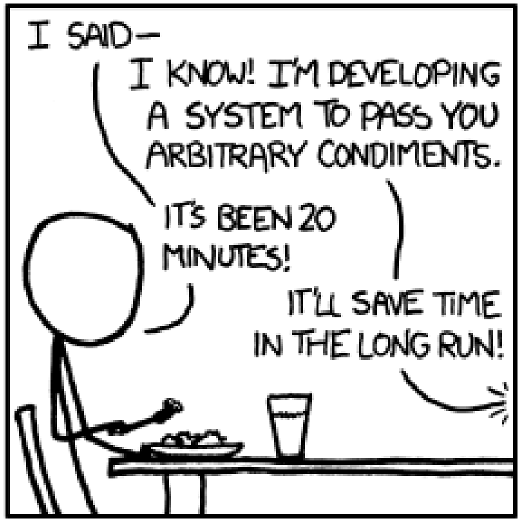
Chapter 3: Introduction to Parameters & Objects


I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

WEEK 3:

PARAMETERS & OVERLOADING METHODS

METHODS THAT RETURN VALUES

STRING OBJECTS

INTERACTIVE METHODS & SCANNER OBJECTS

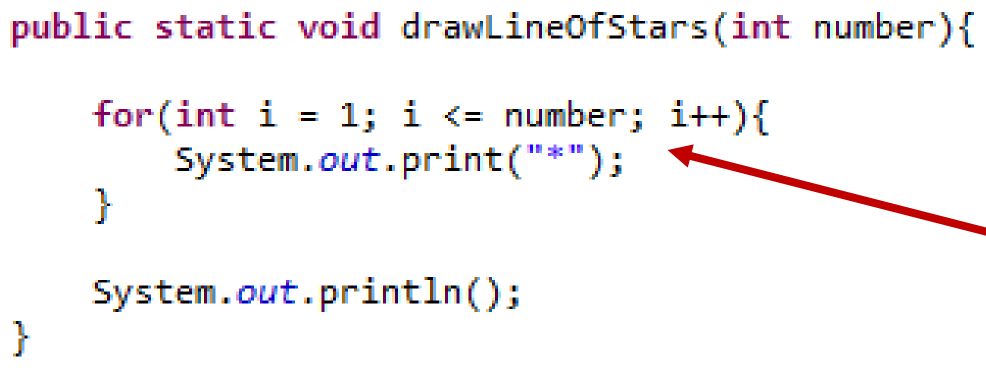
Cartoon from http://xkcd.com/974/

Parameters

- **Parameter:** Any of a set of characteristics that distinguish different members of a family
  - To "**parameterize**" a task is to identify a SET of its parameters
- For example, assume you want to print out a line of asterisks
- How many should be printed? You could write a FAMILY of separate methods to handle a wide range of line lengths (oneStar, twoStars, threeStars, …, manyManyStars, …)
- These methods would obviously be very similar to one another – thus containing much redundancy
- **BETTER APPROACH:** Note that the **PARAMETER** "number of stars" is the only thing distinguishing one program from another – what if there were some way to indicate this, while still using the same "base" code?
- We'd like to be able to specify the length of the line of stars we want to have printed at the time that we call our method – but wouldn't this cause problems with SCOPE (i.e. our method will only "recognize" its own LOCAL VARIABLES

Parameters, cont.

- Fortunately, there's a way around this issue – we can specify one or more parameters to a method and "pass" them in for use in the body of the method
- **Example:**



```
public static void drawLineOfStars(int number){

    for(int i = 1; i <= number; i++){
        System.out.print("*");
    }

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

The **parameter "number"** appears in the **method header**

The value passed on for this **parameter "number"** controls the number of loop iterations

- Now, when you call the method, you need to provide a value that should be used wherever this parameter appears in the execution of the statements in the body of the method
- In the example above, the parameter **number** is determining how many time the loop iterates

Formal Parameters vs. Actual Parameters

- **Formal Parameter:** A variable that appears inside parentheses in the **header of a method** that is used to **generalize** the method's behavior
- **Actual Parameter:** A specific value or expression that appears inside parentheses in a **method call** (also referred to as an **ARGUMENT**)
- The syntax used when declaring **formal parameters** IS NOT THE SAME as that used for passing **actual parameters**
- Thus, calling the method in the following way would result in an **ERROR**:

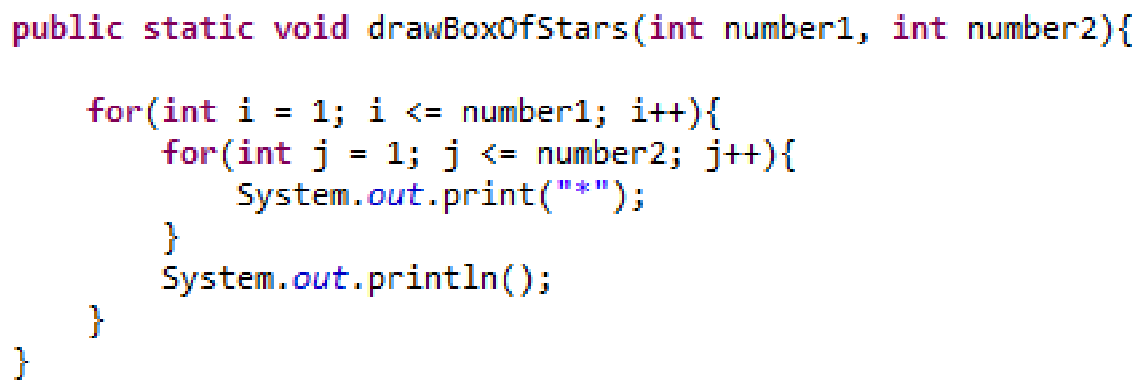
```
drawLineOfStars(int number);
```

ONLY a value or an expression that can be evaluated should be included within the parentheses of a **METHOD CALL**

- The above gives precise definitions for formal parameter and actual parameter. In actual practice, the terms "parameter" and argument are often used interchangeably.

The Mechanics of Parameters

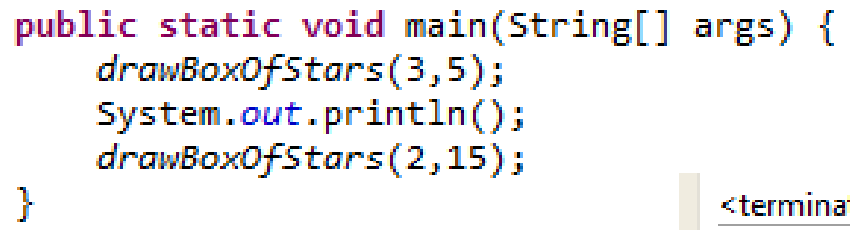
- When Java executes a call on a method, it **INITIALIZES** the method's **PARAMETERS**
- The values or expressions passed as actual parameters (arguments) are used to initialize the local variables whose names match those of the formal parameters
- **Example:** This method prints a box of stars



```
public static void drawBoxOfStars(int number1, int number2){

    for(int i = 1; i <= number1; i++){
        for(int j = 1; j <= number2; j++){
            System.out.print("*");
        }
        System.out.println();
    }
}
```

NOTE: The method header contains 2 FORMAL PARAMETERS

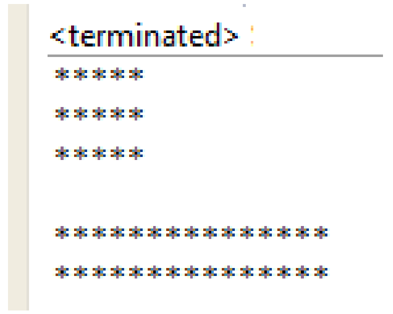
The **main method** below has two **method calls**, using different **ACTUAL PARAMETERS** (**arguments**) each time:



```
public static void main(String[] args) {
    drawBoxOfStars(3,5);
    System.out.println();
    drawBoxOfStars(2,15);
}
```

Output of method call 1 →

Output of method call 2 →



```
<terminated>
*****
*****
*****

***************
***************
```

Limitations of Parameters

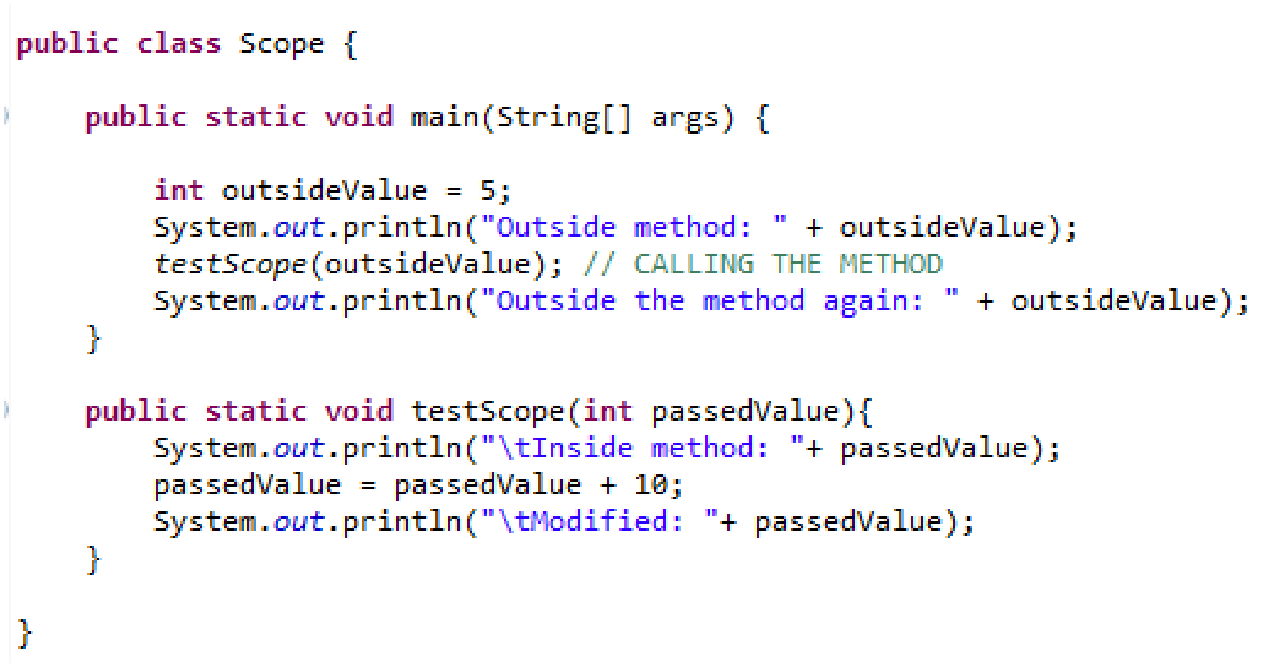
- As we've seen, parameters can be used to provide INPUT to a method – but how do we get values OUT OF methods?

- When a method with a parameter is called, a **local variable** is created within the method and initialized to the value passed in as the ACTUAL PARAMETER
  - In other words, the local variable is created within the **SCOPE** of the method and received only its **INITIALIZING VALUE** from outside the method
  - Since it is a local variable, changes to its value within the method **cannot** affect the values of any variables outside the method

Limitations of Parameters: Example

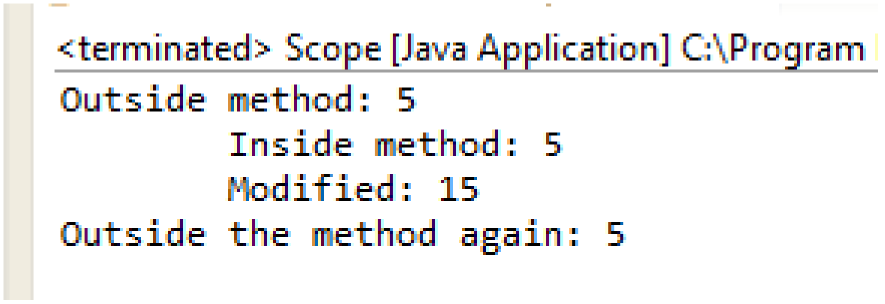
- Changes to a variable's value **INSIDE** a method are made **ONLY** to the method's **LOCAL VARIABLE**
- These changes do not affect any variable **OUTSIDE** the method – the two sets of variables have **DIFFERENT SCOPES**

SAMPLE CODE



```java
public class Scope {

    public static void main(String[] args) {

        int outsideValue = 5;
        System.out.println("Outside method: " + outsideValue);
        testScope(outsideValue); // CALLING THE METHOD
        System.out.println("Outside the method again: " + outsideValue);
    }

    public static void testScope(int passedValue){
        System.out.println("\tInside method: "+ passedValue);
        passedValue = passedValue + 10;
        System.out.println("\tModified: "+ passedValue);
    }

}
```

OUTPUT

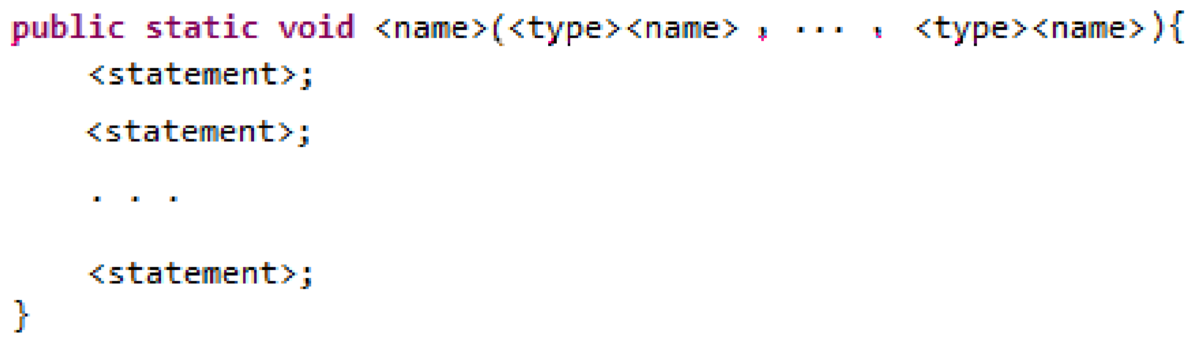


```
<terminated> Scope [Java Application] C:\Program
Outside method: 5
        Inside method: 5
        Modified: 15
Outside the method again: 5
```

NOTE: The changes to the variable **INSIDE** the method (*passedValue*), do not affect the value of the variable **OUTSIDE** the method (*outsideValue*), even though the initial value of *passedValue* is set to equal the value of *outsideValue*

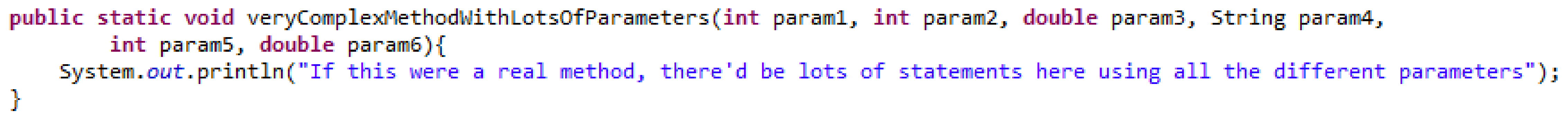
Multiple Parameters

- Here is the precise syntax used to declare static methods with parameters:



```
public static void <name>(<type><name> , ... , <type><name>){
    <statement>;
    <statement>;
    . . .
    <statement>;
}
```

  - From this, we see that methods can accept **MULTIPLE** parameters as input
  - If your method has numerous parameters, the method header can become very long, so Java lets us wrap long lines by inserting a line break after an operator or a parameter and indenting the line that follows by twice the normal indentation width:



```
public static void veryComplexMethodWithLotsOfParameters(int param1, int param2, double param3, String param4,
        int param5, double param6){
    System.out.println("If this were a real method, there'd be lots of statements here using all the different parameters");
}
```

Parameters vs. Constants

- Earlier, we discussed how a constant could be used in a program to easily change the value of a variable that appeared in multiple points in your program with one command
- This gives your program more flexibility – instead of being restricted to specific values, the values of certain variables can be changed by changing the value of the constant variable

- The advantage to using parameters instead of constants is that parameters allow for even more flexibility.
- Constants can only change values for the entire execution of the program, whereas the values of parameters can change within the execution of the program

Overloading of Methods

- You’ll find that, frequently, you want to create slight variations of the **SAME** method
- For example, you might have a drawBox method that allows you to specify the length and height of a rectangular box
- Sometimes this is very useful, however, most of the time, all you want is a generic 2 x 2 square
- You can create a second version of your drawBox method to do this for you

… But you already knew this – so what’s so interesting?

- What’s different is that you can give your **NEW** method the **SAME NAME** as your **OLD** method
  - This is called **OVERLOADING**

Overloading, cont.

- **Method signature:** The name of a method, along with its number and type of parameters
- **Method overloading:** The ability to define two or more different methods with the same name but different method signatures

- Provided they **DIFFER** the **TYPE and/or NUMBER** of **FORMAL PARAMETERS** (i.e. They have DIFFERENT **METHOD SIGNATURES**), Java will be able to distinguish between the two to give you what you want

Methods that Return Values

- **Return:** To send a value out as the result of a method. This value can be assigned to a variable in your program outside of the function that created it, enabling it to be used in expressions in the program.
- **“void”** methods do not return any value

Example:

- Suppose you want to write a method that calculates square roots. However, instead of having the result print to the console, you want to use the calculated square root value in another expression in your program.
- **How can you accomplish this?**

- It is possible to write a method that **RETURNS** a value
- You can tell whether or not a method returns a value by looking at its **method header**

Methods that Return Values, cont.

- In the methods we've written so far, the method headers have started with the following terms:

  public static void …
- The word **void** indicates the return type of the method – namely, that these methods return **NOTHING**

- Compare the method headers we've been writing to the one shown below:

```
public static double squareRoot(double n){
        . . . <body of method> . . .
}
```

- The word **double** indicates the return type of the method – namely, that this methods will return a value of type double

- **FORTUNATELY,** you don't actually need to write a square root calculating method – Java already has one built in that you can use

The Math Class

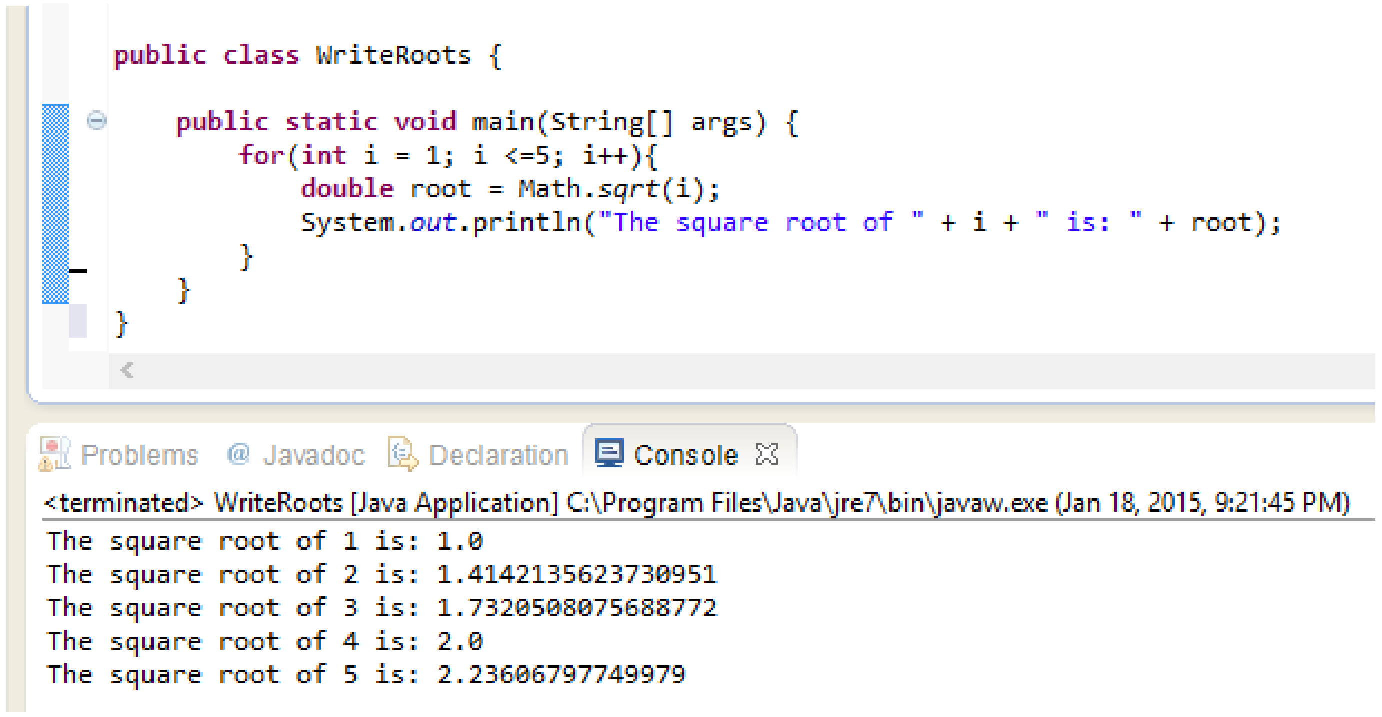
- One of the most useful classes in the Java class libraries is the Math class
- A square root method is one of the methods included in the Math class
- To use a method from another class, you refer to it using **DOT NOTATION:**

  <class name>.<element>

- To use the square root method in the Math class, you would write:

  Math.sqrt(<value>)

Square Root Example:



```
public class WriteRoots {

    public static void main(String[] args) {
        for(int i = 1; i <=5; i++){
            double root = Math.sqrt(i);
            System.out.println("The square root of " + i + " is: " + root);
        }
    }
}
```

```
<terminated> WriteRoots [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 18, 2015, 9:21:45 PM)
The square root of 1 is: 1.0
The square root of 2 is: 1.4142135623730951
The square root of 3 is: 1.7320508075688772
The square root of 4 is: 2.0
The square root of 5 is: 2.23606797749979
```

The Math Class, cont.

- The table to the right lists some of the most frequently used static methods in the Math class

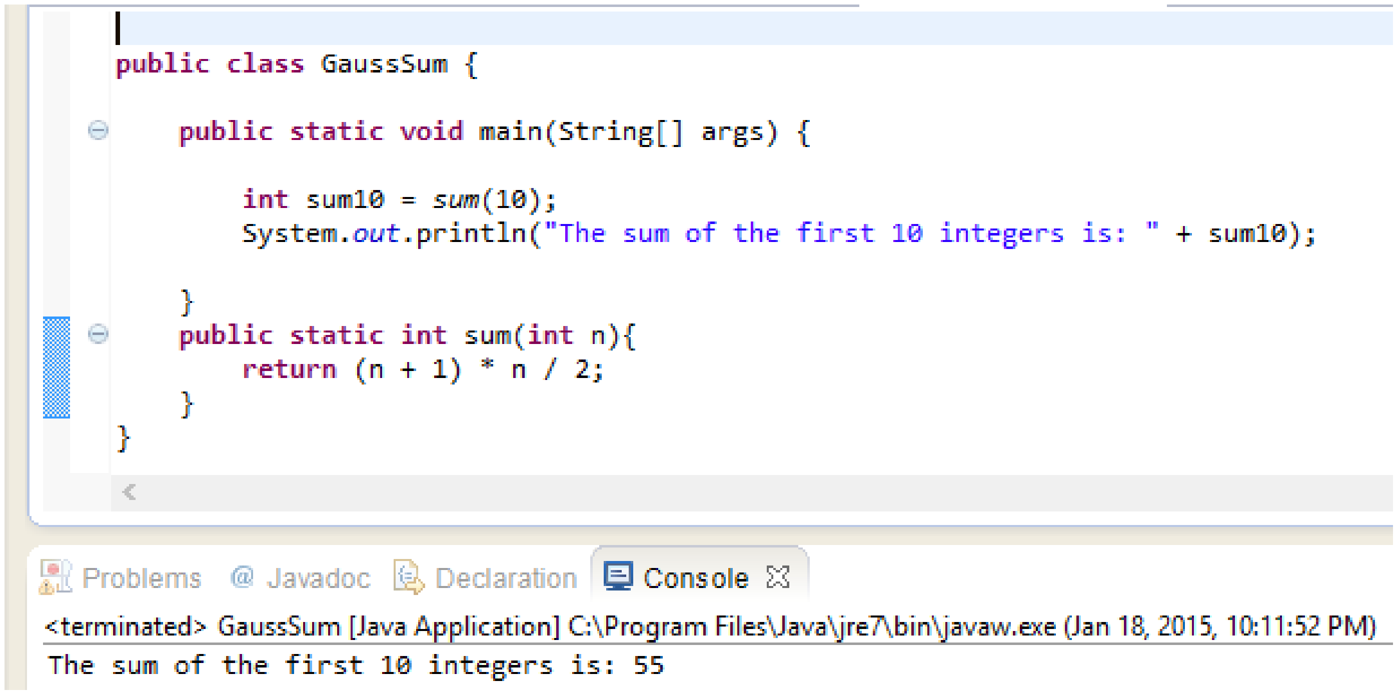
- The Math class also includes two important constants:
  - **PI --** ratio of circumference of a circle to its diameter (3.14159…)
  - **E** – Base used in natural logarithms (2.71828…)

Method	Description
min	Returns the minimum of two values
max	Returns the maximum of two values
exp	Exponent base *e*
pow	Power (general exponentiation)
log	Logarithm base *e*
log10	Logarithm base 10
round	Rounds real numbers to the nearest integer
random	Returns a random double value k such that $0.0 \leq k < 1.0$
abs	Returns the absolute value

Defining Methods that Return Values

- To write your own methods that return values to the program that called them, you will need to use the **return** statement

Example: The method below returns the sum of the first **n** integers by using Gauss's formula



```
public class GaussSum {

    public static void main(String[] args) {

        int sum10 = sum(10);
        System.out.println("The sum of the first 10 integers is: " + sum10);

    }
    public static int sum(int n){
        return (n + 1) * n / 2;
    }
}
```

```
<terminated> GaussSum [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 18, 2015, 10:11:52 PM)
The sum of the first 10 integers is: 55
```

- A **COMMON ERROR** is to ignore the returned value
- The returned value needs to be stored in a variable
- In the program to the left, the value returned by the **sum** method is stored in the variable **sum10**

Objects

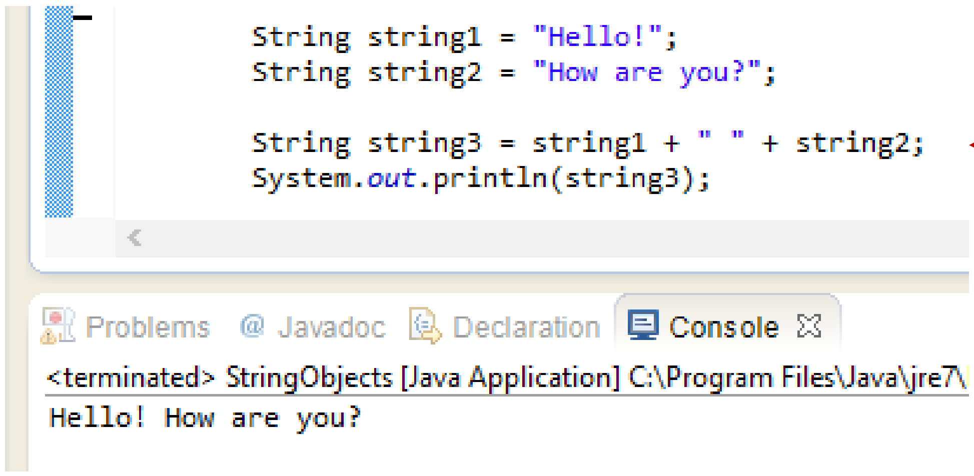
- It is convenient to package data and the operations that need to be performed on that data in to a single entity – Objects let us do this
- **Object:** A programming entity that contains both state (data) and behavior (methods)
- **Class:** A category or type of object

- We can think of a class as a sort of BLUEPRINT of what an object looks like
- Once Java has the class “blueprint,” it can create individual objects that match the blueprint

String Objects

- One of the most commonly used type of objects
- We have discussed string literals already – and we’ve used them in print statements
- These string literals are actually representations of String objects
- An individual object of a certain class type is referred to as an **INSTANCE** of that class

- You can declare a variable of type String to store textual data
- **Example:**



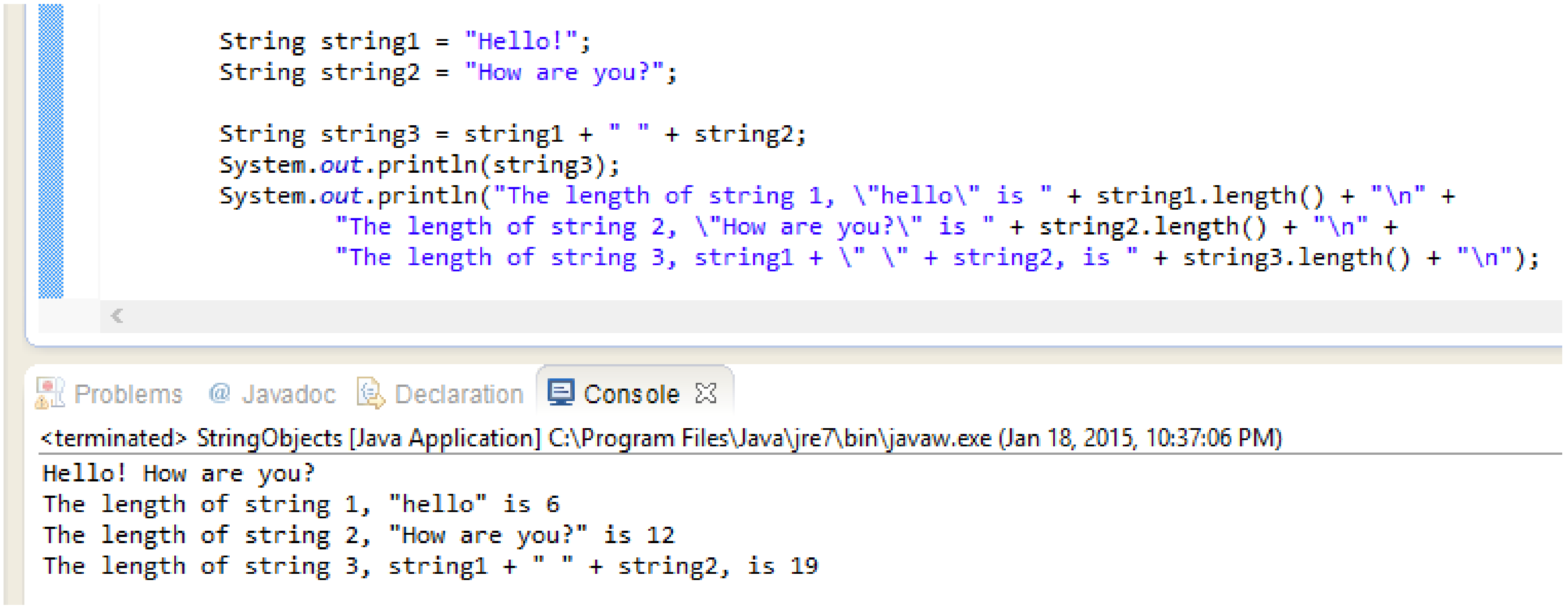
```
String string1 = "Hello!";
String string2 = "How are you?";

String string3 = string1 + " " + string2;
System.out.println(string3);
```

```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\
Hello! How are you?
```

Note use of string concatenation!

String Class Methods

- In addition to giving us a way to store textual data, the String class also contains methods with which to manipulate textual data
- These class methods are accessed via **dot notation** (just as the Math class methods were)
- For example, we can determine the length of any String object by using the **length** method of the String class:



```
String string1 = "Hello!";
String string2 = "How are you?";

String string3 = string1 + " " + string2;
System.out.println(string3);
System.out.println("The length of string 1, \"hello\" is " + string1.length() + "\n" +
        "The length of string 2, \"How are you?\" is " + string2.length() + "\n" +
        "The length of string 3, string1 + \" \" + string2, is " + string3.length() + "\n");
```

```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 18, 2015, 10:37:06 PM)
Hello! How are you?
The length of string 1, "hello" is 6
The length of string 2, "How are you?" is 12
The length of string 3, string1 + " " + string2, is 19
```

Accessing Characters in Strings

- What if we want to access individual characters within a string?
- We can do this by specifying the location of our character of interest – this location is referred to as the INDEX of the character

- **Index:** An integer used to specify a location in a sequence of values

- Java uses a zero-based indexing system
- Accordingly, the word "hello" could be represented by the following diagram:

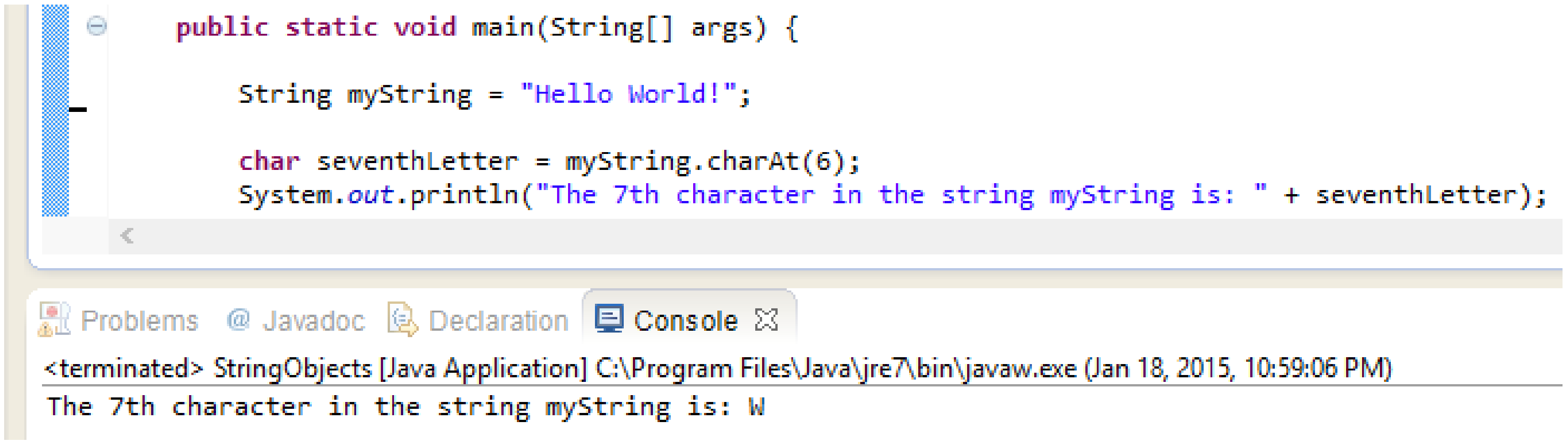
h	**e**	**l**	**l**	**o**	← **characters**
0	1	2	3	4	← indices

Accessing Characters in Strings, cont.

- For longer strings, spaces between words are also individual characters
- The string “**Hello World!**” is shown below:

H	e	l	l	o		W	o	r	l	d	!	← characters
0	1	2	3	4	5	6	7	8	9	10	11	← indices

We can use the strings indices to get the value of any individual character -- provided we use an index whose value satisfies
$0 \le index < string\ length$

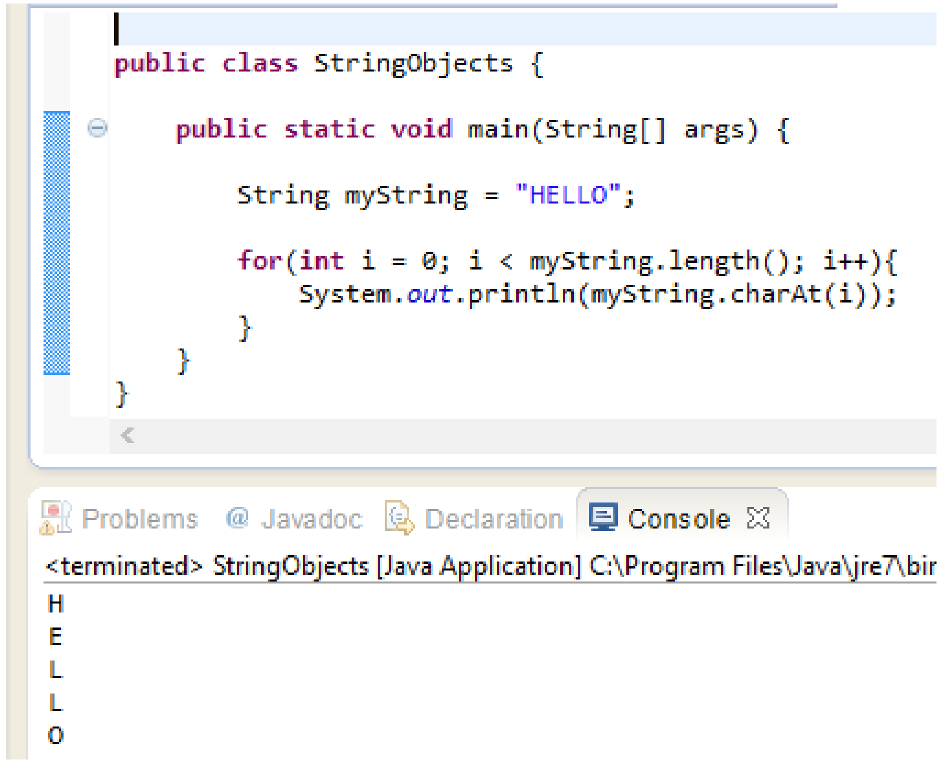


```
public static void main(String[] args) {

    String myString = "Hello World!";

    char seventhLetter = myString.charAt(6);
    System.out.println("The 7th character in the string myString is: " + seventhLetter);
```

```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 18, 2015, 10:59:06 PM)
The 7th character in the string myString is: W
```

Using Loops with String Objects

- What if we want to access individual characters within a string ONE AFTER THE OTHER?
- The length method gives us a handy way of determining an ending condition for a loop that will do this for us:



```
public class StringObjects {

    public static void main(String[] args) {

        String myString = "HELLO";

        for(int i = 0; i < myString.length(); i++){
            System.out.println(myString.charAt(i));
        }
    }
}
```

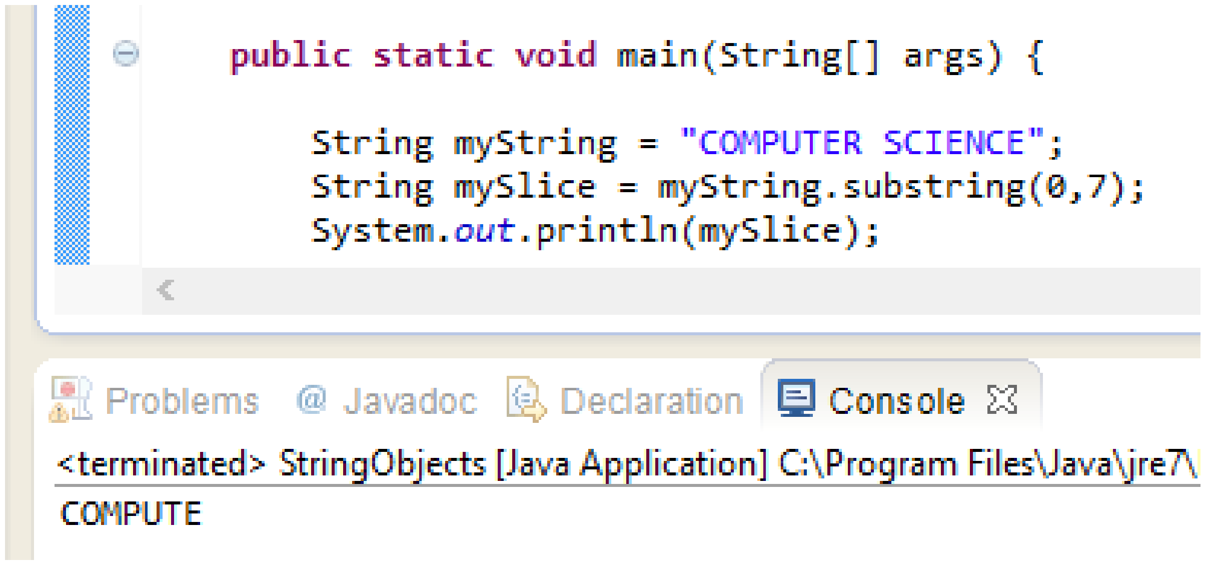
```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\bir
H
E
L
L
O
```

NOTE: It is convenient to start loops that iterate over strings from an index of 0 (**int i = 0**), rather than 1, as this means that the count variable **i** is in alignment with the indices giving the locations of individual characters

The substring Method

- The **substring** method lets you take a “slice” out of a string and store it in a variable

C	O	M	P	U	T	E	R		S	C	I	E	N	C	E	← characters
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	← indices



```
public static void main(String[] args) {

    String myString = "COMPUTER SCIENCE";
    String mySlice = myString.substring(0,7);
    System.out.println(mySlice);
```

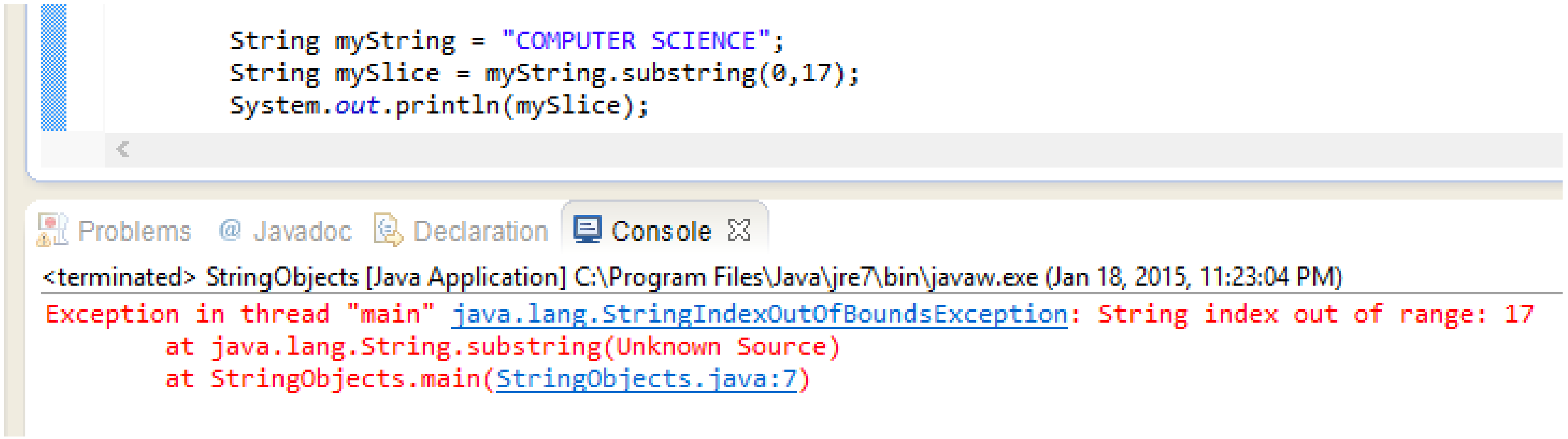
```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\
COMPUTE
```

NOTE: The “**slice**” returned by the substring method is a new (shorter) string that starts from the character at the first index specified, but that ends with the character **ONE CHARACTER BEFORE** the last index specified.

The substring Method, cont.

- What if we tried to use the **substring** method to take too big a “slice”?
- If the indices specified do not exist in your string, this results in an **ERROR**

C	O	M	P	U	T	E	R		S	C	I	E	N	C	E	← characters
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	← indices



```
String myString = "COMPUTER SCIENCE";
String mySlice = myString.substring(0,17);
System.out.println(mySlice);
```

```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 18, 2015, 11:23:04 PM)
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 17
	at java.lang.String.substring(Unknown Source)
	at StringObjects.main(StringObjects.java:7)
```

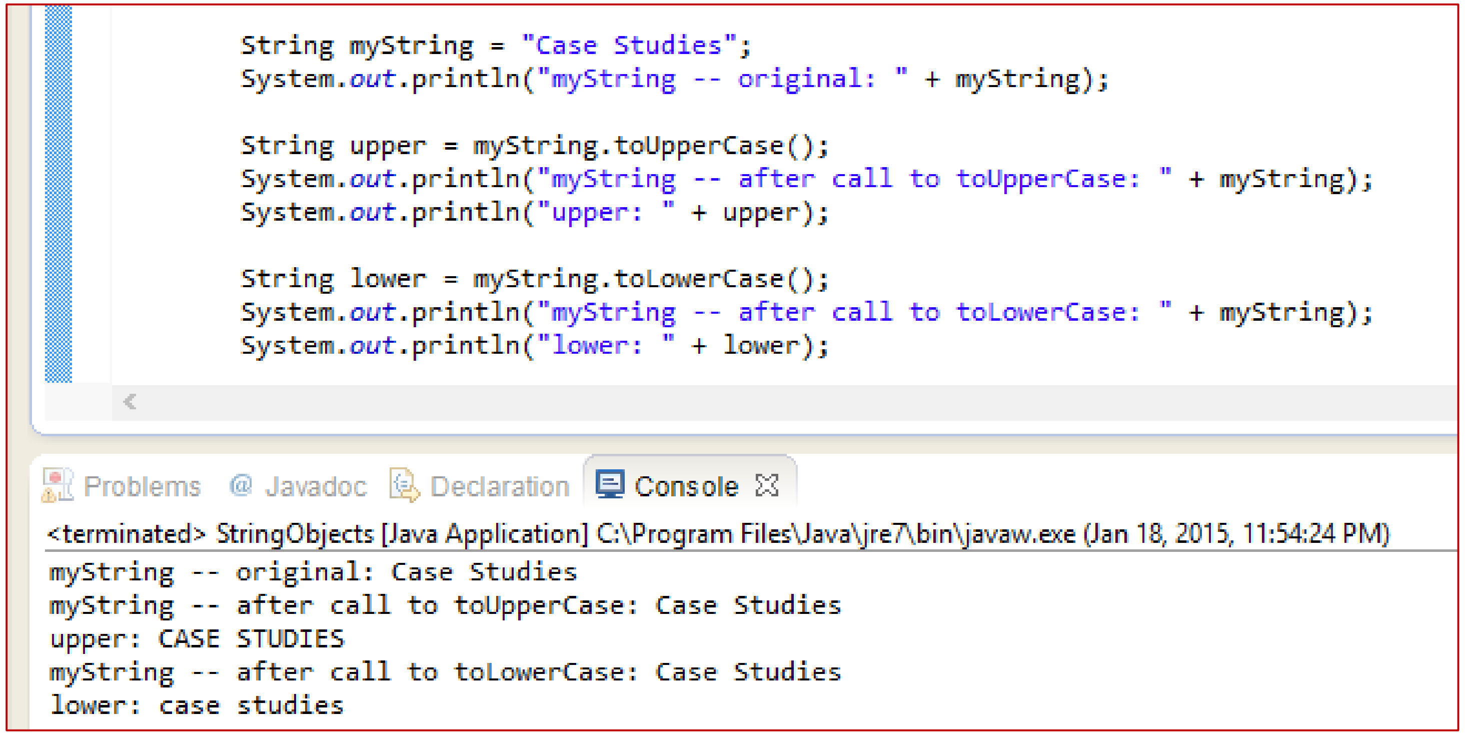
Other String Methods

Method	Description
charAt(index)	Returns the character at a specific index
startsWith(text)	Returns whether or not the string starts with some text
endsWith(text)	Returns whether or not the string ends with some text
length()	Returns the number of characters in the string
indexOf(text)	Returns the index of a particular character of String (or -1 if not present)
Substring(start, stop)	Returns a new string containing the characters from the start index to just before the stop index of the original string
toLowerCase()	Returns a new string with all lowercase letters
toUpperCase()	Returns a new string with all uppercase letters

The Immutability of Strings

- In Java, strings are immutable – once they are constructed, their values **CANNOT BE CHANGED**
- You might think that the existence of such methods as **toLowerCase** and **toUpperCase** violate this rule
  - **THEY DON'T**

- The way these methods work, they CREATE an entirely new string and assign the modified values to this string
- If you want to retain the results of these methods, you need to assign them to a variable
- This variable may have the same name as the original string, but it will NOT be referring to the same object – instead it will be referring to a unique, new object



```
String myString = "Case Studies";
System.out.println("myString -- original: " + myString);

String upper = myString.toUpperCase();
System.out.println("myString -- after call to toUpperCase: " + myString);
System.out.println("upper: " + upper);

String lower = myString.toLowerCase();
System.out.println("myString -- after call to toLowerCase: " + myString);
System.out.println("lower: " + lower);
```

```
<terminated> StringObjects [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 18, 2015, 11:54:24 PM)
myString -- original: Case Studies
myString -- after call to toUpperCase: Case Studies
upper: CASE STUDIES
myString -- after call to toLowerCase: Case Studies
lower: case studies
```

Interactive Programs

- You can write your program so that it pauses at a specified point and waits for input from the user
- Such a program is known as an **interactive program**
- The responses typed by the user are referred to as **console input**

- **How do you tell your program to stop and wait for this user input?**

- When you refer to **System.out**, you are accessing an object in the System class known as the standard output stream, or "**standard out**" for short

- There is a corresponding object in Java for standard input, known as **System.in**, but it is not very easy to use
- Instead, to read console input, we generally rely on **Scanner objects**

Scanner Objects

- Before we can use a Scanner object to read console input, it has to be created
- Most objects have to be explicitly CONSTRUCTED by calling a method known as a **CONSTRUCTOR**

Constructor: A method that creates and initializes an object. Objects in Java programming MUST be constructed before they can be used

- In Java, constructors are called using the keyword **new** followed by the object's type and any necessary parameters
- To construct a new Scanner object, you have to pass information about the SOURCE of the input – where is the Scanner supposed to go to find the input?
- This is where **System.in** comes in handy, as it specifies that we want the Scanner object to read from the console window:

Scanner console = new Scanner(System.in)

Scanner Methods

- Once the Scanner object has been constructed, you can ask it to return a value of a particular type using any of the methods shown below:

Method	Description
next()	Reads and returns the next token as a String
nextDouble()	Reads and returns a double value
nextInt()	Reads and returns an int value
nextLine()	Reads and returns the next line of input as a String

- The first three methods listed above are **token-based** (they read simple elements, not entire lines)
- **Token:** A single element of input (e.g. one word, one number)
- **Whitespace:** spaces, tab characters, and new line characters

Interactive Programs

- You can write your program so that it pauses at a specified point and waits for input from the user
- Such a program is known as an **interactive program**
- The responses typed by the user are referred to as **console input**

- **How do you tell your program to stop and wait for this user input?**

- When you refer to **System.out**, you are accessing an object in the System class known as the standard output stream, or "**standard out**" for short

- There is a corresponding object in Java for standard input, known as **System.in**, but it is not very easy to use
- Instead, to read console input, we generally rely on **Scanner objects**

Tokens

- A Scanner object will divide input into tokens based on the location of **whitespace**
- Punctuation, such as commas, periods, and question marks, will be included in the token along with the word that immediately precedes them
- Examples:

Hello. → Hello (1 token)

How are you? →

$$\left.\begin{array}{l}\text{How}\\ \text{are}\\ \text{you?}\end{array}\right\}\ \text{3 tokens}$$

black-and-white cat →

$$\left.\begin{array}{l}\text{black-and-white}\\ \text{cat}\end{array}\right\}\ \text{2 tokens}$$

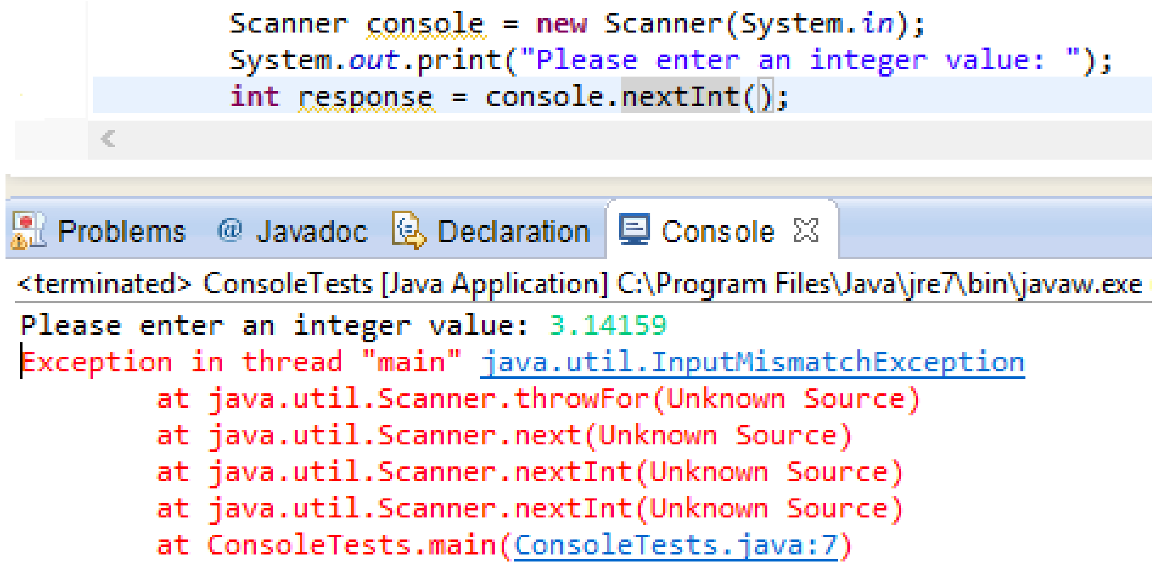
Reading Values with a Scanner Object

- It is possible to read more than one value with a Scanner by making multiple calls:

  double x = console.nextDouble();

  double y = console.nextDouble();

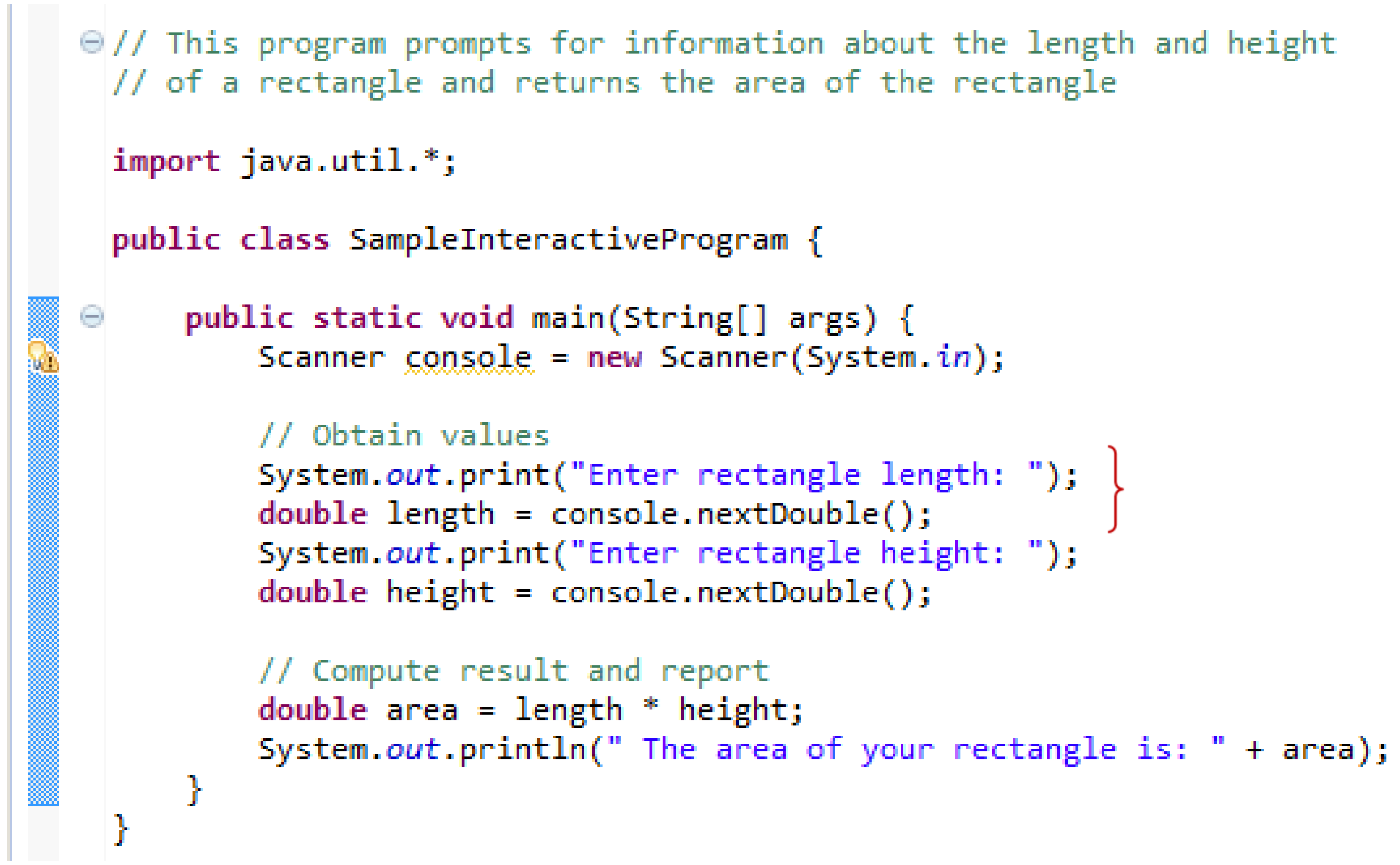
- since there are two calls, the Scanner object will pause until it has received two values as input
- **NOTE:** The Scanner object needs to know what type of object to expect
- If you ask the Scanner object to read an **int**, but you pass it a **double**, you will generate a runtime error



```
Scanner console = new Scanner(System.in);
System.out.print("Please enter an integer value: ");
int response = console.nextInt();
```

```
<terminated> ConsoleTests [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
Please enter an integer value: 3.14159
Exception in thread "main" java.util.InputMismatchException
	at java.util.Scanner.throwFor(Unknown Source)
	at java.util.Scanner.next(Unknown Source)
	at java.util.Scanner.nextInt(Unknown Source)
	at java.util.Scanner.nextInt(Unknown Source)
	at ConsoleTests.main(ConsoleTests.java:7)
```

Sample Interactive Program



```
// This program prompts for information about the length and height
// of a rectangle and returns the area of the rectangle

import java.util.*;

public class SampleInteractiveProgram {

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

        // Obtain values
        System.out.print("Enter rectangle length: ");
        double length = console.nextDouble();
        System.out.print("Enter rectangle height: ");
        double height = console.nextDouble();

        // Compute result and report
        double area = length * height;
        System.out.println(" The area of your rectangle is: " + area);
    }
}
```

- When getting input from the user, we use pairs of statements
- The first statement is called a **PROMPT,** and requests information from the user
- Use a **print()** statement for the prompt so the user input goes on the same line as the prompt
- The second statement calls the nextDouble method of the Scanner object console
- It tells the Scanner object to read a value of type double and to store it in a variable

Concluding Comments

- **Scope of Local Variable:** A local variable can only be used within the block of code (enclosed by curly braces) in which it was created
- **Formal Parameter:** A variable that appears inside parentheses in the header of a method – used to pass information to a method in order to generalize the method's behavior
- **Actual Parameter (argument):** A specific value or expression that appears inside parentheses in a method call
- **Return Statement:** A statement that makes it possible to retrieve the output of a method's statements and store this result in a variable outside the method
- **The Math Class:** A useful class in the Java class libraries that contains methods for performing mathematical calculations
- **Object:** Programming entity that packages state (data) and behavior (methods) together
- **Class:** A category or type of object
- **Scanner Object:** An instance of the Scanner class that lets you read input from the user and create interactive programs