CMSC 132

Intro to Object Oriented Programming II

Ekesh Kumar
Prof. Nelson Padua-Perez • Fall 2019 • University of Maryland
https://www.cs.umd.edu/class/fall2019/cmsc132/

Last Revision: February 5, 2020

Contents

1 Monday, August 26, 2019

Logistics

This is CMSC 131: Object Oriented Programming I. This course is an introduction to Java, and it does not assume any programming knowledge.

- The course homepage is at https://www.cs.umd.edu/class/fall2019/cmsc131-030X/.
- Course announcements are sent out through Piazza.
- Projects are worth 26% of our grade, quizzes and exercises are worth 16%, the three midterms are worth 30%, and the final exam is worth 28%.
- All projects are due at 11 : 30 p.m. on the specified day in the project description. However, you can submit up to 24 hours afterwards with a 12% penalty.
- If you submit a project multiple times, the highest scoring project gets graded.
- All lectures are recorded and posted to Panopto.

Preliminaries

We'll start this course off by introducing some important terminology.

Firstly, we'll briefly discuss two levels of software:

1. **Operating systems** manage the computer's resources; they are typically run as soon as a computer is turned on. Some examples include security-related software, and process management tools.
2. **Applications** are programs that users interact directly with. These are typically explicitly run by the user. This can include word processors, games, music software, or java programs.

Programs are typically executed with the help of **compilers**. Compilers are programs used for translating other programs ("source code") that you write into assembler or machine code. There are many compilers out there, but we only need one. An alternative way to execute programs is through the use of **interpreters**, which take source code as input and execute the source directly. However, these are much slower than compiled programs. **Debuggers** are based on interpreters since they support the step-by-step execution of source code.

2 Wednesday, August 28, 2019

Our First Program

Today, we'll look at our first Java program:

```
public class FirstProgram {
    public static void main(String[] args) {
        System.out.println("Terps are awesome!");
    }
}
```

How does this program work? There are three primary components to this program:

1. The first line uses the keywords "public class" to indicate that everything that follows is part of a new class that is being defined. `FirstProgram` is an identifier that we use to name the class. The entire class definition is contained between an opening curly brace and a closing curly brace.
2. The second component to this program is the `main` method. In the Java programming language, every application is required to have a `main` method, which is declared as "`public static void main(String[] args)`". We will see exactly what each of these keywords mean later on.
3. Finally, the last part of this program consists of the statements to be executed. In this program, we only have one statement: `System.out.println("Hello, World");` This line outputs "Hello, World" followed by a new line on the screen.

Most, but not all, Java statements must end with a semicolon.

Introduction to Variables

A **variable** is a piece of memory that can store a specified type of value. These are similar to the variables that we see in algebra class, like x or y. In Java, there are several different **data types** of variables, for example:

- The `String` type stores text, like "Hello." String values are surrounded by double quotes.
- The `int` type stores integers (whole numbers), like 123 or -123.
- The `float` type stores floating point numbers, with decimals, like 19.99 or -19.99.
- The `char` type stores single characters, like `'a'` or `'B'`. These values are surrounded by single quotes.
- The `boolean` type stores values with two states: either `true` or `false`.

Integer Types

The general procedure to declare a variable in Java is to write the predefined data type (like `int`, `long`, or `short`) followed by an identifier that we are using to refer to that piece of memory. We can subsequently assign values to the variable using a single equal sign (`=`), where the value follows the equal sign.

For example, consider the following Java program:

```
public class FirstProgram {
    public static void main(String[] args) {
        int x;
        x = 20;
        System.out.println(x);
    }
}
```

What's happening here?

- On Line 3, we define an integer variable named `x`. In computer science, the process of defining a variable like so is called a variable **declaration**.
- On Line 4, we assign the value 20 to our previously declared variable `x`. At this point, `x` becomes an alias for 20. The process of assigning a value to a variable for its first time is called **initializing** the variable.
- When we print out the contents variable `x` on Line 5, the number 20 gets printed.

In the above example, we use `x` as the identifier for our variable. However, not all words can be used as variables. Some keywords, like `int`, are **reserved**, so we cannot use them ourselves (for example, we cannot initialize an `int` variable called `int`). The words that appear purple in Eclipse are typically reserved keywords.

It turns out that we can actually make this code even shorter. Java allows us to declare *and* initialize variables at the same time. This is shown below:

```
public class FirstProgram {
    public static void main(String[] args) {
        int x = 20;
        System.out.println(x);
    }
}
```

Instead of declaring the integer variable `x` and assigning it 20 on two different lines, we now declare and initialize it to be 20 at the same time. This is equivalent to the previous example.

Variables are also helpful since they allow us to use pre-defined data type operations. For example, Java supports various arithmetic operations for `int` types, which the following example illustrates:

```
public class FirstProgram {
    public static void main(String args[]) {
        int x = 20;
        int y = 3;
        int a;
        a = x - y;
        System.out.println(a);
        a = x + y;
        System.out.println(a);
        a = x * y;
        System.out.println(a);
        a = x / y;
        System.out.println(a);
    }
```

```
}
```

In the above program, we declare three `int` variables: `x`, `y`, and `a`. At first, we assign `x - y` to `a` and print it, which results in 17 being printed to the screen. Following similar procedures for `a = x + y` and `a = x * y`, we subsequently see 23 and 60 get printed to the screen. Surprisingly, however, the result of assigning `a = x / y` and printing `a` results in 6 getting printed to the screen, rather than, say, 6.6667. Why? Since we are storing the results of dividing the two integers into another integer (which can only store whole numbers), everything after the decimal point gets truncated. It is important to remember that `int` types can only store whole numbers.

We can also shorten the program above by making use of the fact that Java allows us to declare variables with the same type on the same line. Thus, we can move the declarations and initialization of `x`, `y`, and `a` onto the same line as demonstrated below:

```
public class FirstProgram {
    public static void main(String args[]) {
        int x = 20, y = 3, a;
        a = x - y;
        System.out.println(a);
        a = x + y;
        System.out.println(a);
        a = x * y;
        System.out.println(a);
        a = x / y;
        System.out.println(a);
    }
}
```

This code is equivalent to the code that we saw previously.

Next, we'll look at the **modulus** operation, which is defined for integer types in Java. This operation takes the form `x % y`, and it returns the finds the remainder after `x` is divided by `y`.

Example 2.1 (Modulus Operation)
The following examples demonstrate how the modulus operation work:

- The value of `5 % 2` is equal to 1 since dividing 5 by 2 leaves a remainder of 1.
- The value of `100 % 101` is equal to 100 since dividing 100 by 101 leaves a remainder of 100.

Fact 2.2. If `x % y` is equal to 0, then `x` is divisible by `y`.

Floating Point Variables

Previously, we introduced `int` types, which are useful for storing whole numbers. However, what happens if we want to store non-integer values, like `1.3` or `2.5`? In this case, we can use **floating-point data types**. The two primary floating-point data types that we will be using are `float` and `double`. Their usage is very similar to the usage of `int` types.

Consider the following example:

```
public class Example2 {
    public static void main(String args[]) {
```

```
        double salary = 45000.50;
        System.out.println(salary);
    }
}
```

This program compiles successfully (it executes without any errors), and it prints out `45000.50`. This would not have been possible if we were only using `int` data types (we wouldn't be able to store `45000.50` into an `int` type).

We can also perform arithmetic operations with floating point types, like we did with `int` types.

```
public class Example2 {
    public static void main(String args[]) {
        double salary = 45000.50;
        double newSalary = salary * 2;
        System.out.println(newSalary);
    }
}
```

Now, our program prints `90001.0` as we would expect.

Boolean Types

Java supports `boolean` variables, which can only take on the values `true` or `false`. For example, consider the following example:

```
public class Example2 {
    public static void main(String args[]) {
        boolean hungry = false;
        boolean sleepy = true;
    }
}
```

The program above declares two Boolean variables: `hungry` and `sleepy`, which are `false` and `true`, respectively. Why are Boolean variables important? They can be used in conditional statements, which we will introduce later on. For now, it's only important to know that Boolean variables exist.

String Types

A `String` in Java is a sequence of characters. For example, we can write `String name = "John";` to initialize the variable `name` with the contents `John`.

String types have a built-in `+` operation defined for them. We can use the plus (`+`) operator between two strings in order to combine their values. Instead of addition, the `+` acts to **concatenate** (that is, join together) the string sequences. For example, consider the following program:

```
public class Example2 {
    public static void main(String args[]) {
        String s1 = "John", s2 = "Smith";
        System.out.println(s1 + " " + s2);
    }
}
```

On Line 3, we declare the variables `s1` and `s2` with the contents `John` and `Smith`, respectively. On Line 4, `John Smith` gets printed out. Note that we use the string concatenation operation twice.

Comments

In Java, we use **comments** to indicate the programmer's intent. They do not affect the program's execution (they are ignored by the compiler), but they make your code more readable to yourself and others.

There are two types of Java comments:

1. **In-line comments** are one-line comments. They start with `//`, and they terminate as soon as the next line starts.
2. **Multi-line comments** can last for multiple lines. They start with `/*`, and they end with an `*/`.

The following Java program demonstrates how both comments are used:

```
public class Comments {
    public static void main(String args[]) {
        /*
            This is a multi-line comment.
        */

        // This is a single-line comment.

        System.out.println("Hello!");
    }
}
```

This program simply prints out "Hello!". Neither of the two comments affect the execution of the program.

Debugging

There are two primary types of errors that we should be familiar with:

1. **Compile-time errors** are errors that are caught by Eclipse, or your Java compiler. These include **syntax errors** that violate the rules of the language (i.e. `int x <- 5` is an incorrect way to assign 5 to the variable `x`). These also include **type errors**, which come from the misuse of variables.
2. **Run-time errors** are errors that appear during the program's execution. These include semantic errors that obey the rules of the language but do not express the meaning you intended. These can also include division-by-zero errors, or wrong outputs due to mistakes in your programming.

Eclipse helps us identify compile-time errors: it gives us a red flag next to our program when there's an error (and our program won't execute), and it gives us a yellow flag when there's a warning (but we can still execute the program).

3 Friday, August 30, 2019

Last time, we introduced different data types and some of the operations they support. Today, we'll mostly expand on strings.

Immutability of Strings

Before starting a more in-depth discussion of strings, it is important to keep in mind that strings in Java are **immutable**. This means that we cannot change the contents of a string in Java. While it might seem like we're adding on to the end of a string when we're performing string concatenation, this is actually not the case; we are actually creating a new string, and we are combining the two old strings into a new string.

String Concatenation

In the previous lecture, we briefly mentioned that the `+` operator performs addition for integers and floating-point types, but it performs concatenation when working with strings. For example, if we concatenate `von` with `Wienerschnitzel`, we end up with `vonWienerschnitzel`. Note that string concatenation does not automatically add a space character between the strings being concatenated.

When a string is concatenated with a non-string type, the other type is first evaluated and converted into its string representation. For example, if we were to write `(8 * 4) + "degrees"`, then we'd end up with `32degrees`. Likewise, writing `(1 + 2) + "5"` would result in `35`.

This conversion process is also shown through the following code segment:

```
public class Example {
    public static void main(String args[]) {
        System.out.println("Eight times four is: " + 8 * 4);
    }
}
```

The statement that gets printed is `Eight times four is 32`. Note how the integer expression `8 * 4` is evaluated and subsequently converted to a string.

String Comparison

In Java, we can compare numeric values using the `==`, `<`, `<=`, `>`, `>=`, or `!=` operators. For example, the expression `1 == 1` would return true, whereas the expression `2 < 1` would return false. However, strings should not be compared using these operators.

Instead, there are built-in functions that allow us to check whether two strings are equal. If we have a string `s`, and we want to check whether its contents are equal to another string `t`, we can check whether `s.equals(t)` is true. There's also a `.compareTo()` function that compares two strings lexicographically (whichever string would appear first in a dictionary is "greater" than the other string). If we call `s.compareTo(t)` and `s` is less than `t`, then a negative value is returned. If the contents of `s` is equal to `t`, then 0 is returned. Otherwise, a positive value is returned.

Another useful method is `.length()`, which returns the number of characters in the string that the method is invoked on.

Consider the following example:

```
public class StringComparison1 {
    public static void main(String args[]) {
        String katniss = "Katniss";
        String peeta = "Peeta";
        String katniss2 = "Katniss";

        System.out.println(katniss.compareTo(peeta));
        System.out.println(peeta.compareTo(katniss));
        System.out.println(katniss.compareTo(katniss2));

        /* This is how you compare for equality. Do NOT compare using == */
        System.out.println(katniss.equals(katniss2));
    }
}
```

On Lines 3, 4, and 5, we declare and initialize three `String` variables. Both `katniss` and `katniss2` store the contents `Katniss`, whereas `peeta` stores the contents `Peeta`.

On Lines 7, 8, and 9, we print the results of various comparison operations:

- The print statement on Line 7 prints a negative number (it doesn't matter what the number actually is) since `Katniss` is lexicographically smaller than `Peeta` (this means that `Katniss` would appear before `Peeta` in an alphabetically sorted list).
- The print statement on Line 8 prints a positive number (again, it doesn't matter what the number actually is) since `Peeta` is lexicographically larger than `Katniss`.
- Finally, the print statement on Line 9 will print 0 since `Katniss` is lexicographically equal to `Katniss`.

On Line 12, we print the result of an equality check among two equal strings. Consequently, `true` gets printed. It is important to note that we use `.equals()` to compare two strings rather than a double-equal sign.

Here's another example involving string comparisons in which we illustrate why it's important to avoid using `==` and other comparison operators.

```
public class StringComparison2 {
    public static void main(String args[]) {
        String katniss = "Katniss";
        String katniss2 = "Katniss";

        /* Another approach to create strings. */
        String mockingjay = new String("Katniss");


        /* This is the wrong way to compare strings. Use .equals() instead. */
        System.out.println(katniss == katniss2);

        /* This is the wrong way to compare strings. Use .equals() instead. */
        System.out.println(katniss == mockingjay)

    }

}
```

Firstly, we define two `String` variables named `katniss` and `katniss2`. Both of these store the contents `Katniss`. Next, we declare another `String` variable named `mockingjay` that also stores the contents `Katniss`. Note, however, that the way in which we initialized this variable differs from what we've seen so far. For now, we can view this method of declaring and initializing a string as an equivalent alternative to the way that we have been using.

The first print statement prints `true`, which agrees with what we would expect; however, the second print statement surprisingly prints `false`. While we won't go into the details as to why this happens yet, this example illustrates the importance of using the `.equals()` method over the `==` comparison operator.

Introduction to Scanners

In our programs so far, we've been able to provide our users with output by using `System.out.println(..)` statements. However, it's also useful to be able to receive and process user input. One way that this can be done in Java is through the use of a **Scanner**. Scanners are built-in classes provided in the `util` package that allow us to easily obtain the input of various data types. In order to have access to the `Scanner`, we must add the line `import java.util.Scanner;` at the top of our program.

We can subsequently declare a Scanner named `scan` with the line

```
Scanner scan = new Scanner(System.in);
```

The `System.in` that we mention in our initialization of our scanner represents the keyboard.

The following program puts everything together:

```
import java.util.Scanner; // This lets us use scanners.

public class Example3 {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);

    }
}
```

Scanners are useful since they let us interact with the user. For example, if we declare an integer variable named `age`, and we set it equal to `scan.nextInt()`, the variable `age` will be set equal to the next integer that the user enters. This is shown through the following program:

```
import java.util.Scanner; // This lets us use scanners.

public class Example3 {
    public static void main(String args[]) {
        int age;
        Scanner scan = new Scanner(System.in);
        age = scan.nextInt();
        System.out.println("Age is " + age);
        scanner.close();
    }
}
```

This program takes in an integer from the programmer (through the console provided by Eclipse), and it subsequently prints `Age is [age]`, where `[age]` is the integer provided by the user. Finally, we close the

scanner by writing `scanner.close()`, which indicates that we do not want the programmer to be able to provide any more input.

Here's another example:

```
import java.util.Scanner;

/**
 * Shows basic use of scanner
 */
public class Scanner1 {
 public static void main(String args[]) {
    Scanner keyboardInput = new Scanner(System.in);
    int first, second;
    /* Note the use of System.out.print() rather than System.out.println() */
    System.out.print("Enter an integer value: ");

    first = keyboardInput.nextInt();

    System.out.print("Enter another integer value: ");
    second = keyboardInput.nextInt();

    System.out.println("The first number you typed was " + first);
    System.out.println("The second number you typed was " + second);

    System.out.println("Their sum is " + first + second);
    System.out.println("Their product is " + first * second);

    keyboardInput.close();
 }
}
```

This program waits for the user to provide two values. It subsequently tells the user what the two values they provided were, and it finally prints the sum and the product of the two numbers provided. Note that we use `System.out.print()` instead of `System.out.println()` on lines 11 and 15 when we're prompting the user to enter a value so that the prompt is on the same line as where the user inputs their value (the `System.out.println()` prints everything we want to print followed by a new line; `System.out.print()` does not print a new line).

Once again, it is important to close the scanner once we're done with it. This is done in the program above on Line 24.

Here are some more useful tips to keep in mind when using scanners:

- When reading values, scanners will ignore whitespace by default. This means that, in the program above, we can enter any number of leading spaces before our integer, and our program will still work in the same way.
- Scanners cannot deal with incorrect input by default. This means that, in the program above, if we provide a `String` type (e.g. by typing "hello") instead of an `int` type, our program will crash. There are ways that we can handle incorrect input, but we will not worry about it for now.
- Some other operations that scanners support (but we didn't explicitly talk about) are `nextBoolean()`, `nextByte()`, `nextDouble()`, `nextFloat()`, `nextLine()`, and `nextLong()`. However, even this is not an exhuastive list.

Conditional Statements

In Java, statements are typically executed from the top to the bottom. However, we can use **conditional statements** that execute statements only when a certain condition is true in order to modify the control flow.

One way in which this can be done is through the use of "if-statements," which have the general form `if (condition) { statements; }`. The compiler checks whether the `condition` evaluates to true. If so, everything enclosed in curly brackets after the if-statement is executed. Here's an example of if-statements in use:

```
public class SimpleIf {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int value = scanner.nextInt();
        if (value < 0) {
            System.out.println("That was a negative number!");
        }

        System.out.println("The number was " + value);
        scan.close();
    }
}
```

This program reads in an integer into the variable `value` using user input. We subsequently use an if-statement to check whether the value inputted is negative. If the value is negative, then we print "That was a negative number!". Otherwise, we don't print this statement. In either case, we always execute the print statement on Line 10, and we always close the scanner afterwards.

Java also supports if-else statements, which allow us to include a second block of code that is only executed if the condition provided in the if-statement evaluates to false. Here's an example of an if-else statement in use:

```
public class SimpleIf {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int value = scanner.nextInt();
        if (value < 0) {
            System.out.println("That was a negative number!");
        } else {
            System.out.println("That was a positive number!");
        }

        System.out.println("The number was " + value);
        scan.close();
    }
}
```

This program is very similar to the one we just saw; however, we've now added an `else` block. Now, when the user enters a positive number, we print out `That was a positive number!`.

Logical Operators

We just saw how to use if-statements and if-else statements to form more complicated programs. We can also use **logical operators** to form more complex conditions to our conditional statements.

1. In Java, the logical `AND` operator is accessed by using two ampersands, like `&&`. We can use this in an if-statement or an if-else statement when we want more than one condition to be true. For instance, we might write something like,

   ```
   if (temp >= 97 && temp <= 99) { System.out.println("Patient is healthy"); }
   ```

   to check whether a patient's body temperature is in a healthy range.

2. The second logical operator we will cover is the logical `OR` operator, which is accessed in Java with two vertical bars, like `||`. We can use this logical operator when we want at least one of many conditions to be true. For example, we might write,

   ```
   if (grade == "F" || grade == "D) { System.out.println("Ineligible."); }
   ```

   to check whether a student is eligible for a course.

3. The logical `NOT` operator acts on a condition, and it checks whether a condition is false. For example, if we want to check whether the variable `x` is greater than 5, we can equivalently check whether x is *not* less than or equal to 5. Thus, we could write `if (!(x <= 5))` instead of `if (x > 5)`.

4 Wednesday, September 4, 2019

Last time, we introduced logical operators which allow us to create compound Boolean expressions. Today, we'll look at some more instances in which such Boolean expressions can be useful.

More on Conditionals

We've already seen if-statements and if-else statements. Java also supports else-if statements to check if one of many conditions are true. The general syntax for such a statement is

```
if (condition1) { code } else if (condition2) { code } else { code }.
```

Note that we can have arbitrarily many `else if` statements. In an else-if statement, only one code block gets executed. Once a single condition evaluates to "true," we evaluate the corresponding code block and skip all other code blocks (even though more than one condition might evaluate to true).

Here's an example in which such a conditional statement can be helpful:

```
import java.util.Scanner;
public class Example5 {
    public static void main(String args[]) {
        String name;
        String scanner = new Scanner(System.in);
        System.out.print("Enter firstname: ");
        name = scanner.next();
        if (name.equals("Mary")) {
            System.out.println("bff");
        } else if (name.equals("Peter")) {
            System.out.println("wff");
        } else ikf (name.equals("Rose")) {
            System.out.println("classmate");
        } else {
            System.out.println("Do not recognize.");
        }
    }
    scanner.close();
}
```

In this program, the user provides a name. Subsequently, we compare the name to `Mary`. If the name is equal to `Mary`, then we print `bff`. Otherwise, we compare against `Peter`, and so on. If the name does not match with `Mary`, !Peter!, or `Rose`, then we print out `Do not recognize.`

A better way to write the code above would be to use a variable to store the final string that we are printing. This is demonstrated by the example below:

```
import java.util.Scanner;
public class Example5 {
    public static void main(String args[]) {
        String name, ans;
        String scanner = new Scanner(System.in);
        System.out.print("Enter firstname: ");
        name = scanner.next();
```

```
        if (name.equals("Mary")) {
            ans = "bff";
        } else if (name.equals("Peter")) {
            ans = "wff";
        } else ikf (name.equals("Rose")) {
            ans = "classmate";
        } else {
            ans = "Do not recognize.";
        }
        System.out.println(ans);
    }
    scanner.close();
}
```

Note that the program now introduces another string variable named `ans`. This variable stores the contents of what we want to print until we've finished the if-else statements. Finally, we print the answer, which is guaranteed to have a value, after the if-else statements. Why is this better than the previous version? Mainly because it becomes a lot easier to add more conditions to our code in the future. It is also a lot easier to read what the results of each conditions are.

5 Friday, September 6, 2019

Today, we'll discuss the errors associated with uninitialized variables, constants, and compound

Compound Assignment

In Java, we **compound-assignment operators** provide us with shorter syntax to assign the results of arithmetic operations. They perform the operation on the two operands before assigning the result to the first operand. Compound-assignment operators are formed by taking the operator and placing an equals sign immediately after it.

For example, suppose we wanted to increment the variable `x` by 10. One way to do this would be to write `x = x + 10`. However, with compound-assignment operators, we can shorthandedly write `x += 10` to accomplish the same thing. Similarly, we can write `x *= 10` to multiply the old value of `x` by 10 and store the new result in `x`, or we can write `x -= 5` to subtract 5 from the old value of `x` and store the new result in `x`.

Note that compound-assignment operators are not limited to numeric types. Thus, if we have two strings `s1` and `s2`, it would be perfectly fine to write `s1 += s2` to set `s1` to the concatenation of the old value of `s1` with the contents of `s2`.

Uninitialized Variables

Let's look at the following code segment:

```
import java.util.Scanner;

public class Initialization1 {

   public static void main(String[] args) {
      int x;

      Scanner scanner = new Scanner(System.in);
      String s = scanner.next();

      if (s.equals("dog")) {
         x = 10;
      }
      System.out.println("x is " + x);

      scanner.close();
   }
}
```

As we've already seen, Line 6 declares a variable `x`. However, what would happen if we tried to print the variable `x` without initializing it? Surprisingly, we get an error, and our code does not compile.

Due to similar reasons, the code provided above as is does not compile either. Why? Because the value of `x` is *only* assigned a value if the user inputs `dog`. What happens if the user inputs `cat`? Then the conditional statement on Line 12 does not get executed, and `x` is left uninitialized. Subsequently, we try to print `x` on line 14; however, this results in an error due to reasons described previously.

Here's another example of code that does not compile:

```
import java.util.Scanner;

public class Initialization3 {

   public static void main(String[] args) {
      int x;
      boolean foundDog = false; // this is an example of a "flag"
      Scanner scanner = new Scanner(System.in);
      String s = scanner.next();

      if (s.equals("dog")) {
         x = 10;
         foundDog = true;
      }

      if (!foundDog) {
         x = 12;
      }

      System.out.println("x is " + x);
      scanner.close();
   }
}
```

In this code, if the user inputs `dog`, then the value of `x` ends up being 10. On the other hand, if the user *doesn't* input `dog`, then the value of `x` ends up being 12. Thus, no matter what the execution path is, the variable `x` always ends up with a value. However, this program still does not compile since, at compile-time, it is not clear that the variable `x` will eventually take on a value.

Constants

we can use the **final** keyword to define variables whose value cannot change once it has been assigned. Such a variable is typically called a **constant**. Constants are useful since they can make your program more easily read and understood by others. By convention, we typically make the names of constant variables in all uppercase letters.

Here's an example of constants in use:

```
public class Example1 {
    public static void main(String args[]) {
        final double PI = 3.14;
        // This line doesn't compile:
        // PI = 4;
    }
}
```

In the above code segment, we declare a final variable `PI`, which takes on the value 3.14. Note that the `final` keyword comes before the type of the variable (i.e. `final` comes before `double` here). Since the variable is a constant, it would be an error to try and change this variable. Thus, the commented line `PI = 4` would raise a compile-time error if it were uncommented.

While Loops

In computer programming, a **loop** defines a sequence of instructions to be continually repeated until a certain condition is reached. Loops are helpful since we sometimes want to perform an action several times (and using a loop reduces the amount of code we need to write), or we sometimes want to perform an action until some presently unknown condition holds.

Java supports a few types of loops. The first one that we will cover is the **while loop**, which has the following general form:

```
while (condition) {
    statements;
}
```

At the start of the while-loop, we check whether `condition` is true. If so, then we execute everything in the loop block, and we go back to the top of the loop. Otherwise, we skip all of the statements that would have been executed, and we continue executing statements outside of the while-loop.

Here is an illustrative example of the while-loop in use:

```
public class Example2 {
    public static void main(String args[]) {
        int i, limit = 3;
        i = 1;
        while (i <= limit) {
            System.out.println(i);
            i += 2;
        }
    }
}
```

What does this code do?

- First, we declare an integer variable `i` to be 1, and we declare an integer `limit` variable to be 3.
- Next, we reach Line 5. Since the condition `i <= limit` holds (1 is less than or equal to 3), we enter the while loop, and we execute all of the statements inside of the while loop.
- Inside of the while loop, we print out the current value of `i`, which happens to be 1. Subsequently, we increment the variable `i` by 2, and we go back to the top of the while condition.
- Once again, we check whether the condition `i <= limit` holds. This condition holds again (since 3 is less than or equal to 3), so we print `3`, increment `i` by 2 (so `i` is now 5), and go back up to the top of the while-loop.
- At this point, the condition `i <= limit` evaluates to false. Thus, we do not execute the statements inside of the while-loop for a third time, and our program is complete.

It is important to be careful of **infinite loops**. These are loops that never terminate. If we were to remove the `i += 2` statement inside of our while-loop in the program above, we would have an example of an infinite loop (the condition `i <= limit` will never become true).

Suppose we want to print all even numbers between 1 and 100. We can do this very easily now with the help of a loop:

```java
public class Example2 {
    public static void main(String args[]) {
        int i = 1, limit = 100;
        while (i <= limit) {
            if (i % 2 == 0) {
                System.out.println(i);
            }
            i++; /* This is new. */
        }
    }
}
```

In this program, we loop from 1 up to 100. On each iteration of the loop, we check if `i` is divisible by 2. If so, we print `i`. Otherwise, we do nothing. Note that instead of writing `i = i + 1` or `i += 1` as we might be used to, we can instead write `i++`. The expression `i++` increments `i` by one and returns the old value of `i`. This is known as **post-incrementation**. On the other hand, `++i` increments `i` by one and returns the new value of `i`. This is known as **pre-incrementation**.

6 Monday, September 9, 2019

Last time, we introduced while-loops. Today, we'll introduce do-while loops, and the scope of variables.

Do-While Loops

The second type of loop that we'll cover is called the **do-while loop**. The do-while loop is very similar to the while-loop; however, the key difference is that the do-while loop always executes its body at least once. This happens because the condition that checks whether we should perform the next iteration happens at the end of the loop.

Here's an example of the do-while loop in action:

```
public class SimpleDoWhile {
    public static void main(String args[]) {
        int currValue = 1;
        do {
            System.out.println(currValue + " ");
            currValue = currValue + 1;
        } while (currValue <= 5);
        System.out.println("currValue after the loop is " + currValue);
    }
}
```

- At first, we declare a variable `currValue`, which we initialize to 1.
- The `do-while` loop gets executed until `currValue > 5` happens *at the end of the loop*. This means that we print out $1, 2, 3, 4, 5$, and 6 inside of the loop.
- The print statement on Line 8 indicates that the value of `currValue` after the do-while loop is `6`.

Here's another example where do-while loops are useful:

```
public class AskAge {
    public static void main(String args[]) {
        int age;
        Scanner scan = new Scanner(System.in);

        do {
            System.out.print("Enter age: ");
            age = scanner.nextInt();
            if (age < 0) {
                System.out.println("Invalid value!");
            }
        } while (age < 0);
    }
}
```

In this example, we declare a variable `age`, and we scan the user's input for a value of `age` inside of a do-while loop. This lets us keep on reading values from the user's input until we receive a positive value for the age. By using a do-while loop, we're able to keep on reading input until some condition (a positive number is provided) is satisfied. Note how providing a negative number for `age` will make the contents of the loop execute again, whereas providing a positive number for `age` will take us out of the loop. Note that a do-while loop is preferred over a while-loop here since we always want to prompt the user to enter their age at least once.

Variables, Blocks, and Scopes

Variables can be declared anywhere in a Java program. So far, we've only really seen them declared at the top of the `main` function. When are the declarations active? Right after they are executed, and only inside of the block in which they are declared.

There are a set of **scope rules** that formalize which variable declarations are active and when.

- **Global variables** can be accessed from anywhere in a program. We haven't seen these yet.
- **Local variables** are variables whose scope is a block. If we declare a variable at the top of the main method, then that variable goes "out-of-scope" (and hence cannot be used) outside of the main method. Similarly, if we declare a variable inside of the curly brackets of a while-loop, this variable would be inaccessible outside of the closing curly bracket of the while-loop.

Here's a program that makes makes the second bullet point more clear:

```
public class Example {
    public static void main(String args[]) {
        int i = 1;
        while (i <= 5) {
            int x = 2;
            i++;
        }
        /* x cannot be accessed out here. */
    }
}
```

7 Wednesday, September 11, 2019

So far, we've seen both while-loops and do-while loops. Today, we'll introduce a third type of loop.

For Loops

In a **for-loop**, a counter is typically set, and the condition is tested before each body execution. The updated is performed at the end of each iteration. The general form of a for-loop is as follows:

```
for(initialization; condition; update) { statements }
```

Here's a simple example of a program that uses a for-loop:

```
public class ForExample {
    public static void main(String args[]) {
        int i, limit = 3;
        System.out.println("First loop: ");
        i = 1;
        while (i <= limit) {
            System.out.println(i);
            i++;
        }
        System.out.println("Second loop: ");
        for (i = 1; i <= limit; i++) {
            System.out.println(i);
        }
    }

}
```

This example prints the numbers between 1 and 3 inclusive twice using a while-loop and then a for-loop so that we can easily compare how the two loops are structured. Here are the key things to keep in mind:

- The `initialization` portion of the for-loop is only executed once. This means that we only set the variable `i` to 1 once at the start of the for-loop.
- At the beginning of each iteration of the for-loop, we check whether the condition `i <= limit` holds. This is similar to the while-loop in which we also check whether a condition holds at the beginning of each iteration.
- If the condition holds, we execute the body of the for-loop. Finally, we perform the operations stated in the `update` section of the for-loop. In this example, the `update` operation is to increment `i` by one.

We can also declare variables that we've never used before in the `initialization` portion of our for-loop. This is demonstrated through the example below:

```
public class ForExample {
    public static void main(String args[]) {
        for (int k = 1; k <= 3; k++) {
            System.out.println(k);
        }
    }
}
```

It's also valid to leave one of the three components (initialization, condition, or update) of a for-loop blank. However, it's up to the programmer to be sure that the loop is not an infinite loop. Here's an example of a for-loop in which the "update" portion is left blank:

```
public class ForExample {
    public static void main(String args[]) {
        for (int k = 1; k <= 3; )
            System.out.println(k);
            k++;
        }
    }
}
```

Note that this program now increments the loop variable `k` at the bottom of the for-loop. Without this incrementing statement, we would have an infinite loop.

Nested Loops

Now that we've introduced the three types of loops that we will be using in this class, we will now demonstrate some interesting ways in which we can use loops inside of loops to perform various tasks. When we place a loop inside of a loop, we say that our loops are **nested**.

Here's a first example that we can look at:

```
public class NestedWhile {
   public static void main(String[] args) {
      int maxRows = 3, maxCols = 4, row;

      row = 1;
      while (row < maxRows) {
         int col = 1;

         while (col < maxCols) {
            System.out.println("Row: " + row + " Col: " + col);
            col++;
         }
         System.out.println();

         row++; /* Next row */
      }
   }
}
```

When we have two nested loops, every iteration of the outermost loop consists of a full execution of the innermost loop. Thus, for each row starting from `row = 1` up to `row = 2`, we will initialize `col` to `1`, and iterate up to `col = 3`. The inner-most `System.out.println` statement will thus print out the following statements:

```
Row: 1 Col: 1
Row: 1 Col: 2
Row: 1 Col: 3

Row: 2 Col: 1
Row: 2 Col: 2
```

```
Row: 2 Col: 3
```

It is sometimes useful to trace out the values of each variable during an iteration to figure out what a loop is doing.

Now let's rewrite this code but with for-loops:

```
public class NestedFor {
   public static void main(String[] args) {
      int maxRow = 9, maxCol = 9;

      for (int row = 1; row <= maxRow; row++) {
         for (int col = 1; col <= maxCol; col++) {
            System.out.print("Row: " + row + " Col: " + col);
         }
         System.out.println();
      }
   }
}
```

As we can see, this is a much more compact way of doing the same thing that the previous code segment was doing. Now, we're declaring our loop variables as a part of our for-loop, and the inside of our nested for-loop only contains one line.

Using nested loops, we can also draw nice-looking designs. For example, the following program creates a 3×4 grid in which we print an asterisk if the current row number is even and a dollar sign otherwise:

```
public class NestedFor {
   public static void main(String[] args) {
      int maxRow = 4, maxCol = 5;

      for (int row = 1; row <= maxRow; row++) {
         for (int col = 1; col <= maxCol; col++) {
            if (row % 2 == 0) {
                System.out.print("*");
            } else {
                System.out.print("$")
            }
         }
         System.out.println();
      }
   }
}
```

A new line is printed every time a row is completed. The resulting output is shown below:

```
$$$$
****
$$$$
****
```

We can also create a triangle by stopping the innermost for-loop when it reaches the value of `row`. This way, each new line prints one more character than the previous line:

```
public class NestedFor {
```

```
  public static void main(String[] args) {
     int maxRow = 4, maxCol = 5;

     for (int row = 1; row <= maxRow; row++) {
        for (int col = 1; col <= row; col++) { /* Note the change. */
           if (row % 2 == 0) {
               System.out.print("*");
           } else {
               System.out.print("$")
           }
        }
        System.out.println();
     }
  }
}
```

It is a little bit harder to figure out what's going on this program, so it might be helpful to trace out what gets printed for the first few iterations of the loop (write down the values of `row` and `col`, and trace what happens).

The new output is as follows:

```
$
**
$$$
****
```

8 Friday, September 13, 2019

Today, we'll continue talking about nested loops.

Scope Error

In a previous lecture, we talked about how variables are only valid in the scope in which they are defined. Today, we'll briefly look at an example that might seem correct but is actually invalid for that exact reason:

```
import java.util.Scanner;
public class ScopeError {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);
        do {
            System.out.print("Enter an integer from 1 to 10: ");
            int answer = scanner.nextInt();
        } while (answer < 1 || answer > 10);
    System.out.println("Good job");
    scanner.close();
    }
}
```

We want this program to read in an integer between 1 and 10, inclusive. The program should keep on prompting the user to enter an integer between 1 and 10 until valid input is provided. However, the program as shown above does not compile. Why? Because the while-loop's terminating condition on Line 8 depends on the variable `answer` which is only defined inside of the scope of the while-loop. The variable must be visible in the expression in which it is being used. In order to fix this, we can declare `answer` outside of the `do { ... }` block, and we can just update its value (instead of declaring the variable) on Line 7.

Expressions

Now that we are more familiar with the Java programming language, we will now mention what an **expression** is. In Java, an expression is any statement that produces a value. For example, the variable `x` by itself is a value (it can be evaluated to whatever contents `x` is storing). Similarly, `x + 1 - y` and `x == y && z == 0` are both expressions (the former expression evaluates to a numeric type, whereas the latter expression evaluates to a Boolean type).

Expressions have values of a specific type (e.g. `int, boolean`, etc). They can be assigned to variables, appear inside of other expressions, and so on. A statement that *doesn't* evaluate to a value is **not** an expression.

Consider the following code segment:

```
public class Example3 {
    public static void main(String args[]) {
        int x = 20, y;
        x = 1;
        y = (x = 10) + 3; /* This may seem strange. */
    }
}
```

It might seem strange to be able to to assign `y` the expression `(x = 10) + 3`. However, this is valid. The reason why this is valid is because the right-hand side of the equals sign can be evaluated to an `int` type that

we can assign to `y`. This happens because the assignment operator `=` assigns the value on the right-hand side of the equality to the variable on the left-hand side of the equality, and it subsequently assigns the entire expression the newly assigned value.

Thus, the expression `y = (x = 10) + 3` is evaluated as follows:

- Firstly, we assign 10 to the variable `x`.
- Next, we assign the value 10 to the entire expression `(x = 10)`.
- Next, we evaluate the right-hand side of the equality `y = (x = 10) + 3`. This expression evaluates to 13.
- Finally, we assign the evaluated expression to the variable `y`. At this point, the variable `y` stores the value 13.

The resulting value of `x` is 10, and the resulting value of `y` is 13.

While it's hard to read code written like this, this example illustrates how Java evaluates expressions.

Here's another example:

```
public class Example3 {
    public static void main(String args[]) {
        int m = 1, x;
        x = m++;
        System.out.println(x);
        System.out.println(m);
    }
}
```

Recall that `m++` increments the value of `m` and returns the old value of `m`. Here's what's happening in the code above:

- Firstly, we assign the value 1 to the variable `m`.
- Next, we set the value of `m++` to the variable `x`. While doing so, we increment the value of `m`, and the expression `m++` returns the old value of `m` (which was 1). Thus, the variable `x` is assigned `1`.
- When we print the two variables, we find that `x` is equal to 1 and `m` is equal to 2. Note that m's value has increased due to the post-incrementation that took place.

On the other hand, if we were to change Line 4 from `x = m++` to `x = ++m`, the variable `x` would store the value `2` instead of `1` (the preincrementation would return the new value of `m` instead of the old value).

Introduction to Methods

A **method** is a set of code that is grouped together and referred to by a name that can be called (**invoked**) at any point in a program by simply utilizing the method's name. Methods are defined inside of classes, and they are useful since they allow us to reuse portions of code without having to retype the code. So far, we've been using the `main` method, but now we want to create our own methods.

There are two types of methods that we'll talk about:

- **Static methods** are methods that can be called without an object. We just write the name of the class, followed by a period, followed by the name of the method.
- **Non-static methods** are methods that require an object. For example, when we want to use the Scanner's methods, we first instantiate a Scanner with the `new` keyword.

In order to declare a method, we must first write a method header, which takes the following form:

```
[method visiblity] [static] [return type] [method name](args) { ... }
```

The "method visibility" portion of the method header can be either `public` or `private`, depending on where we want our method to be accessible from. For now, we set this to `public`. Next, we can optionally include the keyword `static` to indicate that our method is a static method. Next, we'll add the return type of the method (if the method doesn't return anything, this should be `void`). Finally, we need to give the method a name and specify what arguments it takes in, if any.

Here's an example:

```
public class Rectangle {
    public static void printRectangle() {
        int row, int col, int maxRows = 3, maxCols = 4;

        for (row = 1; row <= maxRows; row++) {
            for (col = 1; col <= maxCols; col++) {
                if (col % 2 == 0) {
                    System.out.print("*");
                } else {
                    System.out.print("$");
                }
            }
            System.out.println();
        }
    }

    public static void main(String args[]) {
        Rectangle.printRectangle(); // Method call.
    }
}
```

Note how we've defined a method called `printRectangle`. Now, whenever we want to draw our rectangle, we can just call `Rectangle.printRectangle()`. This is much more convenient than reusing the same code over-and-over again.

Can we do better? Yes. We can customize our method even more by adding **parameters** that allow our user to specify the dimensions of the rectangle being printed. This is done by changing the code from above to what is shown below:

```
public class Rectangle {
    public static void printRectangle(int maxRows, int maxCols) {
        int row, int col;

        for (row = 1; row <= maxRows; row++) {
            for (col = 1; col <= maxCols; col++) {
                if (col % 2 == 0) {
                    System.out.print("*");
```

```
            } else {
                System.out.print("$");
            }
        }
        System.out.println();
    }
}

public static void main(String args[]) {
    Rectangle.printRectangle(3, 4); // Method call.
}
}
```

Now, when we call our method, we must specify the maximum number of rows and the maximum number of columns. This allows us to easily draw rectangles with different dimensions without having to write more code. For example, calling `Rectangle.printRectangle(3, 4)` results in a 3×4 rectangle, whereas `Rectangle.printRectangle(5, 5)` would print a 5×5 rectangle.

This method can further be customized by providing the symbols we're printing as parameters.

9 Monday, September 16, 2019

Today, we'll continue our discussion on methods.

More on Methods

Recall that there are two types of methods that we can implement: non-static methods (which require an object, like `Scanner`), or static methods (which don't require objects).

When we call a method, the control flow of the program goes to the method. The contents of the methods are executed, and we subsequently return to the `main` method, where we continue execution from where we left off. When a method doesn't return a value, we say that method is `void`. Moreover, this `void` return type must be indicated in the method's header. But, what does this mean? This means that a call to the method cannot be *evaluated*. That is, we cannot set a variable equal to the result of the method call.

Methods can also take **parameters**, which are values that we pass in when we are calling the function. Parameters allow us to "customize" how a method gets executed. Moreover, modifying these parameters inside of the method does not result in the values being changed outside of the method (the Java compiler makes copies of the variables before the method can use the variables).

Consider the following Java program, which has a few methods:

```
import java.util.Scanner;

public class MethodsIntro {

   /* Does not return a value and has no parameters */
   public static void printHeader() {
      System.out.println("****************************");
      System.out.println("****************************");
   }

   /* Returns a value (boolean) and has one parameter */
   public static boolean isValid(int value) {
      if (value >= 1 && value <= 100) {
         return true;
      } else {
         return false;
      }
   }

   /* Does not return a value (void) and has two parameters */
   public static void printRectangle(int width, int height, char symbol) {
      for (int row = 1; row <= width; row++) {
         for (int col = 1; col <= height; col++) {
            System.out.print(symbol);
         }
         System.out.println();
      }
   }

   public static void main(String[] args) {
      int width, height;
```

```
      Scanner scanner = new Scanner(System.in);

      System.out.print("Enter width: ");
      width = scanner.nextInt();

      System.out.print("Enter height: ");
      height = scanner.nextInt();

      if (isValid(width) && isValid(height)) {
         printHeader();
         printRectangle(width, height, '*');
      } else {
         System.out.println("Invalid values");
      }

      scanner.close();
   }
}
```

The first method, `printHeader()`, does not take in any parameters, and it does not return anything either. The method simply prints two lines full of asterisks, which serves as a header.

The second method, `isValid(int value)` takes in one integer parameter named `value`. The function subsequently returns true if `value` is between 1 and 100, inclusive, and the method returns false otherwise. This return statement indicates what the method call will evaluate to.

What happens in this program? A summary is provided below:

- In the main method, we declare two integer variables `width` and `height`. We prompt the user to enter a width and a height, and these values are read in.
- Next, we check if the width and height entered are valid (i.e. they should be in-between 1 and 100, inclusive). If so, then we print our header, and we print a rectangle full of asterisks. Otherwise, we print "Invalid Values." In either case, the scanner closes, and the program terminates afterwards.

Note how methods allow us to easily customize what we are drawing our rectangle with as well as the dimensions of the rectangle being drawn.

Precedence

Precedence rules answer how to evaluate expressions. More precisely, higher-precedence operators are evaluated first (e.g. multiplication before addition), and lower-precedence operators are evaluated afterwards. Below is a list of common operations and operators used in Java in order from highest precedence to lowest precedence:

- Parentheses
- Unary operators (operators that act on a single variable): `++x, x++, x--, !x`.
- Multiplication, division, and the modulus operator
- Addition and subtraction
- Comparison operators, like `<, >, >=, <=`

- Equality checkers, like `==` and `!=`.
- Logical `AND` operator
- Logical `OR` operator
- Assignment operators, like `=`, `+=`, `-=`, etc.

This list isn't exhaustive, and it isn't necessary to memorize this list either.

Short-Circuiting

As soon as Java knows that an entire expression is false, or an entire expression is true, it quites evaluating the expression it is currently looking at. For example, suppose we have a variable named `x` that is equal to 4. If we begin looking at the condition

`if (x > 10 && y == 0) { ... }`,

Java will not even look at the `y == 0` portion of the condition since, right after looking at the `x > 10` condition, it can immediately conclude that there is no way that the entire expression will evaluate to `true` (if we were using the logical `OR` operator instead of the logical `AND` operator, we would still need to look at the remainder of the Boolean expression).

This can have some profound consequences. For example, consider the following code segment:

```
public class Example {
    public static void main(String args[]) {
        int x = 0, y = 1;
        if ((y > 1) && (++x == 0)) {
            --y;
        }
        System.out.println(x);
    }
}
```

Due to short-circuiting, the output of this program is 0. Why? The expression `y > 1` is initially false. Thus, we don't even look at the condition `(++x == 0)`; this statement is not executed.

This process of exiting a Boolean expression early is known as **short-circuiting**.

10 Wednesday, September 18, 2019

Casting Numeric Types

In Java, we can perform **casting** to make a variable of one type to behave as a variable of another type. In order to cast a value or a variable, we can place the type that we wish to cast to in parentheses next to the value or variable that we are casting.

Recall that an assignment like `int x = 7.2` is invalid and will result in a compile-time error. This results in an error because `7.2` is a floating-point value, and we are trying to assign it to an integer variable. We can fix this issue by casting. Writing `int x = (int) 7.2` tells the Java compiler to treat `7.2` as an integer type (so everything after the decimal point gets truncated) and assign the resulting value to `x`. After this statement is executed, `x` will store the value 7.

Here's some code that makes what we just discussed more clear:

```
import java.util.Scanner;

public class ShortCircuiting {
    public static void main(String args[]) {
        double y = 7.2;
        int x = (int) y; // this doesn't give an error!
        System.out.println(y); // prints 7.2
        System.out.println(x); // prints 7.
    }
}
```

Note that casting the variable `y`, which stores the value 7.2, does not change the value of `y`.

When we're adding integers with floats, (e.g. `x += y`, where `y` is an integer and `x` is a float), Java performs implicit casts as necessary.

Floating-Point Calculations

In computers, there are some values that cannot be represented exactly. This issue isn't specific to Java — there are no computers that can represent real numbers exactly. For example, consider the fraction $1/3 \approx 0.333$. This number has an infinitely long decimal representation. But since we have finite space, this leads to complications when we're storing these numbers. Ultimately, some of these bits must get cut off, which can cause some small imprecisions when dealing with floating-point values.

Since many floating-point calculations involve approximations rather than exact results, it is usually *unreasonable* to test the result of a floating-point calculation for equality with another value. Instead, we typically define some small constant `EPSILON`, and we say that two values are equal if they are within `EPSILON` absolute distance of each other.

Here's an example:

```
public class FloatingCalculations {
    private final static double EPSILON = 0.0000000001;

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

```
        double difference = 3.9 - 3.8;
        System.out.println("3.9 - 3.8 = " + difference);
        if ((Math.abs(difference) - 0.10) < EPSILON) {
            System.out.println("Not exactly 0.10, but we will accept it.");
        } else {
            System.out.println("They are different.");
        }
    }
}
```

In the program above, we want to check whether the difference between 3.9 and 3.8 is equal to 0.10. When we print the variable `difference`, it turns out that we get a value that's different from 0.10. This is a result of floating-point imprecision. However, the conditional on Line 7 holds true (the difference is within 0.0000000001 of 0.1), so we end up printing `Not exactly 0.10, but we will accept it.`

Some more details regarding floating-point imprecision are provided here: https://floating-point-gui.de/.

11 Friday, September 20, 2019

Exam 1 is today.

12 Monday, September 23, 2019

Lambda Expressions

Suppose we have the following interface that provides us with the prototype of a `compute` function:

```
package lambda;

public interface Task {
   public int compute(int x);
}
```

One way to use this interface would be to implement it, which would make the compiler force the class to provide a declaration of the function `compute`. This is a great idea if we're planning to create many instances of the class, but what if we only plan to instantiate this class once? As we've learned, we can solve this problem with anonymous classes. An alternative way to solve this problem is with **lambda expressions**, which permit us to write anonymous functions in a simpler way. This means that lambda expressions aren't "necessary" in the sense that there isn't something that can only be done with lambda expressions, but they can be very convenient to use.

The general syntax for a lambda expression is `(type1 parameter1, type2 parameter2, ...) -> expression` or `(type1 parameter1, type2 parameter2, ...) -> {statements;}`. The first form is used when we're returning a value, and the second form is used otherwise.

The following code demonstrates how we can use lambda expressions in the context of our `Task` interface.

```
package lambda;

public class LambdaBasics {
   public static void main(String[] args) {
      Task anonymousVersion = new Task() {
         public int compute(int x) {
            return x + x;
         }
      };
      System.out.println(anonymousVersion.compute(10));

      Task lambdav1 = (int x) -> {
         return x + x;
      };
      System.out.println(lambdav1.compute(10));

      Task lambdav2 = (x) -> x + x;
      System.out.println(lambdav2.compute(10));

      Task lambdav3 = x -> x + x;
      System.out.println(lambdav3.compute(10));

      AnotherTask lambdav4 = () -> 10;
      System.out.println(lambdav4.analyze());

      Processor lambdav5 = (int x, float y) -> x * y;
      System.out.println(lambdav5.increase(10, 5));

      pdata((x, y) -> x * y);
   }
```

```
  public static void pdata(Processor p) {
     System.out.println(p.increase(10, 60));
  }
}
```

The above code firstly declares an anonymous inner class that implements the `Task` interface. This is done by providing a declaration of the `compute` function. Subsequently, we use a lambda expression to declare `lambdav1` using the first "general syntax" form that was specified earlier. Next, we declare a second lambda expression named `lambdav2` using the second "general syntax" form specified earlier. We can note that the key difference between these two declarations is that the first form contains a single expression after the arrowhead, while the second form can contain several statements after the arrowhead (if there's a single statement, like above, the compiler can infer that it needs to return it). Why isn't the `int` type specified in the second declaration? It turns out that our compiler can infer the type to be an `int` based on the context in which the lambda expression is used in.

We can simplify `lambdav2` even more. The lambda expression `lambdav3` drops the parentheses around the parameter of `lambdav2`; parentheses are not necessary when there is only a single parameter.

Now let's introduce another interface named `AnotherTask`:

```
package lambda;

public interface AnotherTask {
  public int analyze();
}
```

This interface provides us with the prototype for an `analyze()` function that returns an integer and doesn't take in any parameters.

As shown in the declaration of `lambda4` in our driver code, we can declare a lambda expression for `analyze` as well by using an empty open-closing parentheses pair, which indicates that there are no parameters.

Finally, let's look at one last interface named `Processor`:

```
package lambda;

public interface Processor {
  public float increase(int x, float y);
}
```

In our driver code, we've declared a lambda expression named `lambdav5` that can be used for the `increase` function.

Finally, in our driver, we show an application of lambda expressions. The `pdata` function takes in a `Processor` type, and it prints out the return value of the `p.increase(10, 60)` call. Instead of creating an instance of a `Processor` and subsequently passing that object into the function call, we can instead use a lambda expression (as done in the driver) in order to avoid writing so much code. This is also helpful since we don't need to declare one-time-use variables.

Revisiting Shallow Copies

Recall the following `Mouse` class that we used to discuss shallow copying:

```
package cloning;
```

```
public class Mouse implements Cloneable {
   private String type;
   private int xPos, yPos;

   public Mouse(String type) {
      this.type = type;
      xPos = yPos = 0;
   }

   public void moveMouse(int xPos, int yPos) {
      this.xPos = xPos;
      this.yPos = yPos;
   }

   public String toString() {
      return type + "-> xPos: " + xPos + ", yPos: " + yPos;
   }

   /* Notice the return type */
   @Override
   public Mouse clone() {
      Mouse obj = null;

      try {
         obj = (Mouse) super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return obj;
   }
}
```

When we create a `Mouse` object, and we use the `.clone()` method to create a copy of that mouse, a shallow copy is created. This means that primitive types are copied, while only the memory addresses of non-primitive types are copied.

In our `Mouse` class, we call `super.clone()` inside of our `.clone()` method. This call returns an `Object` type, but we cast it as a `Mouse` and return a `Mouse` type. Note that this still counts as function overriding (even though we're returning a `Mouse` type instead of an `Object` type) as this is a case of covariant return types. At this point, we're done creating our copy, and we can return this value. This returned value will be independent of the original object since the strings in the class are immutable, and the remaining variables are all primitives, so they are all copied.

So when's an example in which the default `.clone()` method is not enough to produce two independent copies (i.e. copies with the property that changing one does not affect the other)? One example in which `.clone()` wouldn't be enough is if we have a new class named `Computer` that has a `Mouse` object as one of its instance variables. In this case, the `.clone()` method will make both `Computer` objects point to the `Mouse` object. However, since the `Mouse` object contains two primitive types (integers) that can be changed, changing an integer in the `Mouse` object will be reflected as a change in both `Computer` objects.

How do we fix this problem? We can override the `.clone()` method in the `Computer` class, check if a `Mouse` object is present (not null). If it is present, we can call the `.clone()` method on the mouse (which we've already explained does what we want).

Garbage Collection

One of the benefits of using Java as a programming language is that we don't have to worry about freeing memory that we are no longer using in the program. More precisely, if we remove all references to an object, this object becomes **garbage** since it is useless and can no longer affects the program. This is not true in some other programming languages, like C.

This entire process is automated for us in Java, and the process is called **garbage collection**. Essentially, we're able to reclaim memory used by unreferenced objects when we're running low on memory. This process is performed periodically by Java, but it is not guaranteed to occur.

One way in which garbage collection is performed is through the use of **destructors**, which are void methods with the name `finalize()` that contain the necessary actions to be performed when an object is freed. Destructors are only invoked if garbage collection occurs. Here's an example of what a destructor might look like:

```
class Foo {
    void finalize() { ... } // destructor for Foo
}
```

References

[1] Bloch, Joshua. Effective Java. Pearson Education India, 2016.