Readability

Outline for Week 9

I. Readability

II. Polymorphism

  A. Dynamic method invoc.
  B. Overloading vs. overriding

III. Exercise: Singleton

IV. Exercise: Adapter

"Programs must be written for people to read, and only incidentally for machines to execute."—Abelson & Sussman

Guideline: Give a variable the narrowest scope that you can.

Give an example of this principle.

Why is this a good principle?

Guideline: Using standard idioms, make code as concise as possible.

Example: In the following statement, **b** is a boolean variable:

```
if (b == true)
  return true;
else
  return false;
```

This statement is far too verbose. An equivalent and much more readable statement is—

In most cases, this can be made even more readable. How?

Guideline: Variable names should be neither too short nor too long.

Consider a variable that controls whether a **while**-loop is exited.

```
while (variable) {
…
}
```

What is a good name for this variable?

Which should be shorter, in general? Variable names or names of constants?

In general, names that are used less frequently can be longer.

Guideline: Names should be descriptive of the entity they apply to. They should not be vague or overly general.

Give an example of a bad (variable, method, etc.) name you have encountered in code that you were refactoring or interfacing to.

Here are some examples from Expertiza.

Guideline: Names should not be redundant.

Suppose `course_controller.rb` contains a method called **`create_course`**. What should it be?

Suppose it contains a method called **`create_section`**. What should it be?

An excellent discussion of variable naming can be found in *Code Complete*, by Steve McConnell, on electronic reserve for this course.

Guideline: Factor out duplicated code.

If a program has two places where the same sequence of instructions is being executed, it is almost always beneficial to move the duplicated code into a separate procedure.

Example: Suppose you are developing a class of objects one of whose responsibilities is to parse an input string, such as a complicated mathematical expression.

Part of the process of parsing involves checking that the input is valid. So the class might have a method like this:

```
public void parse(String expression)
{
  ...do some parsing...
  if( ! nextToken.equals("+") )  {
    //error
    System.out.println
      ("Expected +, but found " + nextToken);
    System.exit(1);
  }
  ...do some more parsing...
  if( ! nextToken.equals("*") )  {
    //error
    System.out.println
      ("Expected *, but found " + nextToken);
    System.exit(1);
  }
  ...
}
```

How can we clean this code up?

```
  private void handleError(String message) {
    System.out.println(message);
    System.exit(1);
  }

  public void parse(String expression)
  {
    ...do some parsing...
    if( ! nextToken.equals("+") )
    ...do some more parsing...
    if( ! nextToken.equals("*") )


    ...
  }
```

Besides being more readable, this code has another advantage. What?

Guideline: A method should do only one thing and do it well.

Here's an example of a method to avoid:

```
  void doThisOrThat(boolean flag) {
    if( flag ) {
      ...twenty lines of code to do this...
    }
    else {
      ...twenty lines of code to do that...
    }
  }
```

How should we change it?

Polymorphism

Unbounded vs. subtype polymorphism

In a statically typed o-o language like Java or C++, you can declare a variable in a superclass, then assign a subclass object to that type:

```
public class Bicycle {
  protected int gear;
  public void setGear(int nextGear) {
    gear = nextGear;
  }
}
public class MountainBike extends Bicycle {
  protected int seatHeight;
  public void setHeight(int newSeatHeight) {
    seatHeight = newSeatHeight;
  }
}
public class BikeSim {
  public static void main() {
    ...
    Bicycle myBike = new MountainBike();
    ...
    myBike.setGear(3);
    myBike.setHeight(5);
    }
}
```

Which statement is illegal in the code above? Why?

In most dynamically typed o-o languages, including Ruby, that statement would be legal. In Ruby, if a method is defined on an object, the object can respond to the message.

It doesn't matter what class the object is declared as … in fact, the object isn't declared!

This is called *unbounded polymorphism*—the polymorphism is not limited by the declared class of the object.

In contrast, statically typed o-o languages usually have *subtype polymorphism*—the compiler checks that the invoked method is defined in the type that the object is declared as.

Unbounded polymorphism is related to *duck typing*, which was discussed in the Week 3 online lectures [§2.4 of the textbook].

Dynamic method invocation

A call to an inherited method works just as if the inherited method had been defined in the caller's class.

But suppose the subclass (e.g., **`MySpiffyLabel`**) overrides a method of the superclass (e.g., **`JLabel`**).

```
JLabel label = new MySpiffyLabel("A label");
label.paint(g);  //for some Graphics object g
```

`MySpiffyLabel`

- inherits a **`paint`** method from **`JLabel`**, and
- implements its own version of **`paint`**.

Which of those two implementations of **`paint`** will be executed in the second line of above example?

- The paint defined in **`JLabel`**?
- The paint defined in **`MySpiffyLabel`**?

Dynamic method invocation: To invoke a method on an object, the JRE looks at the class of the receiving *object* to choose which version to execute.

For example, when asked to execute **`label.paint(g)`**, the Java environment does not look in the *declared class* of **`label`** (namely, **`JLabel`**).

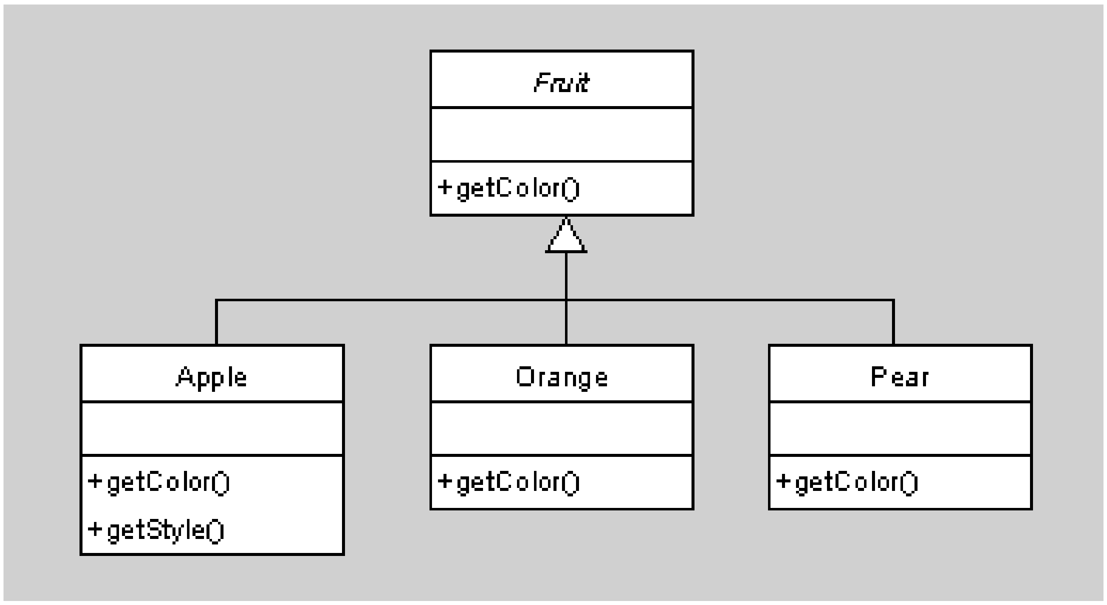
Instead it chooses the **`paint`** method in the *actual class* of the object referred to by **`label`** (namely, **`MySpiffyLabel`**).

When a method is called on an object of a subclass that overrides a superclass method, the *overriding* version of the method is always called.

Let us consider a rather tricky, but illustrative, example.

Abstract class **`Fruit`** has subclasses **`Apple`**, **`Orange`**, and **`Pear`**.

Since it is an abstract class, its name is shown in italics in the class diagram.



Note that Apple has a **`getStyle()`** method to return the kind of apple (**`Delicious`**, **`McIntosh`**, etc.).

Because of subtype polymorphism, it is legal to declare a variable as being of some class and then assign an object of a subclass to it:

```
Fruit fruit = new Apple("McIntosh");
```

Suppose that we have several fruits, and want to print out the colors of each. This code will do the trick:

```
Fruit[] A = new Fruit[3];
```

```
A[0] = new Apple("Granny Smith");
A[1] = new Orange();
A[2] = new Pear();
for( int i = 0; i < A.length; i++ ) {
  if( A[i] instanceof Apple )
    System.out.println(
      ((Apple) A[i]).getColor());
  else if( A[i] instanceof Orange )
    System.out.println(
      ((Orange) A[i]).getColor());
  else if( A[i] instanceof Pear )
    System.out.println(((Pear)
A[i]).getColor());
  else
    System.out.println(A[i].getColor());
```

What’s wrong with this?

How can we simplify it?

What would happen if no **`getColor`** method were defined in **`Fruit`**?

Overloading vs. overriding

Two methods are *overloaded* if they are in the same class, but have different parameter lists.

When a method is *overridden*, one of its *subclasses* declares a method of the same name, with the same signature.

Consider this example. All of our **`Fruit`**s inherit an **`equals`** method from class **`Object`**. Suppose that **`Fruit`** declares its own **`equals`** method:

```
Object>>public boolean equals(Object obj)      (1)

Fruit>>public boolean equals(Fruit fruit)      (2)
```

Has **`Fruit`** overridden the **`equals`** method?

Which **`equals`** method is called in each case below?

```
Object o = new Object();
Fruit f = new Fruit();
Object of = new Fruit();
f.equals(o);
f.equals(f);
f.equals(of);
```

What about these calls, using the same variables?

```
o.equals(o);
o.equals(f);
o.equals(of);
of.equals(o);
of.equals(f);
of.equals(of);
```

Now, let's throw overriding into the picture and declare, in class **`Fruit`**—

```
Object>>public boolean equals(Object obj)      (1)
Fruit>>public boolean equals(Fruit fruit)      (2)
Fruit>>public boolean equals(Object obj)       (3)
```

Which methods are called now?

```
Object o = new Object();
Fruit f = new Fruit();
Object of = new Fruit();
f.equals(o);
f.equals(f);
f.equals(of);
o.equals(o);
o.equals(f);
o.equals(of);
of.equals(o);
of.equals(f);
```

```
  of.equals(of);
```

In summary, the compiler decides which overloaded method to call by looking at the declared type of

- the object being sent the message and
- the declared types of the arguments to the method call.

The particular version of the overloaded method is chosen at runtime by dynamic method invocation using the actual type of the object being sent the message.

The actual classes of the arguments to the method call do not play a role.

This is very different from a language like CLOS, which uses the actual types of the arguments to decide which method to execute.

Exercise: Singleton pattern

In the Week 5 video lecture, we saw the Singleton pattern defined in Ruby.

```
require 'singleton'
class Registry
  include Singleton
  attr_accessor :val
end
r = Registry.new #throws a NoMethodError
r = Registry.instance
r.val = 5
s = Registry.instance
puts s.val   >> 5
s.val = 6
puts r.val  >> 6
s.dup  >> TypeError: can’t duplicate instance of singleton Registry
```

The idea is to prevent more than one object of the class from being defined, and to return the single instance by using a class method.

Here is an exercise with another Singleton pattern, except blanks are left in the code. You need to fill in the blanks to get the code to run.

```
class Balance
  attr_reader __________(1)__________
  def __________(2)__________(balance)
    @balance = balance
    __________(3)__________ = nil
  end

  def __________(4)__________.instance
    @first_instance = __________(5)__________(100)
                                  if @first_instance.nil?
    __________(6)__________
  end

  def withdraw(amount)
    @balance > amount ? (@balance -= amount) :
           (puts 'Insufficient balance')
  end

  def deposit(amount)
    @balance += amount
  end
end

class FamilyMember
  def initialize(name)
    @name = name
    @balance = Balance.__________(7)__________
  end

  def withdraw(amount)
    __________(8)__________(amount)
  end

  def deposit(amount)
    __________(9)__________(amount)
  end

  def balance
    __________(10)__________
  end
end
```

Fill in the blanks in the `Singleton` class and the `FamilyMember` class. Note that Singleton is not implemented as a mixin, though it could be.

Exercise: Adapter Pattern

An *adapter* allows classes to work together that normally could not because of incompatible interfaces.

- It "wraps" its own interface around the interface of a pre-existing class. What does this mean?

- It may also translate data formats from the caller to a form needed by the callee.

One can implement the Adapter Pattern using delegation in Ruby. Consider the following contrived example.

- We want to put a `SquarePeg` into a `RoundHole` by passing it to the hole's `peg_fits?` method.

- The `peg_fits?` method checks the `radius` attribute of the peg, but a `SquarePeg` does not have a radius.

- Therefore we need to adapt the interface of the `SquarePeg` to meet the requirements of the `RoundHole`.

```
class SquarePeg
   attr_reader :width
   def initialize(width)
      @width = width
   end
end
```

```
class RoundPeg
   attr_reader :radius
   def initialize(radius)
      @radius = radius
   end
end
```

```
class RoundHole
    attr_reader :radius

    def initialize(r)
        @radius = r
    end

    def peg_fits?(peg)
        peg.radius <= radius
    end
```

```
        end
```

Here is the Adapter class:

```
        class SquarePegAdapter
            def initialize(square_peg)
                @peg = square_peg
            end

            def radius
                Math.sqrt(((@peg.width/2) ** 2)*2)
            end
        end

        hole = RoundHole.new(4.0)
        4.upto(7) do |i|
            peg = SquarePegAdapter.new( SquarePeg.new(i.to_f) )
            if hole.peg_fits?( peg )
                puts "peg #{peg} fits in hole #{hole}"
            else
                puts "peg #{peg} does not fit in hole #{hole}"
            end
        end
```

```
>>peg #<SquarePegAdapter:0xa038b10> fits in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa038990> fits in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa0388a0> does not fit in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa038720> does not fit in hole
#<RoundHole:0xa038bd0>
```

Here is an exercise on the Adapter pattern. Fill in the blanks.

```
interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void ______(2)______();
}

class Sparrow implements ___(1)___
{
    // a concrete implementation of bird
    public void ____(4)___()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}

interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}

class PlasticToyDuck implements ToyDuck
{
    public void _____(3)____()
    {
        System.out.println("Squeak");
    }
}

class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        this.bird = bird;
    }

    public void squeak()
    {
        bird.______(5)_______();
```

```java
    }
}

class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```