logical operators work with two expressions (except for !) that must resolve to boolean values.
10. In Concatenation operator if there are 3 operands but the 1st 2 are int and 3rd is string then the 1st 2 operands are added and then it gets concatenated with the string.
11. If u have the short-circuit operator followed by the not short circuit then the 1st and 3rd part will execute but not the 2nd part. **Ex:** if( ((5<7) || (++count < 10)) | mask++ < 10 )
12. But if the same condition is with an extra pair of braces then only first part will execute and nothing else.

**Ex:** if( ((5<7) || ((++count < 10)) | mask++ < 10 ))

**LESSON 5:**

1. The StringBuilder class and StringBuffer class has almost similar methods but the only difference is that StringBuilder is faster as it’s methods aren’t synchronized, but both are mutable.
2. In java each character in a string is a 16 but Unicode char. Unicode characters are 16 bits (not the skimpy 7 or 8 bits that ASCII provides), a rich, international set of characters is easily represented in Unicode.
3. String object is immutable so if we give String s=”shubh”; s.concate(“hey”); it won’t change.
4. If we give String s=”shubh”; s=s.concate(“hey”); then it will because the reference variable s earlier referenced to shubh but later shubh object is lost as the s reference variable is referencing to a new string shubhhey.
5. Here total 8 string obj and 2 reference variables are created.

String s1 = "spring ";

String s2 = s1 + "summer ";

s1.concat("fall ");

s2.concat(s1);

s1 += "winter ";

System.out.println(s1 + " " + s2);

**8String Objects:** "spring ", "summer " (lost), "spring summer ", "fall " (lost), "spring fall " (lost), "spring summer spring " (lost), "winter " (lost), "spring winter " (at this point "spring " is lost).

1. String literals occupy large amounts of a program's memory, and there is often a lot of redundancy of String literals in a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the *String constant pool*.
2. When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created.(The existing String simply has an additional reference.) This is y the String obj are immutable so that changes doesn’t affect other var.
3. We can’t override any of the String class’ method as the String class is marked final and as we know that a final class can neither be inherited not can be overridden.
4. String s = "abc"; Will create a string obj in the pool of memory.
5. String s = new String("abc"); Will create a new String obj ina non pool area of the memory
6. **Methods in String Class:**

■ **charAt()** Returns the character located at the specified index (String starts with zero)

■ **concat()** Appends one string to the end of another (+ also works)

■ **equalsIgnoreCase()** Determines the equality of two strings, ignoring case

■ **length()** Returns the number of characters in a string

■ **replace()** Replaces occurrences of a character with a new character

■ **substring()** Returns a part of a string. If it has one argument then it will create a substring from that index and if it has 2 arg then then 2nd will be considered as position and not index.

■ **toLowerCase()** Returns a string, with uppercase characters converted to lowercase

■ **toString()** Returns the value of a string

■ **toUpperCase()** Returns a string, with lowercase characters converted to uppercase

■ **trim()** Removes whitespace from both ends of a string

1. **Arrays have an attribute called length whereas a String class has a method called length(). This is the reason that while accessing the length of an array we just give it as arrayobj.length whereas for string we use stringobj.length().**
2. A StringBuilder class is used when a lot of modifications needs to be done on a string.
3. A common use for StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program.
4. String obj wastes memory when we change any string. So the old string obj is lots thus wasting the memory whereas a string builder or buffer will change the same obj thus saves mem.
5. So far we've seen StringBuilders being built with an argument specifying an initial value. StringBuilders can also be built empty, and they can also be constructed with a specifi c size or, more formally, a "capacity." For the exam, there are three ways to create a new StringBuilder:

new StringBuilder(); // default cap. = 16 chars

new StringBuilder("ab"); // cap. = 16 + arg's length

new StringBuilder(x); // capacity = x (an integer)

1. The two most common ways to work with StringBuilders is via an append() method or an insert() method. In terms of a StringBuilder's capacity, there are three rules to keep in mind when appending and inserting:

• If an append() grows a StringBuilder past its capacity, the capacity is updated automatically.

• If an insert() starts within a StringBuilder's capacity, but ends after the current capacity, the capacity is updated automatically.

• If an insert() attempts to start at an index after the StringBuilder's current length, an exception will be thrown.

1. **Methods in StringBuilder Class:**

■ **append()** Returns the string with the appended value passes as the argument. including boolean, char, double, float, int, long, and others

■**delete(int start, int end)** Here the starting index is 0-based but the end is 1-based.

■**insert(int offset, String s)** Here the offset is a 0-based index.

■**reverse()** Reverses theString

■**toString()** This method returns the value of the StringBuilder object that invoked the method call as a String

1. Arrays are objects in Java that store multiple variables of the same type. Arrays can hold either primitives or object references, but the array itself will always be an object on the heap, even if the array is declared to hold primitive elements. In other words, there is no such thing as a primitive array, but you can make an array of primitives.
2. **Arrays:** An array can be declared with the ([]) just after the type of the array or after the variable name but the former is better than the latter.
3. The size should not be mentioned in the declaration as it is illegal to do so and will throw compilation error.

**Note:** Remember, the JVM doesn't allocate space until you actually instantiate(Construct) the array object. That's when size matters.

1. **Constructing** an array means creating the array object on the heap. So here the size needs to be specified. Once the size is specified and if we try to add or access values greater the size then it throws a runtime exception.

Ex: int[] carList = new int[]; Short-hand: int[] carList = {23,45,323,234};

1. **Multi-Dimensional Arrays:** These are the arrays of arrays. (ie Each element of an array holds the reference of another array) **Ex:** int[][] myArray = new int[3][];

int[][] scores = new int[3][];

scores[0] = new int[4];

scores[1] = new int[6];

scores[2] = new int[1];

**Note:** Only the first brackets are given a size. That's acceptable in Java, since the JVM needs to know only the size of the object assigned to the variable myArray.

1. **Initializing** an array means storing things in it. If we create an array of primitive type then each element of array will hold the value but if we create an array of object type then each element will hold the reference of the obj.
2. Array objects have a single public variable, length, that gives you the number of elements in the array. The last index value, then, is always one less than the length
3. int x = 9; int[] dots = {6,x,8}; It is legal to initialize an array element with an explicit value.
4. Why would anyone use the longer way to initialize an array is that you might not know—at the time you create the array—the values that will be assigned to the array's elements

**Ex:** Dog puppy = new Dog("Frodo"); Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")}; //Legal

**Ex:** int[][] scores = {{5,2,4,7}, {9,2}, {3,4}}; // This code creates a total of four objects on the heap: 1) scores(with length as 4) 2) 1st pair of ({}) 3) 2nd pair of ({}) 4) 3rd pair of ({})

1. **Anonymous array creation:** You can use it to create a just-in-time array. Should not assign size to the array here. The size is derived from the number of items.

**Ex:** int[] testScores; testScores = new int[] {4,7,2}; //Legal

1. Primitive arrays can also have elements of some other primitive type unless it is smaller than the declared type.
2. **In an array of class type can have elements as the objects of the subclasses but not the objects of its superclasses.**
3. Unlike the primitive variables we can’t assign an int array to a float array. Or likewise. Neither implicitly nor explicitly. Not even to its wrapper class types.
4. **ArrayList:** Reasons touse an ArrayList over an array:

■ You need to be able to increase and decrease the size of your list of things.

■ The order of things in your list is important and might change.(Insertion in between)

1. ArrayLists hold only object references, not actual objects, and not primitives. If you see code like this, myArrayList.add(7);

what's really happening is that the int is being autoboxed (converted) into an Integer object and then added to the ArrayList.

1. **Methods in ArrayList:**

■ **add(element)** Adds this element to the **end** of the ArrayList

■ **add(index, element)** Adds this element at the index point and shifts the remaining elements back (for example, what was at index is now at index + 1)

■ **clear()** Removes all the elements from the ArrayList

■ **boolean contains(element)** Returns whether the element is in the list

■ **Object get(index)** Returns the Object located at index

■ **int indexOf(Object)** Returns the (int) location of the element, or -1 if the Object is not found

■ **remove(index)** Removes the element at that index and shifts later elements toward the beginning one space

■ **remove(Object)** Removes the **first** occurrence of the Object and shifts later elements toward the beginning one space

■ **int size()** Returns the number of elements in the ArrayList

**Encapsulation:**

1. In a StringBuilder any changes made to the SB obj will reflect back to the original obj. This is because we have only one SB object which is being referenced by 2 different reference variables. **Ex: Pg 294**

**Two Minutes Drill:**

1. When the JVM finds a String literal, it is added to the String literal pool.
2. StringBuilder methods act on the invoking object, and objects can change without an explicit assignment in the statement. Hence they are mutable.
3. Elements in an array of objects are not automatically created, although primitive array elements are given default values.
4. An array of primitives can accept any value that can be promoted implicitly to the array's declared type—for example, a byte variable can go in an int array.
5. An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
6. You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).
7. ArrayLists can hold only objects, not primitives, but remember that autoboxing can make it look like you're adding primitives to an ArrayList when in fact you're adding a wrapper version of a primitive.
8. You can store any dimension of an array in a 2-D or a 3-D array by explicitly typecasting it into a 1-D array.
9. You can store a 2-D array in an object without even typecasting.
10. But object can’t be stored into an array even if u explicitly typecast it.
11. String me after passing a string to another string var. the change in one obj will not effect in the other but in string buffer or builder it reflects back.

**Chapter 5:**

1. **Decision statements** are the if and switch statements.
2. There are a couple of rules for using else and else if:

■ You can have zero or one else for a given if, and it must come after any else ifs.

■ You can have zero to many else ifs for a given if and they must come before the (optional) else.

■ Once an else if succeeds, none of the remaining else ifs nor the else will be tested.

1. The expression in an if statement must be a boolean expression. Any expression that resolves to a boolean is fine
2. The only legal expression in an if test is a Boolean.
3. A **switch's** expression must evaluate to a char, byte, short, int, an enum (as of Java 5), and a String (as of Java 7). You can’t use long, float, double in a switch case.
4. switch(new Integer(4)) { **// It is legal**

case 4: System.out.println("boxing is OK");

}

1. The case can have a pair of curly braces for itself but the word case can’t be omitted.
2. for a switch statement, two Strings will be considered "equal" if they have the same case sensitive sequence of characters.
3. When the program encounters the keyword break during the execution of a switch statement, execution will immediately move out of the switch block to the next statement after the switch. If break is omitted, the program just keeps executing the remaining case blocks until either a break is found or the switch statement ends.
4. **Fall-through** is defined as a switch case running through all the cases as it can’t detect a break statement within it.

**Note:** Because fall-through is less than intuitive, Oracle recommends that you add a comment such as // fall through when you use fall-through logic.

1. Even if we use a default case somewhere in between all the other cases the fall-through will still work in the same way for the default case as well.
2. **Looping:**
3. **While loop** is used when you want to continue looping as long as some condition is true.
4. The body of the while loop will execute only if the expression (sometimes called the "condition") results in a value of true. Once inside the loop, the loop body will repeat until the condition is no longer met because it evaluates to false.
5. The **do loop** is similar to the while loop, except that the expression is not evaluated until after the do loop's code is executed. Therefore, the code in a do loop is guaranteed to execute at least once. **Ex:** do{}while(condition);
6. Be sure to note the use of the semicolon at the end of the while expression.
7. **For loops** are of 2 types: 1) basic for loop 2) Enhanced for or for-each or for-in loop.
8. The basic for loop is more flexible than the enhanced for loop, but the enhanced for loop was designed to make iterating through arrays and collections easier to code.
9. The for loop is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block. The for loop declaration has three main parts, besides the body of the loop:

■ Declaration and initialization of variables

■ The boolean expression (conditional test)

■ The iteration expression

The three for declaration parts are separated by semicolons.

1. The first part of the for statement lets you declare and initialize zero, one, or multiple variables of the **same type** inside the parentheses after the for keyword.
2. The conditional expression within the for loop can contain only one logical expression but can have a complex Boolean condition.
3. The iteration expression can contain 0 or multiple expressions.
4. Keep in mind that barring a forced exit, evaluating the iteration expression and then evaluating the conditional expression are always the last two things that happen in a for loop!
5. Examples of **forced exits** include a break, a return, a System.exit(), and an exception, which will all cause a loop to terminate abruptly, without running the iteration expression
6. None of the three sections of the for declaration are required

for( ; ; ) { //It is an endless loop

System.out.println("Inside an endless loop");

}

1. **Forced Exits:**
   1. break Execution jumps immediately to the first statement after the for loop.
   2. return Execution jumps immediately back to the calling method.
   3. System.exit() All program execution stops; the VM shuts down.
2. With the absence of the initialization and increment sections, the loop will act like a while loop

int i = 0;

for (;i<10;) {

i++;

// do some other work

}

1. **The 3rd part of a for loop should not be mandatorily an iteration, it can be anything that u want to do after each execution of the loop.**
2. **Enhanced for loop:** It is used to iterate through an array or a collection. **Ex:** for(declaration:expression)
3. Declaration consists of the variable that has the same type as the expression and which can be accessed only within the loop and not out of the loop.
4. The expression can either be a collection or any other method call that returns a collection.
5. Enhanced for loop me u can’t declare a variable outside and use it in the for loop condition.
6. If you're using break or continue, you'll do an if test within the loop, and if some condition becomes true (or false depending on the program), you want to get out immediately. The difference between them is whether or not you continue with a new iteration or jump to the first statement below the loop and continue from there
7. The break statement causes the program to stop execution of the innermost loop and start processing the next line of code after the block.
8. The continue statement causes only the current iteration of the innermost loop to cease and the next iteration of the same loop to start if the condition of the loop is met.
9. Both the continue and break stmts can be either labeled or unlabeled. Unlabled is just giving the words break or continue but labeled are those where we give a specific name(like a checkpoint) that could be used by the continue to start from. **Ex:** label: (before loop)

foo:

for (int x = 3; x < 20; x++) {

while(y > 7) {

y--;

}

}

**Questions from video 1:**

1. String me equals() and ==.
2. System.arrayCopy()

**Importants jo tujhe wapas dekhne hai:**

Pg 204 wala prg.

1. Encapsulation samajh nahi aa raha hai wapas ache se samjhna
2. Tu complex operators ko alag alag pattern me dalkar dekh bohot complications hai isme.

Points:

1. Local variables of Primitive types can’t be left uninitialized in the main() whereas The reference variables can be left uninitialized
2. Whenever u pass an object to sysout() then it internally calls the object’s toString().
3. Pay close attention for misdirection like the following:

if (exam.done())

if (exam.getScore() < 0.61)

System.out.println("Try again.");

Else //Yet it belongs to the innermost if only

System.out.println("Java master!");

1. The variables that were declared and initialized within the loop can’t be accessed out of the loop but if the variable is declared before the loop but initialized in the loop can be accessed.