CMSC 131

Intro to Object Oriented Programming I

Ekesh Kumar

Prof. Nelson Padua-Perez • Fall 2019 • University of Maryland

http://www.cs.umd.edu/class/fall2019/cmsc131-030X/

Last Revision: March 6, 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:

```java
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:

```
1 public class FirstProgram {
2     public static void main(String[] args) {
3         int x;
4         x = 20;
5         System.out.println(x);
6     }
7 }
```

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:

```java
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:

```java
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:

```java
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:

```java
1  public class StringComparison1 {
2      public static void main(String args[]) {
3          String katniss = "Katniss";
4          String peeta = "Peeta";
5          String katniss2 = "Katniss";
6  
7          System.out.println(katniss.compareTo(peeta));
8          System.out.println(peeta.compareTo(katniss));
9          System.out.println(katniss.compareTo(katniss2));
10 
11         /* This is how you compare for equality. Do NOT compare using == */
12         System.out.println(katniss.equals(katniss2));
13     }
14 }
```

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.

```java
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:

```java
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:

```java
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:

```
1 import java.util.Scanner;
2
3 /**
4  * Shows basic use of scanner
5  */
6 public class Scanner1 {
7  public static void main(String args[]) {
8     Scanner keyboardInput = new Scanner(System.in);
9     int first, second;
10    /* Note the use of System.out.print() rather than System.out.println() */
11    System.out.print("Enter an integer value: ");
12
13    first = keyboardInput.nextInt();
14
15    System.out.print("Enter another integer value: ");
16    second = keyboardInput.nextInt();
17
18    System.out.println("The first number you typed was " + first);
19    System.out.println("The second number you typed was " + second);
20
21    System.out.println("Their sum is " + first + second);
22    System.out.println("Their product is " + first * second);
23
24    keyboardInput.close();
25  }
26 }
```

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:

```
1  public class SimpleIf {
2      public static void main(String args[]) {
3          Scanner scan = new Scanner(System.in);
4          System.out.print("Enter an integer: ");
5          int value = scanner.nextInt();
6          if (value < 0) {
7              System.out.println("That was a negative number!");
8          }
9  
10         System.out.println("The number was " + value);
11         scan.close();
12     }
13 }
```

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:

```
1  import java.util.Scanner;
2
3  public class Initialization1 {
4
5    public static void main(String[] args) {
6      int x;
7
8      Scanner scanner = new Scanner(System.in);
9      String s = scanner.next();
10
11     if (s.equals("dog")) {
12       x = 10;
13     }
14     System.out.println("x is " + x);
15
16     scanner.close();
17   }
18 }
```

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:

```java
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:

```java
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:

```
1  public class Example2 {
2      public static void main(String args[]) {
3          int i, limit = 3;
4          i = 1;
5          while (i <= limit) {
6              System.out.println(i);
7              i += 2;
8          }
9      }
10 }
```

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:

```
1  public class SimpleDoWhile {
2      public static void main(String args[]) {
3          int currValue = 1;
4          do {
5              System.out.println(currValue + " ");
6              currValue = currValue + 1;
7          } while (currValue <= 5);
8          System.out.println("currValue after the loop is " + currValue);
9      }
10 }
```

- 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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
public class NestedFor {
```

```java
  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:

```
1  import java.util.Scanner;
2  public class ScopeError {
3      public static void main(String args[]) {
4          Scanner scan = new Scanner(System.in);
5          do {
6              System.out.print("Enter an integer from 1 to 10: ");
7              int answer = scanner.nextInt();
8          } while (answer < 1 || answer > 10);
9      System.out.println("Good job");
10     scanner.close();
11     }
12 }
```

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:

```
1 public class Example3 {
2     public static void main(String args[]) {
3         int m = 1, x;
4         x = m++;
5         System.out.println(x);
6         System.out.println(m);
7     }
8 }
```

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:

```java
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:

```java
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:

```java
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[]) {
```

```java
5         double difference = 3.9 - 3.8;
6         System.out.println("3.9 - 3.8 = " + difference);
7         if ((Math.abs(difference) - 0.10) < EPSILON) {
8             System.out.println("Not exactly 0.10, but we will accept it.");
9         } else {
10            System.out.println("They are different.");
11        }
12    }
13 }
```

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 Monday, September 23, 2019

Today, we'll talk about some more details about returning values from a method, which may have not been mentioned before.

Pass By Value

First, we'll talk about some important details regarding using passed in variables in a method.

Consider the following code segment:

```
public void task(int x) {
    x = x + 1;
}
public static void main(String args[]) {
    int y = 5;
    task(y);
    System.out.println(y);
}
```

1. In our main method, we initialize a variable `y`, and we set it to the integer 5.
2. Next, we call `task` which simply increments its parameter.
3. Finally, we print out the variable `y` in our main method. What value gets printed?

Possibly surprisingly, the answer is 5. Why? Because when variables are passed in to a method in Java, the method first makes a copy of all of the variables. The copied variable is what is used inside of the method. In other words, changes to a method's parameters are only local; they are not exhibited when we return back to the main method.

This can be confusing — here's another example:

```
public static void wrongSwap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    System.out.println("a: " + a + "b: " + b);
}
public static void main(String args[]) {
    int x = 2, y = 3;
    wrongSwap(x, y);
    System.out.println("x: " + x + "y: " + y);
}
```

An unaware programmer might think that, after we call the `wrongSwap` method, the values of `y` and `y` are interchanged. However, as we've discussed, what will actually happen is that we'll make a copy of `x` and `y` to be used in the method. Thus, change takes place (the value of `x` and `y` are still 2 and 3, respectively).

This construct of making a copy of the variable to be used in the method is known as **passing by value** (some other programming languages reflect changes made to a parameter in a method; this construct is known as **passing by reference**).

StringBuffers

As we've discussed in the past, strings are immutable — there isn't any way in which we can change the object itself, but we can change the reference to the object.

The `StringBuffer` class is a peer class of `String`, and it provides many of the functionalities that we'd want to perform on strings. While strings are fixed-length, immutable character sequences, a `StringBuffer` object can be used to represent growable and writable character sequences. More precisely, the `StringBuffer` class is used to create mutable strings.

Some of the most commonly used methods on the `StringBuffer` class are listed below:

- The `append()` method is used to append a sequence of characters to the end of the current string in the `StringBuffer`.
- The `insert()` method is used to insert a sequence of characters into the middle of the current string.
- The `length()` method returns the length (character count) of the current string.

Here's an example:

```
public static void main(String args[]) {
    int x = 2, y = 3;
    StringBuffer a = new StringBuffer("talk");
    a.append("ing");
    System.out.println(a);
}
```

As we'd expect, once we append `ing` to `talk`, we end up printing `talking`.

However, it is important to remember that `StringBuffer` have *references* to strings. Thus, if we pass in a `StringBuffer` to a method, and if we modify the `StringBuffer` inside of that method, then we'll end up modifying the string that it points to. Here's another example:

```
public static void addEnding(StringBuffer b) {
    b.append("ing");
    b = null;
}

public static void main(String args[]) {
    StringBuffer a = new StringBuffer("talk");
    addEnding(a);
    System.out.println(a);
}
```

As mentioned previously, this code segment will end up printing `talking`.

Would this work with just a `String` object (not a `StringBuffer`)? No — recall that `String` objects are immutable; even if we were to append to a string with `+=`, we wouldn't be modifying the original object.

Returning Values

As we have already seen, one of the purposes of returning values in a method is way for us to access the results of a computation performed in the method in our main method. The method call in the `main` is always replaced by what the method returns.

This is demonstrated below:

```
public int add(int x, int y) {
    return (x + y);
}
public static void main(String args[]) {
    int z = add(5, 4);
    System.out.println(z);
}
```

When we execute this code, the `add(5, 4)` call inside of the main method gets replaced with the value that the method returns. In this case, that value is 9. Thus, the print statement that follows the method call prints out 9. Here's another example of a method:

```
public int maximum(int x, int y) {
    int max;
    if (x >= y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

In this method, we take in two integers. We determine which integer is larger than the other, and we return the larger of the two integers. Methods can also have more than one `return` statement. For example, we can rewrite the method above as follows:

```
public int maximum(int x, int y) {
    if (x >= y) {
        return x;
    } else {
        return y;
    }
}
```

In this rewritten method, we no longer create the variable `max` to store the maximum value of `x` and `y`. Instead, we just exit the function and the return the maximum value as soon as we know what it is. Once we execute a `return` statement, we no longer execute any of the statements in the method that proceed the `return` statement.

We can also use `return` statements to exit early out of `void` methods. Here's an example:

```
public void foo(int x) {
    if (x <= 0) {
        return;
    }
    System.out.println(2*x);
}
```

What does this method do? If we pass in a positive number, then we print out the double of that number. Otherwise, we don't do anything.

Software Development

There are many aspects that come into play when developing software or large-scale programs. We can describe the development of software using **the software lifecycle**, which is depicted below:

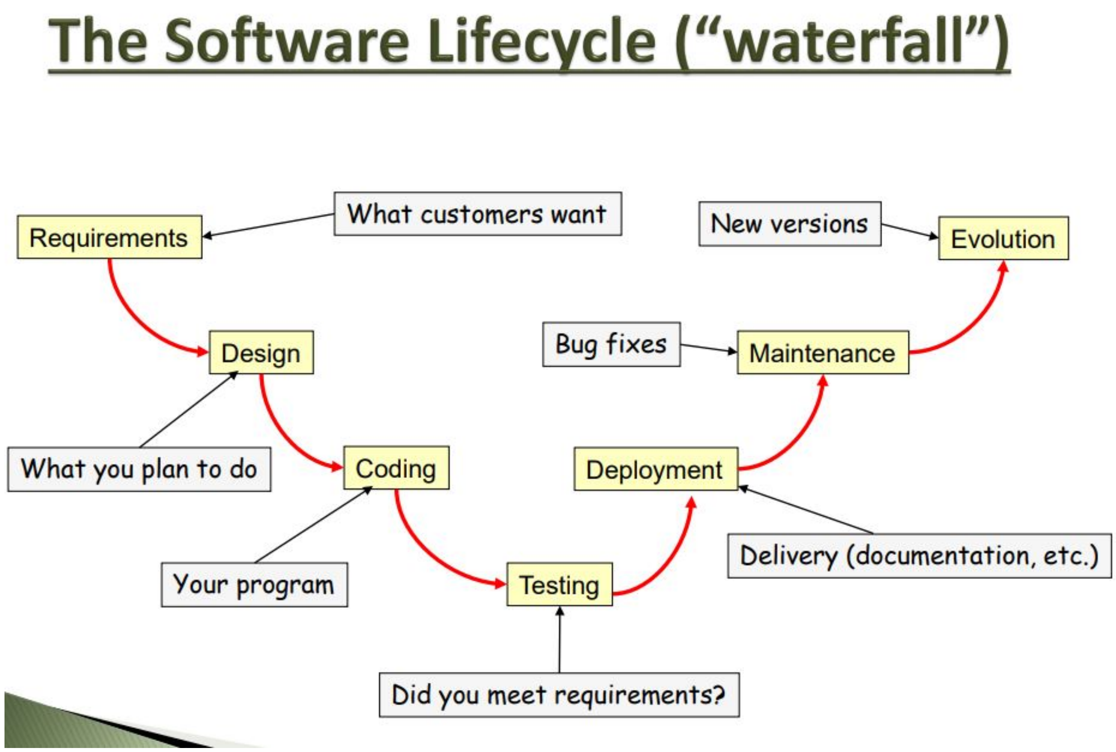

Figure 1: Software Development Lifecycle

As shown above, software development typically starts by recognizing the requirements the software must satisfy. This consideration typically involves thinking about what customers want. Next, we go into the design phase in which we think about what we plan to do. Finally, we go into the coding phase, followed by testing, deployment, maintenance, and finally evolution.

When developing software, it's also usually helpful to take a **top-down approach**. A top-down design is a method of ordering knowledge in which we start from a big idea (i.e. what we want to implement), and we break it down into what we need to achieve what's at the top (and we subsequently break those things down, and so on).

Pseudocode is also useful in creating software. Pseudocode is an English-like description of the set of steps required to solve a problem. Here's an example of some pseudocode for finding the minimum value from numerous inputted numbers.

```
FIND_MINIMUM {
    read the number of values to process; call this value N.
    repeat the following steps until N input values have been processed {
       read the next value into x
       if x is the first value read, set the current minimum to x.
```

```
        otherwise, if x is less than the current minimum, then set the current minimum to x.
    }
}
```

By using pseudocode, we can avoid having to deal with the subtleties of our programming language's syntax and semantics, and we can instead focus on determining the high-level steps necessary to solve a problem.

12 Wednesday, September 25, 2020

Exam 1 is today.

13 Friday, September 27, 2020

Class is cancelled today due to a Football game.

14 Monday, September 30, 2020

Introduction to Classes

In Java, a **class** is a user-defined blueprint or protottype from which objects are created. Classes represent a set of properties or methods that are common to all objects of one particular type. They allow us to group together several variables and methods into one entity.

For instance, we can define a `Superhero` class in one file called `Superhero.java` as follows:

```
public class Superhero {
    String name;
    int strength;
}
```

In this class, we've added a `strength` and `name` variable to represent the corresponding superhero's strength and name. Essentially, we've grouped together both of these variables into one entity (known as a `Superhero`). Now, in another file, we can create instances of the `Superhero` class:

```
public class Driver {
    public static void main(String args[]) {
        Superhero s1 = new Superhero();
        s1.name = "Batman";
        s1.strength = 100;
    }
}
```

We create a `Superhero` object by using the `new` keyword. In the driver program above, we've created a `Superhero` object called `s1`. When we create such an object, the object gets its own `name` and `strength` variables. We can access these variables present in the `Superhero` class by using the dot operator (that is, we put a period after the variable, and we write the name of the variable that we want to access). As shown above, we've set our superhero's name to `Batman`, and we've set its strength to 100.

Classes can also have their own methods. For example, we can modify our `Superhero.java` file to what follows below:

```
public class Superhero {
    String name;
    int strength;

    void print() {
        System.out.println("Name: " + name);
        System.out.println("Strength: " + strength);
    }
}
```

Similar to how we access variables, we can access methods using the dot operator:

```
public class Driver {
    public static void main(String args[]) {
        Superhero s1 = new Superhero();
        s1.name = "Batman";
        s1.strength = 100;
```

```
        s1.print(); /* This will print s1's name and strength. */
    }
}
```

Let's add another method:

```
public class Superhero {
    String name;
    int strength;

    void print() {
        System.out.println("Name: " + name);
        System.out.println("Strength: " + strength);
    }

    void increaseStrength(int delta) {
        if (delta < 0) {
            System.out.println("Invalid strength increment.");
        } else {
            strength += delta;
        }
    }
}
```

Now, once we've instantiated a `Superman` object, we can invoke the `increaseStrength()` method in order to increase the superhero's strength (we don't want our users to be able to decrease the superhero's strength).

However, we are faced with a problem: nothing is stopping a user from simply setting the `strength` variable of a `Superman` object to a negative value (or a decreased value). Here's an example of what this means:

```
public class Driver {
    public static void main(String args[]) {
        Superhero s1 = new Superhero();
        s1.name = "Batman";
        s1.strength = 100;
        /* s1's strength is 100. */
        s1.increaseStrength(-10);
        /* s1's strength is still 100. */
        s1.strength = 90;
        /* s1's strength is 90. */
        System.out.println(s1.strength); /* Prints 90. */
    }
}
```

In order to overcome this issue, we can prevent our users from directly accessing the variables inside of a class by using the **private** keyword. By adding the keyword `private` in front of the variables inside of our `Superhero` class, we won't be able to use the dot operator to access them. Likewise, if we were to make our methods private, then they wouldn't be accessible from the outside (we don't want this, though). It's also a good idea to explicitly write `public` before any of the variables or methods that we intend the user to have access to.

If we're making our variables private, then how do we set the variables of a `Superhero` object? We can do so by simply creating a method that does this for us. Consider the following modified `Superhero.java` file:

```java
public class Superhero {
    /* These variables can't be accessed with the dot operator. */
    private String name;
    private int strength;

    /* This method lets us set the values of our variables. */
    public void init(String nameIn, int strengthIn) {
        name = nameIn;
        if (strengthIn < 0) {
            System.out.println("Invalid strength.");
            strength = 0;
        } else {
            strength = strengthIn;
        }
    }

    public void print() {
        System.out.println("Name: " + name);
        System.out.println("Strength: " + strength);
    }

    public void increaseStrength(int delta) {
        if (delta < 0) {
            System.out.println("Invalid strength increment.");
        } else {
            strength += delta;
        }
    }
}
```

Now, we can update our driver program to the following:

```java
public class Driver {
    public static void main(String args[]) {
        Superhero s1 = new Superhero();
        // s1.name = "Batman" -> this won't compile; name is private.
        s1.init("Batman", 100);
        // s1.strength = -1; -> this won't work; strength is private.
        s1.increaseStrength(5);
        // s1's stsrength is now 105.
    }
}
```

Constructors

Previously, we showed how we can limit a user's access to a class's variables — by making the variables private, and providing public methods to set the variables.

Since this is such a common thing to do, Java allows us to define a special method, known as a **constructor**, that is designed specifically for the purpose of setting up an object's variables upon instantiation. A constructor is declared by creating a method whose name is the same as the class's name. Moreover, this method is called when we're instantiating the object using the `new` keyword.

Here's a modification of the `Superhero` code that we saw earlier:

```java
public class Superhero {
    /* These variables can't be accessed with the dot operator. */
    private String name;
    private int strength;

    /* This is a constructor. */
    public Superhero(String nameIn, int strengthIn) {
        name = nameIn;
        if (strengthIn < 0) {
            System.out.println("Invalid strength.");
            strength = 0;
        } else {
            strength = strengthIn;
        }
    }

    public void print() {
        System.out.println("Name: " + name);
        System.out.println("Strength: " + strength);
    }

    public void increaseStrength(int delta) {
        if (delta < 0) {
            System.out.println("Invalid strength increment.");
        } else {
            strength += delta;
        }
    }
}
```

The only thing that we've changed now is that we've replaced the `init` method with a constructor. Now instead of invoking the `init` method to instantiate our Superhero's variables, we can simply pass in the values we want "on-the-fly" when we're instantiating the object:

```java
public class Driver {
    public static void main(String args[]) {
        Superhero s1 = new Superhero("Batman", 100);
        s1.print();
    }
}
```

When we execute `Superhero s1 = new Superhero("Batman", 100);`, the `name` and `strenth` of `s1` are set to Batman and 100. There's also a special type of constructor, known as a **default constructor**, which is simply a constructor that takes in no arguments. One possible implementation of a default constructor for our `Superman` class might look something like this:

```java
public class Superhero {
    /* Other code omitted. */

    /* Default constructor; takes in no arguments. */
    public Superhero() {
        strength = 0;
        name = "NONAME";
    }
}
```

Now, if we instantiate an object by calling `Superman s1 = new Superman()`, it automatically gets assigned a strength of 0, and a name of `NONAME`.

There's another special method, known as the `toString()` method, which simply tells the Java compiler what to print when we try to print the object. Currently, if we were to add the line `System.out.println(s1);` into our program, our program would not compile. Why? Because `System.out.println()` requires that we insert a string as the parameter. However, if we implement a `toString()` method for the `Superhero` object, then this line will compile, and it'll print whatever our `toString()` method returns.

Here's what our modified code would look like:

```java
public class Superhero {
    /* These variables can't be accessed with the dot operator. */
    private String name;
    private int strength;

    /* This is a constructor. */
    public Superhero(String nameIn, int strengthIn) {
        name = nameIn;
        if (strengthIn < 0) {
            System.out.println("Invalid strength.");
            strength = 0;
        } else {
            strength = strengthIn;
        }
    }

    public void increaseStrength(int delta) {
        if (delta < 0) {
            System.out.println("Invalid strength increment.");
        } else {
            strength += delta;
        }
    }

    public String toString() {
        String answer;
        answer = "Name: " + name;
        answer += ", Strength: " + strength;
        return answer;
    }
}
```

Now, we've gotten rid of our `print()` method, and we've replaced it with a `toString()` method. Instead of calling `s1.print()`, we can instead call `System.out.println(s1)`.

15 Wednesday, October 2, 2019

Last time, we started talking about classes. Recall that classes can be **instantiated** using the `new` keyword (for example, we can instantiate a `Superhero` by writing `Superhero s = new Superhero();`). An **object** is simply an instance of a class.

.equals() method

When we're implementing a class, sometimes it's useful to be able to be able to compare two instances of the class. In order to do so, there's a special function — called `.equals()` ("dot equals") — that we can implement. Recall, for instance, that we use this method when we want to compare two strings (i.e. `"Hello".equals("Hello")` evaluates to `true`), which means that the `String` class in Java implements the `.equals()` method.

How does the `.equals()` method work? Here's an example:

```java
public class Buffalo {
    public String name;
    public int age;

    public Buffalo(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Buffalo(int age) {
        this.age = age;
    }

    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Buffalo buffalo = (Buffalo) other;
        return (this.name == other.age && this.age == other.age);
    }

}
```

Note that we've used a new keyword: **this**. The *this* keyword can be used in Java as a reference to the current object. This is useful particularly when we have two variables of the same name, which allows us to get rid of ambiguity.

In the `Buffalo` class above, we have implemented a method called `equals` that returns a `boolean`. Every time we create our own `.equals()` method, we need to make sure that the function has the same function prototype (the return value and parameter list) as in this example. Moreover, the method needs to return `true` if we deem the other object being passed in as a parameter to be equal to the current object, and it should return false otherwise.

How does our `.equals()` method work?

- First, we take in an object `obj` as a parameter. We want to determine whether this object is equal to the current object that our class is representing.
- Firstly, we check if the parameter that we passed in *is* the class that we're representing. This is done by, once again, using the `this` keyword to obtain a reference to the current object. If the parameter passed in is the same as the class we're representing, then we know that the two objects must be the same.
- Next, we check whether the parameter is `null`. If so, we return `false` since we know that our current object isn't null, meaning that it cannot be equal to the parameter object. Moreover, we call `getClass()` on our current object and the parameter object. This is a special function that Java provides for objects, and it tells us whether or not two objects have the same type. If they do not have the same type, then we can return false.
- At this point, we know that the two objects have the same type. Thus, we can treat the parameter object `obj` as a `Buffalo` object. This is done by casting the parameter as a `Buffalo`. Finally, we can check the criteria that we want in order to check if the two `Buffalo` objects are equal. In particular, we verify that the parameter object's name and age are equal to the current object's name and age. If they are, then we return `true`; otherwise, we return `false`.

Can we use the `.equals()` method without implementing it? Yes – but it may not do what we want it to do. Every class, by default, has a `.equals()` method. How does it work? It simply compares the two memory addresses of the variables we are using. In our `Buffalo` example above, even if we set the `name` and `age` of two `Buffalo` objects to the same values, we would end up finding that the default `.equals()` method asserts that these two objects are different. In summary, it's not a good idea to rely on the default `.equals()` method that Java provides for you.

The "this" Keyword

Previously, we saw that we can use the *this* keyword to obtain a reference to the current object. Here, we'll see a few more applications of the `this` keyword.

Here is an implementation of a `Person` class:

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this(name, 18);
    }

    public person() {
        this("NONAME", 18);
    }

    public Person increaseAge(int delta) {
        age += delta;
        return this; /* Return reference. */
    }
}
```

- In our first constructor, we use the `this` keyword to reference the instance variables inside of our class. As mentioned previously, it can be useful to use this `this` keyword here since our parameter variables have the same names as our instance variables. Using `this` avoids ambiguity.

- In our next two constructors, we use the `this` keyword, almost like a function. What are these two lines doing? This is a special way to call one of the current object's constructors. In this case, since we've provided two parameters of type `String` and `int`, Java will recognize that we're trying to use the first constructor that we created. Thus, we set `name` and `age` according to the values that we provide in our `this()` call. This is helpful since it allows us to reuse the code that we use in one constructor in another.

- Finally, in our `increasingAge()` function, we return a reference to the current object by writing `return this;`. This is allowed since our function prototype indicates that we're returning a `Person`, and that's exactly what we're doing.

Here's another example:

```java
public class HdTv {
    private String make;
    private int cost;

    public HdTv(String make, int cost) {
        this.make = make;
        this.cost = cost;
    }

    public HdTv(HdTv tv) {
        make = new String(tv.make);
        cost = tv.cost;
    }

    public String toString {
        return "Make: " + make + ", Cost: " + cost;
    }

    public static void main(String args[]) {
        HdTv tv = new HdTv("Poly", 100);
        System.out.println("Original--> " + tv);
        HdTv copy = new HdTv(tv);
    }
}
```

In this example, we introduce another application of the `this` keyword: our `HdTv` class implementation provides us with a constructor whose only parameter is another `HdTv` object. The usefulness of this constructor is illustrated in our main method, where we initialize an `HdTv` called `copy` using another `HdTv` called `tv`. Such a constructor that uses an existing object to create a new object is known as a **copy constructor**.

16 Friday, October 4, 2019

Breaking and Continuing

In Java, we can use the **break** keyword to terminate the loop in which the `break` statement is used (i.e. a for-loop, while-loop, or do-while loop). Here's an example:

```java
public class Example {
    public static void main(String args[]) {
        int x = 0;
        while (true) {
            if (x == 40) {
                break; // Jump out of the loop.
            }
            x = x + 1;
            System.out.println(x);
        }
    }
}
```

How does this program work? At a first glance, it might seem like this program has an infinite loop — we have a loop whose loop condition is just `true`. However, as we've just mentioned, this while-loop includes a `break` statement that gets executed when `x` is equal to 40. This `break` statement will get us out of the loop after we print out the integers between 0 and 39, inclusive.

Can a `break` statement be placed anywhere? No — `break` must be used either inside of a loop, or inside of a switch statement (which we will discuss later). If we place a `break` statement somewhere that it is not supposed to be used, then we receive a compilation error.

What happens if we `break` inside of a nested loop? We only exit the inner-most loop we're in (i.e. if we're inside of two for-loops, then we'll go outside of the innermost loop but inside of the outermost loop).

Next, we'll discuss the **continue** keyword in Java. The `continue` keyword can be used inside of a loop to immediately jump to the next iteration of the loop. Here's an example:

```java
public class Example {
    public static void main(String args[]) {
        int x = 0;
        while (true) {
            x = x + 1;
            if (x == 100) {
                break; // Jump out of the loop.
            }
            if (x % 2 == 0) {
                continue;
            }
            System.out.println(x);
        }
    }
}
```

What's happening here? Once again, we have a `while (true) { ... }` loop. However, we can see that we're exiting the loop as soon as our variable `x` becomes 100. Also, we have now introduced a second conditional in which we check whether `x` is an even number. If so, then we `continue` onto the next iteration of the `while` loop. Consequently, this program ends up printing all odd numbers between 1 and 100.

In general, one should be careful when using `break` and `continue` statements since they modify the control flow of the program.

17 Monday, October 7, 2019

Exceptions

Introduction

When executing a program, oftentimes, something unexpected can happen. For instance, we might run out of memory, try to divide a number by 0, or try to open a file that doesn't exist.

How do we recover from these unexpected occurrences? In Java, we can perform **exception handling**. An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions (the examples listed previously are all examples of exceptions). Exception handling lets us recover in the case that an exception occurs.

To illustrate an example, consider the following program:

```
public class MilesPerGallon {
    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter numbre of gallons: ");
        int miles = scanner.nextInt();

        System.out.println("Enter number of gallons: ");
        int gallons = scanner.nextInt();

        int milesPerGallon = miles / gallons;

        System.out.println("Miles per gallon is " + milesPerGallon);

        scanner.close();
```

In this program, we let the user provide integer values for the number of miles and the number of gallons that they use, and we subsequently divide the two quantities to find the miles per gallon used. But, what happens if the user enters the value 0 for `gallons`? We get an exception when we try to compute `milesPerGallon` (since we're trying to divide by 0).

One way to solve this issue is to simply use an `if-else` statement to check whether the user's input is 0. But, what happens if there are several "bad" values that we want to be careful of (in this case, the only "bad" value to be careful of is 0, but if there were too many, then it might not be feasible to enumerate them all)? In this scenario, we would need to use exception handling.

One way in which we can handle exceptions is by using a `try-catch` block. In Java, a `try` statement lets us define a block of code to be tested for errors while it is being executed. Furthermore, the `catch` block can be used to define a block of code to be executed, if an error occurs in the `try` block. Here's the same code from above, but now written with a `try-catch` block:

```
public class MilesPerGallon {
    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter numbre of gallons: ");
        int miles = scanner.nextInt();

```

```
        System.out.println("Enter number of gallons: ");
        int gallons = scanner.nextInt();
        try {
           int milesPerGallon = miles / gallons;

           System.out.println("Miles per gallon is " + milesPerGallon);
        } catch (ArithmeticException e) {
           System.out.println("Invalid value provided!");
           System.out.println("Default message: " + e.getMessage());
        }
        System.out.println("Thank you for using our system.");
        scanner.close();
```

In the code above, we can note the following changes:

- We've placed the `int milesPerGallon = miles / gallons;` statement in the `try` block that we've created. Why? Because this is the portion of code that might result in an exception. More precisely, the exception that we want to avoid is known as an `ArithmeticException` (this is the name of the exception that results from trying to divide by zero).
- Next, we've added a `catch (..) { ... }` block. Inside of the parentheses directly after the `catch` statement, we need to specify what type of exception we want to watch out for. In this case, we've specified that we want to watch out for `ArithmeticException`s since this type of exception takes place when we try to divide by zero. Is it possible to watch out for *all* exceptions? Yes — instead of specifying the specific type of exception, like `ArithmeticException`, we would just write `Exception`.
- Inside of the `catch` block, we specfy the instructions that we want to execute if an exception takes place. In our program, we simply notify the user that they've proivded an invalid value. Moreover, we print the message associated with the `ArithmeticException` exception by calling the `.getMessage()` method on it.
- Finally, we print "Thank you for using our system," outside of the `try-catch` block. This message gets printed regardless of what the user inputs. Finally, we close our scanner, and we're done.

In Java, exceptions are represented by *objects*. Java has a built-in class called `Exception`, and we can treat an `Exception` as an object. This is seen clearly in the example above, where we call the `.getMessage()` method associated with the `ArithmeticException` object `v`.

If a user ever provides the value 0 for `gallons`, then we jump out of the `try` block as soon as the exception takes place. This means that the statement `System.out.println("Miles per gallon is " + milesPerGallon);` is *never executed*. Here are some of the most common types of exceptions that we should be wary of:

- `ArithmeticException` is an exception that occurs when we try to divide by zero (like in the example above).
- `NullPointerException` occurs when we attempt to acces an object with a null reference.
- `IOException` occurs when we attempt to perform illegal input/output operations.`IndexOutOfBoundsException` occurs when we attempt to access part of an array (which we will introduce later) or `String` outside of the range in which it's defined.

Exception Propagation

When an exception takes place, Java always looks in the current method for a catch clause that matches the exception. If one is found, the exception is handled using that `try-catch` block. Otherwise, **exception propogation** takes place.

What is exception propogation? When an exception occurs, Java pops back up the call stack (e.g. goes up through the various methods it is nested in) to see whether the exception is being handled in a catch block of one of the methods. This process is known as exception propogation. The first method that handles the exception defines how we handle the exception. If we get all the way back up to the `main` method and no method catches the exception, then Java will abort your program.

Here's an example:

```java
public class Propagation {
  public static void B() {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter number of miles: ");
    int miles = scanner.nextInt();

    System.out.println("Enter number of gallons: ");
    int gallons = scanner.nextInt();

    int milesPerGallon = miles / gallons;
    System.out.println("Miles per gallon is: " + milesPerGallon);

    scanner.close();
  }

  public static void A() {
    System.out.println("Before calling method B");
    B();
    System.out.println("After calling method B");
  }

  public static void main(String[] args) {
    try {
      System.out.println("Before calling method A");
      A();
      System.out.println("After calling method A");
    } catch (ArithmeticException e) {
      System.out.println("Invalid value provided");
      System.out.println("Default Message: " + e.getMessage());
    }
    System.out.println("Thank you for using our system");
  }
}
```

In the following code, we note that our "dangerous code," which might cause an exception, has now been moved to the function `B()`. However, we've introduced another function called `A()`, which simply prints a statement, calls the function `B`, and prints another statement.

In our `main` method, we have a `try-catch` block in which we print a statement prior to calling `A`, call `A`, and subsequently print another statement.

What happens if the user of the program enters 0 for the number of miles? Even though our function `B` does not have a `try-catch` block, error propagation will occur since the exception is being thrown inside of a function. More precisely, once we try to divide by 0, we'll check "up one level" for a `try-catch` block. Since the function `A` doesn't have a `try-catch` block either, we'll go up again into the `main` method. In the `main` method, we'll find a `try-catch` block, and we'll perform the error handling in the `catch` block.

What happens if we add another `try-catch` block in `A`? Then the `try-catch` block in `A` will be used instead of the `try-catch` block in the main (we always use the `try-catch` block that's closest to the error on the stack).

What happens if we *remove* the `try-catch` block in the `main` method? Our program will abort with an exception when we try to divide by 0.

Throwing Exceptions

So far, we've seen how to handle exceptions. However, it turns out that we can throw our own exceptions as well. Why would we want to do this? Mainly because it helps us debug; by throwing an exception when something goes wrong, we know exactly what went wrong, and we can figure out how to fix the issue.

The general syntax to throw an exception is shown below:

```
throw new [ExceptionName](message);
```

One instance in which it's useful to throw an exception is when we haven't implemented a method yet:

```java
public static double computeCost(int option) {
   throw new UnsupportedOperationException("Not implemented yet");
}
```

In the code segment above, we don't have an implementation of the `computeCost` method yet, so we just throw an exception instead. This is helpful since it allows our code to still compile (we don't need to comment out this method until we implement it). Furthermore, if a user tries to call this method, then they will receive a message telling them that the method is not implemented yet.

Here's another example in which the usefulness of throwing exceptions is clear:

```java
   public static int processCourse(String courseName) {
   if (courseName == null) {
      throw new IllegalArgumentException("Invalid argument");
   } else {
      if (courseName.equals("cmsc131")) {
         return 4;
      } else {
         return 1;
      }
   }
}
```

In this method, we throw an `IllegalArgumentException` if the user provides a `null` reference to a `String` object. Once again, this is very helpful since it tells us right away what the issue is, and we don't have to spend long periods of time searching for the error.

18 Wednesday, October 9, 2019

"Finally" Blocks

Last time, we introduced `try-catch` blocks, which are used to handle exceptions. Today, we'll introduce the **finally** keyword. A `finally` block is typically used along with a `try-catch` block. The purpose of a finally block is to hold all of the crucial statements that must be executed, whether the exception occurs or not. The statements present in the `finally` block will **always** execute, regardless of whether an exception occurs or not.

Here's an example:

```java
public static int getGasAverage() {
  Scanner scanner = new Scanner(System.in);

  System.out.println("Enter number of miles: ");
  int miles = scanner.nextInt();

  System.out.println("Enter number of gallons: ");
  int gallons = scanner.nextInt();

  try {
    int milesPerGallon = miles / gallons;
    System.out.println("Miles per gallon is: " + milesPerGallon);

    return milesPerGallon;
  } finally {
    scanner.close();
    System.out.println("Before leaving method getGasAverage() (finally message)");
  }
}

public static void main(String[] args) {
  try {
    System.out.println("Before calling method getGasAverage() (main)");
    System.out.println("Average: " + getGasAverage());
    System.out.println("After calling method getGasAverage() (main)");
  } catch (ArithmeticException e) {
    System.out.println("Invalid value provided (in main)");
    System.out.println("Default Message: " + e.getMessage());
  }
  System.out.println("Thank you for using our system");
}
```

What's happening here?

- In main method, we have a `try-catch` block in which we call the function `getGasAverage()`. In our `getGasAverage` function, we read in the `miles` and `gallons` variables using a scanner.
- In the `getGasAverage` function, we have another `try-catch` block in which we divide the `miles` by `gallons`. What happens if the user provides the value 0 for `gallons`? An exception is thrown, and Java starts searching for a `catch` block. Since there's no `catch` block in the `getGasAverage` function, we use the `catch` block in the main.
- After executing the `catch` block in the `main` method, we execute the code in `finally`. Recall that the statements in `finally` blocks are *always* executed.

- Inside of the `finally` block, we close the scanner. Note that this is something that we always want to do when we're done with the scanner, whether an exception took place or not.

In the code segment above, we have a `try { ... } finally { ... }` block inside of our `getGasAverage()` method; however, we are also permitted to have `try { ... } catch (..) { .. } finally { ... }` blocks.

In summary, the `finally` keyword allows us to group together some statements that should *always* be executed, whether an exception is thrown or not. Often, these instructions might include closing a scanner, closing a file, or printing some message.

String Methods

In Java, the `String` class has several built-in methods that are readily usable on strings. Today, we'll discuss some of the most commonly used ones.

- The `.charAt()` method takes in an single integer as a parameter, and it returns the character at the given index (starting from 0). For example, `"Hello".charAt(0)` would return `H`, and `"Hello".charAt(1)` would return e
- The `.length()` method returns the number of characters in the `String` (where we start counting from 1). For instance, `"Hello".length()` returns 5.
- `.toLowerCase()` and `.toUpperCase()` return another String variable with all of the letters in the `String` in lowercase or in uppercase. For example, `"Hello".toLowerCase()` returns `hello`, whereas `"Hello".toUpperCase()` returns `HELLO`. Since strings are immutable, these methods return entirely new objects.

There are several other String methods available to us as well. A full list is available in the Java API at https://docs.oracle.com/javase/7/docs/api/java/lang/String.html.

Here's a sample program that illustrates how some of these methods work:

```java
package apisEx;

import javax.swing.*;

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

    String name = JOptionPane.showInputDialog("Enter a word");
    String answer;

    if (name.charAt(0) == name.charAt(name.length() - 1)) {
      answer = "word starts and end with the same letter";
    } else {
      answer = "word does not start and end with the same letter";
    }
    JOptionPane.showMessageDialog(null, answer);

    String str1 = JOptionPane.showInputDialog("Enter string");
    String str2 = JOptionPane.showInputDialog("Enter string");
    System.out.println("Using compareTo: " + str1.compareTo(str2));
    System.out.println("Using compareToIgnoreCase: " + str1.compareToIgnoreCase(str2));

```

```
23      String login = JOptionPane.showInputDialog("Enter login id");
24      answer = "Access Granted";
25      if (!login.equalsIgnoreCase("Hulk")) {
26         answer = "Access Denied";
27      }
28      JOptionPane.showMessageDialog(null, answer);
29
30      String mascot = " Terps ";
31      System.out.println("Character r for Terps is at : " + mascot.indexOf('r'));
32      System.out.println("Before trimming:--" + mascot + "--");
33      String mascotTrimmed = mascot.trim();
34      System.out.println("After trimming:--" + mascotTrimmed + "--");
35      System.out.println("Uppercase: " + mascot.toUpperCase());
36      System.out.println("Lowercase: " + mascot.toLowerCase());
37      System.out.println("Mascot after trimming and case changes:--" + mascot + "--");
38
39      int x = 100;
40      String strIntValue = String.valueOf(x);
41      System.out.println(strIntValue);
42   }
43 }
```

In this example, we also introduce the `JOptionPane` class, which is simply a way to produce graphical user interfaces. This can be used to make our programs more interactive; however, it would also be fine to just use a `Scanner`.

In this example, we read in the variable `name` from the user's input. On Line 11, we call `.charAt(0)` to retrieve the first character of the user's inputted string, and we call `.charAt(name.length() - 1)` to retrieve the last character of the user's inputted string. If these two characters are the same, then we display the message "word starts and ends with the same character."

Next, we read in two more strings, and we display what the `.compareTo()` and `.compareToIgnoreCase()` methods return. Recall that the `.compareTo()` method returns the integer 0 if the two strings are equal, a positive value if the first string is lexicographically greater than the second string, and a negative number otherwise. On the other hand, the `.compareToIgnoreCase()` function does the exact same, while ignoring the case of each character (`A` is considered the same as `a`, etc).

Math Methods

Now, we'll introduce some useful math methods that we can use. Here are some of the most common methods that are made available to us:

- The `Math.abs()` method takes in a number, and it returns the absolute value of that value. For example, `Math.abs(5) = 5`, and `Math.abs(-5) = 5`.
- The `Math.ceil()` method takes in a double, and it returns the ceiling (smallest integer larger than the number) of the number. For example, `Math.ceil(3.5) = 4`, and `Math.ceil(5) = 5`
- The `Math.floor()` method takes in a dobule, and it returns the floor (largest integer less than the number) of the number. For example, `Math.floor(3.5) = 3`, and `Math.floor(3) = 3`.
- The `Math.pow()` method takes in two values, and it returns the first argument raised to the power of the second argument. For example, `Math.pow(2, 3)` evaluates to $2^3 = 8$.

Here's a Java program illustrating some of these methods:

```java
1 public class MathExamples {
2   public static void main(String[] args) {
3 
4     // Math m = new Math(); // not possible
5     System.out.println("Math.PI: " + Math.PI);
6     System.out.println("Maximum between 20 and 10: " + Math.max(20, 10));
7     System.out.println("Minimum between 20 and 10: " + Math.min(20, 10));
8 
9     double value = Double.parseDouble(JOptionPane.showInputDialog("Enter number"));
10    System.out.println("Square root of " + value + ": " + Math.sqrt(value));
11    System.out.println("Floor of " + value + ": " + Math.floor(value));
12    System.out.println("Ceiling of " + value + ": " + Math.ceil(value));
13    System.out.println("Power (2) of " + value + ": " + Math.pow(value, 2));
14    System.out.println("Rounding " + value + ": " + Math.round(value));
15 
16    System.out.println("Set 1");
17    for (int i = 0; i < 100; i++) {
18      System.out.println(Math.random());
19    }
20 
21    System.out.println("Set 2");
22    for (int i = 0; i < 100; i++) {
23      System.out.println(200 * Math.random());
24    }
25 
26    System.out.println("Set 3");
27    for (int i = 0; i < 100; i++) {
28      System.out.println(Math.floor(200 * Math.random()));
29    }
30 
31    System.out.println("Set 4");
32    for (int i = 0; i < 100; i++) {
33      System.out.println(Math.floor(201 * Math.random()));
34    }
35 
36    System.out.println("Set 5");
37    for (int i = 0; i < 100; i++) {
38      System.out.println(Math.floor(200 * Math.random()) + 1);
39    }
40  }
41 }
```

Here are some useful observations that we can make:

- First, we read in a `double` on Line 9. Line 10 prints out the square root of the number, Line 11 prints out the floor of the number, Line 12 prints out the ceiling of the number, Line 13 prints out the square of the number, and Line 14 rounds the number.
- On Line 18, we print out `Math.random()` 100 times. What does `Math.random()` do? It returns a pseudorandom number on the interval $[0, 1]$.
- In the subsequent for-loops, we shift our interval `[0, 1]` to various other intervals so that we can generate random numbers in any interval we want. For example, if we add a constant value c to `Math.random()`, then we end up generating values in the interval `[c, 1 + c]`. Moreover, we can scale the interval by multiplying it by different values.

19 Friday, October 11, 2019

Immutable Classes

In Java, an **immutable class** is a class whose content we cannot change. Recall that the `String` class in Java is immutable. Why is immutability good? Because they make our code more safer and cleaner, which is particularly important when we are multithreading.

The **final** keyword in Java can be used before a variable to prevent it from being changed. The only place in which we can assign a value to a `final` variable is inside of a constructor. After the constructor is executed, a `final` variable cannot be changed.

Here's an example of an immutable class:

```java
public final class Telephone {
  private final String number;

  public Telephone(String number) {
    this.number = number;
  }

  public String getNumber() {
    return number;
  }

  public String toString() {
    return "Telephone [number=" + number + "]";
  }

  public static void main(String[] args) {
    Telephone telephone = new Telephone("1-899-COMPILE_JAVA");

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

In the code segment above, we've also specified our class as `final`. Declaring a class as final prevents it from being extended, which is something that we will discuss later. In the code above, we've declared both our class and our `number` variable as `final`. This is an example of an immutable class since, once an object is created, it cannot be modified.

Ternary Operator

In Java, the **ternary operator** is an operator that takes in three arguments. The first argument is a Boolean condition, the second argument is the result that the operator should return if the condition evaluates to true, and the third condition is the result that the operator should return if the condition evaluates to false.

The ternary operator might look confusing, but with a few examples, it is not too hard to understand. The general form of the ternary operator is shown below:

```
condition ? exprValueIfConditionIsTrue : exprValueIfConditionIsFalse
```

Here's an example:

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

        Scanner scan = new Scanner(System.in);
        int x = scan.nextInt();
        int y = scan.nextInt();
        int ans = (x > y) ? x : y;
        System.out.println("Maximum is " + ans);
        scan.close();
    }

}
```

What's this program doing? First, we just declare a scanner, and we read in two integer variables `x` and `y`. Next, we initialize the variable `ans` using the ternary operator. Here, our condition is the Boolean expression `(x > y)`. If this Boolean expression evaluates to `true`, then we set `ans` equal to `x`; otherwise, we set `ans` equal to `y` In other words, we are simply setting `ans` to the maximum of the two variables that we are reading in.

We could easily change our initialization of `ans` to

```java
int ans = (x < y) ? x : y;
```

if we wanted `ans` to store the minimum value instead.

It might be helpful to think of the ternary operator as a shortened way of writing an if-else statement. If the condition provided is true, then we evaluate the ternary expression to the first expression provided; otherwise, we evaluate it to the second expression provided.

The Switch Statement

A **switch** statement in Java allows a variable to be tested for equality against a list of several values. Each value in the list is called a **case**, and the variable being "switched on" is checked for each case.

Pretty much, a `switch` statement is a more convenient — and often more efficient — way to perform a multi-way conditional based on a single control value.

The general syntax of a `switch` statement is as follows:

```java
switch (variable) {
    case 1:
        System.out.println("Case 1!");
        break;

    case 2:
        System.out.println("Case 2!");
        break;

    // ... ... ....

    default:
        System.out.println("Invalid");
```

```
        break;
}
```

The `switch` statement code presented above would be equivalent to the following code:

```
if (variable == 1) {
    System.out.println("Case 1!");
} else if (variable == 2) {
    System.out.println("Case 2!");
    // ... .... ..
} else {
    System.out.println("Invalid!");
}
```

The case that is selected in a `switch` statement is dependent on the value of the `variable`. The `break` statement within each case is used to exit the `switch` statement afterwards.

Here's a fully working example:

```
public class SwitchExample {
   public static void printDay(int day) {
      String answer;

      switch (day) {
      case 6:
         answer = "Saturday";
         break;
      case 1:
      case 2:
      case 3:
      case 4:
      case 5:
         answer = "Weekday";
         break;
      case 7:
         answer = "Sunday";
         break;
      default:
         answer = "Invalid day value";
         break;
      }
      System.out.println("Value " + day + " corresponds to " + answer);
   }

   public static int printYear(String type) {
      int year;

      switch (type) {
      case "Freshman":
         year = 1;
         break;
      case "Sophomore":
         year = 2;
         break;
      case "Junior":
         year = 3;
         break;
```

```
      case "Senior":
         year = 4;
         break;
      default:
         year = -1;
         break;
      }

      return year;
   }

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

      System.out.println("Enter integer value for day of the week: ");
      int day = scanner.nextInt();
      printDay(day);

      System.out.println("Enter student's classification: ");
      System.out.println(printYear(scanner.next()));

      scanner.close();
   }
}
```

Here's what's happening:

- In the main method, we first read in an integer from the user, and we store it in the variable `day`. Next, we call our function `printDay()` with `day` as an input parameter.
- In our `printDay()` method, we utilize a switch statement. The variable that is being switched on is `day`. In each case, we compare the value of `day` to the case number (first, we compare `day` to 6, then we compare it to 1, and so on).
- If `day` is equal to 6, then we set `answer` equal to `Saturday`, and we break out of the switch statement. If `day` is equal to $1, 2, 3, 4,$ or 5, then we set `answer` equal to `Weekday`, and we break out of the `switch` statement.
- If `day` has any other value, then we set `answer` to `Invalid day value`, and we break out of the `switch` statement.
- Finally, we print out the value of `day`, and we print out the `answer` string that we stored.

The `printYear` function works in a similar manner; however, it demonstrates that `switch` statements don't only have to be used with integers — they can be used with strings as well.

20 Monday, October 14, 2019

Today, we'll start talking about arrays.

Arrays

Suppose we have a `Student` class, and we want to keep track of three students. From what we've learned so far, we could just initialize three `Student` variables. But what happens if we need to keep track of 100 students? Or 10,000 students? Clearly, our current method is not feasible. This is where arrays come into picture.

An **array** in Java is an object used to store a fixed-size sequential collection of elements of the same type. The benefits of an array is that we can treat an array as a single entity, but we can also access each of the individual elements that are stored inside of the array.

How do we declare an array? We can declare an array of a given type with the general format:

```
type[] variablename;
```

For example, we can declare an array of `int`s named `arr` by simply writing `int[] arr;`. Similarly, we can create an array of characters by writing `char[] arr;`.

How do we initialize an array? We use the `new` keyword, we respecify the type, and we finally specify the size of the array. The general syntax is provided below:

```
type[] variablename = new type[size];
```

For example, if we want to declare an array of 10 integers, then we can write,

```
int[] arr = new int[5];
```

The code above would initialize an array of 5 `int`s, and we would now have 5 `int`s at our disposal. By default, the values of an `int` array are all initialized to 0. We view an array as a a contiguous block in memory, so we might visualize `arr` as follows:

$$\boxed{0}\ \boxed{0}\ \boxed{0}\ \boxed{0}\ \boxed{0}$$

We can access each of the individual `int` values in our array by writing the name of the array, followed by square brackets, with a number corresponding to the entry that we wish to access (starting from 0). For example, we can change the first value in our array by writing `arr[0] = 5;`, and we can change the last element in our array as `arr[4] = 7;`. Afterwards, our array will look like the following:

$$\boxed{5}\ \boxed{0}\ \boxed{0}\ \boxed{0}\ \boxed{7}$$

The process of accessing an individual element in an array is called **indexing**.

Here's some code illustrating some of the nice things that we can do with arrays:

```java
public class ReadValues {

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

    System.out.println("How many values would you like to store: ");
    int size = scanner.nextInt();

    int[] nums = new int[size];

    for (int i = 0; i < nums.length; i++) {
      System.out.println("Enter value " + (i + 1) + ": ");
      nums[i] = scanner.nextInt();
    }

    System.out.println("Here are the values you entered: ");
    for (int i = 0; i < nums.length; i++) {
      System.out.println(nums[i]);
    }

    scanner.close();
  }
}
```

A summary of what's going on in this code is presented below:

- Firstly, we ask our user to enter the number of values they want to store. We read in this value, and we initialize an array with that size.
- Next, we read in the corresponding number of values by using a for-loop. In each iteration of the for-loop, we ask the user to enter the next value, and we store it into the next entry in the array. Note that our array has a built-in `.length` field that we can access. This is simply an integer variable that tells us the size of the array.
- Finally, we use another for-loop to iterate over our array. In the `i`th iteration of the for-loop, we print `nums[i]`, which is the `i`th element in our array.

It's important to note that arrays start at 0. This means that, if we're storing 10 elements, then it's only valid to access the array at the indices $0, 1, 2, \ldots, 9$. What happens if we try to access an element that's out of bounds? An exception is thrown, and our Java program will abort if we don't have a `try-catch` block.

To summarize, there are three different situations that we've learned so far in which we can use the square brackets `[..]`.

1. Firstly, we use square brackets when we're trying to declare an array (e.g. `int[] arr;`).
2. We also use square brackets when we're initializing an array (i.e. `arr = new int[10];`).
3. Finally, we also use square brackets when indexing an array (i.e. `arr[0]` accesses the first element, `arr[1]` accesses the second element, and so on).

Copying Arrays

Consider the following code segment:

```java
public class Example {
    public static void main(String args[]) {
        int[] a = new int[5];
        for (int i = 0; i < 5; i++) {
            a[i] = i;
        }
        int[] b = a;
    }
}
```

First, we initialize a variable `a`. Subsequently, we use a for-loop, and we set `a[i]` equal to `i` for $i = 0, 1, 2, 3, 4$. Next, we create a new variable called `b`, and we set it equal to `a`.

Would this copy the elements of `a` into `b` (does `b` contain the elements $0, 1, 2, 3, 4$)? The answer is no — this just makes `a` and `b` aliases for each other. This is similar to creating a `String` variable, and setting a new `String` variable equal to it.

So how do we copy the contents of `a` into `b`? We need to use a for-loop and iterate over the elements of `a`, while assigning the `i`th element of `a` to `b[i]`. This is shown below:

```java
public class Example {
    public static void main(String args[]) {
        int[] a = new int[5];

        for (int i = 0; i < 5; i++) {
            a[i] = i;
        }

        int[] b = new int[5];
        /* Copy contents of a into b. */
        for (int i = 0; i < a.length; i++) {
            b[i] = a[i];
        }
    }
}
```

21 Wednesday, October 16, 2019

Today, we'll continue talking about arrays.

Resizing Arrays

When we initialize an array with a fixed size, this size cannot be changed. For example, the following code will not compile:

```
public class Example {
   public static void main(String args[]) {
      int arr[] = new int[10]; /* Declare an array of size 10. */
      arr.length += 5; /* THIS DOESN'T WORK! */
   }
}
```

We cannot increment the `length` variable associated with an array; it is declared as a constant variable with the `final` keyword. So, how do we resize an array? We need to create a completely new array, and copy over the elements.

Arrays of References

References

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