Java OOP

Java Interview Guide : 200+ Questions

Interfaces

- Interface represent common actions between Class Hierarchies or classes
- Why should you prefer coding to interface rather than implementations
  - Interfaces are contracts can get a feel for the developer's intent
  - Loose Coupling : Can change the underlying data structure.
- Java 1.8 : Interfaces can supply a default method – A method with an implementation inside the interface
  - Example
    - Interface ExampleInterfrace1 {
      - default void method1() {
        - System.out.println("1"); } }
  - created to allow interfaces to leverage lambda expression of java 8 without implementing methods in the implementation class
    - Now list and collection can have a foreach instead fo the foreach needed every class that implements or needing to create an adapter
    - Better then C++ Multiple Inheritance since if the function have the same function name then append the class name
      - Example
      - interface **TestInterface1** { default void show() { } } and interface **TestInteface2** { default void show {})
      - public Test implement TestInterface1, TestInteface2 { public void show1() { TestInterface1.show(); }
  - Enable addition of new functionalities to interfaces without breaking the classes that implement the interface
- Any method describe in an interface is public and abstract
  - Public interface Flyable { void fly(); }
- Variables in an interface are always public , static and final
- Can extend an interface
  - Interface SubInterface1 extends ExampleInterface1 { void method3(); }

Interfaces – Default

- Use Case : Suppose you have an interface which has been implemented by ten classes. After a couple of years you want to introduce a new method in this interface
  - All Ten class will get impacted and it will give compilation errors
  - An interface is a contract and we need to inform all stakeholders
    - Solution
      - We need to implement a new method in all ten class which is practically very difficult.
      - Use default Method
      - Provides common functionality which can be reused in all implementing classes
      - Example
        - public interface DefaultMethodDeomFromJava8 {
          - default public int addTwoNumber(int number1, int number2) {
          - Int sum = number1 + number2
          - return sum
          - }
        - }
          - class Class1 implements DefaultMethodDemoFromJava8 {
            - public int addTwoNumber(int number1, intNumber2) { return addTwoNumber(number1, number2) }
        - }
- A default Method is a method defined in an interface with the keyword default
- A default method is used to provide common functionality which can be reused in all implementing class
- A default method in an interface should have the body and it cannot be empty
- It is not compulsory for implementing classes to override a default method
- The class that implements the interface inherits the interface default methods
- We can have any number of default methods in the interface
- Example
  - Java introduced a new interface Stream in Java 8
  - In Java 9 Java added two new default methods takeWhile, dropWhile
  - Stream is being use by 1000’s of classes

Interfaces – Static

- A static method is a method defined in an interface with keyword static and is used to provide utility method in an interface
- cannot override them in implementation classes which make it good for security
- A static method is used to provide common functionality which can be reused in all implementing classes
- We can have any number of static methods in an interface and cannot use a name of an Object Class Method
- A static method in an interface should have the body and it can not be empty
- Works like a normal static which mean that the static function has no access to “this”
- Usually Utility and Helper methods
- Example
  - In Java 8 introduced Stream and in Java 9 introduced new static methods
    - ofNullable
    - Iterate
- Abstract class still have a small advantage over java 8 interfaces since abstract classes can have protected and they can have a constructor
- Use for methods that apply to instances of that interface

Interfaces – Conflicts Java 8
Java 9 – Private Method in Interface

- Example
  - public interface Int1 { default int SomeMethod() { return(5); } }
  - public interface int2 (default int SomeMethod() { return(6) } }
  - public class ParentClass { public Int someMethod() { return (3); }
  - Question : public class SomeClass implements Int1, Int2 { }
    - Must Include the name of the Interface with the function
    - cannot be resolved automatically
  - Question : public class ChildClass extends parentClass implements Int1 {}
    - Version of someMethod from parentClass wins over the version from Int1
    - resolved automatically
- Java 9 – Private Method in Interface
  - Why private method in interface
    - private only to the interface. Only functions inside the interface can use them
    - Example
- interface from Java 9 {
  - static void displayRandomNumber() { System.out.println(“Random Number is ” + generateRandomNumber); }
  - private static int generateRandomNumber() { randomNumber = 3; }
  - }

Constructors

- When you write a constructor the default constructor is overwritten and cannot be used anymore
- A constructor can call the parent constructor → super()
  - Should be the first line of a constructor
  - super() get called even when there it is not explicitly called
- this(<arguments>) → Call a constructor from another constructor int the same class
- Default constructor → A constructor provided when there the user does not provide a constructor.
- Why should you avoid calling abstract methods inside the constructor
  - Problem is initialization order. The subclass will not have had a chance to run yet and there is no way to force it to run before the parent class
    - The class is not fully initialized and should not be calling abstract methods.
  - Example
    - Public abstract class Widget
      - private final int cacheWidth; private final int cachedHeight;
      - Public Width() { this.cachedWidth = width(); this.cachedHeight = height() }
      - Protected abstract int width();
    - Protected abstract int height
  - Public class SquareWidget extends Widget
    - Private final int size;
    - Public SquareWidget(int size) { this.size = size }
    - @override protected int width() { return size; }
    - @override protected int height() { return height; }
- Bug: For Widget.cachedWidth and Widget.cachedHeight will always be 0 for SquareWidget

Polymorphism

- Polymorphism
  - Example
    - Class Animal { public void speak() { “??” }
    - Class Cat extends Animal { public void speak() { “Meow”}}
    - Class Dog extends Animal { public void speak() {“woof”}}
    - Animal animal = new Cat();
    - Cat.speak() → Meow
- InstanceOf → Can be used to see if an interface is implemented.
  - Looked at the instanceOf example.
- Type of Polymorphism
  - Compile Time Polymorphism : Method Overloading
  - Runtime Polymorphism : Method Overriding
- Polymorphism via class inheritance
  - When an object is polymorphic acting as another object the more specific object is restricted to only using the interface of the more general object
  - Only way to assign an object to a object to an abstract datatype is by using polymorphism
- Polymorphism via interfaces
  - By using the interface the developer has provided a functionality requirement not a strict object data type
- Dynamic binding → Association of function call to function definition during run time.
- Runtime polymorphism : Action Mapping in web app, generics

Abstract Class

- Differences between Abstract Class and Interface
  - Method of an interface can only be public
  - An interface extending another interface need not provide default implementation for methods inherited from the parent interface
  - A child class can only extend a single class, An interface can extend multiple interfaces and a class can implement multiple interface
  - A child class can define abstract methods with the same or less restrictive visibility whereas a class implementing the interface must define all interface methods as public

Inner Classes and Static Inner Class

- Inner Classes are classes which are declared inside other classes.
  - Can access the outer class variables.
  - Because an inner class is associated with an instance, it cannot define any static members itself
- Static inner class → A class inside another class and declared as static
  - Static member variables are not static
  -
  - In a static nested class you cannot access the outer class variables.
- Anonymous Class → A class that does not have a name
  - Example
    - New Comparator<String>() { public int compare(String string1, String string2) { return string2.compareTo(string1); } };
  - Can be used to override a method without creating a new class.
  - Don't know the name of the class, so you cannot use it anywhere. Use it where you will only use it once
- Why use nested classes
  - Simplifies Coding coding
  - Group class that are only used in one place
  - Increases encapsulation
    - Consider two classes A and B where B needs access to members of A that would otherwise be declared private
    - Hide class b into A
      - A members could still be declared private and B can access them
      - B can be hidden from the outside world
- Increases readability

Inner classes and Static Classes

- Can you create an inner class inside a method
- Example
  - Class OuterClass
  - Public void exampleMethod() {
  - Class MethodLocaInnerClass {}; }
- Able to access final local variable from the enclosing function
- Static nested class can be created without needing to create its parent.
- OuterClass.StaticNestedClass staticClass1 = new OuterClass.StaticNestedClass();
- An inner class needs the outer class created
  - Outerclass.InnerClass inOuterclass example = new OuterClass().innerClas() // example.new InnerClass();

Access Modifiers

- Default Class Modifier → The classes are visible inside the same package
  - For variables and methods can be access by the SuperClass are available only to SubClasses in same package
  - For variables and methods from SuperClass of a different package are not available in Subclass
  - Example
  - In this example we have two classes,
    - Test class is trying to access the default method of Addition class,
      - since class Test belongs to a different package
    - This program would throw compilation error,
      - because the scope of default modifier is limited to the same package in which it is declared.
- Protected Access Modifier (Same package + Subclasses)
  - Protected variables and methods can be access in the same package class
  - Protected variables and methods from SuperClass are available to SubClass in any package
  - Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package.
    - You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.

Final & Static Modifier

- Final public class – When there is a final keyword on a class then
- Create a final class
  - immutable → Don’t want a subclass to break the immutability
  - If somebody implements a different hash code it might it might result in security issues
- String, Wrapper classes are immutable
- Methods can be final → Implementing the core logic and you don’t want it change
- Final on arguments → The argument can not be modified
- In static methods you have can static variables (variables that same values for all instances of the class)
- Example : public static class {} will result in a compilation error

Coupling & cohesion & Encapsulation

- Coupling
  - A measure how much a specific class is dependent on other classes
  - Change part of one class then you must change it in another class.
  - Exposing behavior and not instance variables help to have low coupling
- Cohesion
  - How related the responsibilities of a class are
  - When a class need a functionality it can create another class to do the functionality
- Encapsulation
  - Hiding the internal implementation of a class so it can change without affecting the signature of public methods
  - Try to define behavior instead of getters/setters
  -
- Abstraction is the process of separating ideas from specific instances.
- Aim to separate the implementation details from it behavior

Serialization

- Serialization → Convert object state to some internal object representation
  De-Serialization → Convert internal representation to object
- Functions
  - ObjectOutputStream.writeObject ObjectInputStream.readObject
- The object must implement serializable
- Example → Serializing an Object
  - FileOutputStream fileStream= new FileOutputStream(“Rectangle.ser”);
  - ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);
  - ObjectStream.writeObject(new Rectangle(5,6));
  - ObjectStream.close();
- Example Deserializing an object
  - FileInputStream fileInputStream = new FileInputStream(“Rectangle.ser”);
  - ObjectInputStream objectInputStream = new OjbectInputStream(fileInputStream);
  - Rectangle rectangle = (Rectangle)objectInputStream.readObject();
- Transient → Don’t serialize that part of the object
- How do you serialize a hierarchy objects
  - Object of one class might contain objects of other classes
  - All class that are serialized must implement the serializable interface
- NotSerializationException → Does not have the serialize interface
- Static variables are not part of the object and are not serialized
- When the Class is deserialized (constructor and initalizers) are not called.