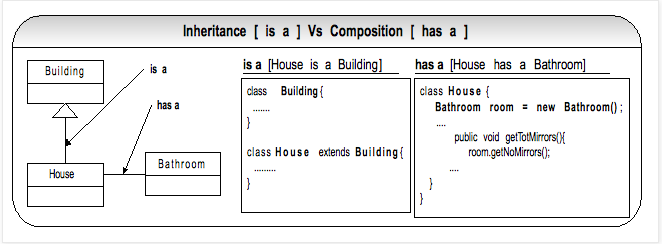
**Why favor composition over inheritance?**

Q11. How do you express an ‘is a’ relationship and a ‘has a’ relationship or explain inheritance and composition?  
A11. 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** is uni-directional. For example “House” is a type of “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.

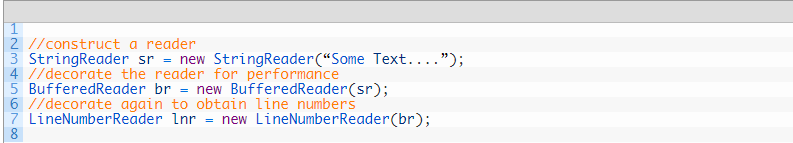
Q12. Which one to favor, composition or inheritance?  
A12. The guide is that inheritance should be only used when subclass ‘is a’ super class. Don’t use inheritance just to get code reuse. If there is no ‘**is a**’ relationship then use composition for code reuse.

**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.

Q13. Can you give an example of the Java API that favors composition?  
A13. 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.

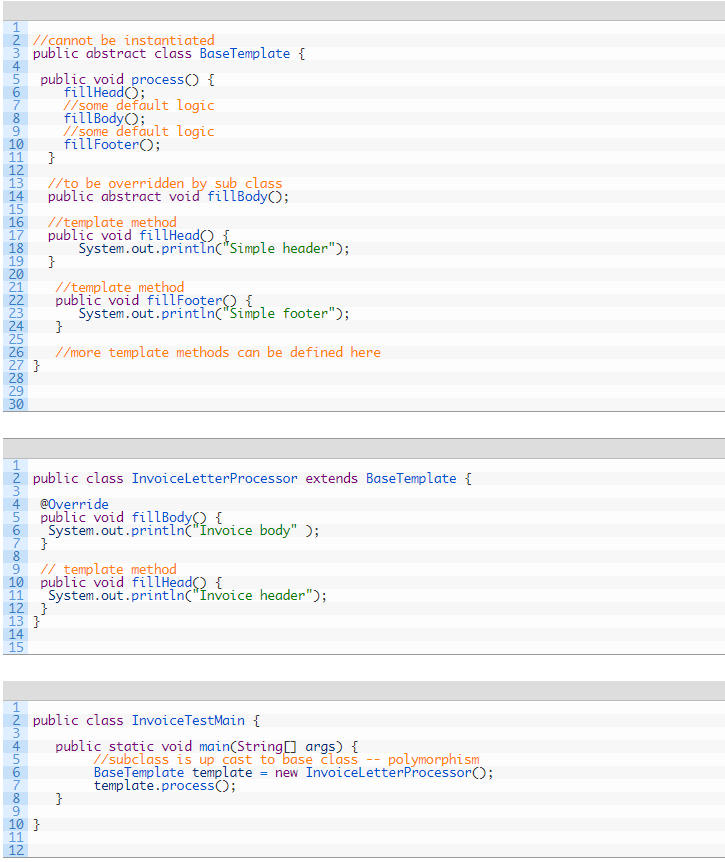


The GoF design patterns like strategy, decorator, and proxy favor composition for code reuse over inheritance. Interesting read ti further your knowledge in OOP & design patterns: [Why do Proxy, Decorator, Adapter, Bridge, and Facade design patterns look very similar? What are the differences?](https://www.java-success.com/proxy-decorator-adapter-bridge-facade-design-patterns-look-similar-differences/)

Q14. Can you a give an example where GoF design patterns use inheritance?  
A14. A typical example of using inheritance for code reuse is in frameworks where the **template method design pattern** is used.

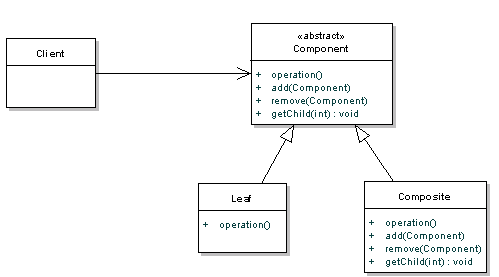
Template Method design pattern is a good example of using an **abstract class** and this pattern is used very prevalently in application frameworks.

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

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.

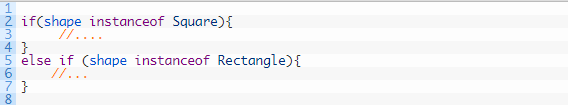
[](https://www.java-success.com/wp-content/uploads/2014/08/Screen-shot-2014-08-12-at-3.12.48-PM.png)

Q15. 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)?  
A15. 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,

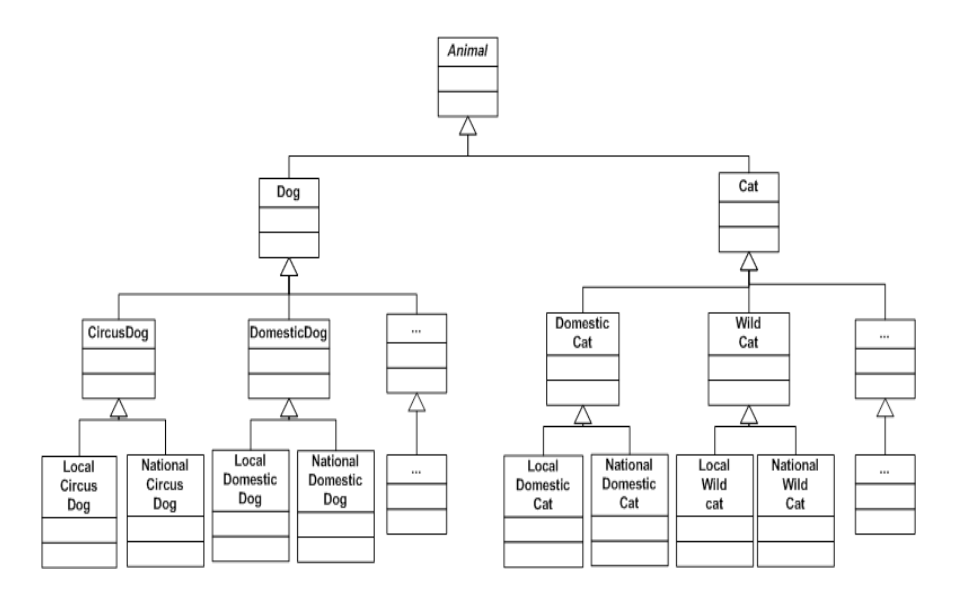


If you cannot truthfully answer yes to the above questions, then favor using “has-a” relationship (i.e. composition). Don’t use “is-a” relationship for just convenience. If you try to force an “is-a” relationship, your code may become inflexible, post-conditions and invariants may become weaker or violated,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

Learn more about [**SOLID** OOP design principles](https://www.java-success.com/design-principles-interview-questions-answers/).

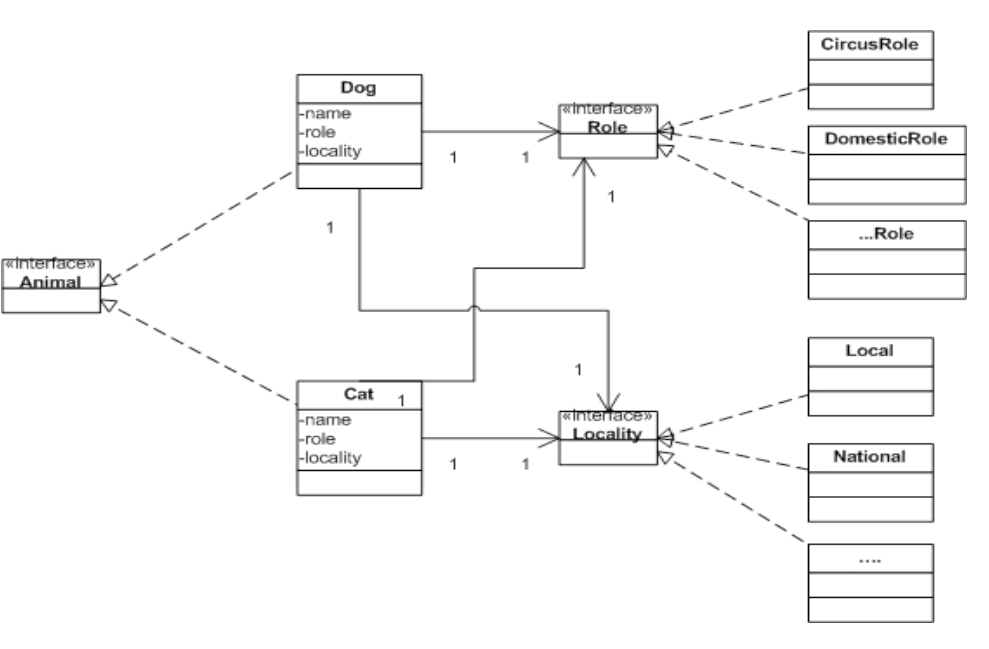
Always ask yourself, can this be modeled with a “**has-a**” relationship to make it more flexible?

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.**

[](https://www.java-success.com/wp-content/uploads/2014/08/Screen-shot-2014-08-11-at-10.04.33-PM.png)

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**.

[](https://www.java-success.com/wp-content/uploads/2014/08/Screen-shot-2014-08-11-at-10.08.29-PM.png)

Composition to the rescue