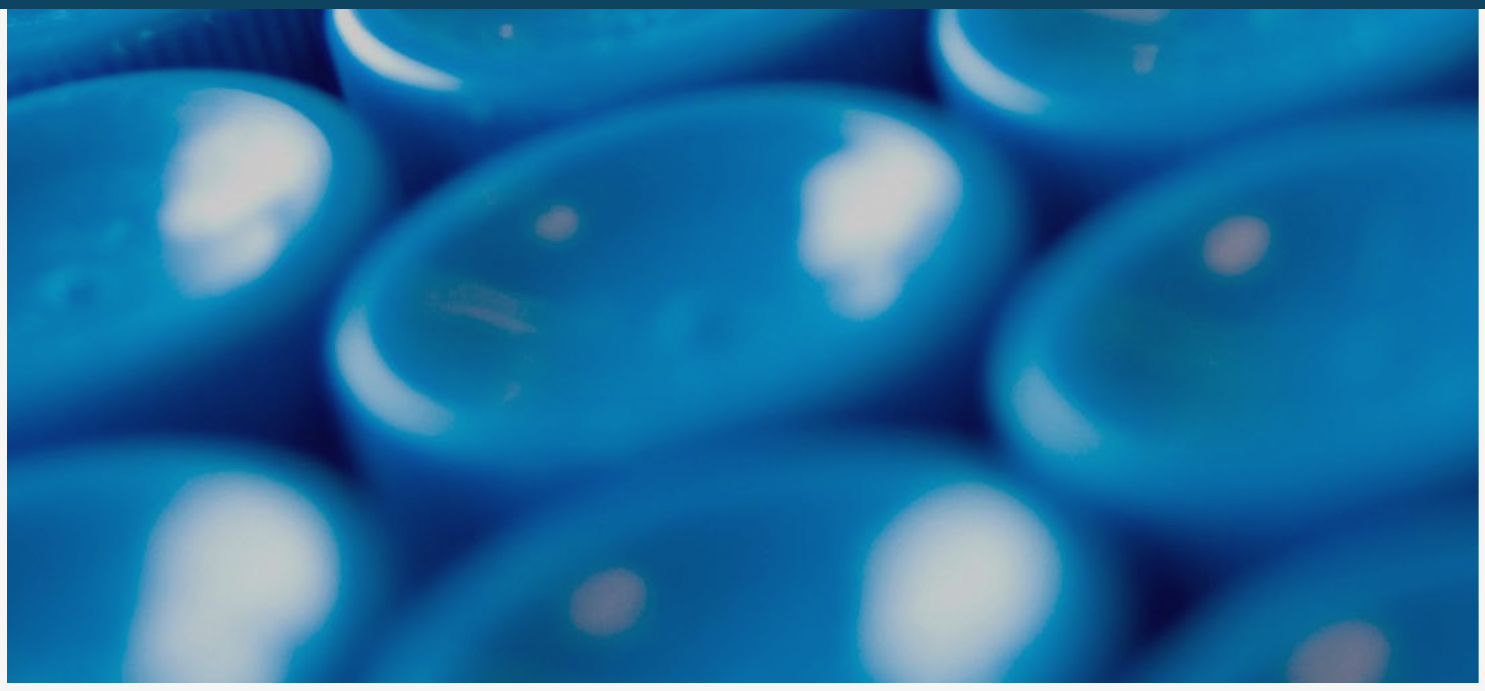
Java 101 primer: Composition and inheritance

Use composition to overcome the problem with inheritance

BY JEFF FRIESEN, JAVAWORLD, OCTOBER 2015



In "Java 101: Inheritance in Java, Part 1," you learned how to leverage inheritance for code reuse, by establishing *is-a relationships* between classes. This free Java 101 primer focuses on *composition*, a closely related programming technique that is used to establish *has-a relationships* instead. Whereas inheritance extends the features of one class to another, composition allows us to compose a class from another class. The distinction may seem subtle, but it will make more sense once you see it in code.

> Download the source code for "Java 101 primer: Composition and inheritance." Created by Jeff Friesen for JavaWorld.

Has-a versus *is-a* relationships

The premise in composition is that one class *has a* field whose type is that of another class. For example, `Vehicle` has a `String` field named `make`. It could also have an `Engine` field named `engine` and a `Transmission` field named `transmission`:

```
class Vehicle
{
   private String make;

   private Engine engine;

   private Transmission transmission;

   // ...
}
```

```
class Transmission
{
   // ...
}

class Engine
{
   // ...
}
```

In this example, we would say that a vehicle is composed of a make, an engine, and a transmission because it has a make field, an `engine` field, and a `transmission` field.

In addition to composing classes from other classes, you can use this technique to compose objects from other objects, by storing object references in another object's fields.

Composition techniques

This primer demonstrates using composition to overcome a well-known problem with Java inheritance. For a more in-depth introduction to composition techniques, see the JavaWorld classic "Inheritance versus composition: Which one should you choose?"

The problem with inheritance

The problem with inheritance is that it breaks encapsulation. You will recall from "Java 101: Classes and objects in Java" that *encapsulation* refers to the combining of constructors, fields, and methods into a class's body. In inheritance, a subclass relies on implementation details in its superclass. If the superclass's implementation details

change, the subclass might break. This problem is especially serious when a developer doesn't have complete control over the superclass, or when the superclass hasn't been designed and documented with extension in mind (see "Java 101: Inheritance in Java, Part 2" for more about working with superclasses).

To understand this problem, suppose you've purchased a library of Java classes that implement a contact manager. Although you don't have access to their source code, assume that Listing 1 describes the main `CM` class.

Listing 1. Implementing part of a contact manager

```
public class CM
{
   private final static int MAX_CONTACTS = 1000;
   private Contact[] contacts;
   private int size;

   public CM()
   {
      contacts = new Contact[MAX_CONTACTS];
      size = 0; // redundant because size is
automatically initialized to 0
                // adds clarity, however
   }

   public void addContact(Contact contact)
   {
      if (size == contacts.length)
         return; // array is full
      contacts[size++] = contact;
   }
```

```
    public void addContacts(Contact[] contacts)
    {
       for (int i = 0; i < contacts.length; i++)
          addContact(contacts[i]);
    }
}
```

The `CM` class stores an array of contacts, with each contact described by a `Contact` instance. For this discussion, the details of `Contact` aren't important; it could be as trival as `public class Contact {}`.

Libraries and public classes

If you're wondering why I declared `CM public`, it's because I believe that a future version of `CM` could be stored in a package, also known as a library in this example. In a package, only `public` classes can be accessed by external applications. (*Helper classes*, which are designed to support `public` classes, and which are not accessible to applications, are not declared `public`.) I would apply the same rationale to constructors and methods that might be called by an external application.

Now suppose you wanted to log each contact in a file. Because a logging capability isn't provided, you extend `CM` with the `LoggingCM` class below, which adds logging behavior in overriding `addContact()` and `addContacts()` methods.

Listing 2. Extending the contact manager with support for logging

```
public class LoggingCM extends CM
{
   // A constructor is not necessary because the Java
compiler will add a
   // no-argument constructor that calls the superclass's
```

```
no-argument
   // constructor by default.

   @Override
   public void addContact(Contact contact)
   {
      Logger.log(contact.toString());
      super.addContact(contact);
   }

   @Override
   public void addContacts(Contact[] contacts)
   {
      for (int i = 0; i < contacts.length; i++)
         Logger.log(contacts[i].toString());
      super.addContacts(contacts);
   }
}
```

The `LoggingCM` class relies on a `Logger` class (see below) whose `void log(String msg)` class method logs a string to a file. A `Contact` object is converted to a string via `toString()`, which is passed to `log()`:

Listing 3. log() outputs its argument to the standard output stream

```
class Logger
{
   static void log(String msg)
   {
      System.out.println(msg);
   }
}
```

Although `LoggingCM` looks okay, it doesn't work as you might expect. Suppose you instantiated this class and added a few `Contact` objects to the object via `addContacts()`:

Listing 4. The problem with inheritance

```
class CMDemo
{
   public static void main(String[] args)
   {
      Contact[] contacts = { new Contact(), new
Contact(), new Contact() };
      LoggingCM lcm = new LoggingCM();
      lcm.addContacts(contacts);
   }
}
```

If you run this code, you will discover that `log()` outputs a total of six messages; each of the expected three messages (one per `Contact` object) is duplicated.

What happened?

When `LoggingCM`'s `addContacts()` method is called, it first calls `Logger.log()` for each `Contact` instance in the `contacts` array that is passed to `addContacts()`. This method then calls `CM`'s `addContacts()` method via `super.addContacts(contacts);`.

`CM`'s `addContacts()` method calls `LoggingCM`'s overriding `addContact()` method, one for each `Contact` instance in its `contacts` array argument. The `addContact()` then executes `Logger.log(contact.toString());`, to log its `contact` argument's string representation, and you end up with three additional logged messages.

Method overriding and base-class fragility

If you didn't override the `addContacts()` method, this problem would go away. But in that case the subclass would still be tied to an implementation detail: `CM`'s `addContacts()` method calls `addContact()`.

It isn't a good idea to rely on an implementation detail when that detail isn't documented. (Recall that you don't have access to `CM`'s source code.) When a detail isn't documented, it can change in a new version of the class.

Because a base class change can break a subclass, this problem is known as the *fragile base class problem*. A related cause of fragility (which also has to do with overriding methods) occurs when new methods are added to a superclass in a subsequent release.

For example, suppose a new version of the library introduces a `public void addContact(Contact contact, boolean unique)` method into the `CM` class. This method adds the `contact` instance to the contact manager when `unique` is `false`. When `unique` is `true`, it adds the `contact` instance only if it wasn't previously added.

Because this method was added after the `LoggingCM` class was created, `LoggingCM` doesn't override the new `addContact()` method with a call to `Logger.log()`. As a result, `Contact` instances passed to the new `addContact()` method are not logged.

Here's another problem: you introduce a method into the subclass that is not also in the superclass. A new version of the superclass presents a new method that matches the subclass method signature and return type. Your subclass method now overrides the superclass method and probably doesn't fulfill the superclass method's contract.

Composition (and forwarding) to the rescue

Fortunately, you can make all of these problems disappear. Instead of extending the superclass, create a `private` field in a new class, and have this field reference an *instance* of the superclass. This workaround entails forming a "has-a" relationship between the new class and the superclass, so the technique you are using is composition.

Additionally, you can make each of the new class's instance methods call the corresponding superclass method and return the called method's return value. You do this via the superclass instance that was saved in the `private` field. This task is known as *forwarding*, and the new methods are known as *forwarding methods*.

Listing 5 presents an improved `LoggingCM` class that uses composition and forwarding to forever eliminate the fragile base class problem and the additional problem of unanticipated method overriding.

Listing 5. Composition and method forwarding demo

```
public class LoggingCM
{
   private CM cm;

   public LoggingCM(CM cm)
   {
      this.cm = cm;
   }

   public void addContact(Contact contact)
   {
      Logger.log(contact.toString());
```

```
        cm.addContact(contact);
    }

    public void addContacts(Contact[] contacts)
    {
        for (int i = 0; i < contacts.length; i++)
            Logger.log(contacts[i].toString());
        cm.addContacts(contacts);
    }
}
```

Note that in this example the `LoggingCM` class doesn't depend upon implementation details of the `CM` class. You can add new methods to `CM` without breaking `LoggingCM`.

Wrapper classes and the Decorator design pattern

Listing 5's `LoggingCM` class is an example of a *wrapper class*, which is a class whose instances wrap other instances. Each `LoggingCM` object wraps a `CM` object. `LoggingCM` is also an example of the Decorator design pattern.

To use the new `LoggingCM` class, you must first instantiate CM and pass the resulting object as an argument to `LoggingCM`'s constructor. The `LoggingCM` object wraps the `CM` object, as follows:

```
LoggingCM lcm = new LoggingCM(new CM());
```

Final notes

In this primer you've learned the difference between composition and inheritance, and how to use composition to assemble classes from other classes. Composition resolves one of the main

programming challenges associated with inheritance, which is that it breaks encapsulation. Composition is an especially important programming technique for situations where future developers are unlikely to have access to or control over the superclass. The need for it emerges especially in cases where a class package or library has not been designed with extension in mind. Some quick F.A.Q.s about extension will complete this primer:

- **What does 'design and document for class extension' mean?** *Design* means to provide `protected` methods that hook into the class's inner workings (to support writing efficient subclasses) and ensure that constructors and the `clone()` method never call overridable methods. *Document* means to clearly state the impact of overriding methods.
- **When should I extend a class versus using a wrapper?** Extend a class when an "is-a" relationship exists between the superclass and the subclass, and either you have control over the superclass or the superclass has been designed and documented for class extension. Otherwise, use a wrapper class.
- **Why shouldn't I use wrapper classes in a callback framework?** A *callback framework* is an object framework where an object passes its own reference to another object (via `this`), so that the latter object can call the former's methods at a later time. Calling back to the former object's method is known as a *callback*. Because the wrapped object doesn't know of its wrapper class, it passes only its reference (via `this`); resulting callbacks don't involve the wrapper class's methods.