Anonymous class

The anonymous class is supposed to provide syntactic sugar. It is more like syntactic salt poured onto a wound, because it is difficult to read and even called ugly by some. This is caused partially by the fact that Java does not allow methods as parameters, partially just because how OO is implemented in Java. But don't take this too seriously. No language is good in all respects, and Java *is* in many respects a great language.

You may never write an anonymous class of your own, but you should be able to recognize one when you see it.

What if Java allowed functions as parameters?

Consider a class C, only part of which is shown to the right. It has fields y and b. C's constructor initializes field y. HAP b is associated with some event, or happening, on a GUI, perhaps a button or a keystroke, and that happening should cause field y to be changed.

```
public class C {
   int y;
   HAP b= new HAP(…);

   public C(int y) {
      this.y= y;

      // Tell b to call m to change y
      b.use(m);
   }

   public void m(int x) {y= x;}
}
```

This is implemented using method m, which assigns its parameter to y. In C's constructor, we set it all up by calling b.use(m), which tells b to call method m to change field y whenever needed.

But this doesn't work, because Java does not allow methods as parameters. Method b.use cannot be written this way. So, we have to do something different: the parameter has to be an object that contains the method.

Use an inner class

Here is how we make the change. First, write an interface In, which contains abstract method m:

```
public interface In {
   void m(int c);
}
```

Second, in class C, write an inner class that implements interface In and overrides method m. You see this inner class to the right (arrow (1)). It is an inner class because it needs access to field y.

Third, change method b.use so that its argument is of type In:

```
public void use(In p) { … }
```

Paameter p is not a function but an object that contains the function, and method b.use can save the object and call the function when needed.

Fourth, as the argument of a call to b.use (arrow (2)), use a new-expression that creates an Inner object.

```
public class C {
   int y;
   HAP b= new HAP (…);

   public C(int y) {
      this.y= y;

      // Tell b to call m to change y
      b.use(new Inner());
   }

   public class Inner implements In {
      public void m(int x) {y= x;}

   }
}
```

(1) (2)

This, then, is legal Java code.

Comments

(1) Java version 8 does introduce a form of function as parameter, making the language more "functional". This is explained partially under "lambdas".

(2) Interface In could be an abstract class instead of an interface.

(3) Class Inner can have many methods and fields —not just one method.

(4) The scheme just explained is the conventional way of associating events in a GUI with methods to be called when the events happen. The interface (e.g. In) and its method(s) (e.g. m) are different for each kind of event on a GUI. A call like b.use(new Inner) is said to *register* the object as a *listener* to the event.

Anonymous class

Creating an anonymous class

In the code on the previous page, which is shown to the left below, only *one* object of class Inner is created, and it seems senseless to have to declare the class with a name just to be able to create one instance of it. So, syntactic sugar is introduced.

To the right below, we have equivalent code using the syntactic sugar: in blue, a new-expression that creates an object of an anonymous, unnamed class.

How was that blue code in the right class constructed? We show the steps taken to create the blue code below the two classes.

```
public class C {
   int y;
   HAP b= new HAP (…);

   public C(int y) {
      this.y= y;

      // Tell b to call m to change y
      b.use(new Inner());
   }

   public class Inner implements In {
      public void m(int x) {y= x;}

   }
}
```

```
public class C {
   int y;
   HAP b= new HAP (…);

   public C(int y) {
      this.y= y;

      // Tell b to call m to change y
      b.use(new In() {
            public void m(int x) {y= x;}
           });
   }
}
```

We start with class C in the left box and show how to create the blue new-expression in class C on the right.

1. Copy new: `new`
2. Append the interface that Inner implements: `new In`
3. Append the argument list of the constructor call `new In()`
4. Append the body of class Inner (shown in red):

```
new In() {
    public void m(int x) {y= x;}
}
```

5. Use the result as the argument of the call to b.use, and delete the inner class.

The result, shown to the right is how one writes a new-expression that creates an object of an anonymous, unnamed class. It is syntactic sugar for, and is translated back into, the code in the box on the left, when the program is compiled.