CISC 3400: Java Programming
Class 2: Java Basics
January 22, 2015
What we’ll cover

• Class 1 Rewrite (Clarification and Extensions from last class)
• Java Basics from Chapter 2 & 3
• Abstract Algorithms
• JAR files, Maven, and the Central Repository
• Next time: we’ll talk about a couple chapters in another recommended book:
  – “Clean Code” by Robert Martin
  – See chapters on good names, good comments
• Note: I’ve added (later) markers for forward references and </examples> markers where we’ll cut over to Eclipse.
Rewrite and Additional Details

**Method declaration:**

```markdown
<optional access modifier>
<optional static keyword>
<optional synchronized keyword>
<optional generic type list>
<return type>
    methodName( formal parameters )
<optional throws clause>
```

</examples: DeckedOutDeclaration>

**Method signature is just the method name and parameters:**

```java
methodName( formal parameters )
```

- Will be handy when we get to method overloading later.
Variable, Class access modifiers

- Last time we talked about their semantics; here’s some more detail:
  - Classes: public or package-private (omit the modifier, I’ll use just “package” to refer to this)
  - Methods (in decreasing order of visibility): public, protected, package, private
  - Inner Classes (later): Any
  - Member variables: Any

- Some rules for **when** they should be used:
  - Make methods private that are not part of an API for the package (if they are, use package), or for the class (if they are, use public).
  - Make methods protected that are not part of an API and that you intend for inheritance (later).
  - Make member variables private unless they are intended for inheritance (if so, use protected). If they are constants, prefer enums (either in an inner (later) or external class).
String Immutability and Caching

• An object is said to be immutable if, once constructed, it’s not possible to change its state. How? Later, when we discuss threads.

• **Don’t do this**, the compiler won’t use the constant pool:

  ```java
  String helloString = new String("hello");
  Favor this: String helloString="hello"; // uses the constant pool (cache)
  ```

  But don’t rely on the constant pool for equality testing. You may not always be certain how the String objects were created and you will make your code dependent on String’s implementation. This is generally a bad idea.

  ```java
  if (helloString == "hello") { ... } // This works or not depending on String’s implementation!
  Favor this instead: if (helloString.equals("hello") ) { ... }
  ```

• Why is there a constant pool for strings?
  – It’s designed to save allocations of frequently reused strings.

• Don’t use them in loops to construct new strings, prefer StringBuffer.
Java Basics from Chapters 2
Types

• Java is strongly typed: all variables must have a type declaration: local variables, member variables, formal method parameters. Contrast this with JavaScript that relies on runtime type inference (it figures it out by context).

• Types direct compiler type checking; the compiler does its best to “help” you out, the rules of which follow.
Primitive Types

- **byte** -2^7 to 2^7 – 1
- **short** -2^15 to 2^15 – 1
- **int** -2^31 to 2^31 – 1
- **long** -2^63 to 2^63 – 1
- **float** -2^-149 to 2^128 (approx, limited precision)
- **double** -2^-1022 to 2^1023 (approx, limited precision)

The compiler will automatically convert a primitive type according to the descending path above.

If you try to convert otherwise, you’ll need to **cast**, something you want to try to avoid. When casting you’re telling the compiler “I promise I know what I’m doing”, but you risk a run-time error if you don’t.

Note that **char** is considered **unsigned** and can hold a numeric value from 0 to 2^16 – 1. When in expressions with ints or longer types, the compiler will auto-convert it to the wider type.

</examples: TypeConversion, ExplicitCasting >
Reference Types...

• Java only has two types: primitive and reference types.
• Every class and (later) interface definition creates a new reference type. A variable declared with that type contains a reference that can either be set to null or to an allocated instance of the declared class. Instances are allocated with the ‘new()’ operator.
• Example: SomeClass anObject = new SomeClass();
• In the example above, ‘anObject’ is a variable of reference type ‘SomeClass’. It points to not-directly-accessible heap memory where the object is allocated.
• Any parameters are passed to the constructor using the same rules as method calls.
• Notes:
  – Pointer arithmetic is not possible with reference types.
  – The compiler creates a default (empty) constructor for a class if you don’t supply one.
Programs and Constants

• Programs are comprised of one or more classes, classes are comprised of zero or more methods, and methods are comprised of zero or more statements. Statements are built from expressions. (Note: classes also have special static initializer blocks we’ll discuss later.)

• Constants: (must use **final** to make them immutable!)
  
  ```
  final double PI = 3.14159265358979; // camelCase exception for CONSTANTS
  final int MAX_NUMBER_OF_SOMETHING= 100; // use underscores here
  ```
Constants and Literals

• Constants can be defined from literals and/or other constants:
  
  final int DAYS_IN_WEEK= 7;
  final int WEEKS_IN_SEMESTER = 14;
  final int DAYS_IN_SEMESTER = DAYS_IN_WEEK * WEEKS_IN_SEMESTER;

• Define constants in the scope in which they are meaningful.
  – Method ( Use final local variable )
  – Class ( Use final member variable )
  – Package & Beyond ( Use enums with the corresponding access modifier )
Operators, Expressions, and Statements

- **Operators** define implicit functions from one or more terms (the domain) to a resulting term (the range), e.g.
  - BOOLEAN, BOOLEAN -> BOOLEAN (this is a special function: Predicate)
  - INT, INT -> DOUBLE (See class java.lang.Math)
  - Rich sets of common functional forms exist... we'll cover this when we get to Java 8.

- **Expressions** are any syntactically correct mix of variables, operators, and literals that return a value. An expression can be just a constant or a literal or a variable. Complex expressions are any of these combined with operators. They don’t compile stand-alone but only as part of a statement. Examples:
  
  \[
  \begin{align*}
  & x \\
  & x + 1 \\
  & y \times y - x \times x \\
  & y > 0 \ ? \ 1 \ : \ 0 \\
  & 10 == x \ // \ equality \ operator \ returns \ boolean \\
  \end{align*}
  \]

Expressions, however, are only useful only as part of a statement:

\[
\begin{align*}
  & x = (y > 0) \ ? \ 1 \ : \ 0; \\
  \end{align*}
\]

- **Statements** are executable sentences of the language. Sequential statements are separated by a semicolon. While you can place more than one on a line, generally avoid this unless they are tightly coupled: e.g. int I, j; // where I, j are used immediately after as loop variables.

Examples:

\[
\begin{align*}
  & x = y \% 5; \\
  & \text{Int } z=x; \\
  & \text{System.out.println(“Hello World, x = ” } + x + “ and y = “ } + y); \\
  \end{align*}
\]
Operators

- Relational comparison: Compare two variables/values of the same type (remember, compiler will promote types if possible) and returns boolean.
- Arithmetic comparison: >, >= , <, <=
- Equality: ==, !=
- Bitwise: & (and), ^ (xor), | (or)
- Conditional: &&, ||
- Ternary: ? :
- Bit shifting: >> (right shift), << (left shift), >>> (right shift with zero-fill)
- Array access: []
- **Object reference member access**: .
- Method invocation: ()
- Increment and Decrement: ++, -- (pre and post forms exist; post forms have higher precedence).
- Logical operator: ! (logical not – inverts booleans)
- Bitwise not: ~ (flips bits on variables or literals)
- **Cast operator**: (<cast-to type>)
- **New operator**: new
- Arithmetic, multiplicative: , *, /, %
- Arithmetic, additive: +, -
- Assignment: =, !=, +=, -=, *=, /=, %= Bitwise: &=, ^=, |=, <<=, =>>, >>>=

<examples-with-exercise-for-non-bold-items/>
Operator Precedence

• ‘Google’ “Java operator precedence” for a complete table.

• Operators are generally grouped by level (example: see p53 of the book).

• Each level associates either left-to-right or right-to-left.

</examples>
Arrays

- Arrays are also reference types.
- Array Elements are always of the same type. They can **contain exactly one** reference **OR** one primitive type per array.
- Declaration syntax examples:
  ```java
  int[] results = new int[100]; // fixed size
  Int[][] results = new int[100][100] // 2D
  Int results[][] = new int[100][100] // variation
  Int[] results[] = new int[100][100] // another...
  ```
- Array contents are auto-initialized like member variables.
Arrays

- Arrays are zero indexed. `someArray[0]` is the first element of `someArray`.
- If you try to access an element past the end of the declared array, an `ArrayIndexOutOfBoundsException` exception will occur.
- Array initialization
  
  ```java
  boolean[] logicalValues = {false, false, true, true};
  ```
- Key array properties:
  - Size fixed to allocation size.
  - Ordered collection of primitives or ref types
  - Inserting “in the middle” requires shifting values down, a relatively expensive operation.
  - Collection data structures are generally a better option; `ArrayList` is built on arrays and handles automatic resizing.

</examples>
Pass by Value

• In Java, pass-by-value means that whatever actual parameters you pass are copied into the formal parameters of the target method. This applies to both primitive and reference type parameters.
• If you change the formal parameters (either a primitive or object reference), it will have no effect on the actual parameter in the calling method. [ Stack vs Heap memory ]
• While all parameters are pass-by-value in Java, if you pass an object reference pointing to mutable data, you can effect changes to what that reference refers to.
Java uses Pass by Value

• A copy of the actual parameters is set into the formal parameters of the called method.

• If you change the data an object reference refers to, it will change the referred-to data for every reference, regardless of where that reference exists.

• Functions that change the state of the system after they are called are called “procedures” and the state changes referred to as “side effects”. Where possible, prefer methods that do not produce side effects. As we’ll see, there are key exceptions to this rule with object-oriented design.

• Note: A key difference between (C++) pointers and references is that references cannot be arithmetically manipulated to refer to another memory location. They can only be dereferenced to access or mutate what they refer to.

<examples/>
 Enums

• Each enum is a class that defines instances that represent enumerated values.
• Enums can have methods, as can each enum instance. But, consider…
• Java Language Spec: “Instance methods declared in enum bodies may be invoked outside the enclosing enum type only if they override accessible methods in the enclosing enum type.”
• This requirement forces uniformity across all enum instances.
• Also, the enum itself cannot by extended like classes.

</examples: TestEnum>
Text and FileIO from the Book

• Handy utilities for reading and writing various data types to the Console and files.
• Has everyone imported this class into Eclipse?
</examples:TextAndFileIO>
Abstract Algorithms

• An algorithm is an effective method of correctly solving a problem in a finite number of steps.
• If it doesn’t meet both those criteria, it isn’t an algorithm.
• Algorithms are therefore referred to as “totally correct”.
• Methods of solving problems that guarantee correctness, but not necessarily in a finite number of steps, are “partially correct”.
Abstract Algorithms

• When solving a problem in any programming language, start here.

• Think through an abstract solution in natural language (see examples in chapter 2).

• For now, refine the solution in steps until you can implement each step in Java.

• Later, we’ll talk about class design where we’ll modify this process.
Apache Maven

• “Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.”
• Create a Java Project, then “Configure > Convert to Maven Project”. This will create a pom.xml (default packaging should be jar).
• A jarfile is a java archive that contains the .class files that hold the compiled Java bytecode.
• Right click on pom.xml, then “Run As > Maven install”.
• Maven knows how to package your project jarfile. You can use it to package your code (find the output jarfile under the target directory).
• Next class we’ll see:
  – How POM dependencies provide access to the Central Repository. You can search it here: [http://search.maven.org/](http://search.maven.org/)
  – How Maven can set up a skeleton project structure.