Data Types, Vars & Arithmetic

Ryan Nixon

Variables

- Every program uses data, and this data needs to be stored somewhere

- Java uses variables to store information in 3 different locations, the class level, the method level and the instance level. We'll see examples of the first two today

- Think of a variable as a named location in memory that contains data and can be retrieved for use at any time

Variables

- All variables must be declared with a data type before they are used. Declaring tells Java how much memory to use when storing the information and what type of information it is

- The declaration generally looks like this:
  Type variable_1, variable_2 ...;

- So to declare an Integer (a whole number) might look like this:
  `int firstVariable, secondVariable;`

- Note that the type of variable will always come first, followed by the name. You may declare additional variables of the same type by separating their names with commas

Variable Identifiers

- Variable identifiers are the names given to your variables. Some limitations are placed on what names are allowed as well as some best practices

- Identifiers may only contain alphanumeric characters (a-z, A-Z, 0-9) and the underscore ‘_’ symbol. Java also supports the $ symbol but this is discouraged

- Identifiers should also have descriptive names according to how they are to be used. Avoid single character names that you might use in math such as x, y, & z. Java has no limitation on the length of the name so better choices might be ‘currentAmount’, ‘errorMessage’, or ‘toReturn’

- Note that each word in these variables are title case, sans the first word which is lower case. This is known as “camel case” and is a common variable naming convention

Primitive Data Types

- There are a number of different data types that can be declared

- This lecture will cover what are known as the primitive types. These types are written in lowercase and are the most basic that variables can get in Java

- Each data type has its own formatting, storage requirement and size limitation. You will want to commit to memory the **byte**, **int**, **double**, **char**, and **bool** data types at the least since they are used often.

Primitive Data Types

Type Name	Kind of Value	Memory Used	Range of Values
`byte`	Integer	1 byte	–128 to 127
`short`	Integer	2 bytes	–32,768 to 32,767
`int`	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
`long`	Integer	8 bytes	–9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
`float`	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
`double`	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
`char`	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
`boolean`		1 bit	True or false

Assigning Values

- To declare these you write their type name along with the variable name:

```
double currency;
int amount, currentCount;
char prefix;
bool outcome;
```

- You can then use them in your program after the declaration using an assignment operator to initialize the variables to a specific value:

```
currentCount = 0;
currency = 5.04;
prefix = 'Z';
outcome = false;
```

- Assignment operators always take this form with the variable on the left-hand side of an equals sign and the value being assigned to it on the right-hand side. As you'll see this value doesn't have to be simple, it may be the result of an expression or an entire section of code

Assigning Values

- Keep in mind the different formats for initializing the variables. Integer types contain whole numbers, floating-point types (float & double) use decimal points, characters use single quotations and booleans use true/false

- Variables by definition vary, so they may be assigned as many times as you need in a program. For example, to increment a counting variable you might write:

```
count = count + 1;
```

- This takes the current value of ‘count’, adds 1 to it, then stores the new value at the same location in memory. This arithmetic follows the same rules that you would expect including order of operations

Order of Operations

Ordinary Math	Java (Preferred Form)	Java (Parenthesized)
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1 / (time + 3 * mass)`	`1 / (time + (3 * mass))`
$\frac{a - 7}{t + 9v}$	`(a - 7) / (t + 9 * v)`	`(a - 7) / (t + (9 * v))`

Order of Operations

- One symbol not mentioned in the previous slide is the modulus “%” operator. This symbol outputs the remainder of a division operation:

```
int modResult = 15 % 8; //Produces 7
int modResult = 25 % 8; //Produces 1
```

- The modulus operator is often overlooked when working with simple problems. For example, to determine if a number is odd or even, mod by 2:

```
int isOdd = 3 % 2; //Produces 1 (true)
int isOdd = 4 % 2; //Produces 0 (false)
```

- You should also remember the modulus if you encounter counting problems. An example of this is on the previous page where each odd row has been ‘striped’ with a background color

Assignment Operators

- The = sign isn't the only assignment operator in Java. There are a few other operators that serve as shorthand for modifying a variable
- `Var1 += 5` adds 5 to Var1
- `Var1 -= 5` subtracts 5 from Var 1
- This also works for multiplication (*=) and division (/=)
- Changing a value by 1 is very common in programming (i.e. counting). There are special operators for this as well
- `Var1++` increments Var1 by 1
- `Var1--` decrements Var1 by 1

Operators to Avoid

- It is highly discouraged to use these increment/decrement operators in the middle of an expression as their behavior may be unintuitive:

```
int count = 10;
int result = 5 + count--; //Result = 15, count = 9
int result = 5 + count++; //Result = 14, count = 10
```

- To make this worse, alternate operators are available that swap the order of execution in the expression:

```
int count = 10;
int result = 5 + --count; //Result = 14, count = 9
int result2 = 5 + ++count; //Result = 15, count = 10
```

- It's best to avoid these misleading operators altogether and use the increment/decrement operators only on their own lines.

Approximate Numbers

- Floating point numbers deserve some additional attention. Because floating point numbers represent fractions they suffer from a storage limitation

- For example, if a float was used to store the fraction 1/3, it would need to store the number 0.3333... with the 3 repeated to infinity

- Memory doesn't like infinity. The number would be stored as an approximation, getting as close as possible to the real value. This would end up being closer to 0.33333333333333000000000

- This problem affects all floating point numbers and often manifests itself when performing multiplication or division on one. To that end, never use floats or doubles when dealing with currency

Named Constants

- There are times in a program where you need to store a value but want that value to never change. Variables wouldn't work well for this (since they vary). Java provides named constants to serve this need instead

- Using a constant is a class-level (static) variable, but with the keyword `final` added

```
public static final int DEFAULT_VOLUME = 4;
```

- Since it is class-level and public, this constant is visible throughout the entire program. However, since it is final, it can not ever be modified again after declaring it which keeps it safe from outside influence

- It is also a good practice with constants to capitalize the entire identifier and separate spaces between words with underscores. This helps differentiate unchangeable constants from dynamic variables

Type Casting

- To force a variable to change its type, you place the new type in parentheses before that variable's name:

```
double exactMileage = 34.257;
int estMileage = (int)exactMileage;
```

- Type casting is only needed when you are reducing the specificity of the variable, such as reducing a double to an int

- Note that when you cast a double to an integer the numbers after the decimal point are truncated, not rounded

Implicit Casts

- ```
  int source = 6;
  double dest = source / 5; //Produces 1 rather than 1.2
  ```

- One thing you will run into is the behavior where division does *not* automatically cast when between two integers

- Because of this, even if you are storing the final result in a floating point variable you will still have to cast at least one of the two integers for them to divide properly

- You can do this by casting one of the variables or, if you are using a numeric literal, you can ensure that it is already a floating point value. You do this by using the floating-point format (use a decimal, even if it's x.0):
  ```
  int source = 6;
  double dest = source / 5.0; //Produces 1.2
  dest = (double)source / 5; //Also produces 1.2
  ```
  ```

Practice is the Best Teacher

- Take some time now to play with the different operators and with the compiler

- I'll be here if you have any questions or if you'd like something demonstrated

- Some ideas to start you off:

  - Converting fahrenheit to celsius

```
double cel = (fah - 32.0) * 5 / 9;
```