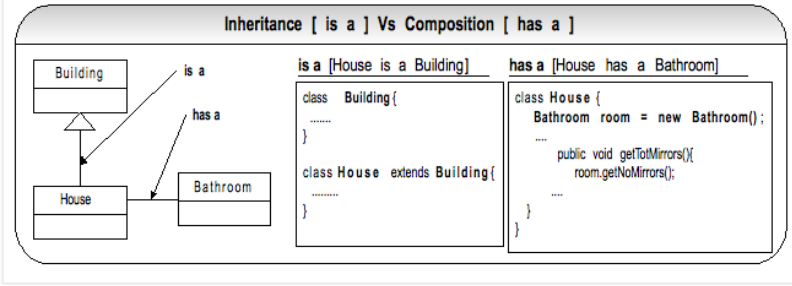
♦ Why favor composition over inheritance? a must know interview question for Java developers

Posted on August 11, 2014 by Arulkumaran Kumaraswamipillai — 2 Comments ↓

This is a very popular job interview question, and the correct answer depends on the problem you are trying to solve. You need to ask the right questions before deciding on one over the other. Understand the concepts properly to handle any follow up questions.

Q1. How do you express an ‘is a’ relationship and a ‘has a’ relationship or explain inheritance and composition?
A1. The ‘is a’ relationship is expressed with inheritance and ‘has a’ relationship is expressed with composition. Both inheritance and composition allow you to place sub-objects inside your new class. Two of the main techniques for **code reuse** are class inheritance and object composition.



inheritance vs composition

Inheritance is uni-directional. For example *House* is a *Building*. But *Building* is not a *House*. Inheritance uses extends key word.

Composition: is used when a *House* has a *Bathroom*. It is incorrect to say *House* is a *Bathroom*. Composition simply means using instance variables that refer to other objects. The class *House* will have an instance variable, which refers to a *Bathroom* object.

Q2. Which one to favor, composition or inheritance?
A2. The guide is that inheritance should be only used when subclass ‘is a’ super class. Don’t use inheritance just to get code reuse. If there is no ‘**is a**’ relationship then use composition for code reuse.

Reason #1: Overuse of implementation inheritance (uses the “extends” key word) can break all the subclasses, 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. Interface inheritance is accomplished by implementing interfaces.

Reason #2: Composition is more flexible as it is easily achieved at runtime while inheritance provides its features at compile time. Don't confuse inheritance with polymorphism. Polymorphism happens at runtime as it states that Java chooses which overridden method to run only at runtime.

Reason #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.

Q3. Can you give an example of the Java API that favors composition?
A3. The Java IO classes that use composition to construct different combinations of I/O outcomes like reading from a file or System.in, buffering the streams, tracking the line numbers, piping the streams for efficiency, etc using the decorator design pattern at run time.

```
1
2 //construct a reader
3 StringReader sr = new StringReader("Some Text....
4 //decorate the reader for performance
5 BufferedReader br = new BufferedReader(sr);
6 //decorate again to obtain line numbers
7 LineNumberReader lnr = new LineNumberReader(br);
8
```

The GoF design patterns like strategy, decorator, and proxy favor composition for code reuse over inheritance.

Q4. Can you a give an example where GoF design patterns use inheritance?
A4. A typical example of using inheritance for code reuse is in frameworks where the **template method design pattern** is used.

Template Method design pattern is a good example of using an **abstract class** and this pattern is used very prevalently in application frameworks.

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

Senior Java developers must have a good handle on

open all | close all
- Best Practice (6)
- Coding (26)
- Concurrency (6)
- Design Concepts (7)
- Design Patterns (11)
- Exception Handling (3
- Java Debugging (21)
- Judging Experience I
- Low Latency (7)
- Memory Managemen
- Performance (13)
- QoS (8)
- Scalability (4)
- SDLC (6)
- Security (13)
- Transaction Managen

80+ step by step Java Tutorials

open all | close all
- Setting up Tutorial (6)
- Tutorial - Diagnosis (2
- Akka Tutorial (9)
- Core Java Tutorials (2
- Hadoop & Spark Tuto

1. Java HTTP Servlet's doGet and doPost methods.
2. Message Driven EJB's and Spring message listener's onMessage(….) method.
3. Spring framework's JdbcTemplate, JmsTemplate, etc.
4. All non-abstract methods of *java.io.InputStream*, *java.io.OutputStream*, *java.util.AbstractList*, *java.util.AbstractMap*, *java.io.Reader*, etc

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
2  //cannot be instantiated
3  public abstract class BaseTemplate {
4
5   public void process() {
6      fillHead();
7      //some default logic
8      fillBody();
9      //some default logic
10     fillFooter();
11   }
12
13   //to be overridden by sub class
14   public abstract void fillBody();
15
16   //template method
17   public void fillHead() {
18       System.out.println("Simple header");
19   }
20
21    //template method
22    public void fillFooter() {
23       System.out.println("Simple footer");
24    }
25
26    //more template methods can be defined here
27 }
28
29
30
```

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

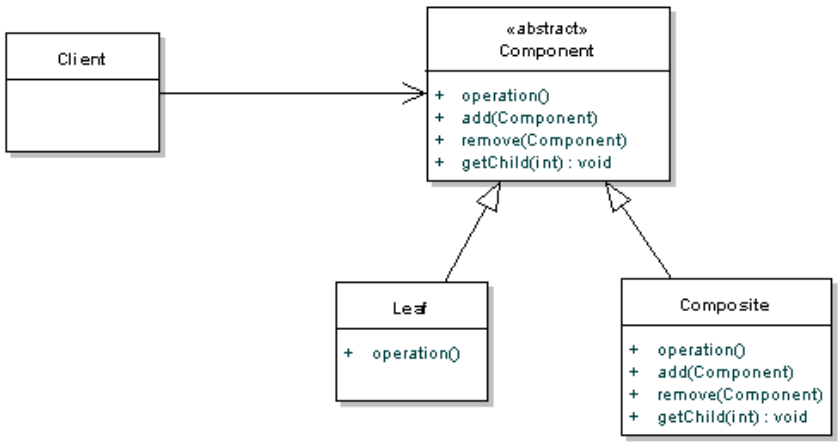
JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

```
11    System.out.println("Invoice header");
12   }
13 }
14
15
```

```
1
2  public class InvoiceTestMain {
3
4     public static void main(String[] args) {
5          //subclass is up cast to base class -- p
6          BaseTemplate template = new InvoiceLette
7          template.process();
8     }
9
10 }
11
12
```

Another common pattern that would use inheritance 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.



Q5. What questions do you ask yourself to choose composition (i.e. has-a relationship) for code reuse over implementation inheritance (i.e. is-a relationship)?
A5. Do my subclasses only change the implementation and

not the meaning or internal **intent** of the base class? Is every object of type *House* really “is-an” object of type *Building*? Have I checked this for “Liskov Substitution Principle”

According to **Liskov substitution principle** (LSP), a Square is not a Rectangle provided they are mutable. Mathematically a square is a rectangle, but behaviorally a rectangle needs to have both length and width, whereas a square only needs a width.

Another typical example would be, an Account class having a method called *calculateInterest(..).* You can derive two subclasses named *SavingsAccount* and *ChequeAccount* that reuse the super class method. But you cannot have another class called a *MortgageAccount* to subclass the above Account class. This will break the Liskov substitution principle because the **intent** is different. The savings and cheque accounts calculate the interest due to the customer, but the mortgage or home loan accounts calculate the interest due to the bank.

Violation of LSP results in all kinds of mess like failing unit tests, unexpected or strange behavior, and violation of **open closed principle** (OCP) as you end up having if-else or switch statements to resolve the correct subclass. For example,

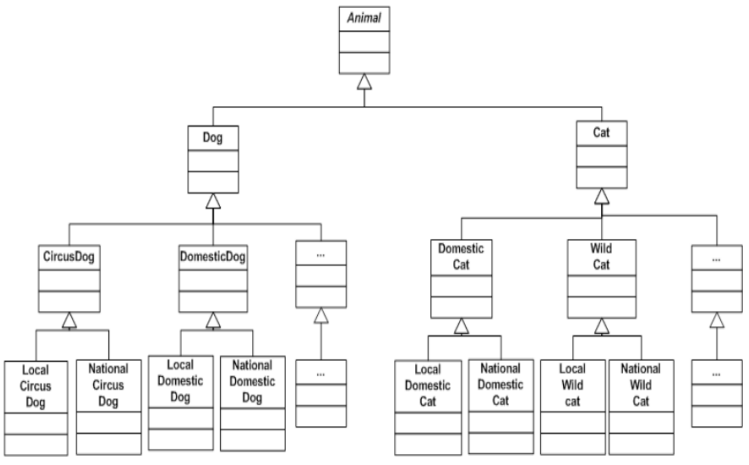
```
1
2 if(shape instanceof Square){
3       //....
4 }
5 else if (shape instanceof Rectangle){
6      //...
7 }
8
```

If you cannot truthfully answer yes to the above questions, then favor using “has-a” relationship (i.e. composition). Don’t use “is-a” relationship for just convenience. If you try to force an “is-a” relationship, your code may become inflexible, post-conditions and invariants may become weaker or violated, your code may behave unexpectedly, and the API may

become very confusing. LSP is the reason it is hard to create deep class hierarchies.

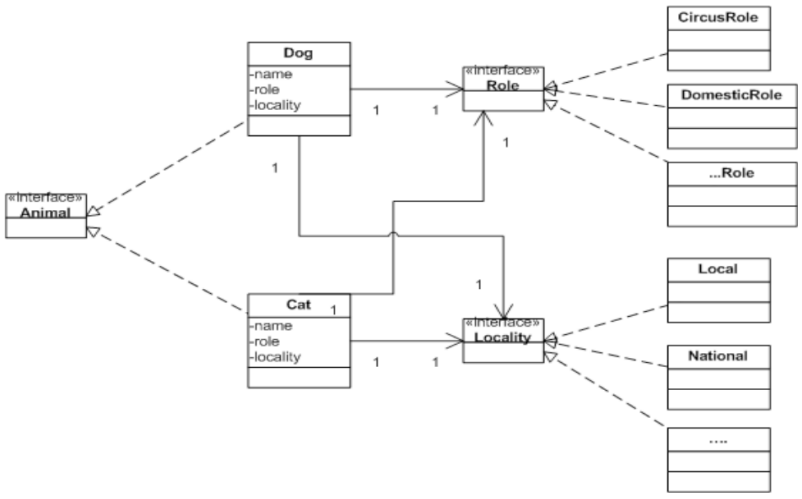
Always ask yourself, can this be modeled with a "**has-a**" relationship to make it more flexible?

For example, If you want to model a circus dog, will it be better to model it with "is a" relationship as in a *CircusDog* "is a" Dog or model it as a role that a dog plays? If you implement it with implementation inheritance, you will end up with sub classes like *CircusDog*, *DomesticDog*, *GuideDog*, *SnifferDog*, and *StrayDog*. In future, if the dogs are differentiated by locality like local, national, international, etc, you may have another level of hierarchy like *LocalCircusDog*, *NationalCicusDog*, *InternationalCircusDog*, etc extending the class *CircusDog*. So you may end up having 1 animal x 1 dog x 5 roles x 3 localities = 15 dog related classes. If you were to have similar differentiation for cats, you will end up having similar cat hierarchy like *WildCat*, *DomesticCat*, *LocalWildCat*, *NationalWildCat*, etc. **This will make your classes strongly coupled.**



Explosion of classes due to inheritance

If you implement it with interface inheritance, and composition for code reuse, you can think of circus dog as a role that a dog plays. These roles provide an abstraction to be used with any other animals like cat, horse, donkey, etc, and not just dogs. The role becomes a “has a” relationship. There will be an attribute of interface type Role defined in the Dog class as a composition that can take on different subtypes (using interface inheritance) such as *CircusRole*, *DomesticRole*, *GuideRole*, *SnifferRole*, and *StrayRole* at **runtime**. The locality can also be modeled similar to the role as a composition. This will enable different combinations of roles and localities to be constructed at runtime with 1 dog + 5 roles + 3 localities = 9 classes and 3 interfaces (i.e. *Animal*, *Role* and *Locality*). As the number of roles, localities, and types of animals increases, the gap widens between the two approaches. You will get a better abstraction with looser coupling with this approach as **composition is dynamic and takes place at run time compared to implementation inheritance, which is static**.



Composition to the rescue

Popular Posts

♦ 11 Spring boot interview questions & answers

861 views

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

829 views

18 Java scenarios based interview Questions and Answers

448 views

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

407 views

♦ 7 Java debugging interview questions & answers

311 views

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

303 views

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

294 views

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

288 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.

‹ ♦ 30+ FAQ Java Object Oriented Programming (i.e. OOP) interview Q&As

How to create a well designed Java application? ›

Posted in Design Concepts, OOP

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

2 comments on "♦ Why favor composition over inheritance? a must know interview question for Java developers"

Pratap Shinde says:

December 24, 2015 at 3:29 am

Sir, This blog is just awesome. What a clear explanation and the examples are just perfect.
Completely satisfied with your explanation.

Thank you
Pratap Shinde

Reply

Arulkumaran Kumaraswamipillai says:

December 24, 2015 at 9:24 am

Glad that you like it. Try to be as practical as possible without being too academic.

Reply

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

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

 ↑