♦ Java abstract classes Vs interfaces

Posted on February 2, 2015 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1. When to use an abstract class over an interface?
Q2. What is a diamond problem?
Q3. Does Java support multiple inheritance?

In design, you want the base class to present only an interface (or a contract) for its derived classes. This means, you don't want anyone to actually instantiate an object of the base class. You only want to **up cast** to it (implicit up casting, 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 with a compile-time error.

Q. What is the purpose of an abstract class?
A. The purpose of abstract classes is to function as base classes which can be extended by sub classes to create a full implementation. The base class is used for code reuse and gives **polymorphism** by up casting to it.

Q. Should it have at least one abstract member?
A. It is subjective, but it is preferred. If your intention is to prevent a class from being instantiated, then the best way to handle this is with a private or protected constructor, and not by marking it abstract.

Q. Can you give an example of an abstract class where it is used prevalently?
A. **Template Method design pattern** is a good example of using an abstract class and this pattern is used very prevalently in application frameworks. The **Template Method** design pattern is about providing partial implementations in the abstract base classes, and the subclasses can complete when extending the Template Method base class(es). Here is an example

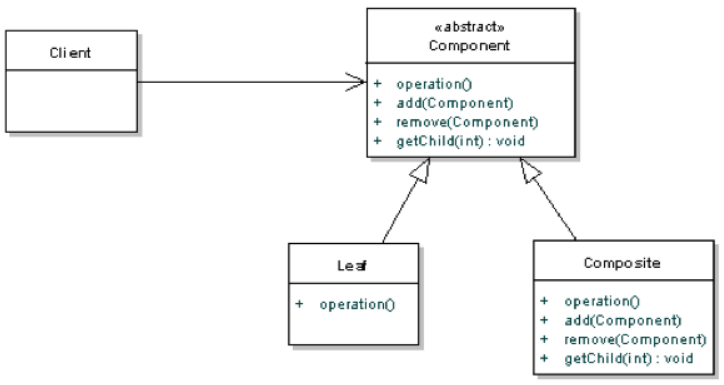
```
1  //cannot be instantiated
2  public abstract class BaseTemplate {
3
4   public void process() {
5      fillHead();
6      //some default logic
7      fillBody();
8      //some default logic
9      fillFooter();
10   }
11
12   //to be overridden by sub class
13   public abstract void fillBody();
14
15   //template method. Sub classes can override or
16   public void fillHead() {
17       System.out.println("Simple header");
18   }
19
20    //template method. Sub classes can override o
21    public void fillFooter() {
22       System.out.println("Simple footer");
23    }
24
```

```
25     //more template methods can be defined here
26 }
27
```

```
1  public class InvoiceLetterProcessor extends Base
2
3   @Override
4   public void fillBody() {
5    System.out.println("Invoice body" );
6   }
7
8   // template method
9   public void fillHead() {
10   System.out.println("Invoice header");
11  }
12 }
13
```

```
1  public class InvoiceTestMain {
2
3     public static void main(String[] args) {
4          //subclass is up cast to base class -- p
5          BaseTemplate template = new InvoiceLette
6          template.process();
7     }
8
9  }
10
11
```

Another design pattern that makes use of abstract classes is the **composite design pattern**. A **node** or a **component** is the parent or base class and derivatives can either be leaves (singular), or collections of other nodes, which in turn can contain leaves or collection-nodes. When an operation is performed on the parent, that operation is recursively passed down the hierarchy. An interface can be used instead of an abstract class, but an abstract class can provide some default behavior for the *add()*, *remove()* and *getChild()* methods.



Composite design pattern

Java 8: If you are using Java 8 or later versions of Java, you can have **default methods** and static helper methods in Java 8 interface definition.

Q. What is the purpose of an **interface**?
A. The interface keyword takes this concept of an abstract class a step further where till Java 7, you can't have implementations in an interface, but from **Java 8** onwards, the concept of **functional interfaces** was introduced where you can implement **default methods** and **static helper methods**.

Q. What are the differences between abstract classes and interfaces?
A.

Before Java 8:

Abstract class	Interface
Can maintain state in instance and static variables.	No state.

Have executable methods and abstract methods.	Have no implementation code. All methods are abstract.
Can only extend one abstract class. public ClassB extends ClassA {}	A class can implement any number of interfaces. public ClassB implements InterfaceA, InterfaceB{}

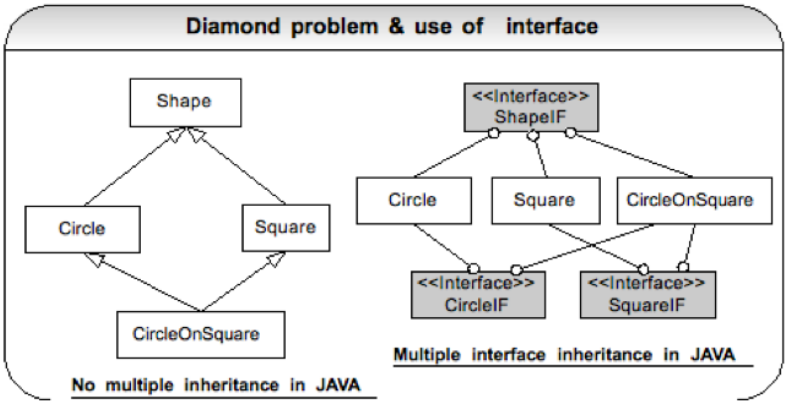
Java 8 onwards:

Abstract class	**Interface**
Can maintain state in instance and static variables.	No state.
Have executable methods and abstract methods.	Have executable default methods and static helper methods.
Can only subclass one abstract class.	A class can implement any number of interfaces.

So, from Java 8 onwards, the key difference is only one class can extend an abstract class, but a class can implement more than one interfaces.

Q. What is the diamond problem?
A.The diamond problem is that in the **multiple-inheritance diagram** on the left hand side below where *CircleOnSquare* inherits **state** and **behavior** from both *Circle* and *Square*. So, when we instantiate an object of class *CircleOnSquare*, any calls to method definitions in class *Shape* will be ambiguous –

because it's not sure whether to call the version of the method derived from class *Circle* or class *Square*.



Multiple inheritance and diamond issue

Java does not have **multiple inheritance**, as you can only extend one class. This means that Java is not at risk of suffering the consequences of the diamond problem. But, Java does support **multiple interface inheritance** as shown above on the right hand side of the diagram as a Java class can implement multiple interfaces. Hang on!!!!! From **Java 8** onwards, you can have functional interfaces where you can implement default and static methods. This means, Java supports **multiple behavior inheritance**, but not full multiple inheritance as state cannot be inherited because you can't define instance variables in an interface.

Before Java 8 interface example:

```
1 public interface Summable {
2   abstract int sum(int input1, int input2);
3 }
4
```

After Java 8 interface example with default and static helper methods:

```
1  @FunctionalInterface
2  public interface Operation<Integer>  {
3
4     Integer operate(Integer operand);
```

```
5
6     default Operation<Integer> add(Integer o){
7         return (o1) -> operate(o1) +   o;
8     }
9
10     default Operation<Integer> multiply(Integer
11         return (o1) -> operate(o1) * o;
12     }
13
14     //ads 5 to a given number
15    static Integer plus5(Integer input) {
16       return input + 5 ;
17   }
18
19 }
20
```

Now, if we have a class ***Calculator***, which implements two functional interfaces *Operation* and *Sum*, and both has a default method *add(Integer o)*. How does it know which default method to use? The one from *Operation* or the one from *Sum*. The *Calculator* class must solve the **multiple behavior inheritance ambiguity** by throwing a compile-time error

java: class Impl inherits unrelated defaults for *add(Integer o)* from types *Operation* and *Sum*.

In order to fix this class, the ***Calculator*** class needs to implement the *add(Integer o)* method to resolve the ambiguity.

Q. When to use an abstract class over an interface?
A.

Abstract classes

With the advent of Default Methods in Java 8, it seems that Interfaces and abstract Classes are same as you can implement behavior in both. However, they are still different concepts as

- An Abstract Class can **define constructor**(s).
- Abstract classes are **more structured and can have a state associated with them**. While in contrast,

default method can be implemented only in the terms of invoking other Interface methods, with no reference to a particular implementation's state.

Hence, both are used for different purposes and choosing between two really depends on the scenario. Abstract methods are good for implementing template method and composite design patterns with state and behavior.

Interfaces

If you need to change your design frequently, you should prefer using interfaces to abstract classes. Coding to an interface reduces coupling and interface inheritance can achieve code reuse with the help of object composition. For example: The Spring frameworks' dependency injection promotes code to an interface principle. Another justification for using interfaces is that they solve the 'diamond problem' of traditional multiple inheritance. Java does not support multiple inheritance, but supports multiple behavior inheritance.

Strategy design pattern lets you swap new algorithms and processes into your program without altering the objects that use them by making use of interfaces.

Q. What are the different ways to get **code reuse**?
A. There are 3 approaches

1. **Implementation inheritance** with **abstract** classes.
2. **Composition**.
3. **Delegation** to a helper class.

Implementation inheritance gives you **polymorphism** in addition to code reuse. You can up cast your child class to your abstract base class. If you are using composition for code reuse, then you need to use ***behavior inheritance*** with interfaces to get **polymorphism (**i.e. code to interface**)**.

The **GoF** design patterns favor **composition for code reuse** with **polymorphism** with **interfaces** over **implementation**

inheritance with **abstract classes for code reuse** and **poylmorphism**.

Why?

1. Code reuse via composition happens at run time whereas code reuse via inheritance happens at compile time. Hence, composition is more flexible and less fragile.

2. It is easy to misuse inheritance when there is really no "is a" relationship, hence breaking the Lithkov's Substitution Principle (**LSP**). For example, a square is not a rectangle. Overuse of implementation inheritance (uses the "extends" key word) can break all the sub classes, if the super class is modified. Do not use inheritance just to get polymorphism. If there is no 'is a' relationship and all you want is polymorphism then use interface inheritance with composition, which gives you code reuse.

3. Composition offers better testability than Inheritance. Composition is easier to test because inheritance tends to create very coupled classes that are more fragile (i.e. fragile parent class) and harder to test in isolation. The IoC containers like Spring, make testing even easier through injecting the composed objects via constructor or setter injection.

Why favor composition over inheritance? detailed discussion.

Q. What is the difference between the default methods introduced in Java 8 and regular methods?
A. In summary, **default methods** are like regular methods, but

- the default methods come with the default modifier.
- the default methods can only access its arguments as Interfaces do not have any state.

Regular methods in Classes can use and modify method arguments as well as the variables (i.e state) of their *Class*.

The default (aka defender) methods allow you to add new methods to interfaces without breaking the existing implementations of your interfaces. This provides an added flexibility by allowing interfaces to provide default implementations in situations where concrete classes fail to provide implementations for a methods.

Popular Posts

♦ 11 Spring boot interview questions & answers

857 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

825 views

18 Java scenarios based interview Questions and Answers

447 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

400 views

♦ 7 Java debugging interview questions & answers

311 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

301 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

292 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

286 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

263 views

8 Git Source control system interview questions & answers

215 views

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.

‹ Part 4: Java Tree structure interview and coding questions

Dealing with concurrent modifications in Java ›

Posted in Classes

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

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

open all | close all
- ♦ Complete the given
- Can you write code? (
- Converting from A to I
- Designing your classe
- Java Data Structures
- Passing the unit tests
- What is wrong with th
- Writing Code Home A
- Written Test Core Jav
- Written Test JEE (1)

How good are your…to go places?

open all | close all
- Career Making Know-
- Job Hunting & Resum

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

 ↑