Reveal the magic behind subtype polymorphism

Behold polymorphism from a type-oriented point of view

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

The interface to an object

So polymorphism relies on separating the concerns of type and implementation, which is often referred to as separating interface and implementation. But that latter statement seems confusing in light of the Java keyword `interface`.

More importantly, what do developers mean by the common phrase *the interface to an object*? Typically, the statement's context indicates that the phrase refers to the set of all public methods defined by the object's class hierarchy -- that is, the set of all publicly available methods that may be called on the object. That definition, however, leans toward an implementation-centric view by concentrating our focus on an object's runtime capability, rather than on a type-oriented view of the object. In Figure 3, the interface to the object refers to the panel labeled "`Derived2` Object." That panel lists all available methods for the `Derived2` object. But to understand polymorphism, we must free ourselves from an implementation level and view the object from the perspective of the type-oriented panel labeled "`Base` Reference." At that level, the reference variable's type dictates an interface to the object. That's *an* interface, not *the* interface. Under the guidance of type conformance, we may attach multiple type-oriented views to a single object. There is no singularly specified *interface to an object.*

So in terms of type, *the interface to an object* refers to the widest possible type-oriented view of that object -- as in Figure 2. A super type reference attached to the same object typically narrows the view -- as in Figure 3. The concept of type best captures the spirit of freeing object interactions from the details of object implementation. Rather than refer to the interface of an object, a type-oriented perspective encourages referring to the reference type attached to an object. The reference type dictates the permissible interaction with the object. Think *type* when you want to know *what* an object can do, as opposed to *how* the object implements its responsibilities.

Java interfaces

The previous examples of polymorphic behavior use subtype relationships established through class inheritance. Java interfaces also declare user-defined types, and correspondingly, Java interfaces enable polymorphic behavior by establishing type inheritance structure. Suppose a reference variable named `ref` attaches to an object whose class contains the following method definition:

```
public String poly2( IType iType )
{
  return iType.m3();
}
```

To explore polymorphic behavior inside `poly2(IType)`, the following code creates two objects from different classes and passes a reference to each into `poly2(IType)`:

```
Derived2 derived2 = new Derived2();
Separate separate = new Separate();
String tmp;
tmp = ref.poly2( derived2 );      // tmp is "Derived.m3()"
tmp = ref.poly2( separate );      // tmp is "Separate.m3()"
```

The above code resembles the previous discussion of polymorphic behavior inside `poly1(Base)`. The implementation code in `poly2(IType)` calls method `m3()` for each object, using a local `IType` reference. As before, code comments note the `String` result of each call. Figure 5 shows the conceptual structure of the two calls to `poly2(IType)`:

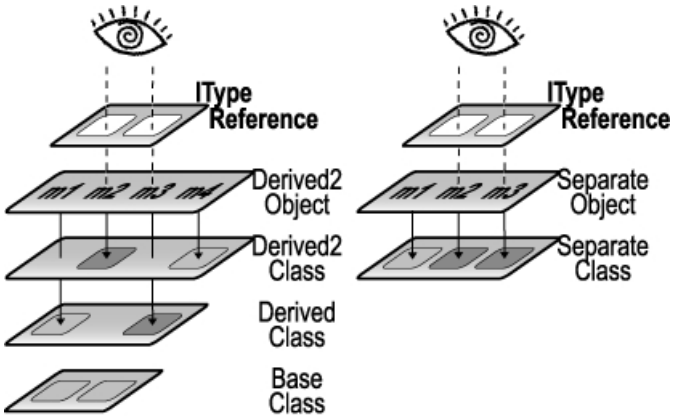

Figure 5. IType reference attached to a Derived2 and a Separate object

The similarity between the polymorphic behavior occurring inside methods `poly1(Base)` and `poly2(IType)` results directly from a type-oriented perspective. Raising our view above the implementation level allows an identical understanding of the two code samples' mechanics. Local super type references attach to incoming objects and make type-restricted calls to those objects' methods. Neither reference knows (nor cares) what implementation code actually executes. The subtype relationship verified at compile time guarantees the passed object's capability to perform appropriate implementation code when called upon.

However, an important distinction manifests itself at the implementation level. In the `poly1(Base)` example (Figures 3 and 4), the `Base-Derived-Derived2` class inheritance chain establishes the requisite subtype relations, and method overriding determines the implementation code mappings. In the `poly2(IType)` example (Figure 5), a completely different dynamic occurs. Classes `Derived2` and `Separate` do not share any implementation hierarchy, yet objects instantiated from those classes exhibit polymorphic behavior through an `IType` reference.

Such polymorphic behavior highlights a significant utility of Java interfaces. The UML diagram in Figure 1 shows that type `Derived` subtypes both `Base` and `IType`. By defining a type completely free of implementation, Java interfaces allow multiple type inheritance without the thorny issues of multiple implementation inheritance, which Java prohibits. Classes from completely separate implementation hierarchies may be grouped by a Java `interface`. In Figure 1, `interface IType` groups `Derived` and `Separate` (and any subtypes of those types).

By grouping objects from disparate implementation hierarchies, Java interfaces facilitate polymorphic behavior even in the absence of any shared implementation or overridden methods. As shown in Figure 5, an `IType` reference polymorphically accesses the `m3()` methods of the underlying `Derived2` and `Separate` objects.

The interface to an object (again)

Note that objects `Derived2` and `Separate` in Figure 5 each possess mappings for method `m1()`. As previously discussed, the interface to each object includes that `m1()` method. But there is no way, using these two objects, to engage method `m1()` in polymorphic behavior. It is insufficient that each object possesses an `m1()` method. A common type must exist with operation `m1()`, through which to view the objects. The objects may seem to share `m1()` in their interfaces, but without a common super type, polymorphism is impossible. Thinking in terms of *the interface to an object* simply confounds this issue.

< 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