Polymorphism

Overloading and overriding methods
Polymorphism: Greek for “multi-shaped”.

In OO, means the ability to use the same method name twice.

- We’ve already seen this with constructors!
  - public Person()
  - public Person(String name)
  - public Person(String name, int age)

- They all have the same name!
Overloading: How’s it Work?

- **Overloading**: Use the same method name with different parameters.
  - Different parameter types, or
  - Different numbers of parameters.

- **Example**: The Random class has methods
  - `public int nextInt()`  //any int value
  - `public int nextInt(int n)`  //between 0 and n

- No confusion! Different parameters.
Examples

- **String has methods**
  - public String substring(int beginIndex)  //from there to end
  - public String substring(int beginIndex, int endIndex)

- **Math has methods**
  - public static double abs(double a)
  - public static float abs(float a)
  - public static int abs(int a)
  - public static long abs(long a)

- **Zillions of other examples.**
public class RudeString {
    private String originalString;
    private String rudeString = “, you jerk”;

    public RudeString(String s) {
        originalString = s;
    }

    public String getString() {
        return originalString + rudeString;
    }

    public String getString(int rudeLevel) {
        String response = originalString;
        for(int i = 0; i<rudeLevel; i++) {
            response = response + rudeString;
        }
        return response;
    }

    public void setRudeString(String rude) {
        rudeString = rude;
    }
}

Note: String class is “final”. Can’t extend it. So used composition – see Software Engineering class!
Overloading “equals”

- Recall all classes ultimately extend Object.
  - Object contains “equals” method
    ```java
    public boolean equals(Object obj)
    ```

- So all of your classes automatically inherit an “equals” method.
  - But you can overload.
    - Make your own equals with different parameter.
      ```java
      public boolean equals(RudeString r)
      {
        boolean same = false;
        if(r.getString().equals(getString()))
        {
          same = true;
        }
        return same;
      }
      ```
      overloaded equals!
Your Turn

Create class that calculates and prints $x^2+1$.

- No constructor or instance variables required.
  - SquarePlusOne class
  - Just create method that prints $x^2+1$.

- Create method for int value of x.
- Overload for double value of x.
- Overload for float value of x.
Overriding: How’s It Work?

**Overriding:** In child class, use the same method name as in parent class. Also use same parameters.

The child’s method gets used instead of the one in the parent.
Example

Suppose have Simpson class with weight instance variable. Homer class wants to lie about the weight. So overrides the original method.

```java
public class Simpson {
    private int weight;

    public Simpson(int weight) {
        this.weight = weight;
    }

    public int getWeight() {
        return weight;
    }

    ...
}

public class Homer extends Simpson {
    public Homer(int weight) {
        super(weight);
        // weight is likely to be embarrassing
        // (like 300), so this overrides (and lies).
        public int getWeight() {
            return 150;
        }
    }
}
```
public class Bart extends Simpson
{
    public Bart(int weight)
    {
        super(weight);
    }

    public void test()
    {
        Homer h = new Homer(300);
        Bart b = new Bart(60);
        Simpson s = new Simpson(100);
        System.out.println( h.getWeight() );
        System.out.println( b.getWeight() );
        System.out.println( s.getWeight() );
    }
}
Accessing Overridden Methods

- Super class still has original methods.

- Can access them by using `super.originalMethod();`
  - Can only do this within a method or constructor of the derived class.
Example

```java
public class Homer extends Simpson {
    private String quote;
    
    public Homer(int weight, String quote) {
        super(weight);
        this.quote = quote;
    }
    
    public int getRealWeight() {
        return super.getWeight();
    }
}
```

// weight is likely to be embarrassing (like 300),
// so this overrides (and lies).
public int getWeight() {
    return 150;
}
```
Instead of creating new “safeDelete” method name, override old “delete”!

```java
import java.io.File;
import javax.swing.JOptionPane;
public class MySafeFile extends File {
    /**
     * Prevents accidental deletion of a file.
     * Original delete method does not ask for confirmation.
     */
    public void delete() {
        String answer = JOptionPane.showInputDialog("This file " +
                "is about to be deleted. Is that ok? (y/n)");
        if (answer.charAt(0) == 'y') {
            super.delete();
        }
    }
}
```

Danger! Need a constructor that calls super(filepath). Why? Hint: look up the constructors for File. Do you see a default?