Okay, it's been three chapters and you still haven't answered my question about **new**. We aren't supposed to program to an implementation, but every time I use **new**, that's exactly what I'm doing, right?

When you see "new," think "concrete."

Yes, when you use the **new** operator you are certainly instantiating a concrete class, so that's definitely an implementation and not an interface. And you make a good observation: that tying your code to a concrete class can make it more fragile and less flexible.

```
Duck duck = new MallardDuck();
```

We want to use abstact types to keep code flexible.

But we have to create an instance of a concrete class!

When we have a whole set of related concrete classes, often we end up writing code like this:

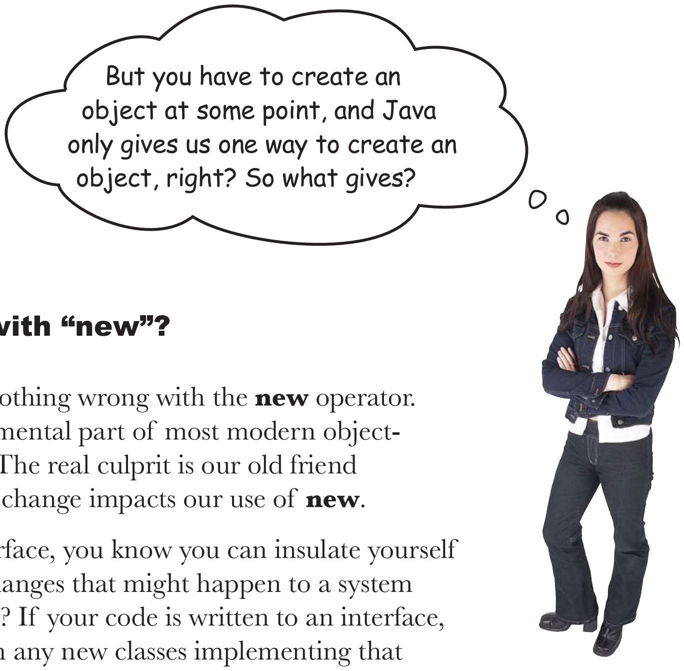
```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

Here we've got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application, making maintenance and updates more difficult and error-prone.


But you have to create an object at some point, and Java only gives us one way to create an object, right? So what gives?

What's wrong with "new"?

Technically there's nothing wrong with the **new** operator. After all, it's a fundamental part of most modern object-oriented languages. The real culprit is our old friend CHANGE and how change impacts our use of **new**.

By coding to an interface, you know you can insulate yourself from many of the changes that might happen to a system down the road. Why? If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism. However, when you have code that makes use of lots of concrete classes, you're looking for trouble because that code may have to be changed as new concrete classes are added. So, in other words, your code will not be "closed for modification." To extend your code with new concrete types, you'll have to reopen it.

Remember that designs should be "open for extension but closed for modification." See Chapter 3 for a review.

So what can you do? It's times like these that you can fall back on OO design principles to look for clues. Remember, our first principle deals with change and guides us to *identify the aspects that vary and separate them from what stays the same.*

BRAIN POWER

How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application?