Composition over Inheritance

The reasons to prefer composition are profound.

* **Composition avoids a proliferation of classes that have to be maintained.** Example vehicle, PowerVehicle, ExpensiveVehicle etc extending the same functionality of super class**. It upholds the "*Single Responsibility Principle*", which is often summarized as "*There should never be more than one reason for a class to change.*",** and it means that every class exists for a specific purpose and it should only have methods that are directly related to its purpose.
* **Delegation is one example of a way to use composition instead of inheritance. Delegation lets you modify the behavior of a class without subclassing**. It can be achieved using Decorator pattern. Composition provides pluggability.
* **Inheritance creates tight, compile-time coupling between the classes whereas Composition in contrast is loose coupling, which among others enables clear separation of concerns, the possibility of switching dependencies at runtime and easier, more isolated dependency testability.**
* **Composition is slightly more language / framework agnostic.**
* **Composition is a very simple and tactical way of building objects.** Using composition you can always choose to define your own behavior or simply expose that part of your composed parts. With composition, it's easy to change behavior on the fly with Dependency Injection / Setters.
* **Composition is often more logical, Composition is about Has-A and Inheritance is about Is-A**

**Disadvantages of Inheritance**

1. You can't change the implementation inherited from super classes at runtime (obviously because inheritance is defined at compile time).
2. Inheritance exposes a subclass to details of its parent's class implementation, that's why it's often said that inheritance breaks encapsulation (in a sense that you really need to focus on interfaces only not implementation, so reusing by sub classing is not always preferred).
3. The tight coupling provided by inheritance makes the implementation of a subclass very bound up with the implementation of a super class that any change in the parent implementation will force the sub class to change.
4. Excessive reusing by sub-classing can make the inheritance stack very deep and very confusing too.

**Use Cases- 1**

public class Parent {  
  
 public void foo() {  
 bar();  
 }  
  
 public void bar() {  
 System.*out*.println("Parent class foo");  
 }  
}

public class Child extends Parent {  
 public void bar() {  
 foo();  
 }  
}

Test

public static void main(String[] args) {  
 Parent p = new Child();  
 p.bar();  
}

Exception in thread "main" java.lang.StackOverflowError due to recursion. In the above case Inheritance violates encapsulation.

**Use Case – 2**

Example: Suppose A overrides all methods in B by first validating input arguments in each method (**for security reasons**). If a new method is added to B and A is not updated, the new inherited method would introduce a security hole.