Class Hierarchy and Polymorphism, continued

First, we demonstrated that if Rectangle has code that uses the height or width fields directly (other than in the constructor), it breaks how Square works.

Square s = new Square(3);

s.setWidth(4)

s.getWidth() <-- returns 4

s.getHeight() <-- returns 4

Rectangle r = s <- legal, a Square "is-a" Rectangle

r.setHeight(2)

r.getHeight() <--- returns 2

r.getWidth() <--- returns 2

- Why did the getWidth return 2? Remember how polymorphism works! r's type is Rectangle so the current type of the object stored in r is Rectangle.

But, above that object's true type is Square. All objects know their true type and the true type is used to call the correct method.

Since the object's true type is Square, it will use the overridden methods.

Next, we looked at why we have private fields and public getter setter methods:

Rule: If you want to make your class easy to extend, you must use getter/setter methods instead of the fields in your code.

Any method that directly accesses the fields directly may need to be overridden in an extending class. By limiting the field access to the getter/setter methods, you reduce the amount of work needed to extend the class.

Example: Suppose we create two methods in Rectangle:

public void doubleWidth() {

width \*= 2;

}

public void doubleHeight() {

height \*= 2;

}

These methods access the fields directly. The methods will break how Square works - no matter which version we decided to write!

So, Square will need to override these methods OR we could have just coded the methods better:

public void doubleWidth() {

setWidth(getWidth() \* 2);

}

public void doubleHeight() {

setHeight(getHeight() \* 2);

}

Now, both methods can be inherited by Square and work correctly without requiring overriding!

Two important Object methods

All (non-private) methods of Object are inherited by every class in Java.

Here are two methods that usually need to be overridden.

1) String toString()

This method produces a String representation of the object. It is used by System.out.print when writing output to the terminal. It is also used by the String concatenation operator + when you concatenate objects to a string.

The method in Object sets the string to be the true type name of the object and it's address in memory.

Usually we want the string to be something more useful, so we override the method.

2) boolean equals(Object o)

This method is used to compare two objects. If you use the == operator on non-primitive values, it is comparing the addresses of the values. Thus, == only returns true if two objects are the exact same object.

For example "Hi" == "Hi" only returns true if the two Strings are really the same string stored at the same location in memory. If you create a String s that contains "Hi" but is stored at a different location in memory,

then s == "Hi" will return false.

The equals method is provided to examine the contents of the object to determine if they are structurally equal.

However, the equals method will not compare the contents of objects by default. The equals method in Object just does an == test. Instead, we must override the method.

For example, the String class overrides the equals method to compare the individual characters of the string. Thus "Hi".equals("Hi") always returns true.

IMPORTANT: To override the method, we must exactly match the method parameter signature.

For example, if we have

public boolean equals(Rectangle r) {

we are not overriding equals, we are overloading it. (Note: many textbooks and on-line sources incorrectly use the above as the correct way to write equals!)

If you overload instead of override, the equals method will not work correctly in all situations!

Suppose we want to rectangles to be equal if they have the same height and width.

Rectangle r1 = new Rectangle(100,200)

Rectangle r2 = new Rectangle(100,200)

With the above definition, r1.equals(r2) returns true. However, r1.equals((Object)r2) returns false! Can you see why? Typecasting should not affect the behavior! This error would not happen if we overrode the equals method correctly.

A correct equals method:

public boolean equals(Object o) {

if (o instanceof Rectangle) {

Rectangle r = (Rectangle)o;

return (this.getWidth() == r.getWidth() && this.getHeight() == r.getHeight());

}

else

return false;

}

instanceof: returns true if object that is the left operand can be typecast as the type that is the right operand. (I.e. is the object's true type equal or "below" the given type in the hierarchy.)

If we do not use instanceof, the method will generate a TypeCastException when typecasting a non-Rectangle to Rectangle.

Remember the polymorphism rules:

- An object is all the types of its hierarchy, from the class it is created as to the classes and interfaces that the class extends and implements.

- The true type of the object is what it was created as by the new operator. The true type is stored in the object instance in the heap.

- The current type is how the object is being used as in a specific Java program line. The current type constantly changes with typecasts, method calls, method returns, variable assignment, etc.

- The compiler uses the current type to verify that all values are used correctly

- The compiler determines that every method that is called exists for that current type.

- The compiler determines which field of the object is accessed.

- The true type is used when the program is running. For each method call, the true type's method is called.

A Final Example: Making a JButton that is square

Most programmers do not take proper advantage Java polymorphism. As an example, try looking up on-line how to make a JButton that is always square. The answers are rarely correct because they all focus on doing things -to- the button.

However, whatever you do to the button, there are other parts of the Java program that can undo your changes.

Instead, we need to think object-oriented. Instead of doing things to the button, have the button know what it is and how it should behave.

Let us create a class SquareButton that knows that it is a SquareButton. Whenever any code requests the button's width or height, the button knows it is a SquareButton and will always respond with the same value for each dimension.

Because the creators of JButton used correct polymorphsim style (they use the getWidth and getHeight methods throughout their code instead of the widht and height fields), we only need to override those methods to make the button always square.

So, the only thing SquareButton needs is to override the getHeight() method to instead return the width of the button.

To make the result display a little nicer, we will also override the getWidth() method to use the lesser of the width and height. This way, we won't have the button's dimension larger than what the layout mananger wants the dimension to be.