# Interchangeable objects with polymorphism

When dealing with type hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type. This allows you to write code that doesn’t depend on specific types. In the shape example, functions manipulate generic shapes without respect to whether they’re circles, squares, triangles, or some shape that hasn’t even been defined yet. All shapes can be drawn, erased, and moved, so these functions simply send a message to a shape object; they don’t worry about how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is the most common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called pentagon without modifying the functions that deal only with generic shapes. This ability to extend a program easily by deriving new subtypes is important because it greatly improves designs while reducing the cost of software maintenance.

There’s a problem, however, with attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a function is going to tell a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile-time precisely what piece of code will be executed. That’s the whole point—when the message is sent, the programmer doesn’t want to know what piece of code will be executed; the draw function can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type. If you don’t have to know what piece of code will be executed, then when you add a new subtype, the code it executes can be different without requiring changes to the function call. Therefore, the compiler cannot know precisely what piece of code is executed, so what does it do? For example, in the following diagram the BirdController object just works with generic Bird objects, and does not know what exact type they are. This is convenient from BirdController’s perspective because it doesn’t have to write special code to determine the exact type of Bird it’s working with, or that Bird’s behavior. So how does it happen that, when move( ) is called while ignoring the specific type of Bird, the right behavior will occur (a Goose runs, flies, or swims, and a Penguin runs or swims)?

The answer is the primary twist in object-oriented programming: the compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler causes what is called early binding, a term you may not have heard before because you’ve never thought about it any other way. It means the compiler generates a call to a specific function name, and the linker resolves this call to the absolute address of the code to be executed. In OOP, the program cannot determine the address of the code until run-time, so some other scheme is necessary when a message is sent to a generic object.

To solve the problem, object-oriented languages use the concept of late binding. When you send a message to an object, the code being called isn’t determined until run-time. The compiler does ensure that the function exists and performs type checking on the arguments and return value (a language in which this isn’t true is called weakly typed), but it doesn’t know the exact code to execute.

To perform late binding, Java uses a special bit of code in lieu of the absolute call. This code calculates the address of the function body, using information stored in the object (this process is covered in great detail in Chapter 7). Thus, each object can behave differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

# In some languages (C++, in particular) you must explicitly state that you want a function to have the flexibility of late-binding properties. In these languages, by default, member functions are not dynamically bound. This caused problems, so in Java dynamic binding is the default and you don’t need to remember to add any extra keywords in order to get polymorphism.

# 7: Polymorphism

# Polymorphism is the third essential feature of an object-oriented programming language, after data abstraction and inheritance.

# Constructors and polymorphism

7: Polymorphism 311Interchangeable objects with polymorphism 447: Polymorphism 311Constructors and polymorphism 330This chapter is an overview of what object-oriented programming is all about, including the answer to the basic question “What’s an object?”, interface vs. implementation, abstraction and encapsulation, messages and functions, inheritance and composition, and the all-important polymorphism. You’ll also get an overview of issues of object creation such as constructors, where the objects live, where to put them once they’re created, and the magical garbage collector that cleans up the objects that are no longer needed. Other issues will be introduced, including error handling with exceptions, multithreading for responsive user interfaces, and networking and the Internet. You’ll learn what makes Java special, why it’s been so successful, and about object-oriented analysis and design. Chapter 7: PolymorphismOn your own, you might take nine months to discover and understand polymorphism, a cornerstone of OOP. Through small, simple examples you’ll see how to create a family of types with inheritance and manipulate objects in that family through their common base class. Java’s polymorphism allows you to treat all objects in this family generically, which means the bulk of your code doesn’t rely on specific type information. This makes your programs extensible, so building programs and code maintenance is easier and cheaper.Java run-time type identification (RTTI) lets you find the exact type of an object when you have a reference to only the base type. Normally, you’ll want to intentionally ignore the exact type of an object and let Java’s dynamic binding mechanism (polymorphism) implement the correct behavior for that type. But occasionally it is very helpful to know the exact type of an object for which you have only a base reference. Often this information allows you to perform a special-case operation more efficiently. This chapter explains what RTTI is for, how to use it, and how to get rid of it when it doesn’t belong there. In addition, this chapter introduces the Java reflection mechanism.Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in this chapter. To demonstrate polymorphism, we want to write a single piece of code that ignores the specific details of type and talks only to the base class. That code is decoupled from type-specific information, and thus is simpler to write and easier to understand. And, if a new type—a Hexagon, for example—is added through inheritance, the code you write will work just as well for the new type of Shape as it did on the existing types. Thus, the program is extensible.What’s impressive about the code in doStuff( ) is that, somehow, the right thing happens. Calling draw( ) for Circle causes different code to be executed than when calling draw( ) for a Square or a Line, but when the draw( ) message is sent to an anonymous Shape, the correct behavior occurs based on the actual type of the Shape. This is amazing because, as mentioned earlier, when the Java compiler is compiling the code for doStuff( ), it cannot know exactly what types it is dealing with. So ordinarily, you’d expect it to end up calling the version of erase( ) and draw( ) for the base class Shape, and not for the specific Circle, Square, or Line. And yet the right thing happens because of polymorphism. The compiler and run-time system handle the details; all you need to know is that it happens, and more important how to design with it. When you send a message to an object, the object will do the right thing, even when upcasting is involved.Technically, OOP is just about abstract data typing, inheritance, and polymorphism, but other issues can be at least as important. The remainder of this section will cover these issues.In object-oriented programming, the most likely way that you’ll create and use code is by simply packaging data and methods together into a class, and using objects of that class. You’ll also use existing classes to build new classes with composition. Less frequently, you’ll use inheritance. So although inheritance gets a lot of emphasis while learning OOP, it doesn’t mean that you should use it everywhere you possibly can. On the contrary, you should use it sparingly, only when it’s clear that inheritance is useful. One of the clearest ways to determine whether you should use composition or inheritance is to ask whether you’ll ever need to upcast from your new class to the base class. If you must upcast, then inheritance is necessary, but if you don’t need to upcast, then you should look closely at whether you need inheritance. The next chapter (polymorphism) provides one of the most compelling reasons for upcasting, but if you remember to ask “Do I need to upcast?” you’ll have a good tool for deciding between composition and inheritance.Both inheritance and composition allow you to create a new type from existing types. Typically, however, you use composition to reuse existing types as part of the underlying implementation of the new type, and inheritance when you want to reuse the interface. Since the derived class has the base-class interface, it can be upcast to the base, which is critical for polymorphism, as you’ll see in the next chapter. It provides another dimension of separation of interface from implementation, to decouple what from how. Polymorphism allows improved code organization and readability as well as the creation of extensible programs that can be “grown” not only during the original creation of the project but also when new features are desired.Encapsulation creates new data types by combining characteristics and behaviors. Implementation hiding separates the interface from the implementation by making the details private. This sort of mechanical organization makes ready sense to someone with a procedural programming background. But polymorphism deals with decoupling in terms of types. In the last chapter, you saw how inheritance allows the treatment of an object as its own type or its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally. The polymorphic method call allows one type to express its distinction from another, similar type, as long as they’re both derived from the same base type. This distinction is expressed through differences in behavior of the methods that you can call through the base class.In this chapter, you’ll learn about polymorphism (also called dynamic binding or late binding or run-time binding) starting from the basics, with simple examples that strip away everything but the polymorphic behavior of the program.That’s exactly what polymorphism allows you to do. However, most programmers who come from a procedural programming background have a bit of trouble with the way polymorphism works.Again, you might expect that Shape’s draw( ) is called because this is, after all, a Shape reference—so how could the compiler know to do anything else? And yet the proper Circle.draw( ) is called because of late binding (polymorphism).// Polymorphism in Java.Now let’s return to the musical instrument example. Because of polymorphism, you can add as many new types as you want to the system without changing the tune( ) method. In a well-designed OOP program, most or all of your methods will follow the model of tune( ) and communicate only with the base-class interface. Such a program is extensible because you can add new functionality by inheriting new data types from the common base class. The methods that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.You can see that the tune( ) method is blissfully ignorant of all the code changes that have happened around it, and yet it works correctly. This is exactly what polymorphism is supposed to provide. Your code changes don’t cause damage to parts of the program that should not be affected. Put another way, polymorphism is one of the most important techniques that allow the programmer to “separate the things that change from the things that stay the same.”The order of constructor calls was briefly discussed in Chapter 4 and again in Chapter 6, but that was before polymorphism was introduced. Let’s take a look at an example that shows the effects of composition, inheritance, and polymorphism on the order of construction:// Constructors and polymorphismOnce you learn about polymorphism, it can seem that everything ought to be inherited because polymorphism is such a clever tool. This can burden your designs; in fact if you choose inheritance first when you’re using an existing class to make a new class, things can become needlessly complicated.That is, the base class can receive any message you can send to the derived class because the two have exactly the same interface. All you need to do is upcast from the derived class and never look back to see what exact type of object you’re dealing with. Everything is handled through polymorphism.Polymorphism means “different forms.” In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the dynamically bound methods.You’ve seen in this chapter that it’s impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like a switch statement can, for example), but instead works only in concert, as part of a “big picture” of class relationships. People are often confused by other, non-object-oriented features of Java, like method overloading, which are sometimes presented as object-oriented. Don’t be fooled: If it isn’t late binding, it isn’t polymorphism.To use polymorphism—and thus object-oriented techniques—effectively in your programs you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it’s a worthy struggle, because the results are faster program development, better code organization, extensible programs, and easier code maintenance.Add a new type of Shape to Shapes.java and verify in main( ) that polymorphism works for your new type as it does in the old types.Add a new type of Instrument to Music3.java and verify that polymorphism works for your new type.Although the features themselves are reasonably straightforward, the use of these features is a design issue, much the same as polymorphism. Over time, you’ll become better at recognizing situations where you should use an interface, or an inner class, or both. But at this point in this book you should at least be comfortable with the syntax and semantics. As you see these language features in use you’ll eventually internalize them.Set has exactly the same interface as Collection, so there isn’t any extra functionality like there is with the two different Lists. Instead, the Set is exactly a Collection, it just has different behavior. (This is the ideal use of inheritance and polymorphism: to express different behavior.) A Set refuses to hold more than one instance of each object value (what constitutes the “value” of an object is more complex, as you shall see). Consider the now familiar example of a class hierarchy that uses polymorphism. The generic type is the base class Shape, and the specific derived types are Circle, Square, and Triangle:This is a typical class hierarchy diagram, with the base class at the top and the derived classes growing downward. The normal goal in object-oriented programming is for the bulk of your code to manipulate references to the base type (Shape, in this case), so if you decide to extend the program by adding a new class (Rhomboid, derived from Shape, for example), the bulk of the code is not affected. In this example, the dynamically bound method in the Shape interface is draw( ), so the intent is for the client programmer to call draw( ) through a generic Shape reference. draw( ) is overridden in all of the derived classes, and because it is a dynamically bound method, the proper behavior will occur even though it is called through a generic Shape reference. That’s polymorphism.As a brief review of polymorphism and upcasting, you might code the above example as follows:Now polymorphism takes over and the exact method that’s called for the Shape is determined by whether the reference is for a Circle, Square, or Triangle. And in general, this is how it should be; you want the bulk of your code to know as little as possible about specific types of objects, and to just deal with the general representation of a family of objects (in this case, Shape). As a result, your code will be easier to write, read, and maintain, and your designs will be easier to implement, understand, and change. So polymorphism is the general goal in object-oriented programming.RTTI allows you to discover type information from an anonymous base-class reference. Thus, it’s ripe for misuse by the novice since it might make sense before polymorphic method calls do. For many people coming from a procedural background, it’s difficult not to organize their programs into sets of switch statements. They could accomplish this with RTTI and thus lose the important value of polymorphism in code development and maintenance. The intent of Java is that you use polymorphic method calls throughout your code, and you use RTTI only when you must.However, using polymorphic method calls as they are intended requires that you have control of the base-class definition because at some point in the extension of your program you might discover that the base class doesn’t include the method you need. If the base class comes from a library or is otherwise controlled by someone else, a solution to the problem is RTTI: You can inherit a new type and add your extra method. Elsewhere in the code you can detect your particular type and call that special method. This doesn’t destroy the polymorphism and extensibility of the program because adding a new type will not require you to hunt for switch statements in your program. However, when you add new code in your main body that requires your new feature, you must use RTTI to detect your particular type.Finally, RTTI will sometimes solve efficiency problems. If your code nicely uses polymorphism, but it turns out that one of your objects reacts to this general purpose code in a horribly inefficient way, you can pick out that type using RTTI and write case-specific code to improve the efficiency. Be wary, however, of programming for efficiency too soon. It’s a seductive trap. It’s best to get the program working first, then decide if it’s running fast enough, and only then should you attack efficiency issues—with a profiler.Inheritance and polymorphism are essential parts of object-oriented programming, but in the majority of cases when you’re putting together an application, what you really want is components that do exactly what you need. You’d like to drop these parts into your design like the electronic engineer puts together chips on a circuit board. It seems, too, that there should be some way to accelerate this “modular assembly” style of programming.The second reason for making FruitQualities a separate object is in case you want to add new qualities or to change the behavior via inheritance and polymorphism. Note that for GreenZebra (which really is a type of tomato—I’ve grown them and they’re fabulous), the constructor calls addQualities( ) and passes it a ZebraQualities object, which is derived from FruitQualities so it can be attached to the FruitQualities reference in the base class. Of course, when GreenZebra uses the FruitQualities it must downcast it to the correct type (as seen in evaluate( )), but it always knows that type is ZebraQualities.Make classes as atomic as possible. Give each class a single, clear purpose. If your classes or your system design grows too complicated, break complex classes into simpler ones. The most obvious indicator of this is sheer size: if a class is big, chances are it’s doing too much and should be broken up.
Clues to suggest redesign of a class are:
1) A complicated switch statement: consider using polymorphism.
2) A large number of methods that cover broadly different types of operations: consider using several classes.
3) A large number of member variables that concern broadly different characteristics: consider using several classes.Watch for switch statements or chained if-else clauses. This is typically an indicator of type-check coding, which means you are choosing what code to execute based on some kind of type information (the exact type may not be obvious at first). You can usually replace this kind of code with inheritance and polymorphism; a polymorphic method call will perform the type checking for you, and allow for more reliable and easier extensibility.constructor · 191; and anonymous inner classes · 370; and exception handling · 562; and exceptions · 561; and finally · 562; and overloading · 194; and polymorphism · 330; arguments · 193; base-class constructor · 332; base-class constructors and exceptions · 281; behavior of polymorphic methods inside constructors · 337; C++ copy constructor · 1042; calling base-class constructors with arguments · 280; calling from other constructors · 205; default · 202; default constructors · 196; initialization during inheritance and composition · 281; name · 192; no-arg constructors · 196; order of constructor calls with inheritance · 330; return value · 193; static construction clause · 228; synthesized default constructor access · 681decoupling: via polymorphism · 46decoupling through polymorphism · 311polymorphism · 44, 311, 346, 660, 685; and constructors · 330; behavior of polymorphic methods inside constructors · 337run-time binding · 316; polymorphism · 311