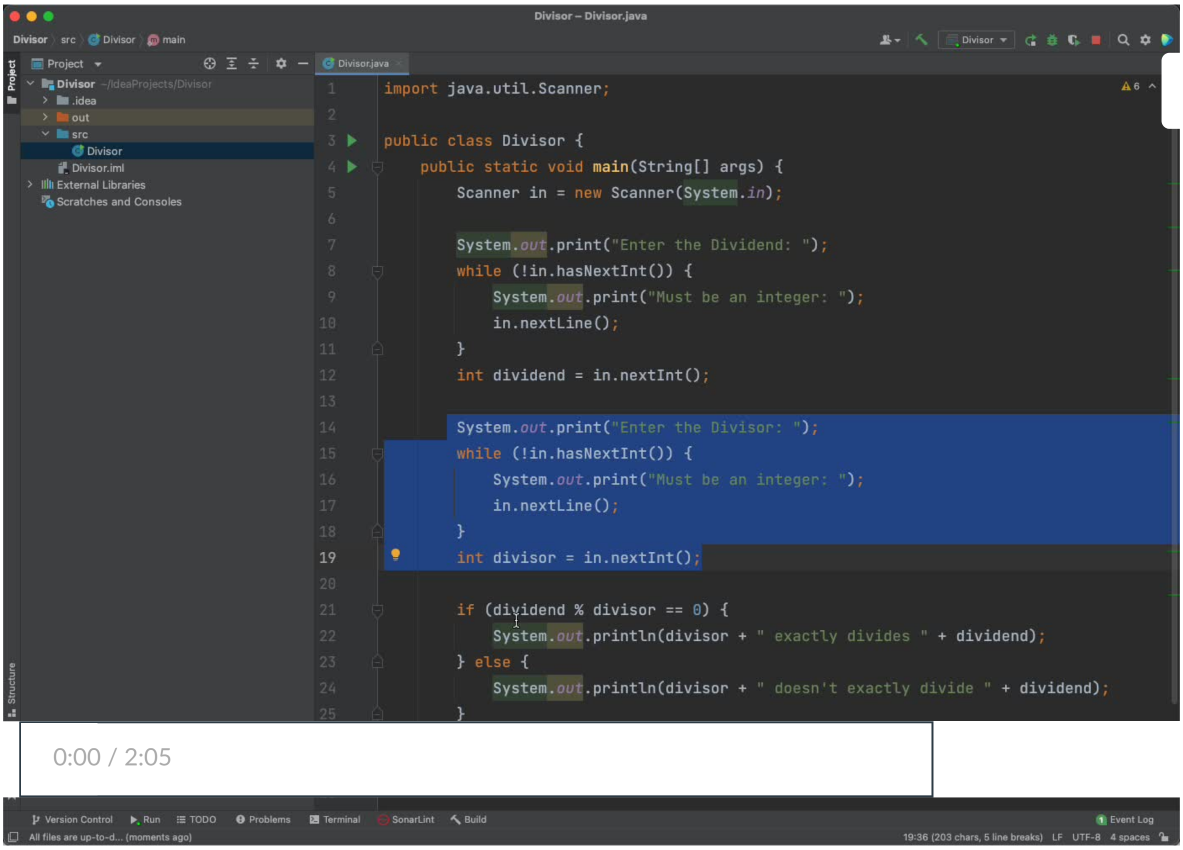
9.1 Introduction to Methods

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

One thing you may have noticed in your solution to the first coursework is that you are repeating (or nearly repeating) chunks of code. It would be nice to only write this code once. Here's an example with quite a large chunk of repeated code:



```java
import java.util.Scanner;

public class Divisor {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        System.out.print("Enter the Dividend: ");
        while (!in.hasNextInt()) {
            System.out.print("Must be an integer: ");
            in.nextLine();
        }
        int dividend = in.nextInt();

        System.out.print("Enter the Divisor: ");
        while (!in.hasNextInt()) {
            System.out.print("Must be an integer: ");
            in.nextLine();
        }
        int divisor = in.nextInt();

        if (dividend % divisor == 0) {
            System.out.println(divisor + " exactly divides " + dividend);
        } else {
            System.out.println(divisor + " doesn't exactly divide " + dividend);
        }
```

Also, even if code is not repeated, programs can get complex and it would be nice to be able to break them into smaller chunks, or *subprograms,* which are easier to manage and understand. Obviously, since there’s a chapter on it, a way of doing this exists. Also we’ve seen some examples already.

- `System.out.println(“Lemon sorbet”); //prints Lemon Sorbet to the screen.`
- `int value = in.nextInt(); //reads in an integer from the keyboard.`

Both System.out.println() and `in.nextInt()` are methods - In fact, *all* the things we’ve seen that end in (..) – are examples of these kinds of subprograms.

Names – Procedures, Functions and Methods

If you look at the two examples above you can see that they are conceptually different – the first one (**System.out.println(...)**) just goes away and does something – prints out whatever is in the brackets; the second one (**in.nextInt**) does something and *returns* a value (in this case an integer) to the program that called or invoked it.

Historically, the first kind of subprogram is called a *procedure* and the second kind a *function* (and is closely related to *mathematical* functions). However, in object oriented programming languages like Java we just call them all *methods*.

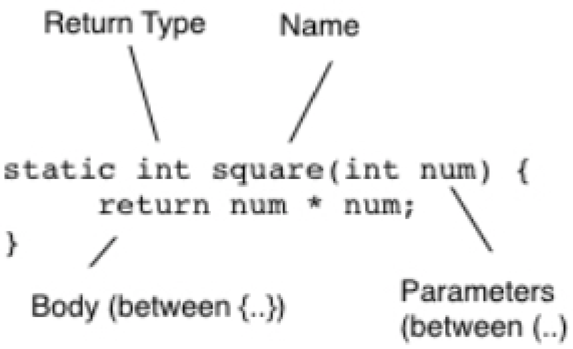
Simple Example – the Square Method

Suppose we are writing a program that needs to compute the squares of integers many times and we want to write a method to do it:

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

The first line defines the *signature* of the method – what it’s called (i.e. it's *name*), the type of data that it *returns*, and the types of the *arguments* or *parameters*: that is, the data that it will work with. In the example above, the method is called `square`, it returns an `int`, and it has one argument/parameter, which is an `int  (int num)`. As well as a *type* the argument also has a *name* (`num`) – this is used inside the body of the method to refer to the value of the parameter.

As this method returns a value, we must use the keyword `return` to specify what that value is – in this case, `num*num`. The various parts are labelled below.



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

KEY POINT – Static

Notice that we’ve used the word static at the start – I’m not going to explain this yet but, **for now**, *you need to put it in* (try taking it out and see what the compiler says). The keyword static is an example of a **modifier** - one of a number of things we can put in front of methods. We'll see a few more later in the module. When we get to the next chapter, we'll start removing static.

We call the `square` method like this:

```
int squareValue = square(5);  // squareValue == 25
System.out.println(square(3)); // prints 9
```

```
int val = 4;
int result;
result = square(val);        // result == 16
```

```
int newVal = 6;
result = square(val + newVal); // result now == 100
```

As you can see it's quite flexible – we can:

- Use numbers (*literals*) like 4 directly as arguments.
- Use variables as arguments (note: they do *not* need to have the same name as the one we use inside the method definition).
- Use more complex expressions (like `a + b`) as arguments.
- Assign the result to a variable.
- Use it directly – for example, print it out.

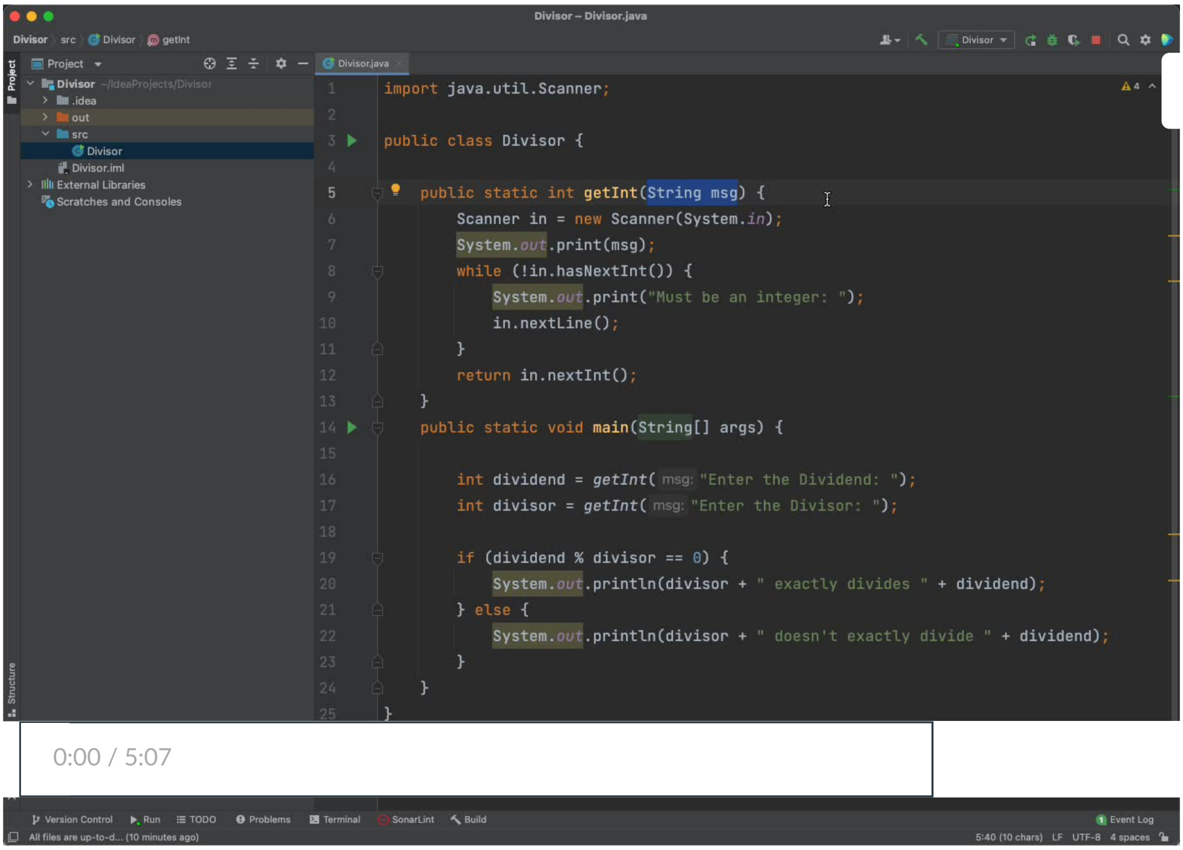
Here's a complete program to show you where the method definition goes.

```
import java.util.Scanner;

public class SquareExample {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          System.out.print("Number? ");

          while(in.hasNextInt()) {
               int squareValue = in.nextInt();

               int result = square(squareValue);
               System.out.println("The square of "
                   + squareValue +
                   " is " + result);

               System.out.print("Another number " +
                   "(anything else to quit)? ");
          }
     }
     //Method goes outside main but inside class
     static int square(int num) {
          return num * num;
     }
}
```

This version uses two `int` variables – one for the argument and one for the result. It keeps printing out the squares of numbers as long as the input is an integer. But we can actually simplify the while loop by calling the method *in the print statement:*

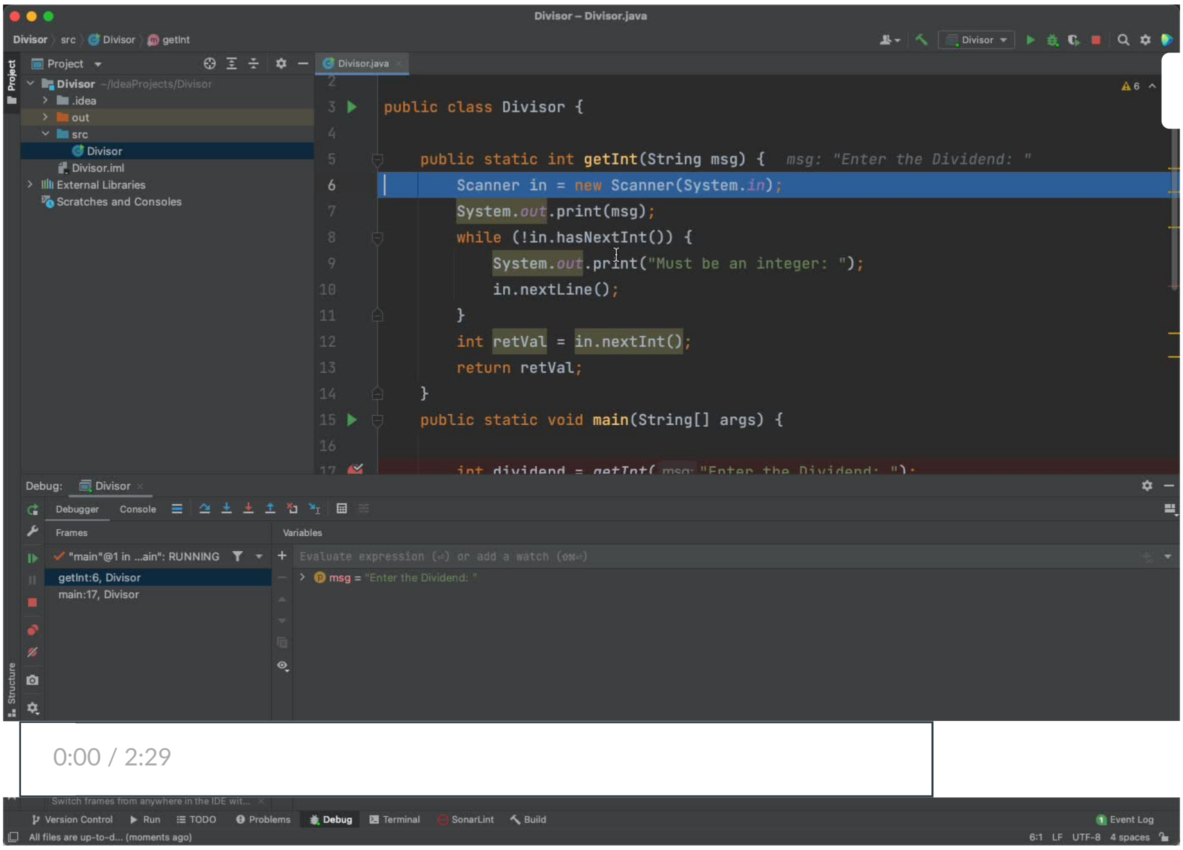
```
while(in.hasNextInt()) {
   int squareValue = in.nextInt();

   System.out.println("The square of "
          + squareValue +
            " is " + square(squareValue)); //Call square here

    System.out.print("Another number " +
          "(anything else to quit)? ");
}
```

Here's an example of how we can shorten and simplify the code from our motivating example, at the top of the section:



```
import java.util.Scanner;

public class Divisor {

    public static int getInt(String msg) {
        Scanner in = new Scanner(System.in);
        System.out.print(msg);
        while (!in.hasNextInt()) {
            System.out.print("Must be an integer: ");
            in.nextLine();
        }
        return in.nextInt();
    }
    public static void main(String[] args) {

        int dividend = getInt( msg: "Enter the Dividend: ");
        int divisor = getInt( msg: "Enter the Divisor: ");

        if (dividend % divisor == 0) {
            System.out.println(divisor + " exactly divides " + dividend);
        } else {
            System.out.println(divisor + " doesn't exactly divide " + dividend);
        }
    }
}
```

And finally, here's what happens when we step through this code using the debugger, so see what happens line-by-line:



```
public class Divisor {

    public static int getInt(String msg) {  msg: "Enter the Dividend: "
        Scanner in = new Scanner(System.in);
        System.out.print(msg);
        while (!in.hasNextInt()) {
            System.out.print("Must be an integer: ");
            in.nextLine();
        }
        int retVal = in.nextInt();
        return retVal;
    }
    public static void main(String[] args) {

        int dividend = getInt( msg: "Enter the Dividend: ");
```

```
"main"@1 in ...ain": RUNNING
getInt:6, Divisor
main:17, Divisor
```

```
msg = "Enter the Dividend: "
```

0:00 / 2:29