Chapter 7 : Polymorphism

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

Polymorphism allows improved code organization and readability as well as the creation of *extensible* programs.

Inheritance allows the treatment of an object as its own type or its base type.

This ability 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 types.

7.1 Upcasting

Upcasting -- Sample : Music.java

Overloading -- Sample : Music2.java

How about just writing a single method that takes the base class as its argument?

7.1.1 Method-call binding

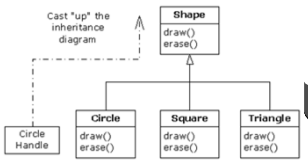
early binding ⟷ *late binding*

compile time | run time

When a language implements late binding, there must be some mechanism to determine the type of the object at run-time and to call the appropriate method. ---- Some sort of type information must be installed inside the objects.

All method binding in Java uses late binding except final methods.

7.1.2 Producing the right behavior

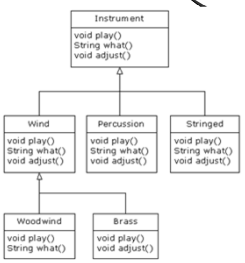


Sample : Shapes.java

The base class **Shape** establishes the common interface to anything inherited from it. The derived classes override these definitions to provide unique behavior.

7.1.3 Extensibility

You can add new functionality by inheriting new data types from the common base class. ---- extensible



Sample : Music3.java

The **tune( )** method is blissfully ignorant of all the code changes that have happened around it, and yet it works correctly.

7.2 Abstract classes and methods

Java provides a mechanism called the *abstract method*. This is a method that is incomplete; it has only a declaration and no method body.

```
abstract void f();
```

A class containing abstract methods is called an *abstract class*. You can't create any instance of it.

If a class contains one or more abstract methods, the class must be qualified as **abstract**.

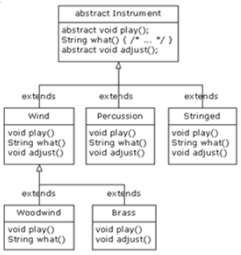
Making a class abstract doesn't force you to make all the methods **abstract**.

If you inherit from an abstract class and want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class.

It's possible to create a class as **abstract** without including any **abstract** methods. ---- just prevent any instances

Sample : Music4.java

It's helpful to create **abstract** classes and methods because they make the abstractness of a class explicit, and tell both the user and the compiler how it was intended to be used.



7.3 Constructors and polymorphism

7.3.1 Order of constructor calls

Sample : Sandwich.java

The order of constructor calls for a complex object :

1. The base-class constructor is called. This step is repeated recursively.
2. Member initializers are called.
3. The body of the derived-class constructor is called.

7.3.2 Behavior of polymorphic methods inside constructors

Sample : PolyConstructors.java

The actual process of initialization is : The storage allocated for the object is initialized to binary zero before anything else happens.

A good guideline for constructors is, "Do as little as possible to set the object into a good state, and if you can possibly avoid it, don't call any methods."

7.4 Designing with inheritance

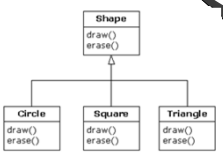
A better approach is to choose composition first, when it's not obvious which one you should use.

Composition is more flexible since it's possible to dynamically choose a type (and thus behavior).

Sample : Transmogrify.java

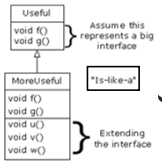
A general guideline is "Use inheritance to express differences in behavior, and fields to express variations in state."

7.4.1 Pure inheritance vs. extension



is-a

The base class can receive any message you can send to the derived class.

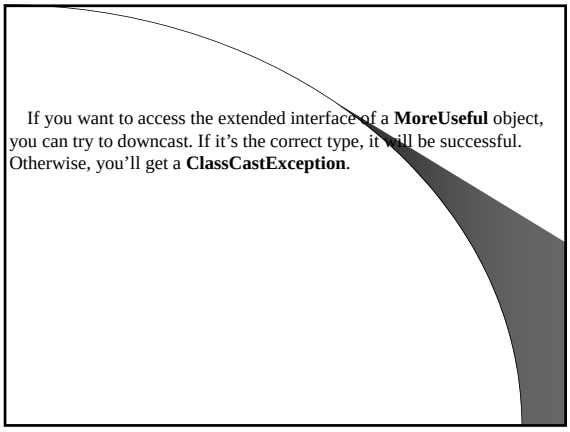


The extended part of the interface in the derived class is not available from the base class, so once you upcast you can't call the new methods.

7.4.2 Downcasting and RTTI

Even though it looks like an ordinary parenthesized cast, at run-time this cast is checked to ensure that it is in fact the type you think it is.

Sample : RTTI.java


If you want to access the extended interface of a **MoreUseful** object, you can try to downcast. If it's the correct type, it will be successful. Otherwise, you'll get a **ClassCastException**.