**Fundamental Question of Java language**

**Q1) What is a class and an object in java?**

Class 🡪 A template that describe the kind of state and behavior its object support

Objects 🡪 At runtime when the JVM encounters a **new** keyword, it will use an appropriate class to make an object which is instance of that class. That object will have its own state and access to all of the behaviors defined by the class.

**Q2) Name some access modifiers and non-access modifiers (access specifires)?**

Access modifiers 🡪

public private

default protected

Non-Access modifiers 🡪

final transient

static synchronized

abstract native

**Q3) List types of variables in java with their default values?**

Variables in java can be

**Primitive** variables

🡪 These are the **char, boolean, byte, short, int, long, float, double**

**Reference** variables

* The variables referencing an object are called reference variables, they **can be static, local or instance**

**Instance** variables

* Variables declared at class level are instance variables
* When a class is instantiated the instance variables are initialized with the below values:

o char - \u0000 (Hex representation of 0)

o boolean – false o long – 0L

o byte – 0 o foat – 0.0f

o short - 0 o double – 0.0d

o int - 0 o String – null

🡪 Instance variables can be **final**, and **can** have **default**, **public**, **private** and **protected** access modifiers.

**Local** variables

* Variables declared inside a method are known as local variables
* They **do not** get and **default** values and cannot be accessed outside the method they are declared in.
* They **cannot** have access modifiers like **public**, **protected**, **private**, **transient**, **abstract** and **static**. However they **can only be final**.

**Note**: **Instance** variable are created in **HEAP** and **local** variables are created in **STACK**

**Q4) What is method overloading?**

Method Overloading means to have two or more methods with same name in the same class with different arguments. The benefit of method overloading is that it allows you to implement methods that support the same semantic operation but differ by argument number or type.  
Note:

* 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

Q5) **What is Constructor**?

* A constructor is a special method whose task is to initialize the object of its class.
* It is special because its name is the **same as the class name**.
* They do not have any return type, not even **void** and therefore they cannot return values.
* They **cannot be inherited**, though a derived class can call the base class constructor.
* Constructor is invoked whenever an object of its associated class is created.
* Constructors can have all the access modifiers , public, private , protected and default

Q6) **How does the Java default constructor be provided?**

If a class defined by the code does **not** have any constructor, compiler will automatically provide one no-parameter-constructor (default-constructor) for the class in the byte code. The access modifier (public/private/etc.) of the default constructor is the same as the class itself.

Q7) **Can constructor be inherited**?

**No**, constructor cannot be inherited, though a derived class can call the base class constructor.

Q8) **What are the differences between Contructors and Methods?**

|  |  |  |
| --- | --- | --- |
|  | **Constructors** | **Methods** |
| **Purpose** | Create an instance of a class | Group Java statements |
| **Modifiers** | Cannot be *abstract, final, native, static*, or *synchronized* | Can be *abstract, final, native, static*, or *synchronized* |
| **Return Type** | No return type, not even void | void or a valid return type |
| **Name** | Same name as the class (first letter is capitalized by convention) -- usually a noun | Any name except the class. Method names begin with a lowercase letter by convention -- usually the name of an action |
| *this* | Refers to another constructor in the same class. If used, it must be the first line of the constructor | Refers to an instance of the owning class. Cannot be used by static methods. |
| *super* | Calls the constructor of the parent class. If used, must be the first line of the constructor | Calls an overridden method in the parent class |
| **Inheritance** | Constructors are not inherited | Methods are inherited |

Q9) **How are this() and super() used with constructors**?

* Constructors use *this* to refer to another constructor in the same class with a different parameter list.
* Constructors use *super* to invoke the superclass's constructor. If a constructor uses *super*, it must use it in the first line; otherwise, the compiler will complain.

**Q10) What are the differences between Class Methods and Instance Methods?**

|  |  |
| --- | --- |
| **Class Methods** | **Instance Methods** |
| Class methods are methods which are declared as **static**. The method can be called without creating an instance of the class | Instance methods on the other hand require an instance of the class to exist before they can be called, so an instance of a class needs to be created by using the **new** keyword. Instance methods operate on specific instances of classes. |
| Class methods can only operate on class members and not on instance members as class methods are unaware of instance members. | Instance methods of the class can also not be called from within a class method unless they are being called on an instance of that class. |

**Q11) What are Access Specifiers?**

One of the techniques in object-oriented programming is ***encapsulation***. It concerns the hiding of data in a class and making this class available only through methods. Java allows you to control access to classes, methods, and fields via so-called ***access specifiers*.**.

**Q12) What are Access Specifiers available in Java?**

Java offers four access specifiers, listed below in decreasing accessibility:

* **Public**- *public* classes, methods, and fields can be accessed from everywhere.
* **Protected**- *protected* methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package.
* **Default(no specifier)-** If you do not set access to specific level, then such a class, method, or field will be accessible from inside the same package to which the class, method, or field belongs, but not from outside this package.
* **Private**- *private* methods and fields can only be accessed within the same class to which the methods and fields belong. *private* methods and fields are not visible within subclasses and are not inherited by subclasses.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Situation | public | protected | default | private |
| Accessible to class   from same package? | yes | yes | yes | no |
| Accessible to class   from different package? | yes | no, *unless it is a subclass* | no | no |

Q13) **What is final modifier**?

The final modifier keyword makes that the programmer cannot change the value anymore. The actual meaning depends on whether it is applied to a class, a variable, or a method.

* *final* **Classes**- A final class cannot have subclasses.
* *final* **Variables**- A final variable cannot be changed once it is initialized.
* *final* **Methods**- A final method cannot be overridden by subclasses.

Q15) **What is static block?**

Static block which exactly executed exactly once when the class is first loaded into JVM. Before going to the main method the static block will execute.

Q16) **What are static variables?**

Variables that have only one copy per class are known as static variables. They are not attached to a particular instance of a class but rather belong to a class as a whole. They are declared by using the static keyword as a modifier.

static type varIdentifier;

where, the name of the variable is varIdentifier and its data type is specified by type.

Note: Static variables that are not explicitly initialized in the code are **automatically initialized** with a default value which same as that of instance variables.

Q17) **What is the difference between static and non-static variables?**

A static variable is associated with the class as a whole rather than with specific instances of a class. Non-static variables take on unique values with each object instance.

Q18) **What are static methods?**

Methods declared with the keyword static as modifier are called static methods or class methods. They are so called because they affect a class as a whole, not a particular instance of the class. Static methods are always invoked without reference to a particular instance of a class.  
Note:The use of a static method suffers from the following restrictions:

* A static method can only call other static methods.
* A static method must only access static data.
* A static method **cannot** reference to the current object using keywords *super* or *this*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Static Methods | Non-Static Methods | Static Variables | Non-Static Variables |
| Static Method | yes | no | yes | no |
| Non-Static Methods | yes | yes | yes | yes |

**Q19) What are the uses of final method?**

There are two reasons for marking a method as final:

* Disallowing subclasses to change the meaning of the method.
* Increasing efficiency by allowing the compiler to turn calls to the method into inline Java code.

**Q20)** What **is transient and volatile (keywords) variables?**

* Java defines two interesting type modifiers: **transient** and **volatile**.

These modifiers are used to handle somewhat **specialized** situations.

* When an instance variable is declared as transient, then its value need not persist when an object is stored.

For example:

class T {

**transient** int a; // will not persist

int b; // will persist

}

Here, if an object of type T is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.

* The **volatile** modifier tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of your program. One of these situations involves **multithreaded** programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable.
* The real (or master) copy of the variable is updated at various times, such as when a synchronized method is entered. While this approach works fine, it may be inefficient at times.
* In some cases, all that really matters is that the master copy of a variable always reflects its current state.
* To ensure this, simply specify the variable as volatile, which tells the compiler that it must always use the master copy of a volatile variable (or, at least, always keep any private copies up-to-date with the master copy, and vice versa). Also, accesses to the master variable must be executed in the precise order in which they are executed on any private copy.

**Q21) What are real-world practices of volatile modifier?**

* One of the real-world use of the volatile variable is to generate interpretation double and long atomic. Equally double and long are 64-bit extensive and they are reciting in two parts, primary 32-bit first time and following 32-bit another time, which is non-atomic but then again volatile double in addition long read is atomic in Java. Additional use of the volatile variable is to deliver a recall barrier, just like it is cast-off in Disrupter framework.
* Fundamentally, Java Memory model pull-outs a write barrier subsequently you write to a volatile variable besides a read barrier beforehand you read it. That means, if you inscribe to volatile field then it’s definite that any thread retrieving that variable will see the worth you wrote and everything you did beforehand doing that correct into the thread is certain to have occurred and any rationalized data values will also be noticeable to all threads, since the memory barrier flushed all additional writes to the cache.

**Q22) What is the difference between abstraction and encapsulation?**

* **Abstraction**

Abstraction refers to the act of representing essential features without including the background details or explanations.

The meaning of abstract is, existing in thought or as an idea but not having a physical or concrete existence

* **Encapsulation**

Encapsulation is a technique used for hiding the properties and behaviors of an object and allowing outside access only as appropriate. It prevents other objects from directly altering or accessing the properties or methods of the encapsulated object.

* **Difference**
* Abstraction focuses on the outside view of an object (i.e. the interface) Encapsulation (information hiding) prevents clients from seeing it’s inside view, where the behavior of the abstraction is implemented.
* Abstraction solves the problem in the design side while Encapsulation is the Implementation.
* Encapsulation is the deliverables of Abstraction. Encapsulation barely talks about grouping up your abstraction to suit the developer needs.

**Q23) What is association?**

Association is a relationship where all object have their own lifecycle and there is no owner. Let’s take an example of **TEACHER** and **STUDENT**. Multiple students can associate with a single teacher and a single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. This relationship can be one to one, one to many, many to one and many to many.

**Q24) What do you mean by aggregation?**

Aggregation is a specialized form of Association where all object have their own lifecycle but there is ownership and child object cannot belongs to another parent object. Let’s take an example of **DEPARTMENT** and **TEACHER**. A single teacher cannot belong to multiple departments, but if we delete the department teacher object will not destroy.

**Q25) What is composition in Java?**

Composition is again specialized form of Aggregation and we can call this as a “**death**” relationship. It is a strong type of Aggregation. Child object does not have their lifecycle and if parent object is deleted all child objects will also be deleted. Let’s take again an example of relationship between **SCHOOL** and **CLASS ROOMS**. House can contain multiple rooms there is no independent life of room and any room cannot belongs to two different house if we delete the house room will automatically delete.

**Q) Why is the Equals method and the hashcode method, why is hashcode after equals is required?**

**Override only equals**

* If only equals is overridden, then when you call **myMap.put(first,someValue)** first will hash to some bucket and when you call **myMap.put(second,someOtherValue)** it will hash to some other bucket (as they have a different hashCode). So, although they are equal, as they don't hash to the same bucket, the map can't realize it and both of them stay in the map.

**Override only hashCode**

Imagine you have this

MyClass first = new MyClass("a","first");

MyClass second = new MyClass("a","second");

* If you only override hashCode then when you call myMap.put(first,someValue) it takes first, calculates its hashCode and stores it in a given bucket. Then when you call myMap.put(second,someOtherValue) it should replace first with second as per the Map Documentation because they are equal (according to the business requirement).
* But the problem is that equals was not redefined, so when the map hashes second and iterates through the bucket looking if there is an object k such that second.equals(k) is true it won't find any as second.equals(first) will be false.

**Q) How hash code and equals method works internally?**

**Q) What are the Equals & Hashcode method contracts?**

**Equals Contract:**

* 1. **Reflexivity** - For any object X, X.equals(X) should return true.
  2. **Symmetry** - For any two objects X and Y, X.equals(Y) should return true if Y.equals(X) returns true.
  3. **Transitivity** - For any objects X,Y and Z, if X.equals(Y) return true and Y.equals(Z) returns true then X.equals(Z) should return true.
  4. **Consistency** - For multiple invocations of X.equals(Y) should return same result unless any of the object properties is modified that is being used in the equals() method.

**HashCode Contract:**

Java Object hashCode() is a native method and returns the integer hash code value of the object. The general contract of hashCode() method is:

* Multiple invocations of hashCode() should return the same integer value, unless the object property is modified that is being used in the equals() method.
* An object hash code value can change in multiple executions of the same application.
* If two objects are equal according to equals() method, then their hash code must be same.
* If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.

**Q) What is Hash Collision?**

The phenomenon when two **keys** have **same hash code** is called hash collision. If hashCode() method is not implemented properly, there will be higher number of hash collision and map entries will not be properly distributed causing slowness in the get and put operations. This is the reason for prime number usage in generating hash code so that map entries are properly distributed across all the buckets.

**Q) Explain Un-Boxing and Auto-Boxing?**

Un-Boxing is a automatic conversion of wrapper objects to primitive.

Auto-Boxing is automatic conversion of primitives to wrapper objects.