Lesson 2A

Data

Data Types, Variables, Constants, Naming Rules, Limits

Based on the O(N)CS Lesson Series
License for use granted by John Owen to the University of Texas at Austin, Center for STEM Education
Table of Contents

- Objectives
- Data types and expressions
- Variables and constants
- Memory storage
- Lesson Summary
Objective #1

• The student will understand the basics of standard data types, and how to create, initialize, declare, assign and output variables and constants.
Objective #2

- The student will also gain an in-depth understanding of the numerical representations and limits of integers and decimals.
DATA TYPES AND EXPRESSIONS

• In the previous lesson group, the three most commonly used types of data were introduced:
  • Strings
  • Integers
  • Decimals
Primitive data types – int and double

- The official JAVA data type designation for an integer is *int*, and for a decimal value, *double*.
- These are called *primitive* data types, because they are *simple*.
- There is a bit more to it than that, but for now this explanation will suffice.
Object data type - String

• A **String** is not a primitive because it is not simple, but more complex.
• Instead it is called an **object**.
• Again, later on we’ll go into more detail regarding the distinction between primitives and objects.
Other primitives - boolean and char

- The two other commonly used primitive data types are `boolean` and `char`.
- The `boolean` data type represents `true/false` values, which are used quite a bit in logic processing, as you will soon see.
Other primitives - boolean and char

• **char** is short for character *(rhymes with “car”), and can be any single character you find on the keyboard, plus a few more as you will see later on.*

• **A complete listing of all Java primitives is on the next page.**
All eight primitive data types

Integer types:
   byte, char, short, int, long

Decimal types:
   float, double

Boolean type:
   boolean

As your programming grows in sophistication, you will explore these other types, but for now we’ll stick with int, double, char and boolean.
Data expressions – int, double

Each type of data is expressed in its own way.

- **int** values are expressed as whole numbers like 7 or –83.
- **double** values are expressed as decimals like 5.95 or –12.0
Data expressions – String, char

- **String** values are always enclosed within double quotes, like "Hello World", "A", or even " "", the empty string.
- **char** values are always enclosed within single quotes, like 'A', '7', or '$'.

Lesson 2A - Intro to Data Types, Variables and Constants

Identifier Creation Rules, Range and Precision Limits
'A' is NOT "A"

• Note that 'A' is NOT the same as "A". The first is a char (inside single quotes), and the other a String (inside double quotes).
• The more significant difference is in how they are stored in the computer’s memory (more on that later).
Data expressions – char, boolean

- A **char** is never empty! It must always contain a single character.
- A **blank space** is a character and can be contained in a **char** variable…’ ’.
- **boolean** values are simply the two words **true** and **false**; they are NOT enclosed in quotes, and they are NOT Strings.
VARIABLES and CONSTANTS

Let’s talk more about variables and constants:

• How to create and assign values to them
• How to output them
• What they really are
• Some limits
Outputting data expressions

In an earlier lesson, you learned how to output literal values like this:

• `System.out.println("Hello");`
• `System.out.println(45);`
• `System.out.println(3.14);`
• `System.out.println('A');`
• `System.out.println(true);`
You also got a brief glimpse into the world of variables, where data literals can be stored in memory, and then used in output statements. Here are some examples:

• `String name = "John Owen";`
• `int age = 55;`
• `double wage = 54.65;`
• `char initial = 'B';`
• `boolean sailor = true;`
Initializing variables

```java
String name = "Julie Jones";
```

There are five parts to a variable initialization statement:

- The **data type**
- The **identifier**
- The “=“ sign (called the **assignment operator**)
- The **actual value**
- A semi-colon to mark the end of the statement.
Separate **declare** and **assign**

It is also possible to separately declare, and then assign variables.
The five parts are still there, but you must restate the identifier in the second part.

- **String** name;  //declare
- name = "John Owen";  //assign
Separate **declare** and **assign**

You can actually do this with a constant when you first create one, but once assigned the first time, it cannot be changed later \(\textit{more about constants later in this lesson}\). The word "**final**" at the front is what makes it a constant.

• final String name;
• name = "Julie Jones";
Here is how you would output the variables:

- System.out.println(name);
- System.out.println(age);
- System.out.println(wage);
- System.out.println(initial);
- System.out.println(sailor);
This program demonstrates how to **initialize** (declare and give beginning values to) and output variables.
Variables are memory locations

Variables are simply *locations in the computer’s RAM*, or memory.

When you declare a variable, like

```java
int age = 55;
```

the JAVA compiler finds some available RAM memory, carves out enough room for an `int` value, marks that memory location with the *identifier* (variable name) `age`, and places the value `55` into that memory location.
Primitives vs objects

```java
int age = 55;
```

- The main reason this is called a *primitive* is that the memory location actually contains the value.
- Objects are stored in a different way than primitives...see next slide.
Object storage

This String object is **not** stored like a primitive. The memory location indicated by the variable does not contain the String itself, but instead references another memory location, which DOES contain the String. This may seem odd, but later on you will see more clearly a good reason for it.

```java
String name = "John Owen";
```
Memory Locations...

Other primitive types work in the same way as an `int`, in that the memory location actually stores the data value.

Each type of data requires different amounts of memory to store...more on that later in this lesson.

- `double wage = 54.65;`
- `char initial = 'B';`
- `boolean sailor = true;`
Constants

Constants work just like variables, in that they store values in memory.

- final String NAME = "John Owen";
- final double WAGE = 54.65;
- final char INITIAL = 'B';
- final boolean SAILOR = true;
The difference is that once they are given a beginning value in the execution of a program, they CANNOT BE CHANGED!

- `final double WAGE;`
- `WAGE = 54.65;`
- `WAGE = 55.65;`

Notice the stand-alone assignment statement for this constant. This is legal to do the first time it receives a value, but you can’t change it later in the program.
Before we proceed to a discussion of memory requirements, let’s discuss some basic rules for creating *identifiers*. *Identifiers* are simply words YOU create as a programmer to represent variables, constants, methods and class names, or labels.
Identifiers can be created for just about anything you need in programming, but there are certain restrictions on how to do this. The main restriction is that Java reserved words, what some call “*magic words*”, CANNOT be used as identifiers, for obvious reasons...
The obvious reason is that the compiler would not consider the reserved word as an identifier, but would instead try to use it as a command, get very confused, and throw a “fit” (a compile error). Now, let’s move on to what you CAN do to create identifiers…
Rules for creating JAVA identifiers

1. Always use descriptive names for identifiers. If you are creating a variable to store total pay, then name it something like `double totalPay;`

2. Avoid non-descriptive single letter identifiers, like `double a;` or `int x;` (for utility purposes, such as loop control, these are generally OK to use…more on this later)

3. Always start with a letter or an underscore, never a digit.
Rules for creating JAVA identifiers

4. Use only letters, digits, and the underscore character…*no symbols allowed!*

*Exception: String identifiers may contain the ‘$’ character, anywhere - beginning, middle, or end*

5. Do not use spaces, but instead use the underscore if you want separation between words.
Valid and invalid examples

Can you tell which of these are valid identifiers?

- name
- $name
- My name
- _address
- #amount
- num1
- 1st_num
Valid and invalid examples

Answers

• name OK
• $name OK for String only
• My name not OK...has a space
• _address OK
• #amount not OK...has a symbol
• num1 OK
• 1st_num not OK...starts with digit
There are several agreements or “conventions” in programming related to creating identifiers which, although not required, are strongly suggested.

Some software development companies DO require these conventions, perhaps others, and strongly enforce them, or you could get fired!

We will follow these conventions, shown on the next few slides.
• **Multiple word** identifiers will lowercase the first word and capitalize all remaining words, or put underscore characters between words

• **Examples:** myName, hourlyWage, startTime, my_street_address

• Uppercasing words in the middle of an identifier is called **CamelCase**
Agreements or “conventions”

- Variable and method identifiers will always begin with lowercase letters, like `sum` or `getName`.
- Constant identifiers will always be in all CAPS, like `PI` and `MAX_VALUE`.
- Class identifiers will always be capitalized….the `String` and `System` classes are two you have already seen.
Agreements or “conventions”

- **Multiple word** constant names will capitalize everything and separate words with the underscore character.

- Examples: **MY_WAGE**, **ZIP_CODE**, **COMPANY_NAME**
Examples of multi-word variables and constants

number1
streetAddress
hourlyWage
surface_area
PI
NICKEL_VALUE
E
MAX_VALUE
And now it is time to talk about the various memory requirements and limits of the different data types, specifically integers and decimals.
Bit – binary digit

One bit, or binary digit, of memory is the smallest unit of memory in a computer. It is an electronic signal or state, either around 5 volts in strength or virtually zero volts. Typically a bit is represented as a 1 (5 volt signal) or a 0, also sometimes represented as on/off, or true/false.
Binary system, boolean logic

These two values, 1 and 0, (true and false) are the foundation of the **binary number system** and the **Boolean logic** field of mathematics, on which all of computing and logical processing is based. We’ll explore the **binary numbers** and **Boolean logic** systems in much greater depth later on.
Bits, bytes, etc.

- Memory consists of bits, bytes, kilobytes (KB), megabytes (MB), gigabytes (GB), terabytes (TB), petabytes (PB), exabytes (EB), zettabytes (ZB), yottabytes (YB), etc.…
- Bits are typically grouped in series of 8 bits called **bytes**.
Bits, bytes, etc.

• An int variable requires 32 bits, or 4 bytes of memory to store it.
• On the next page you will see all the integer family of primitive data types along with the memory requirements and range limits for each.
### Memory and range limits – integer family of data types

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>MEMORY SIZE</th>
<th>RANGE LIMITS</th>
<th>MAXIMUM VALUE EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>-128…127</td>
<td>$2^{7} - 1$</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>-32768…32767</td>
<td>$2^{15} - 1$</td>
</tr>
<tr>
<td>char</td>
<td>16 bits</td>
<td>0…65535</td>
<td>$2^{16} - 1$</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>-2147483648 … 2147483647 (approx 2 billion)</td>
<td>$2^{31} - 1$</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>-9223372036854775808 … 9223372036854775807 (approx 9 quintillion)</td>
<td>$2^{63} - 1$</td>
</tr>
</tbody>
</table>
## Memory and range limits – integer family of data types

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>MEMORY SIZE</th>
<th>RANGE LIMITS</th>
<th>MAXIMUM VALUE EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>-128…127</td>
<td>$2^{7}$-1</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>-32768…32767</td>
<td>$2^{15}$-1</td>
</tr>
<tr>
<td>char</td>
<td>16 bits</td>
<td>0…65535</td>
<td>$2^{16}$-1</td>
</tr>
</tbody>
</table>

Although the char data type is technically an integer data type, has minimum and maximum values, and is truly stored as an integer value, it represents characters and is output as letters, digits, and numerous other symbols in the *Unicode system*, which we’ll explore later on.
Each data type has its maximum and minimum values stored in special constant memory locations called MIN_VALUE and MAX_VALUE. Below is how you can output those values.
These values can also be stored in variables and output using the `%d` format specifier of the `printf` output command, as we discussed in Lesson 1.
Out of bounds?

```java
public class limits {
    public static void main (String [] args) {
        byte b = Byte.MIN_VALUE; b--; System.out.printf("%+28d\n", b);
        b = Byte.MAX_VALUE; b++; System.out.printf("%+28d\n", b);
        short s = Short.MIN_VALUE; s--; System.out.printf("%+28d\n", s);
        s = Short.MAX_VALUE; s++; System.out.printf("%+28d\n", s);
        int i = Integer.MIN_VALUE; i--; System.out.printf("%+28d\n", i);
        i = Integer.MAX_VALUE; i++; System.out.printf("%+28d\n", i);
        long l = Long.MIN_VALUE; l--; System.out.printf("%+28d\n", l);
        l = Long.MAX_VALUE; l++; System.out.printf("%+28d\n", l);
    }
}
```

Now look carefully at this code.
After each variable is initialized (declared and assigned a beginning value), it is either decreased by 1 (b--), or increased by 1 (b++), seemingly causing it to “step out of bounds”, beyond it’s normal range limit.
Out of bounds?

```java
public class limits {
    public static void main (String [] args) {
        byte b = Byte.MIN_VALUE;       // System.out.printf("%+28d\n", b);
        b++; System.out.printf("%+28d\n", b);
        short s = Short.MIN_VALUE;     // System.out.printf("%+28d\n", s);
        s++; System.out.printf("%+28d\n", s);
        int i = Integer.MIN_VALUE;     // System.out.printf("%+28d\n", i);
        i++; System.out.printf("%+28d\n", i);
        long l = Long.MIN_VALUE;       // System.out.printf("%+28d\n", l);
        l++; System.out.printf("%+28d\n", l);
    }
}
```

However, observe what really happens!!!

Can you see it??
Wrap around!

In the first line, `b` is assigned `-128`, the minimum value for a byte.
In the first line, b is assigned $-128$, the minimum value for a byte. It is then decreased by 1, which seems like it should be the value $-129$. 

```java
public class limits {

    public static void main (String [] args) {
        byte b = Byte.MIN_VALUE; // System.out.printf("%+28d\n", b);
    }

    // Wrap around!
```
In the first line, b is assigned \(-128\), the minimum value for a byte.

It is then decreased by 1, which seems like it should be the value \(-129\).

However, since that is “out of range” for a byte, the compiler “wraps around” to the other side of the range and assigns it the maximum value instead, which is 127!!!
This phenomenon is true for all the other examples, stepping out of bounds in either direction, referred to as “wrap-around”.

```
public class limits {
    public static void main (String [] args) {
        byte b = Byte.MIN_VALUE; b--; System.out.printf("%+28d\n", b);
        b = Byte.MAX_VALUE; b++; System.out.printf("%+28d\n", b);
        short s = Short.MIN_VALUE; s--; System.out.printf("%+28d\n", s);
        s = Short.MAX_VALUE; s++; System.out.printf("%+28d\n", s);
        int i = Integer.MIN_VALUE; i--; System.out.printf("%+28d\n", i);
        i = Integer.MAX_VALUE; i++; System.out.printf("%+28d\n", i);
        long l = Long.MIN_VALUE; l--; System.out.printf("%+28d\n", l);
        l = Long.MAX_VALUE;
        l++; System.out.printf("%+28d\n", l);
    }
}
```
Study each example carefully to see how this works.
Decimal max and min

The decimal data types also have maximum and minimum values as you can see below, but the precision of the values they can hold is more significant in programming, rather than the max or min values.
Below you can see the limits of the decimal data types. The most significant item and the easiest to observe is the number of decimal places of precision.

On the next few slides, a brief case study is shown to take a closer look at this.

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>MEMORY SIZE</th>
<th>MAX AND MIN</th>
<th>PRECISION LIMITS IN BITS</th>
<th>PRECISION LIMITS IN DECIMAL PLACES</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
<td></td>
<td>23 bits</td>
<td>7 places</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td></td>
<td>52 bits</td>
<td>15 places</td>
</tr>
</tbody>
</table>
Decimal precision – a brief case study

Let’s take a quick look at the two most famous irrational and transcendental numbers in mathematics – PI and E.

**PI** is the ratio of a circle’s circumference to its diameter – $3.1415…$

**E** is the approximate base of the natural logarithms – $2.71828…$
Math.PI and Math.E

The JAVA Math class defines the two values, PI and E, as constants. Each one is a double, defined with 15 places of precision.
The two double variables receive the constant values of PI and E from the Math class.

The two float variables must “cast” the larger memory double values into the smaller memory float values (more on casting in the next lesson).

The result is a clear loss of precision in the storage and output for the float values.
Lesson Summary

• This lesson introduced the basics of standard data types, and how to create, initialize, declare, assign, and output variables and constants.

• It also explored the numerical representations, ranges and limits of integers and decimals.
Lesson Summary

• You now have a greater understanding of the data types used in JAVA, as well as how to create and use variables and constants.

• Now proceed to the exercises and labs associated with this lesson.