In this lecture, we'll make sure you're familiar with some of the more important Java concepts.

**Keywords**

Let's look for compiler errors in the code below.

```java
package edu.upenn.cis573.sytykj;

public class ClassOne {

    public static int intOne = 1;
    private final int intTwo = 2;
    protected int intThree = 3;
    volatile int intFour = 4;
    final static int intFive;
    final int intSix;

    public ClassOne() {
        intTwo = 22;
        intFive = 5;
        intSix = 6;
    }

    protected final static int methodOne() {
        return intFour;
    }

    int methodTwo(int intSix) {
        intSix = 14;
        return intFive;
    }
}
```

There are three compiler errors:

- Line 13: can't assign value to intTwo since it's final and has been initialized
- Line 14: can't assign value to intFive since it's final and static (even though it hasn't been initialized)
- Line 19: can't access intFour in a static method since it's not static

There are some “red herrings” here, too: things that look like problems, but actually aren't.

- Line 8: volatile is a legal keyword that tells the compiler that this field may be accessed simultaneously by two or more threads
- Line 15: it's okay to assign a value to intSix even though it's final, because it hasn't been initialized; however, if you didn't initialize it, the compiler would complain
- Line 23: it's okay to assign a value to intSix because this is the parameter and not the field
Assuming that we got ClassOne to compile, let's look for compiler problems in this class:

```java
package edu.upenn.cis573.sytykj;

public final class ClassTwo extends ClassOne {
    static {
        this.intOne = 11;
    }

    protected int intSeven;

    public ClassTwo() {
        super();
        intSeven = intTwo + 5;
    }

    protected final static int methodOne() {
        return 1;
    }

    public float methodTwo(int intSix) {
        if (intSix > 0) continue;
        return intFive * 7;
    }
}
```

There are five errors:
- Line 6: can't access “this” in a static context because “this” refers to a particular object, whereas the static block is used when the class is loaded
- Line 13: can't access intTwo because it's declared as private in ClassOne
- Line 16: can't override methodOne because it's declared as final in ClassOne
- Line 20: can't override methodTwo and have it return float, since in ClassOne it returns int
- Line 21: can't use continue outside of a loop (same with break)

There are two red herrings:
- Line 3: final class is okay; it just means it can't be extended
- Line 5: the static initialization block is fine; it is invoked when the class is loaded, i.e. the first time it is referred to in the program

Now let's try this last class:

```java
package edu.upenn.cis573;

import edu.upenn.cis573.sytykj.ClassOne;
```
public class ClassThree {
    public Void fun(ClassOne one) {
        System.out.println(one.intOne);
        System.out.println(one.intThree);
        System.out.println(one.intFive);
        System.out.println(ClassTwo.intFive);
        System.out.println(one.methodTwo(6));
        return null;
    }
}

There are four errors:
• Line 9: intThree is not public in ClassOne
• Line 10: intFive is not public in ClassOne (note that this way of accessing it is fine)
• Line 11: intFive is not public in ClassOne (this is the preferred way of accessing a static
  variable)
• Line 12: methodTwo is not public in ClassOne

And one red herring:
• Line 7: there is a Void type in Java. It's just a class that represents “nothing”

Inheritance

When it comes to inheritance (i.e., one class extending another), the thing to remember is that when
you a call a method on an object, you use the one that's defined in the type of that object itself, i.e. what
you used when you said “new”, and not based on the type of the variable.

Assume we have the following classes:

public class Parent {
    public String name() {
        return "Parent";
    }
    public String toString() {
        return "I am the " + name();
    }
}

public class Child extends Parent {

public String name() {
    return "Child";
}

public class Printer {
    public static String print(Parent p) {
        return p.toString();
    }
}

What gets printed on each of the following System.out.println calls?

Child c = new Child();
System.out.println(c);
Parent p = new Parent();
System.out.println(p);
Parent k = new Child();
System.out.println(k);
System.out.println(Printer.print(k));
Child z = new Child();
System.out.println(Printer.print((Parent)z));
Parent r = (Parent)c;
System.out.println(r);
Child s = (Child)p;
System.out.println(s);

Line 2: Prints “I am the Child”. Since c is of type Child, Child.name is called from Parent.toString
Line 5: Prints “I am the Parent”. Since p is of type Parent, Parent.name is called from Parent.toString
Line 8: Prints “I am the Child”. Even though k is declared to be a Parent, it is actually a Child, so Child.name is called from Parent.toString. It is the type of the object, not the variable, that matters.
Line 10: Prints “I am the Child”. Even though Printer.print takes a Parent, when it calls Parent.toString, that still uses the object that was passed, which is k, which is a Child.
Line 13: Prints “I am the Child”. Casting z to a Parent doesn't change its type.
Line 16: Prints “I am the Child”. Although we cast c to a variable of type Parent, the object is still a
Child.

Line 18: This is a trick! This line results in a ClassCastException. The variable p is a Parent and we cannot simply cast it to a subclass.

**Polymorphism**

The rules for interfaces are pretty much the same as those for inheritance.

Consider the following interface and classes:

```java
public interface MyInterface {
    public int methodOne();
    public int methodTwo();
}

public class MyClassOne implements MyInterface {
    public int methodOne() { return 1; }
    public int methodTwo() { return 2; }
}

public class MyClassTwo extends MyClassOne {
    public int methodOne() { return 2; }
    public int methodTwo() { return 3 * methodOne(); }
}

public class Printer {
    public static int printOne(MyInterface m) {
        return m.methodOne();
    }
    public static int printTwo(MyInterface m) {
        return m.methodTwo();
    }
}
```
What gets printed on each of the following System.out.println calls?

```
1 MyClassOne m1 = new MyClassOne();
2 System.out.println(Printer.printOne(m1));
3 System.out.println(Printer.printTwo(m1));
4
5 MyClassTwo m2 = new MyClassTwo();
6 System.out.println(Printer.printOne(m2));
7 System.out.println(Printer.printTwo(m2));
8
9 MyClassOne r1 = Factory.create(1);
10 System.out.println(Printer.printTwo(r1));
11
12 MyClassOne r2 = Factory.create(2);
13 System.out.println(Printer.printTwo(r2));
14
15 MyInterface i1 = new MyClassOne();
16 System.out.println(Printer.printOne(i1));
17
18 MyClassOne k1 = new MyClassTwo();
19 System.out.println(Printer.printOne(k1));
20 System.out.println(Printer.printTwo(k1));
21
22 MyClassTwo k2 = new MyClassOne();
23 System.out.println(Printer.printOne(k2));
24 System.out.println(Printer.printTwo(k2));
```

Line 2: prints 1, because m1 is of type MyClassOne, so Printer.printOne calls MyClassOne.methodOne
Line 3: prints 2
Line 6: prints 2, because m2 is of type MyClassTwo, so Printer.printOne calls MyClassTwo.methodOne
Line 7: prints 6 because Printer.printTwo calls MyClassTwo.methodTwo, which calls MyClassTwo.methodOne
Line 10: prints 2, because r1 is of type MyClassOne
Line 13: prints 6, because even though r2 is declared to be of type MyClassOne and even though Factory.create returns a MyClassOne, r2 is actually a MyClassTwo
Line 16: prints 1, because i1 is of type MyClassOne

Line 19: prints 2, because even though k1 is declared as a MyClassOne, it is actually a variable of type MyClassTwo
Line 20: prints 6

Line 22: this is a trick! You cannot create an instance of MyClassOne and then refer to it with a variable of type MyClassTwo