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

5.3: The Protected Access Level

If you think about it, it makes very little sense to ever create a generic Shape object. A Shape couldn't draw itself, for example, since it would have no idea what it looked like; only specific kinds of shapes (in OOP terminology, Shape *subclasses*; the class being extended, like Shape, is called the *superclass*) could do that. It probably makes sense to prevent clients from ever creating generic Shape objects. (Shape *variables* are quite useful, however, because they can refer to any Shape subclass.)

A bit more introduction to the terminology is in order here. Superclasses are often also called "base" or "parent" classes, while subclasses are often called "derived" or "child" classes. I tend to use all three pairs of terms interchangeably. Their meanings are all equivalent.

One way to prevent code outside your drawing package from creating Shape objects, while still allowing the use of Shape variables, is to make all the constructors of the Shape class *protected*. Protected members have a level of access control that is in-between those of public members and default (package-protected) members: they are accessible from all code in the current package, plus all code in subclasses. By making the constructors of Shape protected, you ensure that only subclasses of Shape can call them (See Footnote). This makes it harder, but far from impossible, for bare Shapes to be created and (mis)used.

Footnote

Actually, other code in your drawing package could call them too, but you should know better! I've always considered this a wart in Java: it should be possible, as it is in C++, to restrict access to *only* subclasses, but it is not. In the earliest released versions of Java, you could mark members *private protected* to achieve exactly this, but that option was removed. The motive for this change is reasonable, if not necessarily convincing. Note that a subclass is allowed to redefine a method to be "more public" than it is in a class being extended:

```
(in file foo.java) package foobar; public class foo { void doSomething( ) {}}
```

```
(in file bar.java)
package foobar;
public class bar extends foo
{
    public void doSomething()
    {
      super.doSomething();
    }
}
```

```
Note that code outside the foobar package could not call foo.doSomething(), but it could call bar.do
Something(). The reverse is not allowed:
```

```
(in file foo.java)
package foobar;
public class foo { public void doSomething() {}}
```

```
(in file bar.java)
package foobar;
public class bar extends foo
{
    void doSomething()
    {
       super.doSomething();
    }
}
```

Compiler Says:

```
Foo.java:5: Methods can't be overridden to be more private.
Method void doSomething() is public in class foo.
   void doSomething()
        ^
```

Here bar redeclares the doSomething method to be invisible outside package foobar, while foo had made it public. This is illegal. If it were not, then the following code could call bar.doSomething() from outside foobar, even though bar wants that to be impossible:

```
bar b = new bar();

((foo) b).doSomething();
// we called doSomething() on the bar object!
```

This is a simple rule. The four levels of protection (private, package (default), protected, and public) can be placed in a strict ordering, so that "more public" has a clear definition. Private members are accessible only within a class; package members are accessible to the class and the whole package; protected members are accessible to the class, package, and subclasses; and public members are accessible everywhere. If we added "private protected" (one person has proposed the term "progeny," which is nice in that it starts with *p* and clearly suggests subclass access), the neat hierarchy of levels would be destroyed, and the rule about redeclaring at different protection levels would become complex. One of Java's major design goals is simplicity. Hence, the "progeny" level had to go.