Inheritance (pt 3)

Based on slides by Prof. Burton Ma

Static Methods and Inheritance

- There is a big difference between calling a static method and calling a non-static method when dealing with inheritance

- *There is no dynamic dispatch on static methods*

```
public abstract class Dog {
  private static int numCreated = 0;
  public static int getNumCreated() {
    return Dog.numCreated;
  }
}

public class Mix {
  private static int numMixCreated = 0;
  public static int getNumCreated() {
    return Mix.numMixCreated;
  }
}

public class Komondor {
  private static int numKomondorCreated = 0;
  public static int getNumCreated() {
    return Komondor.numKomondorCreated;
  }
}
```

notice no @Override

notice no @Override

```
public class WrongCount {
  public static void main(String[] args) {
    Dog mutt = new Mix();
    Dog shaggy = new Komondor();
    System.out.println( mutt.getNumCreated() );
    System.out.println( shaggy.getNumCreated() );
    System.out.println( Mix.getNumCreated() );
    System.out.println( Komondor.getNumCreated() );
  }
}
```

```
prints 2
       2
       1
       1
```

What's Going On?

- *There is no dynamic dispatch on static methods*

- Because the declared type of **`mutt`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

- Because the declared type of **`shaggy`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

Hiding Methods

- Notice that **`Mix.getNumCreated`** and **`Komondor.getNumCreated`** work as expected
- If a subclass declares a static method with the same name as a superclass static method, we say that the subclass static method hides the superclass static method
  - *You cannot override a static method, you can only hide it*
  - Hiding static methods is considered bad form because it makes code hard to read and understand

- The client code in **`WrongCount`** illustrates two cases of bad style, one by the client and one by the implementer of the **`Dog`** hierarchy
  1. The client should not have used an instance to call a static method
  2. The implementer should not have hidden the static method in **`Dog`**

Interfaces

- Recall that you typically use an abstract class when you have a superclass that has attributes and methods that are common to all subclasses
  - The abstract class provides a partial implementation that the subclasses must complete
  - Subclasses can only inherit from a single superclass
- If you want classes to support a common API then you probably want to define an interface

Interfaces

- In Java an *interface* is a reference type (similar to a class)
- An interface says what methods an object must have and what the methods are supposed to do
  - I.e., an interface is an API

Interfaces

- An interface can contain *only*
  - Constants
  - Method signatures
  - Nested types (ignore for now)

- There are no method bodies
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces

Interfaces Already Seen

access—either public or package-private (blank)

interface name

```
public interface Comparable<T>
{
  int compareTo(T o);
}
```

Interfaces Already Seen

```
public interface Iterable<T>
{
  Iterator<T> iterator();
}
```

access—either public or package-private (blank)

interface name

parent interfaces

```
public interface Collection<E> extends Iterable<E>
{
  boolean add(E e);
  void    clear();
  boolean contains(Object o);
  // many more method signatures...
}
```

Interfaces Already Seen

```
public interface List<E> extends Collection<E>
{
  boolean add(E e);
  void    add(int index, E element);
  boolean addAll(Collection<? extends E> c);
  // many more method signatures...
}
```

Creating an Interface

- Decide on a name
- Decide what methods you need in the interface

- This is harder than it sounds because...
  - Once an interface is released and widely implemented, it is almost impossible to change
    - If you change the interface, all classes implementing the interface must also change

Function Interface

- In mathematics, a real-valued scalar function of one real scalar variable maps a real value to another real value

$$y = f(x)$$

Creating an Interface

- Decide on a name
  - **`DoubleToDoubleFunction`**

- Decide what methods you need in the interface
  - **`double   evaluate(double x)`**
  - **`double[] evaluate(double[] x)`**

Creating an Interface

```
public interface DoubleToDoubleFunction {
  double   at(double x);
  double[] at(double[] x);
}
```

Classes that Implement an Interface

- A class that implements an interface says so by using the **`implements`** keyword
  - Consider the function $f(x) = x^2$

```
public Square implements DoubleToDoubleFunction {
  public double at(double x) {
    return x * x;
  }

  public double[] at(double[] x) {
    double[] result = new double[x.length];
    for (int i = 0; i < x.length; i++) {
      result[i] = x[i] * x[i];
    }
    return result;
  }
}
```

Implementing Multiple Interfaces

- Unlike inheritance where a subclass can extend only one superclass, a class can implement as many interfaces as it needs to

superclass

```
public class ArrayList<E>
   extends AbstractList<E>
   implements List<E>,
              RandomAccess,
              Cloneable,
              Serializable
```

interfaces