Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-five)

5.6: Redefining Nonabstract Methods and Extending Concrete Classes

```
class Foo
{
   int x() { return 3; }
}

class Bar extends Foo
{
   int x() { return 4; }
}

...

// Object is really a Bar
Foo f = new Bar();
// Prints '4'
System.out.println(f.x());
```

The only methods you can't override are those marked with the *final* qualifier:

```
class Foo
{
  final int x() { return 3; }
}

class Bar extends Foo
{
  int x() { return 4; }
}
```

Compiler Says:

```
Foo.java:8: Final methods can't be overridden. Method int x() is final in class Foo.
    int x() { return 4; }
        ^
```

The Java core APIs use the final keyword in this way to prevent programmers from breaking system security or integrity. For example, look at the fairly important class java.lang.ClassLoader. All of its methods are marked final, except for one abstract method that subclasses are required to override.

In many circles, it is considered bad form to design a system in which *concrete* classes serve as superclasses. (A concrete class is one that you can create an instance of. Any class with no abstract methods is concrete.) I've deliberately made our Shape class abstract, and most likely you understand the wisdom behind this: whereas the idea of a Shape is of value, an actual undifferentiated Shape object can only cause trouble if actually created at runtime. Nevertheless, inheritance from concrete classes is

an option, and is frequently used to add to or modify the functionality of code you cannot change (such as the Java API itself).