Introduction to Object-Oriented Programming

Values and Variables

Models, Languages, and Machines

Computing is

> *any purposeful activity that marries the representation of some dynamic domain with the representation of some dynamic machine that provides theoretical, empirical or practical understanding of that domain or that machine.* [1]

- Computing is fundamentally a modelling activity.
- A *model* is a representation of some information, physical reality, or a virtual entity in a manner that can then be interpreted, manipulated, and transformed.
- A *language* is a means of representation.
  - A language enables reasoning and manipulation of the model.
- A computational *machine* allows us to execute our models.

[1] Isbell, et. al., *(Re)Defining Computing Curricula by (Re)Defining Computing*, SIGCSE Bulletin, Volume 41, Number 4, December 2009

Languages and Computation

Every powerful language has three mechanisms for combining simple ideas to form more complex ideas:[2]

- primitive expressions, which represent the simplest entities the language is concerned with,
- means of combination, by which compound elements are built from simpler ones, and
- means of abstraction, by which compound elements can be named and manipulated as units.

In this lecture we'll focus on primitive expressions and basic abstraction.

[2] SICP 1.1

A Model of Course Average

```
public class CourseAverage {

    public static void main(String[] args) {
        double homeworkAvg = 74.2;
        double examAvg = (81 + 91 + 93) / 3;
        double finalExam = 89;
        double courseAverage = (.2 * homeworkAvg) + (.6 * examAvg)
            + (.2 * finalExam);
        System.out.println("Course Average: " + courseAverage);
    }
}
```

- `74.2, 81, 93, 95, 89` are *values* (primitive expressions)
- `homeworkAverage, examAvg, finalExam` are *abstractions* which *name* values.
- The value assigned to `courseAverage` is computed by a `combination` of primitive values.
- Our *model* of course average is expressed in a *language* that allows us to reason about, manipulate, and *run* the model.

Identifiers

An identifier is a string of characters. Identifiers are used as names for classes, methods, and variables

- Java identifiers can contain letters, digits, and the underscore symbol and may not start with a digit.
- Java identifiers are case-sensitive: `this` is not the same as `This`.
- Identifiers used by the Java language compiler are called reserved words, or keywords.
  - Identifiers used by Java, like `class`, `public`, `if` and so on.
  - Identifiers that aren't currently used but are reserved, like `goto` and `const`
  - You can't use reserved keywords for your own identifiers.
  - Full list is here: `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html`

Variable Declarations

Variables are identifiers that name a value. A variable has:

- a type, and
- a storage location for the variable's value.

Variables must be declared before they are used. Here's a declaration:

```
float twoThirds;
```

- `float` is the variable's type
- `twoThirds` is the variable name

The value of `twoThirds` after the declaration statement above depends on whether `twoThirds` is an instance variable or a local variable. More on that later.

Assignment Statements

= is the assignment operator.

- The identifier on the left side of a = must be a variable identifier (an lvalue)
- The right side of the = must be an expression
  - An expression has a value;
  - `2 + 3` is an expression. It has the value `5`.
  - A variable is also an expression. It has whatever value it was last assigned.

```
float twoThirds;
twoThirds = 2/3;
```

We usually combine declaration and assignment into an initialization statement:

```
float twoThirds = 2/3;
```

Type Compatibility

Legal assignments:

```
int x = 1;                          // int literal
float y = 3.14159f;                 // float literal
double z = 3.1415;                  // double literal
boolean thisSentence = false;       // boolean literal
String goedel = "incomplete";       // String literal
```

Illegal assignments:

```
int x = 1.0;                    // 1.0 is a double value
float y = 3.14159;              // 3.14159 is a double value
boolean thisSentence = 1;       // 1 is an int value
```

Core concepts:

- every value has a type
- every variable has a type
- assignment of values to variables must be type compatible at compile-time

Syntax and Semantics

- Syntax - the form to which your source code must conform
- Semantics - the meaning of the code, i.e., what it does

Consider:

```
public class Expressions {
    public static void main(String[] args) {
        float twoThirds = 2/3;
        System.out.println(twoThirds);
    }
}
```

- The code inside `main` conforms to the Java syntax: a sequence of statements that each end with a semicolon.
- The meaning of the program, its semantics, is that we initialize the variable `twoThirds` with the value `.667` and then print it out to the console (or so we think ...)

Compile and run Expressions.java and see what it prints.

Type Conversion

When we run `Expressions.java` we get this:

```
$ javac Expressions.java
$ java Expressions
0.0
```

What happened?

- `twoThirds` is a `float`, so it can hold fractional values.
- But `2` and `3` are literal representations of `int` values.
- `2/3` performed integer division, resulting in a value of `0`.
- Since a `float` variable can hold integer values, Java performed an automatic conversion to `float` upon assignment to `twoThirds`, which ended up with the value `0.0`.

Type Conversions

The previous example showed an implicit widening conversion

- `float` is *wider* than `int` because all intergers are also floating point values.
- Java will perform widening conversions automically because no precision is lost.
- To perform a narrowing conversion, you must explicitly cast the value.

This won't compile because an `int` can't hold a fractional value; converting may cause a loss of precision (note that we're using `double` values by including a decimal part):

```
int threeFourths = 3.0/4.0;
```

You have to cast the `double` to an int:

```
int threeFourths = (int) (3.0/4.0);
```

What happens if we leave off the parentheses around `(3.0/4.0)`?

Integral Primitive Types

- `byte`: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive).
- `short`: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).
- `int`: The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values `int` is generally the default choice.
- `long`: The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by int.

Floating Point Primitive Types

- `float`: The float data type is a single-precision 32-bit IEEE 754 floating point. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.
- `double`: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, `double` is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

`boolean` and `char`

- `boolean`: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- `char`: The char data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

Shortcut Assignment Statements

Like C and C++, Java allows shortcut assignments:

- A binary operation that updates the value of a variable:

```
x += 2; // same as x = x + 2;
```

- Pre- and post-increment and decrement:

```
x++; // post-increment; same as x = x + 1;
x--; // pre-decrement; same as x = x - 1;
```

Pre-increment: variable incremented before used in expression

```
int x = 1;
int y = ++x;
// x == 2, y == 2;
```

Post-increment: variable incremented after used in expression

```
int x = 1;
int y = x++;
// x == 2, y == 1;
```

What's the value of `x` after `x = x++`?

Precedence and Associativity

If an expression contains no parentheses, Java evaluates expressions according to precedence in a three-step process:

1. Associate operands with operators, starting with highest-precedence operators. This step effectively parenthesizes expression
2. Evaluate subexpressions in left to right order (possibly in multiple sweeps if deeply nested)
3. Evaluate outer "top-level" operation once all subexpressions have been evaluated

The expression `6 + 7 * 2 - 12` is evaluated in the following steps:

```
((6 + (7 * 2)) - 12) // Associate operands with operators
((6 + 14) - 12)      // Evaluate subexpressions ...
(20 - 12)
8
```

Side-Effects in Expressions

Beware of side-effects. Consider the evaluation of
`((result = (++n)) + (other = (2*(++n))))`
for `n = 2`:

```
((result = (++n)) + (other = (2*(++n))))
((result = 3) + (other = (2*(++n))))
(3 + (other = (2*(++n))))
(3 + (other = (2*4))) // Note that n was 3 from the first pre-increment
(3 + (other = 8))
(3 + 8)
11
```

- An assignment statement has the value that was assigned
- Pre-increment (`++n`) means `n` is incremented before it's used in the expression in which it appears
- Three side-effects: result = 3, other = 8, and n = 4

Don't write code like this!

`String` Values

A `String` is a sequence of characters.

- `String` literals are enclosed in double quotes

```
"foo"
```

- `String` variables

```
String foo = "foo";
```

Note that, unlike the other types we've seen, `String` is capitalized. `String` is a class.

`String` Concatenation

The `+` operator is overloaded to mean concatenation for `String` objects.

- Strings can be concatenated

```
String bam = foo + bar + baz; // Now bam is "foobarbaz"
```

- Primitive types can also be concatenated with `Strings`. The primitive is converted to a String

```
String s = bam + 42; // s is "foobarbaz42"
String t = 42 + bam; // t is "42foobarbaz"
```

Note that `+` is only overloaded for `String`s.

The `String` Class

`String` acts like primitive thanks to syntactic sugar provided by the Java compiler, but it is defined as a class in the Java standard library

- See http://docs.oracle.com/javase/8/docs/api/java/lang/String.html for details.
- Methods on objects are invoked on the object using the `.` operator

```
String empty = "";
int len = empty.length(); // len is 0
```

- Look up the methods `length`, `indexOf`, `substring`, and `compareTo`, and `trim`
- Because `String`s are objects, beware of null references:

```
String boom = null;
int aPosInBoom = boom.indexOf("a");
```

Play with Strings.java

Closing Thoughts

Every powerful language has three mechanisms for combining simple ideas to form more complex ideas:[3]

- primitive expressions, which represent the simplest entities the language is concerned with,
  - Values are the atoms of programs
- means of combination, by which compound elements are built from simpler ones, and
  - Programs combine and manipulate values
- means of abstraction, by which compound elements can be named and manipulated as units.
  - Variables are the simplest form of abstraction - naming values

[3] SICP 1.1