9.3 Methods that Don't Return Anything and How Parameters are Passed

One of the methods we've been using since the very start of the module (`System.out.println`) doesn't actually return anything – it just *does* something. In some languages sub-programs that return results and those that don't are separate things – but in Java, those that don't return results are just a special case: instead of putting in a return type like `int` or `String`, we use the special 'no return value' type (or *keyword*) `void`. For example:

```
static void sayHello(String name) {
     System.out.println("Hello " + name + "!");
}
```

Notice the word `void` in place of the return type names we used in the examples above. Also notice that there is no `return` statement – because it doesn't return a result.

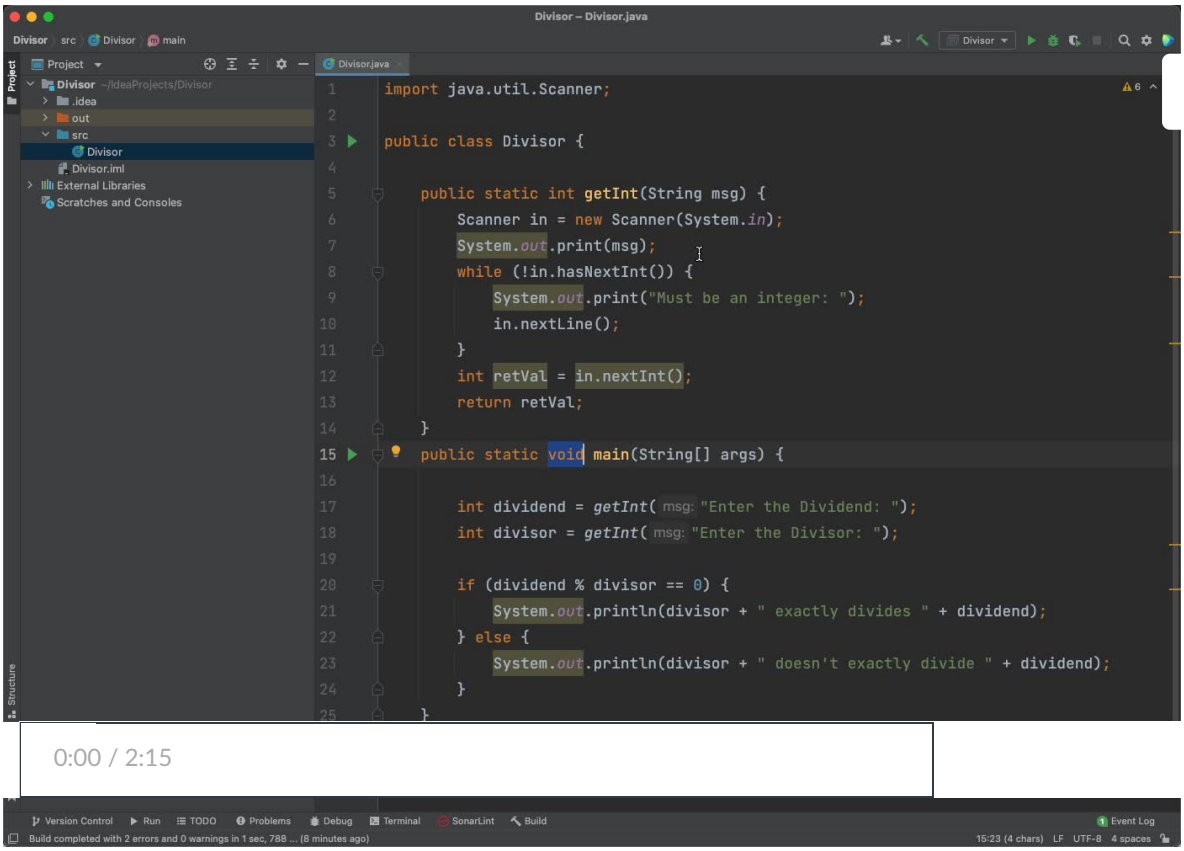
To call this method, we can do things like:

```
sayHello("Mr President");
```

```
String name = "Dave";
sayHello(name);
```

```
String forename = "Darth";
String surname = "Vader";
sayHello(forename + " " + surname);
```

As before you can see that there are lots of options for how you construct the arguments – they can be constants, variables, or operations based on constants and/or variables. Here's a video:



The Return Statement

We have seen the **return** statement in our **getInt** where it indicates the value that will be *returned* to the calling code - the code that called the method in the first place. There's a couple of questions you may have about this?

Can there be more than one return statement? I.e. if you write a method which can end up computing what it needs to in more than one place (usually say in either half of an if-else) can we return directly from that point or do we need to just have one return at the end? i.e.:

```
if (...) {
        //Calculate something
        return myValue;
} else {
        //Calculate something else
        return myOtherValue;
}

vs.

String retVal;
if (...) {
        //Calculate something
        retVal =  myValue;
} else {
        //Calculate something else
        retVal = myOtherValue;
}
return retVal;
```

The first one is obviously shorter - *and it is allowed BUT be careful!* This is a bit like the **break (https://canvas.swansea.ac.uk/courses/44525/pages/6-dot-8-programming-style-and-loops-structured-programming)** statement that we strongly discouraged earlier in the course - you are basically 'jumping out' of the middle of your code. So be careful as it can make the code hard to read - but it's a matter of judgement.

Can a void method have a return statement? Yes it can - but it's just the keyword return on it's own (nothing after it because there's nothing to return). This can be useful if you need to exit early from a method for some reason (maybe the parameters that were passed in didn't make sense - e.g. if it was a method that needed to compute something based on a parameter that was a positive integer, then you might want to exit early if the parameter was negative. But again *be careful* because it's (again) a bit like **break (https://canvas.swansea.ac.uk/courses/44525/pages/6-dot-8-programming-style-and-loops-structured-programming)** .

Parameters and What You Can/Can't Do

Let's go back to our first example in **section 9.1 (https://canvas.swansea.ac.uk/courses/44525/pages/9-dot-1-introduction-to-methods)**

```
static int square(int num) {
     return num * num;
}
```

Now we've seen void methods, it seems we could change it to this:

```
static void square(int num) {
     num = num * num;
}
```

That is, instead of *returning* the new value, we're *changing* (or trying to change…) the value of the parameter we passed in. Does this work? Here's a complete program with both methods to test it (we've changed the name of one because you can't have two methods with the same name in this case):

```
public class SquareTests {
     public static void main(String[] args) {
          //Call the one that returns an int
          int number = 5;
          int squareVal = square(number);
          System.out.println("The 'returning' method: "
             + squareVal);

          //Call the one that tries to change it's argument
          //Try to change number to number*number
          squareVoid(number);
          System.out.println("The 'void' method: "
             + number);

     }
     //Now the two methods
     static int square(int num) {
          return num * num;
     }
```

```
    static void squareVoid(int num) {
        num = num * num;
    }
}
```

The result you should get from this is:

`The 'returning' method: 25`

`The 'void' method: 5`

That is, the second void method does *not* change the value of the parameter. This is because parameters are ***copies*** – *not* the original variables. Try changing the program by putting a `println` statement into the void method:

```
static void squareVoid(int num) {
    num = num * num;
    System.out.println("Inside 'void' method: " + num);
}
```

You should see:

`The 'returning' method: 25`

`Inside 'void' method: 25`

`The 'void' method: 5`

As you can see the value of num *inside* the method *does* change – but because it's a *copy*, and that copy is *thrown away* after the method finishes, the original parameter *does not change*. If you think about it, if the value of a parameter could change outside the method then you couldn't do things like:

```
squareVoid(5);
```

because this would mean trying to change the value of 5 to 25 – which doesn't make sense. On the other hand:

```
int result = square(5);
```

does – because it returns a *new* value and *doesn't* try to change the parameter. What actually happens when you execute:

```
squareVoid(5);
```

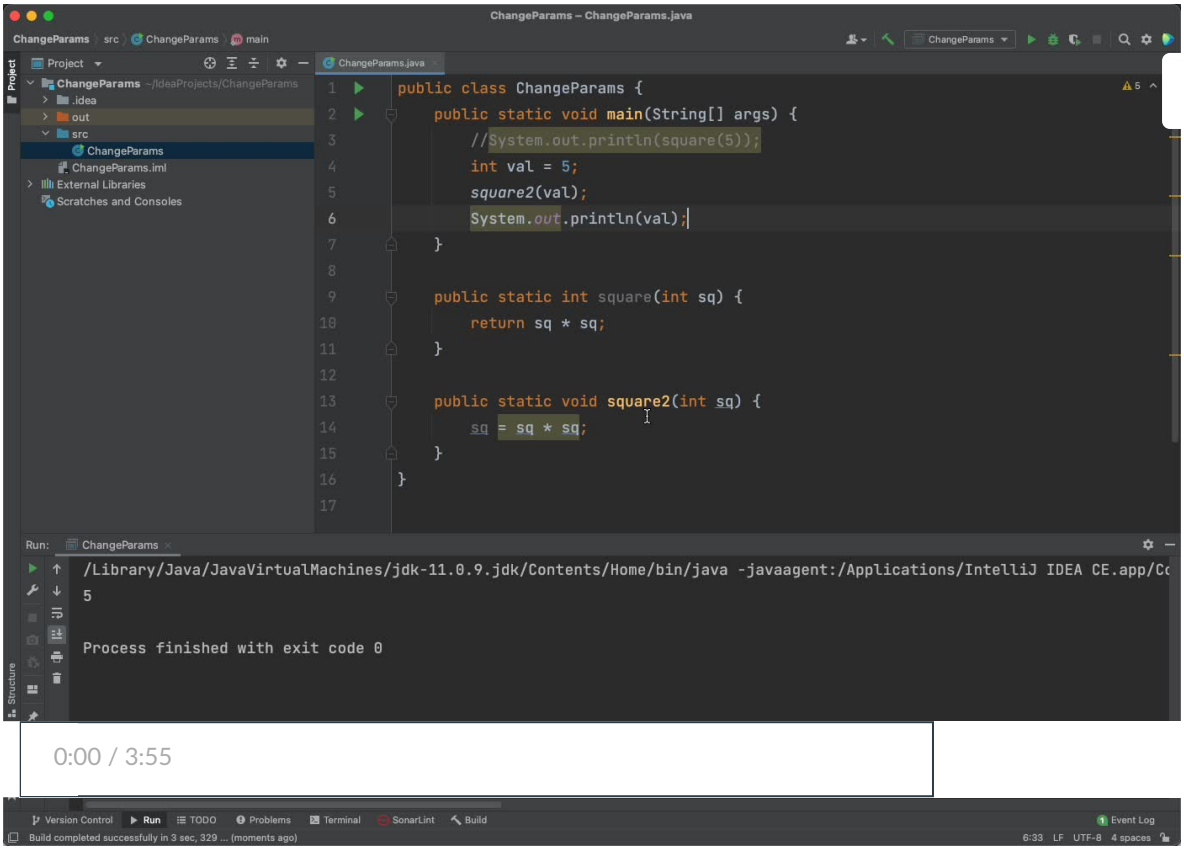
is:

- A copy is made of the value 5;
- that value is copied into a new variable `num` inside the method;
- that new variable is squared and becomes 25;

- if we are using the version which has the `println` in it, that new value of 25 is printed out;
- if the method returns a value, that value is copied back into whatever variable it's going to be stored in;
- finally, the new variable is thrown away when the method ends.

KEY POINT: Parameters are Copies

When you pass a parameter to a method, a copy is made and the method works on the copy. This means you cannot change the value of *simple types* like `int`, `double`, `boolean` which are passed as parameters – you can only return new values. **It *specifically says simple types* in the previous sentence because the situation with *complex types* – like `ArrayLists` – is more complicated as we'll see in the next section.** You can see this example here:



Advanced Aside: Call by Value

The way parameters are passed in Java is known as *Call by Value* – because it is the *value* that is passed, *not* the actual variable. This is the simplest parameter passing mechanism, and the way Java does it, the only one needed. But there are other ways of doing it - for example:

- *Call by Reference* – the actual variable is passed, so you can change it. Java doesn't do this but has a way of 'simulating' it (see later).
- *Call by Value-Result* – the parameter is passed as a copy, but then the value is copied back out to the original variable. Java doesn't do this (most languages don't). In most cases this behaves the same as Call by Reference except in some obscure cases.

Advanced Aside: Final Parameters

As we've seen, you cannot change 'simple' parameters in a method and then expect the changes you make to be kept after the method has finished. However, you *can* use the parameters as local variables inside the method - as long as you know that the changes you make are only 'local' to the method. Having said that, people usually *don't* do that - they just treat the parameter as a source of data and don't change it.

That means you can tell the compiler that the parameter is final - *and you really should:* not doing so amounts to a *code smell* in my opinion. So, for example, if you have a parameter **int thisWontChange** then you can rewrite it to **final int thisWontChange**.