Inheritance (Part 5)

Odds and ends

Static Methods and Inheritance

- there is a significant difference between calling a static method and calling a non-static method when dealing with inheritance

- *there is no dynamic dispatch on static methods*
  - therefore, you cannot override a static method
  - if you use a variable name instead of the class name to invoke the static method, you get the method that belongs to the declared type of the variable

```
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() );
  }
}
```

Dog version

Dog version

Mix version

Komondor version

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`**

Using superclass methods

Other Methods

- methods in a subclass will often need or want to call methods in the immediate superclass
  - a new method in the subclass can call any **`public`** or **`protected`** method in the superclass without using any special syntax
- a subclass can override a **`public`** or **`protected`** method in the superclass by declaring a method that has the same signature as the one in the superclass
  - a subclass method that overrides a superclass method can call the overridden superclass method using the **`super`** keyword

Dog equals

- we will assume that two **Dog**s are equal if their size and energy are the same

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if(obj != null && this.getClass() == obj.getClass())
  {
    Dog other = (Dog) obj;
    eq = this.getSize() == other.getSize() &&
         this.getEnergy() == other.getEnergy();
  }
  return eq;
}
```

Mix equals (version 1)

- two Mix instances are equal if their Dog subobjects are equal and they have the same breeds

```
@Override public boolean equals(Object obj)
{ // the hard way
  boolean eq = false;
  if(obj != null && this.getClass() == obj.getClass()) {
    Mix other = (Mix) obj;
    eq = this.getSize() == other.getSize() &&
         this.getEnergy() == other.getEnergy() &&
         this.breeds.size() == other.breeds.size() &&
         this.breeds.containsAll(other.breeds);
  }
  return eq;
}
```

subclass can call public method of the superclass

Mix equals (version 2)

- two Mix instances are equal if their Dog subobjects are equal and they have the same breeds
  - Dog equals already tests if two Dog instances are equal
  - Mix equals can call Dog equals to test if the Dog subobjects are equal, and then test if the breeds are equal
- also notice that Dog equals already checks that the Object argument is not null and that the classes are the same
  - Mix equals does not have to do these checks again

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if(super.equals(obj))
  { // the Dog subobjects are equal
    Mix other = (Mix) obj;
    eq = this.breeds.size() == other.breeds.size() &&
         this.breeds.containsAll(other.breeds);
  }
  return eq;
}
```

subclass method that overrides a superclass method can call the original superclass method

Dog toString

```
@Override public String toString()
{
  String s = "size " + this.getSize() +
             "energy " + this.getEnergy();
  return s;
}
```

Mix toString

```
@Override public String toString()
{
  StringBuffer b = new StringBuffer();
  b.append(super.toString());
  for(String s : this.breeds)
    b.append(" " + s);
  b.append(" mix");
  return b.toString();
}
```

size and energy of the dog

breeds of the mix

Dog hashCode

```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = 1;
  result = prime * result + this.getEnergy();
  result = prime * result + this.getSize();
  return result;
}
```

use this.energy and this.size to compute the hash code

Mix hashCode

```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + this.breeds.hashCode();
  return result;
}
```

use this.energy, this.size, and this.breeds to compute the hash code

Review

Review

1. Inheritance models the _____ relationship between classes.
2. Dog is a _____ of Object.
3. Dog is a _____ of Mix.
4. Can a Dog instance do everything a Mix instance can?
5. Can a Mix instance do everything a Dog instance can?
6. Is a Dog instance substitutable for a Mix instance?
7. Is a Mix instance substitutable for a Dog instance?

8. Can a subclass use the private fields of its superclass?
9. Can a subclass use the private methods of its superclass?
10. Suppose you have a class X that you do not want anyone to extend. How do you enforce this?
11. Suppose you have an immutable class X. Someone extends X to make it mutable. Is this legal?
12. What do you need to do to enforce immutability?

13. Suppose you have a class Y that extends X.
    a. Does each Y instance have a X instance inside of it?
    b. How do you construct the X subobject inside of the Y instance?
    c. What syntax is used to call the superclass constructor?
    d. What is constructed first–the X subobject or the Y object?
    e. Suppose Y introduces a brand new method that needs to call a public method in X named xMethod. How does the new Y method call xMethod?
    f. Suppose Y overrides a public method in X named xMethod. How does the overriding Y method call xMethod?

14. Suppose you have a class Y that extends X. X has a method with the following precondition:

```
@pre. value must be a multiple of 2
```

If Y overrides the method which of the following are acceptable preconditions for the overriding method:

a. `@pre. value must be a multiple of 2`
b. `@pre. value must be odd`
c. `@pre. value must be a multiple of 2 and must be less than 100`
d. `@pre. value must be a multiple of 10`
e. `@pre. none`

14. Suppose you have a class Y that extends X. X has a method with the following postcondition:

```
@return – A String of length 10
```

If Y overrides the method which of the following are acceptable postconditions for the overriding method:

a. `@return – A String of length 9 or 10`
b. `@return – The String "weimaraner"`
c. `@return – An int`
d. `@return – The same String returned by toString`
e. `@return – A random String of length 10`

15. Suppose Dog toString has the following Javadoc:

```
/*
 * Returns a string representation of a dog.
 * The string is the size of the dog followed by a
 * a space followed by the energy.
 * @return The string representation of the dog.
 */
```

Does this affect subclasses of Dog?

Inheritance Recap

- inheritance allows you to create subclasses that are substitutable for their ancestors
  - inheritance interacts with preconditions, postconditions, and exception throwing
- subclasses
  - inherit all non-private features
  - can add new features
  - can change the behaviour of non-final methods by *overriding* the parent method
  - contain an instance of the superclass
    - subclasses must construct the instance via a superclass constructor