Java Basics

Thanks to Julie for this great handout.

Basic syntax

Much of Java syntax will be quite familiar to you since it is highly derivative of C and its comrade language C++. However, it is not true that Java is a superset of C++, or even a superset of C. Many good additions were made, but to simplify, improve safety, and eliminate redundancy, there were also things removed. James Gosling, chief architect of Java, states they omitted "many rarely used, poorly understood, confusing features of C and C++ that in our experience bring more grief than benefit." Pointers, operator overloading, multiple inheritance, friend functions, extensive automatic coercions, gotos, unions, etc. all didn't make it past the cutting room floor. And even some straightforward language constructs were axed in the name of simplicity—for example, Java has no enums, structs, or global variables. In pursuit of guaranteed portability, various "implementation details" usually left to the compiler are fully specified in Java: the size of various data types, layouts of objects, order of evaluation of subexpressions, etc.

This handout isn't the end-all Java resource (instead check out Javasoft's web site at `http://java.sun.com`, in particular, all the standard packages and classes are documented at `http://java.sun.com/products/jdk/1.1/docs/api/packages.html`)). Here's a quick list of some basic Java language facts, expressed as the deltas from C:

Miscellany

- Java is case-sensitive, like C. This fact is sometimes exploited in the system libraries to use the same name with different capitalization (`float` and `Float`, `true` and `TRUE`).
- The comment characters include the `/* ... */` from C to mark an entire region and the `//` from C++ which marks the rest of the current line as a comment.
- You can declare new variables at any line within a method, not just at the beginning of a new scope as in C.

Data types

- The primitive data types include `byte`, `char`, `short`, `int`, `long`, `float`, `double`.
- There is a true built-in `boolean`, a data type that can hold `true` or `false`. It is not the case that all non-zero values are true in Java, you must use explicit `boolean` values in test expressions.
- All primitive types are signed, they have no unsigned variants.
- In the name of portability, the size of each type is fully specified. A `short` is always 2 bytes, `int`s and `float`s are 4 bytes, `double`s and `long`s are 8, etc. This can lead to some inefficiency when the mandated sizes aren't a best fit for the underlying hardware,

but the portability guarantee was considered more important. It also turns out there is no `sizeof` operator since it isn't needed.

- Characters are 16-bit in order to support the Unicode character set and allow for true internationalization. (The 8-bit extended ASCII is inadequate for anything beyond the Roman alphabet.) Characters are not treated as a numeric type, but you can use a cast to go from a `char` to a numeric type and back.
- You can freely assign the values from smaller-sized data types to larger, so assigning a `short` to an `int` is fine. Going in the reverse always requires a cast since you will potentially truncate and lose information. This is unlike C which allows these conversions without a cast. You cannot cast to a `boolean` (use the `?:` operator if you need to turn an `int` into a `boolean`).
- There are also object wrapper versions of the primitives: `Boolean`, `Integer`, `Float`, etc. which respond to methods like `equals`, `intValue`, `toString`, etc.
- Except the built-in primitives, all data types are objects. Arrays and strings are objects. There is no ability to create new primitive types or define `enums`, `structs`, or `unions`.

Arithmetic and logical operators

- Java has the same binary `+`, `-`, `*`, `/`, `%` operators. Each has a unary-assignment (`+=`, `*=`) form and there is the same shorthand increment / decrement (`++`, `--`) from C.
- The usual precedence and associativity rules apply. Unlike C, Java mandates the order of sub-expression evaluation (which is from left to right). Thus previously ambiguous expressions like (`i++ * a[i]`) are guaranteed to get exactly the same results on all platforms, ensuring portability even for programs that do bogus things.
- The + operator is overloaded to also do string concatenation. There is no general facility to overload operators in Java (unlike C++), but in a few select cases, the language itself overloads an operator in the name of simplicity and convenience.
- The usual relational operators exist (==, `!=`, <, <= ,> ,>=) as well as the logical connectives (`&&`, `||`, `!`), all with the same precedence, associativity, and short-circuiting evaluation.

Control statements

- Java has the same `if/then/else` and `?:` constructs as C. The test expression must be a `bool` value, not just any zero / non-zero expression.
- There are the same iteration constructs (`for`, `while`, `do-while`) from C. All test expressions need to evaluate to a `true bool` result. Java has a comma operator like that of C, but it can be used only in the initialization and increment parts of a `for` loop. You have the same `break` and `continue` options for quitting the loop or skipping to the next iteration.
- Java has the familiar `switch/case` statement, warts and all (i.e. need to use `break` to avoid undesired fall-through behavior).
- The `return` statement for exiting a function and returning its result is the same.

Arrays

- Java arrays are syntactically similar to C, but are implemented quite differently. An array is not a pointer to a block of contiguous elements, it is, in fact, an object of its own. It's probably best to not think too much about what is going on behind the scenes and just focus on how you use an array from a client perspective.
- When you declare an array, you leave the size unspecified: `int[] scores`. Before you use it, you need to allocate the array to the length you need using the `new` operator:

```
scores = new int[100];
```

- Alternatively, you can declare and allocate the array in one step with the = {elem, elem, elem} syntax like C. Once allocated, an array's length cannot change. (For a growable array, see `Vector` class in the `java.util` package.)
- At all times an array knows its length, and you can retrieve it by accessing the public data member of the array object: `scores.length`
- Array elements are indexed from `0` to `length - 1`. You can read or assign an array location using the usual subscript notation: `scores[i]`. Array accesses are bounds-checked at runtime and throw an exception if the index is out of range.

Strings

- Java strings are neither character arrays nor pointers to null-terminated character sequences. They are objects in their own right.
- There are two built-in string classes, `String` and `StringBuffer`. A `String` is an immutable object that is initialized to hold a sequence of characters and cannot then change. Think of it as a string constant. A `StringBuffer` object can grow and shrink as well edit the characters in the buffer. We will mostly use the simple `String` class.
- A `String` object responds to messages such as `length`, `charAt`, `compare` (both case-sensitive and insensitive versions), `startsWith`, `substring`, `concat`, and more. The functionality is similar to that of Eric Roberts' string library you are familiar with from CS106. A `StringBuffer` object includes operations that allow you to append, extract, and insert characters into the buffer.
- As a shorthand, any usual C string literal (i.e. sequence of characters enclosed in double-quotes) will be converted to a `String` for you.
- As mentioned before, `+` applied to two or more strings will concatenate them. Also since all basic data types have the ability to convert to a string representation, you can "add" strings to `int`s and `float`s, which converts the numbers to their string representations and then combines them into the resulting string. You will often use this feature when constructing strings of data you would like to print out.

Exceptions

- Not surprisingly considering Java's emphasis on safety, there is a built-in facility for handling runtime exceptions. The language provides facilities you can use to gracefully handle exceptions if they arise, raise exceptions when error conditions

come up, `try` operations and `catch` any raised exceptions, etc. We probably will not get much into this feature, but it is worth knowing it exists.

- There are many built-in exceptions that can arise in the course of execution: divide by zero, out of memory, stack overflow, use of `null` pointer, index out of bounds, runtime cast failure, etc. If you provide no special handling, these will default to stopping the application and printing an error message. (This will be surprisingly reminiscent of LISP...)
- All stack variables are cleared (set to zero) when a function begins executing. You should not depend on this, and like usual, should initialize all local variables before use. Zeroing the memory enables detection of uninitialized pointer access, and ensures consistent results, even for programs that are poorly written in this regard.

Memory

- You do explicitly allocate memory using `new`, but it is a more structured allocator than `malloc` in that you state what you want (an array of `10` `int`s, a `String` object) rather than the total number of bytes you need. This tends to be less error-prone.
- One of the most pleasant features of Java is its inclusion of garbage collection (remember what I told you about the LISP hiding underneath the C syntax...). You do not deallocate anything. heap-allocated memory is reclaimed when the system detects you are no longer using it. (Most current systems do a mark-and-sweep garbage collector).

Project organization

- There are no header files in Java. The class definition serves as both the interface and the implementation. This removes the problems related to `.h`'s and `.c`'s getting out of sync. In fact, there is no preprocessor at all, so no `#include`, `#define`, `#ifdef`, or macros either.
- Each file / class belongs to a *package*, which is a Java namespace feature for grouping related classes. At the very start of a file, you can indicate the package to which it belongs to with a `package` statement. If omitted, the class is placed in a global default package. This default will be fine for our purposes.
- The built-in classes are already grouped into packages. For example, the `PrintStream` and `File` classes are part of the `java.io` package, the `Socket` and `InetAddress` classes are in the `java.net` package, and so on. Packages are similar to C libraries.
- All of the installed public packages are completely accessible to any Java program. You can refer to classes, methods, constants, etc. using their full names, e.g. `java.util.Date.getMonth`. You do not need to include any files or link with anything special.
- The `import` directive allows you to shorten the names required to access elements out of another package. For example, if you import `java.util.Date`, then you can declare a variable of just `Date` class without the package prefix.

- You can use the wildcard `*` to import an entire package, for example, if you import `java.util.*`, you can then refer to any of the classes in that package (`Vector`, `Hashtable`, etc.) by just the class name, without the package prefix.
- There is a fixed relationship between the name of a class, the name of its package, and the path to where that class is stored, e.g. the class `Point` in the package `cs107.demos` must reside in the file `cs107/demos/Point.class`. This is how Java avoids any `#include` nonsense— given the name of a class, the interpreter knows exactly where to go find its definition.

Built-in packages

- The `java.lang` package includes general purposes classes such as `Math`, `Process`, `Thread`, `System`, etc., the class wrappers for the primitive types, the general exception classes, the `Object` base class, and so on. The entire contents of this package are always visible, so you never need to import anything from here in order to have shorthand name access to its public components.
- The `java.util` package defines a set of built-in utility classes including `Date`, `Stack`, `Vector`, and `Hashtable`, among others. The `Vector` object is the equivalent of the LISP list built-in and parallels the `DArray` we had to work so hard to create for ourselves in C and was so difficult to use.
- The `java.io` package contains all the classes to read and write data streams. A stream can be a disk file, the user's terminal, an in-memory buffer, or a pipe. There is functionality for single-character and block read/write with optional buffering, formatted I/O, and even simple tokenizing (like the CS106 scanner abstraction).
- The `java.net` package contains a set of classes designed to help manage network activity: `InetAddress`, `Socket`, `DatagramPacket`, `URL`, and more. This type of facility is a somewhat unusual inclusion in a language's built-in libraries, but clearly an important future direction, especially for a language like Java that is attempting to meet the needs of the Web.
- The `java.awt` package contains the classes which make up the "abstract window toolkit". The idea of the `awt` to provide a consistent interface for the Java programmer for creating windows, drawing graphics, using various controls (buttons, sliders, text fields, etc.), handling user events, working with text and fonts, providing menus, bringing up dialogs, etc. On any particular platform this abstract interface is then mapped to the underlying native window system (be it MacOS, Windows95, XWindows, or whatever). This concept is an important step toward the Holy Grail of cross-platform deployment. However, to achieve the portability goal, the `awt` is necessarily limited to only providing the lowest common denominator that can be supported on all those platforms. The `awt` is the weakest link in the Java packages, and there are a lot of future changes coming as they try to increase functionality and improve usability of the GUI elements with new Java Foundation Classes and Swing.
- The `java.applet` package contains a few little glue classes that are necessary for a Java applet operating in the context of a browser.