Inheritance pt. 2

Ch. 10

Review

- Classes inheriting from other classes do so by using `extends`
  - An object of subclass has access to all public members of superclass. The inverse is not true.
  - A subclass object treated as a superclass loses capabilities.
  - Using `super` references superclass.

```
class Instrument{
   int lowestNote;
   int highestNote;
   Instrument(int lowestNote, int highestNote){
      this.lowestNote = lowestNote;
      this.highestNote = highestNote;
   }
   void transpose(int x){
      lowestNote += x;
      highestNote += x;
   }
}
class Brass extends Instrument{
   int numKeys;
   Brass(int lowestNote, int highestNote, int numKeys){
      super(lowestNote, highestNote);
      this.numKeys = numKeys;
   }
}
```

When we use a superclass reference to refer to a subclass, then we only have access to the superclass's fields and methods, UNLESS there are overridden methods in the subclass.

```
class Super{
   int x = 50;
   void f(){
      System.out.println("Super");
   }
}
class Sub extends Super{
   int x = 100;
   void f(){
      System.out.println("Sub");
   }
}
class InherTest{
   public static void main(String[] args){
      Super a = new Super();
      Sub b = new Sub();
      Super c = b;
      System.out.println(a.x + "" + b.x + "" + c.x);
   }
}
```

What does the main method print?

```
class Super{
   void f(){
      System.out.println("Super");
   }
}
class Sub extends Super{
   void f(){
      System.out.println("Sub");
   }
}
class InherTest{
   public static void main(String[] args){
      Super a = new Super();
      Sub b = new Sub();
      Super c = b;
      a.f();
      c.f();
   }
}
```

What does the main method print?

```
public class InherTest{
   public static void main(String[] args){
      Super a = new Super();
      Sub b = new Sub();
      foo(a);
      foo(b);
      Super c = b;
      foo(c);
   }
   public static void foo(Super s){
      System.out.println("Super main");
      s.f();
   }
   public static void foo(Sub s){
      System.out.println("Sub main");
      s.f();
   }
}
```

What does the main method print?

The Object Class

- All classes in Java are implicitly inherited from Object
- Several methods defined in documentation
- We're concerned with:
  - toString
  - equals

toString

`public String toString()`

Returns a string representation of the object.

It is recommended that all subclasses override this method.

Object's `toString()` method returns:

```
getClass().getName() + '@' +
Integer.toHexString(hashCode())
```

```
public class Rational{
   int numerator;
   int denominator;
   Rational(int numerator, int denominator){
      this.numerator = numerator;
      this.denominator = denominator;
   }
   // Overrides Object's toString() method
   public String toString(){
      return numerator + "/" + denominator;
   }
}
```

```
// numbers in the form of a + bi, s.t. i² = -1
public class Complex{
   int imag;
   int real;
   Complex(int imag, int real){
      this.imag = imag;
      this.real = real;
   }
   // Overrides Object's toString() method
   public String toString(){
      return (imag != 0 && real != 0) ?
                (real + " " + imag + "i") : ((real == 0) ?
                   (imag + "i") : Integer.toString(real));
   }
}
```

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

For objects x and y, `equals` returns true if and only if x and y refer to the same object (x == y has the value true)

equals

We will need to use `instanceof`

You can use `instanceof` to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

Ex:

`"Hello World" instanceof String` evaluates to true

equals

Once we determine if object is `instanceof` class, then we can cast that object to that class.

```
class Rational{
   int num;
   int denom;
   Rational(int num, int denom){
      this.num = num;
      this.denom = denom;
   }
   // overrides Object's equals method
   @Override
   public boolean equals(Object obj){
      if(obj instanceof Rational){
         Rational p = (Rational)obj;
         return (p.num == num && p.denom == denom);
      }
      else
         return false;
   }
}
```

Why can't our equals method signature be defined: `public boolean equals(Rational obj) ...?`

```
class Rational{
   int num;
   int denom;
   Rational(int num, int denom){
      this.num = num;
      this.denom = denom;
   }
   // overrides Object's equals method
   @Override
   public boolean equals(Object obj){
      if(obj instanceof Rational){
         Rational p = (Rational)obj;
         return (p.num == num && p.denom == denom);
      }
      else
         return false;
   }
}
```

```
class Rational{
    int num;
    int denom;
    Rational(int num, int denom){
        this.num = num;
        this.denom = denom;
    }
    // overrides Object's equals method
    @Override
    public boolean equals(Object obj){
        if(obj instanceof Rational){
            Rational p = (Rational)obj;
            return (p.num == num && p.denom == denom);
        }
        else
            return false;
    }
}
```

Complete the static method `find` such that the index is returned of the first object in the array that is equal to `el`. If no equal object is found, `-1` is returned.

```
public static int find(Rational[] ar, Rational el)
```

```
public class RationalTest{
   public static void main(String[] args){
      Rational[] r = {new Rational(1,2),
                      new Rational(2,3), new Rational(3,4)};
      System.out.println(find(r, new Rational(2,3)));
   }
   public static int find(Object[] ar, Rational el){
      // Assume the same implementation of find that was
      // just discussed
   }
}
```

Does the equals method of Rational get called or the equals method inherited from Object?