Refactoring methods

Eclipse has refactoring tools to:

(1) *Extract* a method, i.e. make a method out of a selected sequence of statements or an expression and replace the selection by a call on the method. This is useful when a method body is complicated or too long or when the statements are duplicated. Of course, a good specification has to be placed on the new method.

(2) *Inline* a method call, i.e. replace a call by the body of the method. This can be used to make a program run faster, since a frame for the call does not have to be pushed onto the call stack and later popped off.

We give an example of each of these.

Extracting a method

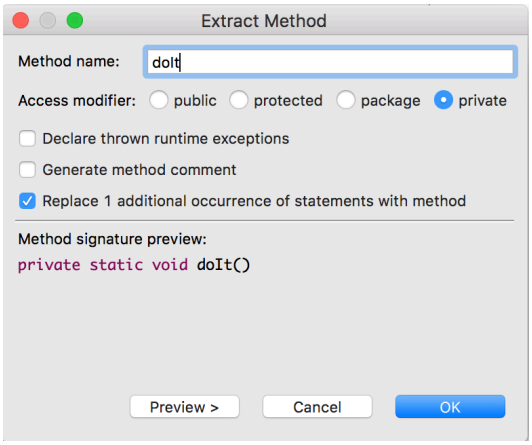
Consider nonsense method *m* to the right. All variables are static fields.

```
public static void m() {
    if (x+y <= y+z) {
        ans= x+y;
    }
    while (x < y) {
        if (x+y <= y+z) {
            ans= x+y;
        }
        x= x+1;
    }
}
```

The if-statement appears twice, and we decide to make a method of it. To do this using the Eclipse refactoring tool, do the following:

1. Select all the text that is to be extracted into a method body.

2. Choose menu item *Refactor -> Extract method*. This causes this window to pop up:



We already typed in method name *doIt*. After we typed *doIt*, the Method signature preview appeared.

The method will be private. Eclipse has decided that there will be no parameters.

Eclipse recognized that the statement to be replaced appears in two places and has checked the box to have both replaced.

When we hit button *OK*, the new method shown to the left was created and method *m* was changed to what you see on the right.

There is no specification on method *doIt*, and we should put one there.

```
private static void doIt() {
        if (x+y <= y+z) {
            ans= x+y;
        }
    }
```

```
public static void m() {
    doIt();
    while (x < y) {
        doIt();
        x= x+1;
    }
}
```

Be careful!

```
public static void m() {
    int ans= 0;
    if (x+y <= y+z) {
        ans= x+y;
    }
    while (x < y) {
        if (x+y <= y+z) {
            ans= x+y;
        }
        x= x+1;
    }
}
```

Suppose *ans* is a local variable, as shown to the left. Performing the same extraction procedure produces the method shown to the right, which has *ans* as a local variable. Thus, the effect we wanted —a call of doIt assigns to local variable *ans* in method *m*— does not happen.

```
private static void doIt() {
        int ans;
        if (x+y <= y+z) {
            ans= x+y;
        }
    }
```

The moral of the story is that the result of refactoring a method must be looked at carefully to make sure the intended effect was achieved.

Refactoring methods

Inlining a method

To the right are methods *m* and *doIt* as produced from extracting a method on the previous page. All variables are fields.

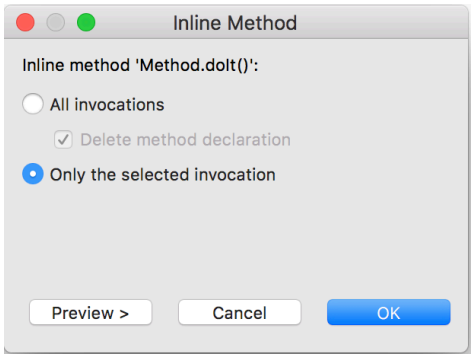
```
public static void m() {
    doIt();
    while (x < y) {
        doIt();
        x= x+1;
    }
}

private static void doIt() {
    if (x+y <= y+z) {
        ans= x+y;
    }
}
```

We now show how to carry out the reverse process, called *inlining*:

1. Place the cursor in the name *doIt* of one of the calls (or, select the whole name) in procedure *m* on the previous page.

2. Choose menu item *Refactor -> Inline*....

3. The window to the right pops up. The name of the method to be inlined is *Method.doit*. That's because all static methods are in class *Method*.



You have the option of inlining only this one call on *doIt* or all of them. And, if you request to inline them all, you have the option of deleting the method declaration.

We decide to change only the selected call, which was the first one.

4. Click button *OK*. That changes method *m* as show on the right. Neat!

```
public static void m() {
    if (x+y <= y+z) {
        ans= x+y;
    }
    while (x < y) {
        doIt();
        x= x+1;
    }
}

private static void doIt() {
    if (x+y <= y+z) {
        ans= x+y;
    }
}
```