Lecture 3 - Booleans, Equality, and Methods

Bentley James Oakes

January 17, 2018

Assignment 1

- Due **January 31st** now
- Start as soon as you can

- Please ask for help if you need it
- Discussion forum, TAs, my office hours (after class)

This Lecture

Recap

- Talked about variables and calculations
- Emphasized step-by-step nature of programming
- Introduced typing of variables and operations in Java

Section 1

Types

Type Agreement

- Operations in Java have to agree on types
- Most of these rules are straightforward

Assignment

- *int* values can be stored in *double* variables
- *double/String/boolean* values cannot be stored in *int* variables

Math Operators

- For multiplication * and subtraction -, and addition +
- Between two *int* values, the result is an *int*
- If there's a *double* value, then the result is a *double*
- Multiplication and subtraction can't be performed on *String* or *boolean* values

Confusion

Two examples of confusion:

- The plus operator +
  - Used for both addition and concatenation
- The division operator /
  - Used for both division and integer division

Concatenation vs Addition

- Java has consistent rules about what the + sign means
- But it can be confusing

```
System.out.println(3 + 5);
```

Prints 8 - Addition

```
System.out.println("3" + "5");
```

Prints 35 - Concatenation

```
System.out.println("3" + 5);
```

Prints 35 - Concatenation

Concatenation

```
System.out.println("3" + (1+4));
```

Prints 35
Calculation then concatenation

```
System.out.println(3 + 5 + "7");
```

Prints 87
Calculation then concatenation

Rule: If there's a *String* on either side of the + sign, then it's concatenation. The order of evaluation is left-to-right.

Division vs Integer Division

- Java can be confusing with the / sign
- When it is between two ints, the result is an int
- Otherwise, it will produce a double value

```
double w = 99.0/25.0;
```

Result: 3.96 - Division

```
double x = 20/30.0;
```

Result: 0.666 - Division

```
int y = 99/25;
```

Result: 3 - Integer Division

```
int z = 3/4;
```

Result: 0 - Integer Division

```
int h = 4/2.0;
```

Result: Error - Right side is double

Integers vs Doubles

- Very common question:
- Why don't we use *doubles* for everything?
- About 40% slower to use *doubles* in math operations
- Doesn't make sense to store age of a person in a decimal number

```
double tenCents = 0.10;
double account = tenCents + tenCents + tenCents;
System.out.println(account);
System.out.println(account==0.3);
```

Result: 0.30000000000000004
This is not equal to 0.30!

Section 2

Booleans

Comparison

- Often in programs we need to make comparisons and tests
- For example, if you are over a certain age, you get a senior's discount
- We could write this in English as:
  - Is your age over 65?
- And this will give either a *true* or *false* result

Comparison Operators (For Numbers)

Recall these from math class

- $>$
  - Greater than
  - $4 > 3$ is *true*
- $<$
  - Less than
  - $5 < 2$ is *false*

- The alligator must be eating the bigger number for the **expression** to be *true*

And Equal

- We also have
- $>=$
  - Greater than or equal
  - $4 >= 3$ is true
  - $4 >= 4$ is also true
- $<=$
  - Less than or equal

Booleans

- So we are dealing with tests that give a *true* or *false* answer
- In programming, these values are called **Boolean values**
- And the calculation is called a **Boolean expression**
- Therefore, they are stored in **Boolean variables**

```
boolean b = 4 > 3;
System.out.println(b);
```

- Here, a **Boolean variable** stores the result of a **Boolean expression**
- And the type of the variable is **boolean** (lower-case b)

Testing Variables

```java
public class TestingVars
{
    public static void main(String[] args)
    {
        //a meal is 45 dollars
        int mealPrice = 45;

        //is mealPrice less than 10?
        boolean mealIsCheap = mealPrice < 10;

        System.out.println("The meal is: $" + mealPrice);
        System.out.println("This meal is cheap: " + mealIsCheap);
    }
}
```

```
The meal is: $45
This meal is cheap: false
```

Note that we can use `System.out.println();` to print out the value of a boolean variable

Section 3

Mod Operator

Remainders

- Let's take detour for a second
- How do we know if a number is odd or even?
- What's special about 2 and 4 where they are even?

- They are divisible by 2
- That is, when you divide 4 by 2, there is no remainder
- Wouldn't it be nice if there was a remainder operator so we could test odd/even?

Mod Operator

- The remainder operator is called 'mod'
- Its symbol is the number sign: %

Examples:

- $2/2 = 1$
  - 2 divided by 2 gives 1
- $2\%2 = 0$
  - The remainder of 2 divided by 2 is 0

Mod Examples

Let's look at examples mod 2

- 14 divided by $2 = 7$
- $14\%2 = 0$

- 19 divided by $2 = 9.5$
- $19\%2 = 1$

- If the remainder of dividing a number by 2 is 0,
- then the number is even
- otherwise, the number is odd

- Now that we have the mod operator to get remainders, we need to test if the remainder is 1 or 0

Section 4

Equality

Equality

$== \text{ Test if equal}$
$! = \text{ Test if unequal}$

Examples:

- $3 == 3$ is true
- $3! = 3$ is false
- $4 == 3$ is false
- $4! = 3$ is true

Testing Equality Example

```
int x = 3;
boolean isThree = x == 3;
System.out.println("X is three: " + isThree);
```

```
X is three: true
```

Careful!

```
int x = 3;
boolean isThree = x == 3;
System.out.println("X is three: " + isThree);
```

- = and == look alike, but do different things
- = is **assignment**
- == is **testing**
- You will have errors if you use them incorrectly

Remember:

- = is one word **assigned**
- == is two words **is equal**

Finishing Up Our Odd/Even Test

- To test for evenness
- Get the remainder of the value when dividing by 2
  - Calculate the value mod 2
- Test if this remainder is equal to 0
- If this test is true, the value is even

```
int remainder = 5 % 2; //gives 1
boolean isEven = remainder == 0; //gives false
System.out.println("5 is even: " + isEven);
```

Comparison Types

- So an expression like `4 > 3` compares two ints
- And the result is a Boolean value

- An expression like `9.5 == 6.2` compares two doubles
- And the result is a Boolean value

Can we compare values of different types?

Comparing Different Variable Types

- Comparing different variable types works as you would expect
- $3.5 < 4$ is true
- $3 == 3.0$ is true
- $10.0 == true$ doesn't make sense and won't compile
- 5 ==“5” doesn't compile
  - We can't directly compare these types
  - 5 is an int, and “5” is a String

Boolean Expressions

```java
public class BooleanExpressions
{
    public static void main(String[] args)
    {
        System.out.println(3.5 < 4); //true
        System.out.println(2 == (3-1)); // true

        System.out.println(3 == 3.0); // true
        System.out.println(0.1 != 0.1); // false

        //System.out.println(5 == "5"); //Compiler error
        //System.out.println(5 != "5"); // compiler error

        System.out.println(true == false); // false
        System.out.println(true == true); // true
        System.out.println(true != false); // true

    }
}
```

Comparing Strings

- Comparing Strings is very different
- Strings have a special algorithm for comparing them:
  - The `.equals()` method
  - Let's take a look at methods

Section 5

Methods

Methods Summary

- Methods are pieces of **reusable code**
- They take **parameters** as input
- They **return** a value as output
- They can contain any number of instructions within
- Therefore, a method is just an algorithm

System.out.println Method

A method you've already seen: `System.out.println();`

This method is **called** on the `System.out` variable (this is a special variable), and does not **return** anything. Note that this method will accept ints, doubles, Strings, or booleans.

```
System.out.println("I'm a String!");
System.out.println(123);
System.out.println(4.56);
System.out.println(true);
```

Math Methods

The next few slides will show a number of math methods
These are selected because it's easy to understand what they do, and the types they accept and return.

Math.sqrt

Another good example of a method:

Math.sqrt()

This method takes one double value, and returns a double value.
"Returns the correctly rounded positive square root of a double value."

Math.sqrt Use

Here we are **calling** the sqrt method with the second instruction:

```
//create x
double x = 123;

//get the square root of x
//using the Math.sqrt method
double y = Math.sqrt(x);

//print out the answer
//gives 11.09
System.out.println(y);
```

We then store the **return** value of the method in a variable.

Math.sqrt

Note that Java will automatically make sure that the call has the correct type

For example, we can't call the Math.sqrt() method and pass it a String

```
12        double y = Math.sqrt("hello");
13
14
```

Interactions | Console | Compiler Output

```
File: /home/dcx/Dropbox/COMP 202/Lecture 3 -
Methodical/MathMethods.java  [line: 12]
Error: incompatible types: java.lang.String cannot
be converted to double
```

Documentation

How do we know how to use the Math.sqrt() method?
We can look at the official documentation online:
`https://docs.oracle.com/javase/9/docs/api/java/lang/Math.html`

```
double   sqrt(double a)
```

Returns the correctly rounded positive square root of a `double` value.

Notice the four parts:

- The **return type**
- The method name
- The **parameters** the method accepts
- A brief description/purpose of the method

Analogy for Methods

- Remember the pinata cake recipe?
- The instructions said ‘Make the chocolate cake mix according to the recipe on the boxes”
- This is similar to a call to a method
- So methods are mini-recipes

Why Use Methods?

Why do we use methods?

- Makes it easy to divide our program into smaller algorithms
- Reusing the Math.sqrt method allows us to avoid re-writing it
- We hide all the unnecessary details

Repeated Use

```
double x = 454549;

double y = Math.sqrt(x);

double z = Math.sqrt(y);

System.out.println(x + " " + y + " " + z);
```

Note that we are calling the Math.sqrt method multiple times
Instead of figuring out repeatedly how to get a square root, we can reuse the same instructions.

Math.pow

Another documentation example

```
double   pow(double a, double b)
```

Returns the value of the first argument raised to the power of the second argument.

Note that we have all the information we need to call this method.
Note that this method is in the Math **class**, which we'll talk about later

```
double value = Math.pow(2, 6);
System.out.println(value);
```

Gives 64.0

Nested Method Calls

The Math.sin() and Math.cos() method calls take a double value, and return a double value
You could write this:

```
double x = 0.5;
double c = Math.cos(x);
double s = Math.sin(c);

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

The above way is easy to read, but you could also write:

```
double s = Math.sin(Math.cos(0.5));

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

Why does this work? The cos method call takes a double and returns a double. The sin method call then takes that returned double value and returns another double.

Looking at the .equals() Method

```
//create a String
String s = "hello";

//check if that String equals something
boolean b = s.equals("hello");

//print out the result
System.out.println(b);
```

Here we are testing if the variable equals the String literal "hello"
Note how it is the variable's name followed by `.equals()` with the String literal inside the `.equals()` brackets

.equals() Method Types

Note that there are types involved with this method.

```
//create a String
String s = "hello";

//check if that String equals something
boolean b = s.equals("hello");

//print out the result
System.out.println(b);
```

This method is called on a String variable, it **accepts** or **is passed** a String, and it **returns** a boolean value of *true* or *false*.

More .equals() Method

We can test two String variables, to see if one equals the other

```
String first = "Hello";
String second = "Hello";
String third = "He";

boolean firstPair = first.equals(second);
boolean secondPair = second.equals(third);

System.out.println("The first pair are equal: " + firstPair);
System.out.println("The second pair are equal: " + secondPair);
```

The first pair are equal, and the second pair are not

We say that we are **calling** the equals method on the first variable, and **passing** the second variable

== vs .equals()

We can't use == to test String equality

```
String phrase = "Hello World";

String world = "World";
String test = "Hello " + world;

boolean areStringsEqual = phrase.equals(test);
boolean areEqual = phrase == test;
```

The first boolean stores true (they both contain *Hello World*)
The second boolean stores false (for reasons we'll see later)

Rule: Use the `.equals()` method for comparing Strings, == for everything else

Section 6

Writing Methods

Dividing Up Our Program

To make our program easier to understand and test, we break up the larger program into smaller algorithms

"I recommend that a novice trying to making something like the tart think of it not as one elaborate recipe but as a series of simpler preparations - a custard, a pastry, and a glaze.
Broken down into its component parts, any recipe will appear less intimidating and more manageable." - Kitchen Wisdom

Four Parts of a Method

Recall what the documentation tells you about a method:

- The return type
- The method name
- The **parameters** the method accepts
- A brief description or purpose
  - To be placed in comments to let others know what this method does

Let's apply this to a method that says "Hello World!"

- **Return type** - void (it returns nothing)
- **Method name** - sayHello
- **Parameters** (what the method accepts) - nothing
- **Description/Purpose** - "This method prints hello world."

Method Example

```
//This method prints hello world.
public static void sayHello()
{
    System.out.println("Hello World");
}
```

- The 'public static' is needed for now
- **Return type** - void (it returns nothing)
- **Method name** - sayHello
- **Parameters** (what the method accepts) - nothing
- **Description/Purpose** - "This method prints hello world."

Using the sayHello() Method

- Note that this is a method, just like the main method
- Both the sayHello method and the main method have to be inside the class
  - The order doesn't matter but usually the main method goes at the top or bottom
- Don't put another method inside the main method

```java
public class SayHelloClass
{
    public static void main(String[] args)
    {
        sayHello();
    }

    //This method prints hello world.
    public static void sayHello()
    {
        System.out.println("Hello World");
    }
}
```

Using the sayHello() Method

- To use the sayHello method, we call it from the main method
- Note that Java starts executing instructions inside the main method
- If the sayHello method is not called, it will not execute!

```
1  public class SayHelloClass
2  {
3      public static void main(String[] args)
4      {
5          sayHello();
6      }
7
8      //This method prints hello world.
9      public static void sayHello()
10     {
11         System.out.println("Hello World");
12     }
13 }
```

Using the sayHello() Method Repeatedly

- Just like calling other methods, we can call the sayHello method repeatedly
- This will print out *Hello World* twice:

```java
public class SayHelloExample
{

    public static void main(String[] args)
    {
        //call the sayHello method
        sayHello();

        //call the sayHello method again
        sayHello();
    }

    //prints out a greeting
    public static void sayHello()
    {
        System.out.println("Hello World");
    }
}
```

Execution Location

- Note that when we do this, Java is executing instructions in both the main method and the sayHello method
- We say that **control** is switching back and forth
  - The main method is executed first
  - The sayHello method is called
    - *Hello World* is printed
  - The main method goes to the next line
  - The sayHello method is called
    - *Hello World* is printed
- **Always step-by-step!**

```java
public class SayHelloExample
{

    public static void main(String[] args)
    {
        //call the sayHello method
        sayHello();

        //call the sayHello method again
        sayHello();
    }

    //prints out a greeting
    public static void sayHello()
    {
        System.out.println("Hello World");
    }
}
```

Another Method

Let's write another method. Let's start with this description:
"Write a method addNumbers. This method should accept two int values as parameters. This method should return an int value which is the sum of the inputs."

Always identify the four parts!

- Method name - addNumbers
- Input - Two integers
- Output - An integer
- Purpose - To add the two input parameters.

A method is just a named algorithm

Another Method

The four parts turn into the first line of the method:

- Method name - addNumbers
- Input - Two integers
- Output - An integer
- Purpose - To add the two input parameters.

```
public static int addNumbers(int a, int b)
```

This first line is called the **method header**
Always always figure out the method header first!

Variable Names

```
public static int addNumbers(int a, int b)
```

Why is it int a and int b?

You need to give the input parameters names so you can refer to them in the method's instructions.
Just like other variable names, these names can be anything (reasonable)

Another Method Example

We add in the calculation of the result, but...

```
3     public static int addNumbers(int a, int b)
4     {
5         int c = a + b;
6         //??????
7     }
8 }
```

Interactions | Console | Compiler Output

```
1 error found:
File: /home/dcx/Dropbox/COMP 202/Lecture 3 -
Methodical/Code/Test.java  [line: 7]
Error: missing return statement
```

Compiler
JDK 8.0-op

If we run this, Java complains that we said this method produces an int, but it doesn't know to **return** c

Return Statement

```
public static int addNumbers(int a, int b)
{
    int c = a + b;
    return c;
}
```

- The return statement says which variable to output
- We can only output one variable, and the type must match the header

Calling the Method

- Let's look at the header again

```
public static int addNumbers(int a, int b)
```

- This methods takes two ints as its parameters, and outputs an int
- We use this method by writing:
  - `int x = addNumbers(56, 34);`
  - We must pass two ints to addNumbers, and store the result in an int variable

Main Method

- Let's look at the main method header

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

- This is a method that does not return anything
- And has one parameter called *args*

String[] args

- These are parameters passed to the program
- This is used on your assignment
- You type *run Calculator 5 5 1* to enter the arguments
- Then some methods turn the Strings into ints and doubles

```
public static void main(String[] args)
{
    System.out.println("First argument: " + args[0]);
```