Today ...

- Quiz 3
- New office hours: 4-5pm (except Wednesdays)
- Misc.
  - Note on Applet example
  - Casting
  - Implicit vs. explicit parameters
- More on Inheritance
  - Constructors
  - The cosmic superclass
  - Overriding
  - Abstract classes and Interfaces
- Lab 1
- Reading assignments
  - Core: Ch. 5 pages 184-192
  - Core: Ch. 6 pages 241-249
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

public class HelloWorld extends JApplet {
    static final long serialVersionUID = 13L; // for serializable
    public void init() {
        Runnable r = new Runnable() {
            public void run() {
                add(new JLabel("Hello World!", SwingConstants.CENTER));
            }
        };
        try {
            SwingUtilities.invokeLater(r);
        } catch (Exception e) {
            e.printStackTrace();
            http://www.cs.gonzaga.edu/~yourlogin/cpsc324/HelloWorld.html
        }
    }
}

import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

public class HelloWorld extends JApplet {
    static final long serialVersionUID = 13L;
    public void init() {
        Runnable r = new Runnable() {
            public void run() {
                add(new JLabel("Hello World!", SwingConstants.CENTER));
            }
        };
        try {
            SwingUtilities.invokeLater(r);
        } catch (Exception e) {
            e.printStackTrace();
            http://www.cs.gonzaga.edu/~yourlogin/cpsc324/HelloWorld.html
        }
    }
}
Anything Seem Strange Here?

import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

public class HelloWorld extends JApplet {

    static final long serialVersionUID = 13L;

    public void init() {
        Runnable r = new Runnable() {
            public void run() {
                add(new JLabel("Hello World!", SwingConstants.CENTER));
            }
       );
        try {
            SwingUtilities.invokeLater(r);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

We are creating a new class on the fly!!!

This is called an "anonymous inner class"

Casting objects

• Explicit casting also works for objects

• For example:

    Todo todo = new Task();
    Task task = (Task) todo;
    Deadline d = (Deadline) todo; // Will this compile? Run?

• We are circumventing the type system when we cast

• So you have to be very careful!
Implicit vs. explicit parameters

- Methods define their “explicit” parameters
  
  ```
  public void bark(int numOfBarks)
  ```

  - The explicit parameter is `numOfBarks`

- When a method is called on an object
  
  … it is passed an “implicit” parameter

- As a reference to the object the method is called on
  
  ```
  Dog d = new Dog();
  d.bark(5);  // like calling “bark(d, 5)”
  ```

  - The implicit parameter here is `d`

Accessing the implicit parameters

- Use the `this` keyword to access the implicit parameter …
  
  ```
  public void bark(int numOfBarks) {
      for(int I = 0; i < numOfBarks; i++)
          this.bark();
  }
  ```

  - Sometimes used in constructors
  
  ```
  public Dog(String name) {
      this.name = name; // this distinguishes ‘name’
  }
  ```

  - There are times when it is needed (more later)
Inheritance and Constructors

- Constructors are not inherited
- We must define needed constructors for each class

```java
public class Pet {
    private String name;
    public Pet(String theName) {
        name = theName;
    }
}
```

```java
public class Dog extends Pet {
    public Dog(String theName) {
        name = theName; // Will this work?
    }
}
```

We don’t have access to name in the subclass.
What should we do?

In Java, we call the superclass constructor
- In C++ we would do this in the initialization list
- In Java, we do this using super(…)

```java
public class Dog extends Pet {
    public Dog(String theName) {
        super(theName); // Calls the Pet(String) constructor in Pet
        Must be the first statement in constructor
    }
}
```
The cosmic superclass

All classes (implicitly) extend the Object class

- So, if we have:
  ```java
  public class MyLoneClass {
      ...
  }
  ```
- We really have:
  ```java
  public class MyLoneClass extends Object {
      ...
  }
  ```

Having an Object class is pretty handy

- Always a common superclass
  ```java
  Object obj1 = new Dog();
  Object obj2 = new Cat();
  ```
- For instance, for collections (ADTs)
  ```java
  List aList = new ArrayList();
  aList.add(obj1);
  aList.add(obj2);
  for(Object obj : aList) {
      ...
  }
  ```
  We’ll see later how to check the class of an object
  We already know how to cast
The cosmic superclass

*The Object class defines some useful methods*

- **public String toString()**
  - Returns a string representation of the object
  - Called whenever cast to a String
  - ("" + obj) -- or -- System.out.println(obj)

- **public Object clone()**
  - Creates and returns a copy of the object

- **public boolean equals(Object obj)**
  - Checks equality

- **public Class getClass()**
  - Returns the class of the object

We will look at these methods in more detail later.

Overriding Methods in Java

- Sometimes in a subclass we want to redefine the behavior of superclass methods
  - For example, if we only want Dog objects to eat DogFood

- We can do this by **overriding** methods
  - Implementing the exact same base class method in the derived class
  - The method must have the same parameter and return types
  - How do overriding and overloading differ?
Overriding Methods in Java

public class Dog extends Animal {
    private int barkVolume;
    public void bark() {
        ...
    }
    public void eat(Food theFood) {
        if (theFood is not a dog food)
            return;
        else
            eat(theFood);
    }
}

But is there a problem here?
This calls the same method!
Overriding Methods in Java – super keyword

```java
public class Dog extends Animal {
    private int barkVolume;
    public void bark() {
        ...
    }
    public void eat(Food theFood) {
        if(!theFood instanceof DogFood)
            return;
        else
            super.eat(theFood);
    }
}
```

Calls the parent’s eat(Food) method

Use the "instanceof" keyword here

Checks if an object is an instance of a class (a kind of "reflection")
Dynamic Binding in Java

When we do this:
Animal a = new Dog();
And then this:
a.eat(new DogFood());

• What should happen?
  – The object’s version of eat(Food) is called
  – Not the more generic version in Animal

• This is called “dynamic” binding
  – Methods are bound at runtime
  – As opposed to “static” binding, which is at compile time
  – All methods are “virtual” in Java (woohoo!)

Java abstract classes

• Java supports abstract classes
  – A class where some of its methods are not implemented
  – Often useful when there is no “default” implementation
  – But, you want all subclasses to have the method

• Specify abstract methods & classes with the abstract keyword

  abstract public class Animal {
    public abstract void speak();
    public void move(Location newLoc) {
      ...
    }
  }  

  A class can be declared abstract without any abstract methods
Java abstract classes

• Abstract classes cannot be instantiated

• You can extend an abstract class, but ...
  – You must implement each abstract method
  – Or else declare the class abstract

```java
public class Dog extends Animal {
    public void speak() {
        bark();
    }
    ...
}
```

Java interfaces

• Recall that a class can extend only one class
  ```java
  public ChildClass extends ParentClass
  ```

• Interfaces allow you to conform to many “contracts”
  – An interface is like a “fully” abstract class
  – None of the methods in an interface are implemented
  – Every class that implements an interface must implement all methods declared in the interface (or be declared abstract)

• Interfaces provide a simple version of multiple inheritance
  ```java
  public ChildClass extends ParentClass implements ThisInterface, ThatInterface, TheOtherInterface
  ```
Java interfaces

```java
public interface Animal {
    public static final int MAX_LENGTH = 109; // in feet
    public void speak();
    public void move(Location newLoc);
    ...
}

public class Dog implements Animal {
    public void speak() {
        ...
    }
    public void move(Location newLoc) {
        ...
    }
}
```

Technically, all methods in a public interface are public...

Can include constants

Access as: Animal.MAX_LENGTH

We can use the interface as a type:

```java
Animal a = new Dog();
```

Why would we do this?

- Various uses of interfaces in Java
- As we will see, used a lot in GUI programming
  - Mainly for event handling
- Also in the Collection APIs
  - E.g., List is an interface whereas ArrayList and LinkedList implement the List interface

Can include constants

Access as: Animal.MAX_LENGTH

We can use the interface as a type:

```java
Animal a = new Dog();
```

Why would we do this?

- Various uses of interfaces in Java
- As we will see, used a lot in GUI programming
  - Mainly for event handling
- Also in the Collection APIs
  - E.g., List is an interface whereas ArrayList and LinkedList implement the List interface
• Lab 1