Basics of Java Programming

Lecture 2
CGS 3416 Fall 2019

August 29, 2019

Components of a Java Program

- **statements** - A statement is some action or sequence of actions, given as a command in code. A statement ends with a semi-colon (;).
- **blocks** - A block is a set of statements enclosed in set braces { }. Blocks can be nested.
- **classes** - A class is a blueprint for building objects in Java.
  - Every Java program has at least one class.
  - Programmers can define new classes
  - There are many pre-built classes in the Java SDK
- **methods** - A method is a function (i.e. subroutine) that belongs to a class.
  - In Java, all functions are methods, meaning they are always contained in some class

Components of a Java Program

- A Java program can be made up of multiple classes, spread across multiple code files.
- It will typically make use of some SDK libraries as well
- **The main method** - Every Java application must have a main method, which defines where the program begins.
- In Java, the main method belongs to a class. Any class can have a main method. The main method looks like this:

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

Java Source Code Files

The Java compiler imposes some specific rules on the naming of source code files.

- A Java source code file has a base name, along with the file extension ".java"
- A source file can contain one or more classes (and/or interfaces, to be discussed later in the course)
- If there are multiple classes in a code file, one and only one of them should be declared to be public
  - The base name for the filename must match the name of the class that is declared to be public in the file.
  - If there's only one class in the file, the filename must match that class name
  - class names in Java are case sensitive. Be careful – in Windows, for example, filenames are not case sensitive, but in Unix, they are.

Java Source Code Files

This class goes in "HelloWorld.java"

```
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

Java Source Code Files

This file must be named "Daffy.java"

```
class Bugs
{
     public static void main(String[] args)
     {
          System.out.println("What's up, doc?");
     }
}

public class Daffy
{
     public static void main(String[] args)
     {
          System.out.println("You're dethpicable.");
     }
}
```

Statements

- **reserved words** - words that have pre-defined meanings in the Java language
- **identifiers** - words that are created by programmers for names of variables, functions, classes, etc.
- **literals** - literal values written in code, like strings or numbers
  - integer literal - an actual integer number written in code (4, -10, 18)
  - float literal - an actual decimal number written in code (4.5, -12.9, 5.0)
  - character literal - a character in single quotes: ('F', 'a', ' ')
  - string literal - a string in double quotes: ("Hello", "Bye", "Wow!")
- **operators** - special symbols that perform certain actions on their operands
  - A unary operator has one operand
  - A binary operator has two operands
  - A ternary operator has three operands (there's only one of these)
- Calls to methods (functions)

Escape Sequences

String and character literals can contain special *escape sequences* that represent single characters that cannot be represented with a single character in code.

Escape Sequence	Meaning
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage Return
\"	Double Quote
\'	Single Quote
\\	Backslash

Comments

Comments are used to improve the readability of code. Comments are ignored by the compiler. There are two styles of comments in Java:

- block style - comment enclosed in a block that starts with /* and ends with */

```
/* This is a comment */
```

- Line style - comment follows the double slash marker //. Everything after this mark, to the end of the line, is a comment.

```
int x; // This is a comment
x = 3; // This is a comment
```

Variables

Variables are used to store data. Every Java variable has a:

- **Name** – chosen by the programmer (aka identifier)
- **Type** – specified in the declaration of the variable
- **Size** – determined by the type
- **Value** – the data stored in the variable's memory location

Identifiers

Identifiers are the names for things (variables, functions, etc) in the language.
Some identifiers are built-in, and others can be created by the programmer.

- User-defined identifiers can consist of letters, digits, underscores, and the dollar-sign $
- Must start with a non-digit
- Identifiers are case sensitive (count and Count are different variables)
- Reserved words (keywords) cannot be used as identifiers
- an identifier can be any length

Style-conventions (for identifiers)

While you can legally pick any name for a variable that follows the rules, it's also a good idea to follow common programming conventions, for easy-to-read code.

- Here are some conventions used in the Java SDK
  - class and interface names start with an uppercase letter
  - variable names and method names start with a lowercase letter
  - constants are usually in ALL CAPS
  - When using names that are made up of multiple words, capitalize the first letter of each word after the first. Example: numberOfMathStudents
- In addition, it's good to pick mostly meaningful identifiers, so that it's easy to remember what each is for
  - numStudents, firstName // good
  - a, ns, fn // not so good

Primitive Data Types

Java has a small set of what are known as *primitives*. These are basic data types that are predefined for the language.

- **char** - used for storing single characters (letters, digits, special symbols, etc)
  - 16 bits, unicode character set.
- **boolean** - has two possible values, true or false
- **integer types** - for storage of integer values
  - **byte** - 8 bits
  - **short** - 16 bits
  - **int** - 32 bits
  - **long** - 64 bits
- **floating point types** - for storage of decimal numbers (i.e. a fractional part after the decimal)
  - **float** - 32 bits
  - **double** - 64 bits

Declaring Variables

- Inside a block, variables must be declared before they can be used in later statements in the block
- Declaration format: typeName variableName1, variableName2, ...;

```
 int numStudents; // variable of type integer
double weight; // variable of type double
char letter; // variable of type character
boolean flag; // variable of type boolean

// Examples of multiple variables of the same type in
// single declaration statements

int test1, test2, finalExam;
double average, gpa;
```

Initializing Variables

- To declare a variable is to tell the compiler it exists, and to reserve memory for it
- To initialize a variable is to load a value into it for the first time
- One common way to initialize variables is with an assignment statement. Examples:

```
int numStudents;
double weight;
char letter;

numStudents = 10;
weight = 160.35;
letter = 'A';
```

Initializing Variables

Variables of built-in types can be declared and initialized on the same line, as well

```
int numStudents = 10;
double weight = 160.35;
char letter = 'A';

int test1 = 96, test2 = 83, finalExam = 91;
double x = 1.2, y = 2.4, z = 12.9;
```

Constant Variables

A variable can be declared constant by using the keyword *final*

```
final double PI = 3.14159;
```

After this, PI cannot be changed. The following would not work:

```
PI = 15;
```

Type Conversions

When working with mixed primitive types, conversions can take one of two forms:

- **Automatic type conversion**: when appropriate, the complier will automatically convert a smaller numeric type to a larger one (where the floating point types are always considered "larger" than the integer types).
- **Explicit cast operations**: for all other conversions, the programmer must specify with a cast operation. To cast, put the type in parentheses before the expression whose value you are casting.

Type Conversions

```
int i1 = 5, i2;
short s1 = 3;
double d1 = 23.5, d2;
float f1 = 12.3f;
byte b1 = 10;

d2 = i1; // automatically allowed
i1 = b1; // automatically allowed
s1 = (short)i1; // requires cast operation (some data
                may be lost)
i2 = (int)d1; // requires cast operation (decimal data
                may be lost)

d2 = f1 + d1; // automatically allowed
i2 = b1 + s1; // automatically allowed
```

Operators

Special built-in symbols that have functionality, and work on operands

- **operand** – an input to an operator
- **Arity** - how many operands an operator takes
  - unary operator – has one operand
  - binary operator – has two operands
  - ternary operator – has three operands

Examples:

```
int x, y = 5, z;

z = 10;      // assignment operator (binary)
x = y + z;      // addition (binary operator)
x = -y;      // -y is a unary operation (negation)
x++;      // unary (increment)
```

Operators

- **cascading** - linking of multiple operators, especially of related categories, together in a single statement:

```
x = a + b + c - d + e;    // arithmetic operators
x = y = z = 3;     //assignment operators
```

  This works because the result of one operation sends back the answer (i.e. a return value) in its place, to be used in the next piece of the statement. In the above, (a + b) happens first, then the answer becomes the first operand in the next + operation.

- **Precedence** - rules specifying which operators come first in a statement containing multiple operators

```
x = a + b * c;    // b * c happens first,
     // since * has higher precedence than +
```

- **Associativity** - rules specifying which operators are evaluated first when they have the same level of precedence.
  Most (but not all) operators associate from left to right.

Assignment Operator

- Value on the right side (R-value) is assigned to (i.e. stored in) the location (variable) on the left side (L-value)
  - **R-value** – any expression that evaluates to a single value (name comes from "right" side of assignment operator)
  - **L-value** – A storage location! (not any old expression). A variable or a reference to a location. (name comes from "left" side of assignment operator
  - Typical usage: `variable_name = expression`
- The assignment operator returns the L-value (which now stores the new value).
  Examples

```
x = 5;
y = 10.3;
z = x + y; // right side can be an expression
a + 3 = b; // ILLEGAL! Left side must be a variable
```

Assignment Operator

- Associates right-to-left

```
x = y = z = 5;   // z = 5 evaluated first, returns z
```

- Use appropriate types when assigning values to variables:

```
int x;
x = 5843;   // assigning integers to int variables
double a;
a = 12.98;   //assign decimal numbers to type double
float c;
c = 12.98f;   // 'f' indicates float
char letter;
letter = 'Z';   //assign character literals to char
boolean flag;
flag = true;
```

- Be careful to not confuse assignment = with comparison ==

Arithmetic Operators

Name	Symbol	Arity	Usage
Add	+	Binary	x + y
Subtract	-	Binary	x - y
Multiply	*	Binary	x * y
Divide	/	Binary	x / y
Modulus	%	Binary	x % y
Minus	-	Unary	- x

- An operation on two operands of the same type returns the same type
- An operation on mixed primitive types (if compatible) returns the "larger" type
- Floating point types are "larger" than integer types, because no data is lost converting from integer to decimal precision.

```
 int x = 5;
double y = 3.6;
z = x + y;   // what does z need to be?
```

Arithmetic Operators

Division is a special case

- For types 'float' and 'double', the / operator gives the standard decimal answer

```
 double x = 19.0, y = 5.0, z;
z = x / y;    // z is now 3.8
```

- For integer types, / gives the quotient, and % gives the remainder (as in long division)

```
int x = 19, y = 5, q, r;
q = x / y;    // q is 3
r = x % y;   // r is 4
```

Operator Precedence

- Arithmetic has usual precedence
  1. parentheses
  2. Unary minus
  3. *, /, and %
  4. + and -
  5. operators on same level associate left to right
- Many different levels of operator precedence
- When in doubt, can always use parentheses
- Example:

```
z = a - b * -c + d / (e - f);
```

  7 operators in this statement

  What order are they evaluated in?

Some short-cut assignment operators (with arithmetic)

```
v += e;   means   v = v + e;
v -= e;   means   v = v - e;
v *= e;   means   v = v * e;
v /= e;   means   v = v / e;
v %= e;   means   v = v % e;
```

Increment and Decrement Operators

```
++x;     // pre-increment (returns reference to new x)
x++;     // post-increment (returns value of old x)
            // shortcuts for x = x + 1

--x;     // pre-decrement
x--;     // post-decrement
           // shortcuts for x = x - 1
```

Increment and Decrement Operators

- Pre-increment: incrementing is done before the value of x is used in the rest of the expression
- Post-increment: incrementing is done after the value of x is used in the rest of the expression
- Note - this only matters if the variable is actually used in another expression. These two statements by themselves have the same effect:

```
x++;
++x;
```

- Examples

```
int x = 5, count = 7;
result = x * ++count;   // result = 40, count = 8

int x = 5, count = 7;
result = x * count++;   // result = 35, count = 8
```