JavaWorld Solutions for Java developers™

Search Advanced search

DEVELOPMENT TOOLS

HOME

RESEARCH CENTERS

+ Java Standard Edition

+ Java Enterprise Edition

+ Java Micro Edition

Development Tools

- Application Management
- Data Access Tools
- Gaming Tools
- Web Development Frameworks
- Security & Testing
- Java Application Servers
- Profiling and Monitoring
- Reporting

SITE RESOURCES

- Featured Articles
- News & Views
- JW Blogs
- Forums
- Podcasts
- Newsletters
- Whitepapers
- RSS Feeds

CAREERS

PARTNER SITES

- Demo.com
- LinuxWorld.com
- NetworkWorld.com

ABOUT US

Reveal the magic behind subtype polymorphism

Behold polymorphism from a type-oriented point of view

By Wm. Paul Rogers, JavaWorld.com, 04/13/01

Page 3 of 7

At first glance, the above `List` abstraction may seem to be the utility of the class `java.util.List`. However, Java does not support true parametric polymorphism in a type-safe manner, which is why `java.util.List` and `java.util`'s other collection classes are written in terms of the primordial Java class, `java.lang.Object`. (See my article "A Primordial Interface?" for more details.) Java's single-rooted implementation inheritance offers a partial solution, but not the true power of parametric polymorphism. Eric Allen's excellent article, "Behold the Power of Parametric Polymorphism," describes the need for generic types in Java and the proposals to address Sun's Java Specification Request #000014, "Add Generic Types to the Java Programming Language." (See Resources for a link.)

Inclusion

Inclusion polymorphism achieves polymorphic behavior through an inclusion relation between types or sets of values. For many object-oriented languages, including Java, the inclusion relation is a subtype relation. So in Java, inclusion polymorphism is subtype polymorphism.

As noted earlier, when Java developers generically refer to polymorphism, they invariably mean subtype polymorphism. Gaining a solid appreciation of subtype polymorphism's power requires viewing the mechanisms yielding polymorphic behavior from a type-oriented perspective. The rest of this article examines that perspective closely. For brevity and clarity, I use the term polymorphism to mean subtype polymorphism.

Type-oriented view

The UML class diagram in Figure 1 shows the simple type and class hierarchy used to illustrate the mechanics of polymorphism. The model depicts five types, four classes, and one interface. Although the model is called a class diagram, I think of it as a type diagram. As detailed in "Thanks Type and Gentle Class," every Java class and interface declares a user-defined data type. So from an implementation-independent view (i.e., a type-oriented view) each of the five rectangles in the figure represents a type. From an implementation point of view, four of those types are defined using class constructs, and one is defined using an interface.

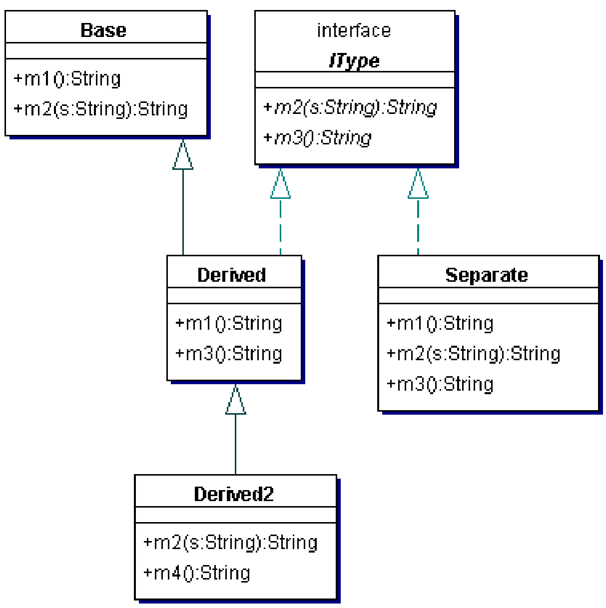

Figure 1. UML class diagram for the example code

The following code defines and implements each user-defined data type. I purposely keep the implementation as simple as possible:

```
/* Base.java */
public class Base
{
 public String m1()
 {
  return "Base.m1()";
 }
 public String m2( String s )
 {
  return "Base.m2( " + s + " )";
 }
}
/* IType.java */
interface IType
{
 String m2( String s );
 String m3();
}
/* Derived.java */
public class Derived
 extends Base
 implements IType
{
 public String m1()
 {
  return "Derived.m1()";
 }
 public String m3()
 {
  return "Derived.m3()";
 }
}
/* Derived2.java */
public class Derived2
 extends Derived
{
 public String m2( String s )
 {
  return "Derived2.m2( " + s + " )";
 }
 public String m4()
 {
  return "Derived2.m4()";
 }
}
/* Separate.java */
public class Separate
 implements IType
{
 public String m1()
 {
  return "Separate.m1()";
 }
 public String m2( String s )
 {
  return "Separate.m2( " + s + " )";
 }
 public String m3()
 {
  return "Separate.m3()";
 }
}
```

Using these type declarations and class definitions, Figure 2 depicts a conceptual view of the Java statement:

```
Derived2 derived2 = new Derived2();
```

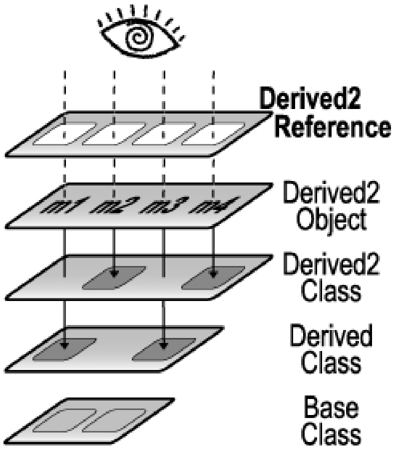

Figure 2. Derived2 reference attached to Derived2 object

The above statement declares an explicitly typed reference variable, `derived2`, and attaches that reference to a newly created `Derived2` class object. The top panel in Figure 2 depicts the `Derived2` reference as a set of portholes, through which the underlying `Derived2` object can be viewed. There is one hole for each `Derived2` type operation. The actual `Derived2` object maps each `Derived2` operation to appropriate implementation code, as prescribed by the implementation hierarchy defined in the above code. For example, the `Derived2` object maps `m1()` to implementation code defined in class `Derived`. Furthermore, that implementation code overrides the `m1()` method in class `Base`. A `Derived2` reference variable cannot access the overridden `m1()` implementation in class `Base`. That does not mean that the actual implementation code in class `Derived` can't use the `Base` class implementation via `super.m1()`. But as far as the reference variable `derived2` is concerned, that code is inaccessible. The mappings of the other `Derived2` operations similarly show the implementation code executed for each type operation.

Now that you have a `Derived2` object, you can reference it with any variable that conforms to type `Derived2`. The type hierarchy in Figure 1's UML diagram reveals that `Derived`, `Base`, and `IType` are all super types of `Derived2`. So, for example, a `Base` reference can be attached to the object. Figure 3 depicts the conceptual view of the following Java statement:

```
Base base = derived2;
```

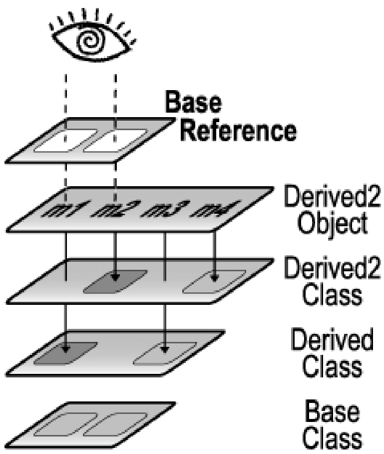

Figure 3. Base reference attached to Derived2 object

There is absolutely no change to the underlying `Derived2` object or any of the operation mappings, though methods `m3()` and `m4()` are no longer accessible through the `Base` reference. Calling `m1()` or `m2(String)` using either variable `derived2` or `base` results in execution of the same implementation code:

```
String tmp;
// Derived2 reference (Figure 2)
tmp = derived2.m1();            // tmp is "Derived.m1()"
tmp = derived2.m2( "Hello" );    // tmp is "Derived2.m2( Hello )"
// Base reference (Figure 3)
tmp = base.m1();               // tmp is "Derived.m1()"
tmp = base.m2( "Hello" );       // tmp is "Derived2.m2( Hello )"
```

Realizing identical behavior through both references makes sense because the `Derived2` object does not know what calls each method. The object only knows that when called upon, it follows the marching orders defined by the implementation hierarchy. Those orders stipulate that for method `m1()`, the `Derived2` object executes the code in class `Derived`, and for method `m2(String)`, it executes the code in class `Derived2`. The action performed by the underlying object does not depend on the reference variable's type.

< Prev 1 2 3 4 5 6 7 Next >

Print E-Mail article Feedback Add to del.icio.us

Related Article

Resources

"On Understanding Types, Data Abstraction, and Polymorphism," Luca Cardelli and Peter Wegner from *Computing Surveys,* (December, 1985) -- an academic treatise of three important object-oriented concepts
http://research.microsoft.com/Users/luca/Papers/OnUnderstanding.pdf

"Behold the Power of Parametric Polymorphism," Eric Allen (*JavaWorld,* February 2000) -- an excellent overview of the need for introducing generic types to the Java language
http://www.javaworld.com/jw-02-2000/jw-02-jsr.html

"Add Generic Types to the Java Programming Language," (Java Community Process Program, JSR #000014) -- the Java Specification Request regarding extending the Java language to incorporate parametric polymorphism
http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html

Read more from Wm. Paul Rogers:

"Thanks Type and Gentle Class" (*JavaWorld,* January 19, 2001) explores the importance of separating the object-oriented concepts of type and class.

"A Primordial Interface?" (*JavaWorld,* March 9, 2001) uses a type-oriented perspective to explore the implicit existence of a primordial interface in Java.

Wm. Paul Rogers comoderates the **Java Beginner** discussion. Ask him your beginner-level questions here
http://www.itworld.com/jump/jw-0413-polymorph/forums.itworld.com/webx?230@@.ee6b804!skip=2899

Sign up for the *JavaWorld This Week* free weekly email newsletter and keep up with what's new at *JavaWorld*
http://www.idg.net/jw-subscribe

Browse *JavaWorld*'s Topical Index
http://www.javaworld.com/javaworld/topicalindex/jw-ti-index.html