The Essence of O-O

Classes vs. instances

Define the difference between a class and an instance.

Class variables/methods vs. instance variables/methods

How are class variables and class methods declared in Java?

Outline for Week 8

I. Classes vs. instances

III. UML diagrams

IV. Reasons for inheritance

V. Inheritance and interfaces

VI. Polymorphism

A. Dynamic method invoc.

B. Overloading vs. overriding

What are some ways that class variables can be used?

What are some ways that class methods can be used?

Sometimes, we have a choice of whether to use a class or instance method.

Suppose you are defining a Set class and need a method to determine whether one Set intersects another.

Do you define it as an instance method …

public Set intersect(Set other)

or as a class method?

public static Set intersect(Set first, Set second)

What are the advantages of defining it as an instance method?

•

•
What are the advantages of defining it as a class method?

•
•

Try to think of another example of where a method can be written as an instance method or a class method. Which is better?

Type vs. class

A type can be thought of as a set of data values and the operations that can be performed on them.

For example the int primitive type in Java can be thought of as the set of all 32-bit integers together with the set of operations that can be performed on integers.

In this course, an object type consists of a set of operations (a “protocol”) and a set of objects that can perform those operations.

In Java, types can be defined in two ways:

• by classes
• _________

When I declare a variable, I need to give its type. Its type may be either a class

    JLabel myLabel;

or an interface

    Runnable r;
Note that objects of a subclass \( S \) of a class \( T \) are considered to be of type \( S \) and of type \( T \).

A subclass of a class defines a subtype of the superclass’s type.

One type \( S \) is a *subtype* of another type \( T \) (which, in turn, is called a *supertype* of \( S \)) iff

- the set of objects of type \( S \) are a subset of the set of objects of type \( T \) and
- the set of operations of \( S \) are a superset of the operations of \( T \).

Note that, if type \( S \) is a subtype of type \( T \), the set of operations of \( S \) must include all the operations of \( T \) (and possibly more).

**Polymorphism**

In Ruby, polymorphism is *unbounded*. What does that mean?

Java has *subtype* polymorphism: An object of a subtype can legally invoke any operation that an object of its supertype could invoke.

*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.

![Class Diagram](image)

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:

```java
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:

```java
Fruit[] A = new Fruit[3];
A[0] = new Apple("Granny Smith");
A[1] = new Orange();
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**

[Skrien §2.8] 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 Fruits inherit an equals method from class Object. Suppose that Fruit declares its own equals method:

Object>>public boolean equals(Object obj)  

Fruit>>public boolean equals(Fruit fruit)

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)  
Fruit>>public boolean equals(Fruit fruit)  
Fruit>>public boolean equals(Object obj)

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 last week’s video lecture, we saw the Singleton pattern defined in Ruby:

```ruby
class Single
  def initialize
    # Initialize an instance of the class
  end

  def self.instance
    return @@instance if defined? @@instance
    @@instance = new
  end
end
```
end

def private_class_method :new
end

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.

Now in the .zip file associated with today’s lecture, we have the Singleton pattern defined in Java, except blanks are left in the code. You need to fill in the blanks to get the code to run.

```java
class Balance {

    _______ ______ Balance firstInstance = null; //>>1. Fill in the keywords private int balance;

    ______________________ { //>>2. Fill in the header for the missing routine
        this.balance = balance;
    }

    public static Balance getInstance (){,
        if (____________________) { //>>3. Fill in the condition
            firstInstance = new Balance(10000);
        } return firstInstance;
    }

    public void withdraw (int amount){
        if (balance>amount)
            balance -= amount;
        else
            System.out.println("Insufficient balance");
    }

    public void deposit (int amount){
        balance += amount;
    }

    public int getBalance (){,
        return balance;
    }
}
```

© 2017 Edward F. Gehringer    CSC/ECE 517 Lecture Notes, Spring 2017
class FamilyMember {
    __________ b; //>> 4. Fill in the blank
    String name;

    public FamilyMember (String name) {
        this.name = name;
        b = __________; //>> 5. Fill in the blank
    }

    public void withdraw (int val) {
        __________.withdraw(val); //>> 6. Fill in the blank
    }

    public void deposit (int val) {
        __________.deposit(val); //>> 7. Fill in the blank
    }

    public int getBalance() {
        return __________.getBalance(); //>> 8. Fill in the blank
    }
}

Note that in Java, the Singleton pattern needs to be defined in each class that uses it, rather than being required as a mixin. Fill in the blanks in the Singleton class and the FamilyMember class.