Register | Login | Logout | Contact Us

Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

search here … Go

Home | Java FAQs | 600+ Java Q&As | Career | Tutorials | Member | Why?

Can u Debug? | Java 8 ready? | Top X | Productivity Tools | Judging Experience?

Home › Interview › Core Java Interview Q&A › Classes › ♦ Java classes and interfaces are the building blocks

♦ Java classes and interfaces are the building blocks

Posted on August 13, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1. Which class declaration is correct if A and B are classes and C and D are interfaces?

a) class Z extends A implements C, D{}
b) class Z extends A,B implements D {}
c) class Z extends C implements A,B {}
d) class Z extends C,D implements B {}

A1. a). class Z extends A implements C, D{}

A class is a template. A class can extend only a single class (i.e. single inheritance. Java does not support multiple implementation inheritance), but can implement multiple interfaces to achieve multiple interface inheritance. An interface can also extend more than one other interfaces.

```

interface E extends C,D {  //.... }

```

Q2. What happens when a parent and a child class has the same variable name?
A2. When both a parent class and its subclass have a field with the same name, this technique is called **variable shadowing or variable hiding**. The variable shadowing depends on the static type of the variable in which the object's reference is stored at compile time and NOT based on the dynamic type of the actual object stored at runtime as demonstrated in polymorphism via method overriding.

Q3. What happens when the parent and the child class has the same non-static method with the same signature?
A3. Unlike variables, when a parent class and a child class each has a non-static method (aka an instance method) with the same signature, the method of the child class overrides the method of the parent class. The method overriding depends on the dynamic type of the actual object being stored and NOT the static type of the variable in which the object reference is stored. The dynamic type can only be evaluated at runtime. As you can see, the rules for variable shadowing and method overriding are directly opposed. The **method overriding enables polymorphic behavior**.

Q4. What happens when the parent and the child class has the same static method with the same signature?
A4. The behavior of static methods will be similar to the **variable shadowing or variable hiding**, and not recommended. It will be invoking the static method of the referencing static object type determined at compile time, and NOT the dynamic object type being stored at runtime.

Q5. What is the difference between an abstract class and an interface and when should you use them?
A5. In design, you want the base class to present only an interface for its derived classes. This means, you don't want anyone to actually instantiate an object of the base class. You only want to upcast to it (implicit **upcasting**, which gives you polymorphic behavior), so that its interface can be used. This is accomplished by making that class abstract using the **abstract** keyword. If anyone tries to make an object of an abstract class, the compiler prevents it.

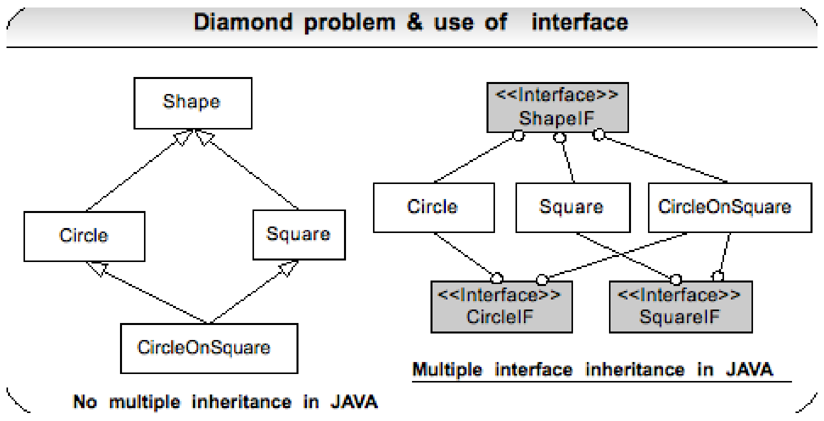
The **interface** keyword takes this concept of an abstract class a step further by preventing any method or function implementation at all. You can only declare a method or function but not provide the implementation till Java 7. The class, which is implementing the interface, should provide the actual implementation. The interface is a very useful and commonly used aspect in OO design, as it provides the separation of interface and implementation and enables you to

— Capture similarities among unrelated classes without artificially forcing a class relationship.
Declare methods that one or more classes are expected to implement.
— Reveal an object's programming interface without revealing its actual implementation.
— Model multiple interface inheritance in Java, which provides some of the benefits of full on multiple inheritances, a feature that some object-oriented languages support that allow a class to have more than one super class.

Q6 When will you use an abstract class?
A6 In case where you want to use implementation inheritance then it is usually provided by an abstract base class. Abstract classes are excellent candidates inside of application frameworks. Abstract classes let you define some default behavior and force subclasses to provide any specific behavior. Care should be taken not to overuse implementation inheritance as discussed before. The **template method design pattern** is a good example to use

an abstract class where the abstract class provides a default implementation.

Q7. When will you use an interface?
A7. For polymorphic interface inheritance, where the client wants to only deal with a type and does not care about the actual implementation, then use interfaces. If you need to change your design frequently, you should prefer using interface to abstract class. Coding to an interface reduces coupling. Another justification for using interfaces is that they solve the '**diamond problem**' of traditional multiple inheritance as shown in the diagram. Java does not support multiple inheritance. Java only supports multiple interface inheritance. Interface will solve all the ambiguities caused by this 'diamond problem'. Java 8 has introduced functional interfaces, and partially solves the diamond problem by allowing default and static method implementations in interfaces to inherit multiple behaviors.**Strategy design pattern** lets you swap new algorithms and processes into your program without altering the objects that use them.



Diamond problem and multiple inheritance in Java

Q8. What is a marker or a tag interface? Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java?
A8. The interfaces with no defined methods act like markers. They just tell the compiler that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. For example, java.io.Serializable, java.lang.Cloneable, java.util.EventListener, etc. Marker

interfaces are also known as “tag” interfaces since they tag all the derived classes into a category based on their purpose.

Now with the introduction of **annotations** in Java 5, the marker interfaces make less sense from a design standpoint. Everything that can be done with marker or tag interfaces in earlier versions of Java can now be done with **annotations at runtime using reflection**. One of the common problems with the marker or tag interfaces like Serializable, Cloneable, etc is that when a class implements them, all of its subclasses inherit them as well whether you want them to or not. You cannot force your subclasses to un-implement an interface. Annotations can have parameters of various kinds, and they’re much more flexible than the marker interfaces. This makes tag or marker interfaces obsolete, except for situations in which empty or tag interfaces can be checked at compile-time using the type-system in the compiler

Q9. What is a functional interface?
A9. **Functional interfaces** are introduced in Java 8 to allow default and static method implementations to enable functional programming (aka closures) with lambda expressions.

```

@FunctionalInterface
public interface Summable {
        abstract int sum(int input1, int input2
}

```

Q10. What does the @FunctionalInterface do, and how is it different from the @Override annotation?
A10. The @Override annotation takes advantage of the compiler checking to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does. Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.

The annotation @FunctionalInterface acts similarly to @Override by signaling to the compiler that the interface is intended to be a functional interface. The compiler will then throw an error if the interface has multiple abstract methods.

Q10. Does Java 8 solve the "diamond problem" discussed earlier? If yes how?
A10. Partially yes.

We know that Java does not support multiple implementation inheritance to solve the diamond problem (till Java 8). Java did only support multiple interface inheritance. That is, a class can implement multiple interfaces. By having default method implementations in interfaces, you can now have multiple behavioral inheritance in Java 8. Partially solving the diamond problem.

Q11. Does Java 8 solve the need to have a separate helper classes? For example, you have

1) *java.util. Collection* interface and a separate *java.util.Collections* helper class with static methods.
2) *java.nio.file.Path* interface and a separate *java.util.Paths* helper or utility class with static methods.

A11 Yes. In Java 8, interfaces can have static helper methods. Here is an example of a Java 8 interface with both default and static methods.

```

package com.java8.examples;

import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.Objects;

@FunctionalInterface
public interface Operation<Intetger>  {

   //SAM -- Single Abstract Method.
   //identifier abstract is optional
   Integer operate(Integer operand);

    default Operation<Integer> add(Integer o){
        r eturn (o1) -> operate(o1) +   o;
    }

    default Operation<Integer> multiply(Integer 
         return (o1) -> operate(o1) * o;
    }

    //define other default methods for divide, s
    default Integer getResult() {
       return operate(0);
    }

   default void print(){
      System.out.println("result is = " + getRes
    }


   //helper -- adds 5 to a given number
   static Integer plus5(Integer input) {
        return input + 5 ;
   }
 }

```

Now, how to invoke the default and static methods via **Lambda expressions**:

```

package com.java8.examples;

import static java.lang.System.out;

public class OperationTest {

 public static void main(String[] args) {

  //plus5 is an expressive static helper method
  Operation<Integer> calc = (x) -> Operation.plu

  Operation complexOp = calc.add(3)
         .multiply(4)
         .multiply(2)
         .multiply(2)
         .add(4);

  complexOp.print();

  int result = complexOp.getResult();

 }
}

```

So, if you are coding in Java 8, a separate helper class with static methods may no longer be required.

Q12. Does this mean that abstract classes may not be required as you can have default and static methods in Java 8 interfaces?

A12 No. You still can't have states in Java 8 interfaces. You can only have behavior. So, Java 8 interfaces only gives your **multiple behaviour implementation inheritance**. You still need abstract classes to maintain default states via instance or member variables.

Popular Posts

Bio | **Latest Posts**

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ ♦ Java class loading interview Q&A to ascertain your depth of Java knowledge

♦ 12 Java Generics interview Q&A ›

Posted in Classes, member-paid

Tags: Core Java FAQs, Java/JEE FAQs, Novice FAQs

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

- Unit Testing Q&A (2
  - BDD Testing (4)
    - Java BDD (Be
    - jBehave and E
    - jBehave and j
    - jBehave with t
  - Data Access Uni
    - ♥ Unit Testing
    - Part #3: JPA H
    - Unit Test Hibe
    - Unit Test Hibe
  - JUnit Mockito Sp
    - JUnit Mockito
      - Spring Con
      - Unit Testing
    - Part 1: Unit te
    - Part 2: Mockit
    - Part 3: Mockit
    - Part 4: Mockit
    - Part 5: Mockit
  - Testing Spring T
    - Integration Un
    - Unit testing Sp
  - ♦ 5 Java unit tes
  - JUnit with Hamc
  - Spring Boot in ur
- Other Interview Q&A 1
  - Finance Domain In
    - 12+ FX or Forex
    - 15 Banking & fin
  - FIX Interview Q&A
    - 20+ FIX basics ir
    - Finding your way
  - Groovy Interview Q
    - Groovy Coding (
      - Cash balance
      - Sum grades G
    - ♥ Q1 – Q5 Groov
    - ♦ 20 Groovy clos
    - ♦ 9 Groovy meta
    - Groovy method c

As a Java Architect

Java architecture & design concepts interview Q&As with diagrams | What should be a typical Java EE architecture?

Senior Java developers must have a good handle on

80+ step by step Java Tutorials

Preparing for Java written & coding tests

How good are your...to go places?

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from "Blah blah" to "Wow"? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼

© Disclaimer