Abstraction, Encapsulation, and Inheritance

CS211
Fall 2000

Why Objects?

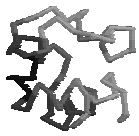
The use of objects encourages

- Abstraction
  - An abstraction denotes the essential properties of an object
  - One of the fundamental ways in which we handle *complexity*
  - Programming goal: choose the right abstractions
- Encapsulation (information hiding)
  - No direct access to the parts of an object
  - No dependence on the object's *implementation*

2

Multiple Abstractions

- A single thing can have multiple abstractions
- Example: a protein is…
  - a sequence of amino acids
  - a complicated 3D shape (a *fold*)
  - a surface with "pockets" for *ligands*

3

Choosing Abstractions

- Abstractions can be about
  - tangible things (a vehicle, a car, a map) or
  - intangible things (a meeting, a route, a schedule)
- An example:

- What are the essential properties of this "thing"?
  - Abstraction name: *light*
  - Light's wattage (i.e., energy usage)
  - Light can be *on* or *off*
- There are other possible properties (shape, color, socket size, etc.), but we have *decided* those are less essential
- The *essential* properties are determined by the problem

4

Modeling Abstraction using Classes

A class defines

- all attributes/properties and
- all behaviors/operations of an abstraction

- In Java…
  - Attributes/properties correspond to *fields* (or *variables*)
  - Behaviors/operations correspond to *methods*

```
class light {
    // Instance variables
    private int wattage;
    private boolean on;

    // Instance methods
    public void switchOn ( ) { on = true; }
    public void switchOff ( ) { on = false; }
    public boolean isOn ( ) { return on; }
    public int getWattage ( )
            { return wattage; }
}
```

5

Encapsulation

- Classes support a particular kind of abstraction, encouraging separation between
  - an object's *operations* and
  - the *implementations* of those operations
- This allows and encourages *encapsulation*
  - Objects are regarded as "black boxes" whose internals are hidden
  - Separation of *contract* (i.e., what operations are available) and *implementation*

6

Contract vs. Implementation

- A class can be viewed as a *contract*; the contract specifies
  - *what* operations are offered by the class
  - In Java, this corresponds to the method headings for the methods that are *public*
- A class can be viewed as an *implementation*; the implementation specifies
  - *how* the desired behavior is produced
  - In Java, this corresponds to the method-bodies and the (nonpublic) instance variables

7

Programming Implications

- Encapsulation makes programming easier
  - As long as the contract is the same, the client doesn't care about the implementation
  - In Java, as long as the method signatures are the same, the implementation details can be changed
- In other words, I can write my program using *simple* implementations; then, if necessary, I can replace some of the simple implementations with *efficient* implementations

8

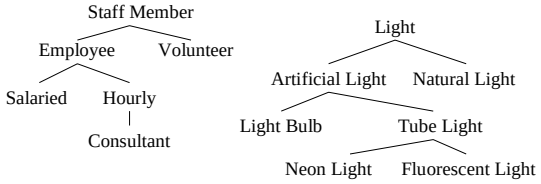
Recall Basics Ideas of OOP

- Objects
  - Allow and encourage
    - ▲ Abstraction
    - ▲ Encapsulation
- Classes
  - Templates for producing multiple objects
- *Inheritance*
  - Allows and encourages
    - ▲ Extensibility
    - ▲ Code reuse
- A distinction is sometimes made between
  - Object *Based* Programming
    - ▲ Objects
    - ▲ Classes
  - Object *Oriented* Programming
    - ▲ Objects
    - ▲ Classes
    - ▲ Inheritance

9

Inheritance

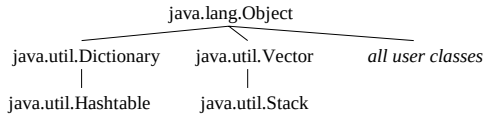
- Inheritance = natural, hierarchical way of organizing things
- Based on the "is-a" relationship



10

Another Inheritance Hierarchy

- Higher in the hierarchy implies
  - More generalized
- Lower in the hierarchy implies
  - More specialized
  - with additional properties and behaviors



11

Example: Animal Class

```
class Animal {
    protected String name = "";
    protected String noise = "";

    public setName (String myName) {
          name = myName;
          }
    public void identifySelf ( ) {
          System.out.println(
            "My name is " + name);
          }
    public void perform ( ) {
          }
    }
```

- An Animal has a name and a noise, it can identify itself and perform.
- What happens?

```
Animal harpo = new Animal( );
harpo.setName("Harpo");
harpo.perform( );
// Output:
//   Says nothing
```

12

A Dog is an Animal

```
class Dog extends Animal {
    public Dog ( ) {
        noise = "woof";
        }
    public void perform ( ) {
        identifySelf( );
        System.out.println("I am a Dog");
        System.out.println(noise);
        }
    }
```

- What Happens?

```
Dog snoopy = new Dog( );
snoopy.setName("Snoopy");
snoopy.perform( );
// Output:
//   My name is Snoopy
//   I am a Dog
//   woof
```

- Dog inherits name, noise, setName( ) and identifySelf( ) from Animal
- Method perform( ) is

A BigDog is a Dog

```
class BigDog extends Dog {
    public BigDog ( ) {
        noise = "WOOF";
        }
    }
```

- What Happens?

```
BigDog fang = new BigDog( );
fang.setName("Fang");
fang.perform( );
// Output:
//   My name is Fang
//   I am a BigDog
//   WOOF
```

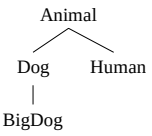
- BigDog inherits name, noise, setName( ) and identifySelf( ) from Animal
- BigDog inherits perform( ) from Dog

A Human is an Animal

```
class Human extends Animal {
    public Human ( ) {
        noise = "I think, therefore I am";
        }
    public void perform ( ) {
        identifySelf( );
        System.out.println(
                "I am a sentient being");
        System.out.println(noise);
        }
    }
```

- What Happens?

```
Human descartes = new Human( );
descartes.setName("Rene");
descartes.perform( );
// Output:
//   My name is Rene
//   I am a sentient being
//   I think, therefore I am
```



Inheritance and Scope in Java

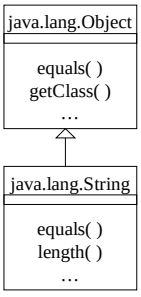
For Variable (e.g., noise)

- Java first examines current methods for local variables or parameters
- Then examines variables of current class
- Then examines variables of superclass
- Continues up hierarchy until no more superclasses

For Methods (e.g., perform( ) & identifySelf( ))

- Java first examines methods of current class
- Then examines methods of superclass
- Continues up hierarchy until no more superclasses

Illustrating Inheritance




```
class Example {
    public static void Main (String[ ] args) {

        String string = new String("Java");

        System.out.println(string.getClass( ));
        System.out.println(string.length( ));

        Object obj = string; string = null;
//      System.out.println(obj.length( )); // error
        System.out.println(obj.equals("Java"));
        System.out.println(obj.getClass( ));

        string = (String) obj;
        System.out.println(string.equals("C++");
        }
    }
```


Illustrating Inheritance - Output

Output:

```
class Java.lang.String
4

true
class Java.lang.String

false
```

```
class Example {
    public static void Main (String[ ] args) {

        String string = new String("Java");

        System.out.println(string.getClass( ));
        System.out.println(string.length( ));

        Object obj = string; string = null;
//      System.out.println(obj.length( )); // error
        System.out.println(obj.equals("Java"));
        System.out.println(obj.getClass( ));

        string = (String) obj;
        System.out.println(string.equals("C++");
        }
    }
```

What Was Illustrated?

- Inheriting from the superclass

```
System.out.println(string.getClass( ));
```

- Extending the superclass (with the new method length( ))

```
System.out.println(string.length( ));
```

- Upcasting

```
Object obj = string;
```

  - But can't use methods exclusive to subclass

```
System.out.println(obj.length( )); // error
```

- Method Overriding

```
System.out.println(obj.equals("Java"));
```

- Polymorphism and Dynamic Method Binding

```
System.out.println(obj.equals("Java"));
System.out.println(obj.getClass( ));
```

- Downcasting

```
string = (String) obj;
System.out.println(string.equals("C++");
```

19

Overriding vs. Overloading

- Overriding
  - New method has same *method signature* and same return type
  - The syntax super.method( ) can be used to access the method in the superclass
  - Occurs only in subclasses
- Overloading
  - Requires *different* method signature, *same* method name
  - Return type is *not* part of the signature; cannot overload by just changing return type
  - Can occur in subclasses or in same class

20

Polymorphism & Dynamic Method Binding

- Polymorphism
  = the ability of a variable to hold objects of its own class and its subclasses at runtime

```
Object obj = string;
```

- Dynamic Method Binding
  = the method invoked depend on the *actual type* of the reference

```
System.out.println(obj.equals("Java"));
System.out.println(obj.getClass( ));
```

- Note: the method called depends on the *declared type* of any arguments, *not* the actual type

21

Downcasting

- To cast a superclass variable to a subclass, explicit casting is required

```
string = (String) obj;
```

- Downcasting can be *invalid* at runtime
  - A ClassCastException can be thrown
  - Use the operator instanceof to determine the runtime type of an object

```
if (obj instanceof String) {
    string = (String) obj;
    System.out.println(string.length( ));
    }
```

22