Think Java

CHAPTER 4: METHODS AND TESTING

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- So far we’ve written programs that have only one method (main). In this chapter, we’ll show you how to organize programs into multiple methods.
- We’ll also learn how to trace the order in which a program runs. Finally, we’ll discuss strategies for incrementally developing and testing your code.

Topics

- Math Class and its methods
- Composition of Functions
- Defining new functions
- Flow of Execution
- Parameter
- Return
- Stack Diagram

LECTURE 1

Math methods

Utility Methods

- The Java library includes a Math class that provides common mathematical operations. Math is in the java.lang package, so you don't have to import it.

```
double root = Math.sqrt(17.0);
double angle = 1.5;
double height = Math.sin(angle);
```

- The first line sets root to the square root of 17. The third line finds the sine of 1.5 (the value of angle).

Math Class

- Values for the trigonometric functions – sin, cos, and tan – must be in radians. To convert from degrees to radians, you can divide by 180 and multiply by π. Conveniently, the Math class provides a constant double named PI that contains an approximation of π:

```
double degrees = 90;
double angle = degrees / 180.0 * Math.PI;
```

- Notice that PI is in capital letters. Java does not recognize Pi, pi, or pie. Also, PI is the name of a variable, not a method, so it doesn’t have parentheses. The same is true for the constant Math.E, which approximates Euler’s number.

Math Class

- Converting to and from radians is a common operation, so the Math class provides methods that do that for you.

```
double radians = Math.toRadians(180.0);
double degrees = Math.toDegrees(Math.PI);
```

- Another useful method is round, which rounds a floating-point value to the nearest integer and returns a long. The following result is 63 (rounded up from 62.8319).

```
long x = Math.round(Math.PI * 20.0);
```

Math Class

- A long is like an int, but bigger. More specifically, an int uses 32 bits of memory; the largest value it can hold is 231−1, which is about 2 billion. A long uses 64 bits, so the largest value is 263−1, which is about 9 quintillion.
- Take a minute to read the documentation for these and other methods in the Math class. The easiest way to find documentation for Java classes is to do a web search for “Java” and the name of the class.

LECTURE 2

Composition

Composition of Methods

- You have probably learned to evaluate simple expressions like sin(π/2) and log(1/x). First, you evaluate the expression in parentheses, which is called the argument of the function. Then you can evaluate the function itself, either by hand or by punching it into a calculator.
- This process can be applied repeatedly to evaluate more complex expressions like log(1/sin(π/2)). First we evaluate the argument of the innermost function (π/2 = 1.57), then evaluate the function itself (sin(1.57) = 1.0), and so on.

Composition of Methods

- Just as with mathematical functions, Java methods can be composed to solve complex problems. That means you can use one method as part of another. In fact, you can use any expression as an argument to a method, as long as the resulting value has the correct type:

```
double x = Math.cos(angle + Math.PI / 2.0);
```

- This statement divides Math.PI by two, adds the result to angle, and computes the cosine of the sum. You can also take the result of one method and pass it as an argument to another:

```
double x = Math.exp(Math.log(10.0));
```

Composition of Methods

- In Java, the log method always uses base e. So this statement finds the log base e of 10, and then raises e to that power. The result gets assigned to x.
- Some math methods take more than one argument. For example, Math.pow takes two arguments and raises the first to the power of the second. This line computes 210 and assigns the value 1024.0 to the variable x:

```
    double x = Math.pow(2.0, 10.0);
```

Composition of Methods

- When using Math methods, beginners often forget the word Math. For example, if you just write x = pow(2.0, 10.0), you will get a compiler error:

```
File: Test.java  [line: 5]
Error: cannot find symbol
  symbol:   method pow(double,double)
  location: class Test
```

- The message "cannot find symbol" is confusing, but the last two lines provide a useful hint. The compiler is looking for a method named pow in the file Test.java (the file for this example). If you don't specify a class name when referring to a method, the compiler looks in the current class by default.

LECTURE 3

Defining new methods

In-Class Demo Program

- printf(“\n”);
- println();

NEWLINE.JAVA

NewLine Class

- The name of the class is NewLine. By convention, class names begin with a capital letter. NewLine contains two methods, newLine and main. Remember that Java is case-sensitive, so NewLine and newLine are not the same.
- Method names should begin with a lowercase letter and use “camel case”, which is a cute name for jammingWordsTogetherLikeThis. You can use any name you want for methods, except main or any of the Java keywords.
- newLine and main are public, which means they can be invoked (or called) from other classes. And they are both void, which means that they don’t return a result (unlike the Math methods, for example).

newLine()

- The output of this program is:

```
  First line.
  Second line.
```

- Notice the extra space between the lines. If we wanted more space between them, we could invoke the same method repeatedly. Or we could write yet another method (named threeLine) that displays three blank lines.
- In the following program, main invokes threeLine, and threeLine invokes newLine three times. Because newLine is in the same class as threeLine, we don’t have to specify the class name like NewLine.newLine().

In-Class Demo Program

- printf(“\n”);
- println();

NEWLINE.JAVA

LECTURE 4

Flow of execution

Flow of Execution

- When you look at a class definition that contains several methods, it is tempting to read it from top to bottom. But that is not the flow of execution, or the order the program actually runs. The NewLine program runs methods in the opposite order than they are listed.
- Programs always begin at the first statement of main, regardless of where it is in the source file. Statements are executed one at a time, in order, until you reach a method invocation, which you can think of as a detour. Instead of going to the next statement, you jump to the first line of the invoked method, execute all the statements there, and then come back and pick up exactly where you left off.

Flow of Execution

- That sounds simple enough, but remember that one method can invoke another one. In the middle of main, the previous example goes off to execute the statements in threeLine. While in threeLine, it goes off to execute newLine. Then newLine invokes println, which causes yet another detour.
- Fortunately, Java is good at keeping track of which methods are running. So when println completes, it picks up where it left off in newLine; when newLine completes, it goes back to threeLine; and when threeLine completes, it gets back to main.

Flow of Execution

- Beginners often wonder why it's worth the trouble to write other methods, when they could just do everything in main. The NewLine example demonstrates a few reasons:
- Creating a new method allows you to name a block of statements, which makes the code easier to read and understand.
- Introducing new methods can make the program shorter by eliminating repetitive code. For example, to display nine consecutive newlines, you could invoke threeLine three times.

Flow of Execution

- A common problem-solving technique is **to break problems down into sub-problems**. Methods allow you to focus on each sub-problem in isolation, and then compose them into a complete solution.
- Perhaps most importantly, organizing your code into multiple methods allows you to test individual parts of your program separately. It’s easier to get a complex program working if you know that each method works correctly.

LECTURE 5

Parameters and arguments

Parameters

- Some of the methods we have used require **arguments**, which are the values you provide in parentheses when you invoke the method.
- For example, the Math.sin method takes a double argument. To find the sine of a number, you have to provide the number: Math.sin(0.0). Similarly, the System.out.println method takes a String argument. To display a message, you have to provide the message: System.out.println("Hello").

In-Class Demo Program

- Method with parameters.

PRINTTWICE.JAVA

printTwice()

- The printTwice method has a parameter named s with type String. When you invoke printTwice, you have to provide an argument with type String.
- Before the method executes, the argument gets assigned to the parameter. In this example, the argument "Don't make me say this twice!" gets assigned to the parameter s.
- This process is called parameter passing because the value gets passed from outside the method to the inside. An argument can be any kind of expression, so if you have a String variable, you can use its value as an argument:

```
String message = "Never say never.";
printTwice(message);
```

printTwice()

- The value you provide as an argument must have the same (or compatible) type as the parameter. For example, if you try:

```
printTwice(17);  // syntax error
```

- You will get an error message like this:

```
File: Test.java  [line: 10]
Error: method printTwice in class Test cannot be applied
       to given types;
  required: java.lang.String
  found: int
  reason: actual argument int cannot be converted to
          java.lang.String by method invocation conversion
```

printTwice()

- Sometimes Java can convert an argument from one type to another automatically. For example, Math.sqrt requires a double, but if you invoke Math.sqrt(25), the integer value 25 is automatically converted to the floating-point value 25.0. But in the case of **printTwice**, Java can’t (or won’t) convert the integer 17 to a String.
- Parameters and other variables only exist inside their own methods. Inside main, there is no such thing as s. If you try to use it there, you’ll get a compiler error. Similarly, inside **printTwice** there is no such thing as message. That variable belongs to main. Because variables only exist inside the methods where they are defined, they are often called local variables.

printTime()

- Here is an example of a method that takes two parameters:

```
public static void printTime(int hour, int minute)
{
    System.out.print(hour);
    System.out.print(":");
    System.out.println(minute);
}
```

- In the parameter list, it may be tempting to write:

```
public static void printTime(int hour, minute) {
// error
```

- But that format (without the second int) is only allowed for local variables. For parameters, you need to declare the type of each variable separately.

printTime()

- To invoke this method, we have to provide two integers as arguments:

```
int hour = 11;
int minute = 59;
printTime(hour, minute);
```

- Beginners sometimes make the mistake of “declaring” the arguments:

```
int hour = 11;
int minute = 59;
printTime(int hour, int minute);   // syntax error
```

- That’s a syntax error, because the compiler sees int hour and int minute as variable declarations, not expressions. You wouldn’t declare the types of the arguments if they were simply integers:

```
printTime(int 11, int 59);   // syntax error
```

In-Class Demo Program

- Method with parameters.

PRINTTIME.JAVA

	Procedural Abstraction	**Algebraic Abstraction**
object	operation (procedure, function, method)	Algebra (abstract data types)
components	a sequence of operations	several operations operating on the same sorts
applied to	values	sorts (types)
use	call	instantiation
formal parameters	formal parameters for values	formal parameters for sorts
actual parameters	values	representable data types
combination	call of procedure within another abstraction	use as a sub-algebra within another algebra

LECTURE 6

Stack diagrams

Stack diagrams

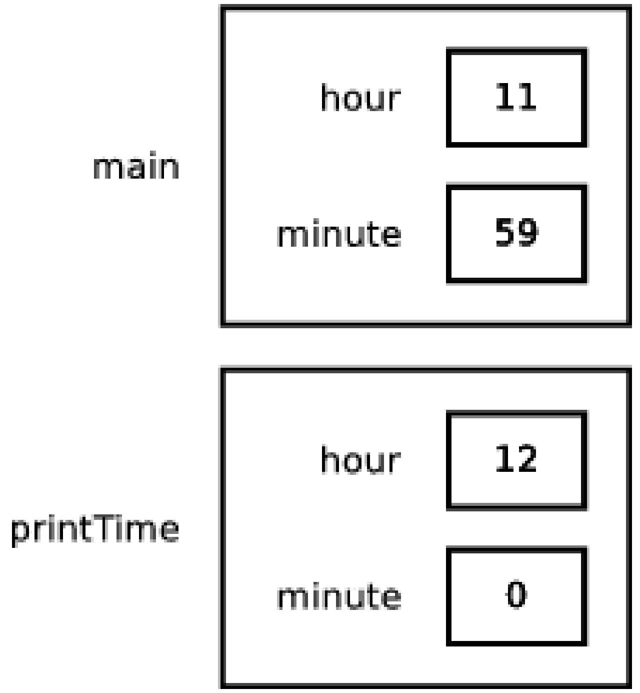
- printTime has two parameters, named hour and minute. And main has two variables, also named hour and minute. Although they have the same names, these variables are not the same. The hour in printTime and the hour in main refer to different memory locations, and they can have different values. For example, you could invoke printTime like this:

```
int hour = 11;
int minute = 59;
printTime(hour + 1, 0);
```

- Before the method is invoked, Java evaluates the arguments; in this example, the results are 12 and 0. Then it assigns those values to the parameters. Inside printTime, the value of hour is 12, not 11, and the value of minute is 0, not 59. Furthermore, if printTime modifies one of its parameters, that change has no effect on the variables in main.

Stack diagrams

- One way to keep track of everything is to draw a stack diagram, which is a memory diagram (see Section 2.3) that shows currently running methods. For each method there is a box called a frame that contains the method’s parameters and local variables. The name of the method appears outside the frame; the variables and parameters appear inside.



Stack diagrams

- As with memory diagrams, stack diagrams show variables and methods at a particular point in time. Figure 4.1 is a stack diagram at the beginning of the printTime method. Notice that main is on top, because it executed first.
- Stack diagrams are a good mental model for how variables and methods work at run-time. Learning to trace the execution of a program on paper (or on a whiteboard) is a useful skill for communicating with other programmers.

LECTURE 7

Return values

Return

- When you invoke a void method, the invocation is usually on a line all by itself. For example:

```
printTime(hour + 1, 0);
```

- On the other hand, when you invoke a value-returning method, you have to do something with the return value. We usually assign it to a variable or use it as part of an expression, like this:

```
double error = Math.abs(expect - actual);
double height = radius * Math.sin(angle);
```

Return

Compared to void methods, value-returning methods differ in two ways:

1. They declare the type of the return value (the return type);
2. They use at least one return statement to provide a return value.

Here’s an example from a program named Circle.java. The calculateArea method takes a double as a parameter and returns the area of a circle with that radius (i.e., π r2).

```
public static double calculateArea(double radius) {
    double result = Math.PI * radius * radius;
    return result;
}
```

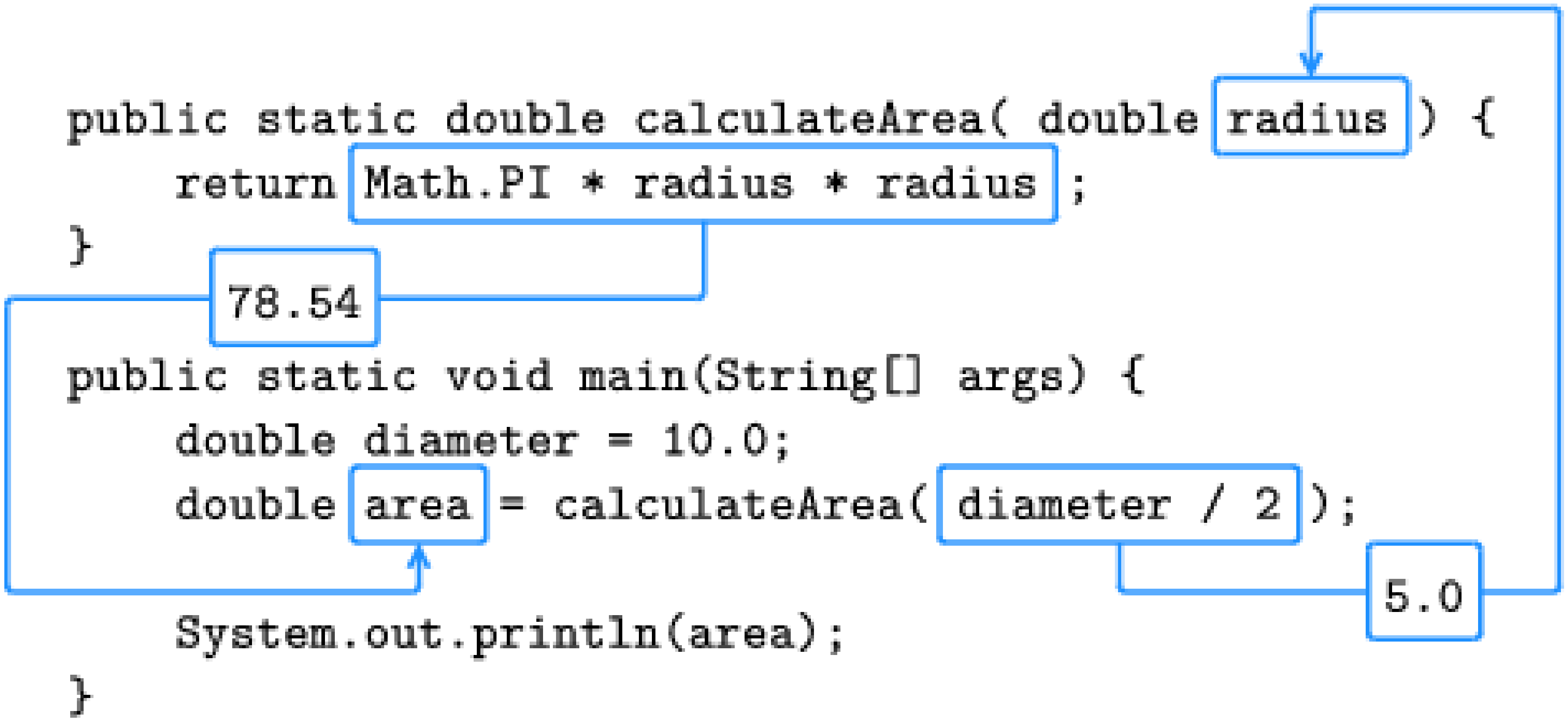
Return

- As usual, this method is public and static. But in the place where we are used to seeing void, we see double, which means that the return value from this method is a double
- The last line is a new form of the return statement that means, “return immediately from this method, and use the following expression as the return value.” The expression you provide can be arbitrarily complex, so we could have written this method more concisely:

```
public static double calculateArea(double radius) {
    return Math.PI * radius * radius;
}
```

Return

- On the other hand, temporary variables like result often make debugging easier, especially when you are stepping through code using an interactive debugger (see Appendix A.6).
- Figure 4.2 illustrates how data values flows through the program. When the main method invokes calculateArea, the value 5.0 is assigned to the parameter radius. calculateArea then returns the value 78.54, which is assigned to the variable area.



```
public static double calculateArea( double radius ) {
    return Math.PI * radius * radius ;
}

public static void main(String[] args) {
    double diameter = 10.0;
    double area = calculateArea( diameter / 2 );

    System.out.println(area);
}
```

Return

- The type of the expression in the return statement must match the return type of the method itself. When you declare that the return type is double, you are making a promise that this method will eventually produce a double value.
- If you try to return with no expression, or return an expression with the wrong type, the compiler will give an error.

LECTURE 8

Incremental development

Incremental Development

- People often make the mistake of writing a lot of code before they try to compile and run it. Then they spend way too much time debugging. A better approach is what we call incremental development. The key aspects of incremental development are:
- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know where to look.
- Use variables to hold intermediate values so you can check them, either with print statements or by using a debugger.
- Once the program is working, you can consolidate multiple statements into compound expressions (but only if it does not make the program more difficult to read).

Incremental Development

- As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition:
- The first step is to consider what a distance method should look like in Java. In other words, what are the inputs (parameters) and what is the output (return value)? For this method, the parameters are the two points, and it is natural to represent them using four double values. The return value is the distance, which should also have type double.

Incremental Development

- Already we can write an outline for the method, which is sometimes called a stub. The stub includes the method declaration and a return statement:

```
public static double distance
        (double x1, double y1, double x2, double y2)
{
    return 0.0;  // stub
}
```

- The return statement is a placeholder that is only necessary for the program to compile. At this stage the program doesn't do anything useful, but it is good to compile it so we can find any syntax errors before we add more code.

Incremental Development

- It’s usually a good idea to think about testing before you develop new methods; doing so can help you figure out how to implement them. To test the method, we can invoke it from main using the sample values:
- double dist = distance(1.0, 2.0, 4.0, 6.0);
- With these values, the horizontal distance is 3.0 and the vertical distance is 4.0. So the result should be 5.0, the hypotenuse of a 3-4-5 triangle. When you are testing a method, it is necessary to know the right answer.
- Once we have compiled the stub, we can start adding code one line at a time. After each incremental change, we recompile and run the program. If there is an error, we have a good idea where to look: the lines we just added.

Incremental Development

- The next step is to find the differences x2 – x1 and y2 – y1. We store those values in temporary variables named dx and dy, so that we can examine them with print statements before proceeding. They should be 3.0 and 4.0.

```
public static double distance
          (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    System.out.println("dx is " + dx);
    System.out.println("dy is " + dy);
    return 0.0;   // stub
}
```

- We will remove the print statements when the method is finished. Code like that is called scaffolding, because it is helpful for building the program, but it is not part of the final product.

Incremental Development

- The next step is to square dx and dy. We could use the Math.pow method, but it is simpler (and more efficient) to multiply each term by itself.

```
    public static double distance
            (double x1, double y1, double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        double dsquared = dx * dx + dy * dy;
        System.out.println("dsquared is " + dsquared);
        return 0.0;   // stub
    }
```

- Again, you should compile and run the program at this stage and check the intermediate value, which should be 25.0. Finally, we can use Math.sqrt to compute and return the result.

```
public static double distance
        (double x1, double y1, double x2, double
y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx * dx + dy * dy;
    double result = Math.sqrt(dsquared);
    return result;
}
```

- As you gain more experience programming, you might write and debug more than one line at a time. But by using incremental development, scaffolding, and testing, your code is more likely to be correct the first time

Homework

Homework

- Textbook Exercise
- Project 4