CSE 413 Java Overview
April, 1999

A Few References:

CSE413 text — *Understanding Object-Oriented Programming with Java* by Tim Budd, Addison-Wesley, 1998.

Good quick overview for experienced programmers — *Java in a Nutshell*, O'Reilly & Associates. First edition covers Java 1.0; second edition covers Java 1.1 and deletes some of the Java 1.0 examples.

Sun's Java web site at www.java.sun.com — includes online Java class library web reference pages.

Reference information — *The Java Language Specification* by Gosling, Joy, and Steele, *The Java Class Libraries* by Chan and Lee, *The Java Programming Language* by Arnold and Gosling,… .

Some History

- 1973 C (Ritchie)
- 1978 *C Programming Language*, 1st ed. (K&R)
- C with Classes (Stroustrup)
- 1983 C++ (Stroustrup)
- 1985 *C++ Programming Language*, 1st ed.
- 1988 *C Programming Language*, 2nd ed.
- 1990 C standard adopted
- 1991 *C++ Programming Language*, 2nd ed.
- 1993 Oak project at Sun
- 1995 Java announced
- 1996 Java available
- 1997 (March) Java 1.1
- 1997 *C++ Programming Language*, 3rd ed.
- 1997 (September) Java 1.2 beta
- 1998 (July) C++ Standard adopted
- 1998 (October) ~~Java 1.2 final~~ Java 2
- ???? Java Standardized? Pure Java? MS Java?

Java

Java originated in the Oak project at Sun to support software for consumer electronics gadgets. C++ was used at first, but that turned out to be unwieldy. So they invented a new language, Java, based on good ideas from languages like C++, Eiffel, Smalltalk, Objective C, and Cedar/Mesa.

Java's design goals include:

- Support secure, high performance, robust applications that run as-is on many platforms and over networks.
- "Architecture-neutral", portable, support for dynamic updates and adaptation to new environments.
- Culturally familiar so the C++ world will buy into it.
- Support for object-oriented programming.
- Support for concurrent, multithreaded applications.
- Simple (particularly compared to C++).

"Hello World" in Java

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

Basic Data Types

- `int` — 32-bit integers, ± 2,147,483,647: -17, 42
- `double` — 64-bit IEEE floating point: 3.14, 1.0e-6
- `char` — 16-bit Unicode: 'a', '?', '¥ ', 'é ', 'π '
- `boolean` — truth values: `true`, `false`

Notes:

- `boolean` and `int` values are not interchangeable.
- There are additional signed numeric types:
  - — integer types: `byte` (8 bits), `short` (16), `long` (64)
  - — floating point type: `float` (32 bits)
- An integer constant normally has type `int`; a floating-point constant normally has type `double`. `byte` and short values are widened to `int` before arithmetic operations are performed.
- Nothing in Java is "implementation-defined" or "implementation-dependent".

Variables, Arithmetic Expressions, and Assignment

Almost the same as in C/C++

```
int k = 17;
double x, y, z;
boolean maybe = true;
char ch;
```

Initialization is optional; if omitted, all class variables are initialized to binary `0` (`false`, `null`). Local variables in methods and constructors are not initialized.

```
k = k/2;          // truncates
x = 3.5*k+42.0;
y = x/2;          // no truncation
```

Assignment quietly coerces types as long as no information is lost:

```
y = (k+6)*11/2;
```

Explicit coercion is required to, for example, truncate a floating-point value to integer.

```
k = (int) x * 0.5;
```


Conditional Statements

Java's `if` statement is the same as in C/C++. Braces (`{}`) are used to create compound statements.

```
if (x < y)
   x = y;

if (x > y) {
   int tmp = x;
   x = y;
   y = x;
}

if (x != y) {
   x = y;
} else {
   y = 0;
}
```


`switch`

The `switch` statement also works just like it does in C/C++ (unfortunately). An explicit `break` is needed after each case; if it is omitted, execution falls through to the next one. If `default` is not provided and the expression does not match any case label, execution proceeds with the next statement.

```
switch (k/2) }
   case 0:
      <do something>
      break;
   case 1:
      <do something else>
      break;
   default:
      <do something different>
}
```

Indefinite Iteration — `while`

The fundamental iteration construct is `while`.

```
while (k < 100)
   k = 2*k;

while (k < n && a[k] != x) {
   sum = sum + a[k];
   k++;
}
```

Notes:

- Logical and (&&) and or (||) are short-circuit — the second operand is not evaluated if not needed.
- Watch for use of bitwise & and |. These typos are legal Java but probably don't do what you want.

Definite Iteration — `for`

The `for` statement works as in C/C++.

`for` (*initialize*; *test*; *increment*)
    statement;

is equivalent to

initialize;
`while` (*test*) {
    statement;
    increment;
}

(except new variables can be declared in *initialize*.)

~~Functions~~ Methods

"Java has no functions. Object-oriented programming supersedes functional and procedural styles. Mixing the two styles just leads to confusion and dilutes the purity of an object-oriented language. Anything you can do with a function you can do just as well by defining a class and creating methods for that class."

The Java Language Environment: A White Paper
James Gosling and Henry McGilton

There are no free-standing, global functions or variables in Java. Everything is a member of some class.

Classes & Objects

The basic use of a class is to define a template for objects (instances) of that class. Objects are used both for "object-oriented" programming and for data aggregates like `struct` in C/C++ or record in Pascal.

```
// a very simple class
class Blob {
   private int val;    // Blob's value

   // yield the current value of this Blob
   public int getVal( ) {
      return val;
   }

   // set the current value of this Blob to n
   public void setVal(int n) {
      val = n;
   }

   // Yield string representation of this Blob
   public String toString( ) {
      return "Blob: val = " + val;
   }
}
```

Access to a member is restricted to other members of the class if it is qualified with `private`. If `public` is used, it may be accessed from anywhere.

Class `Blob` should normally be in file `Blob.java`.

Creating Objects, Calling Methods

Objects are allocated by `new` and fields are referenced with the usual dot notation.

```
Blob b;
b = new Blob( );
int k = b.getVal( );    // k = ____?
b.setVal(17);
Blob x = new Blob( );
x.setVal(b.getval( ) + 25);
```

All parameters are passed by value (either a primitive value or a reference to an object allocated by `new`.)

Notes:

- The declaration

  ```
  Blob b;
  ```

  only declares that `b` has type "reference to `Blob`". It does not actually allocate a new `Blob`. The declaration and allocation are often combined.

  ```
  Blob b = new Blob( );
  ```

- The value `null` can be used anywhere a reference is needed; it points to no object. If an attempt is made to select a field from `null`, a `NullPointerException` is thrown.

- Storage allocated by `new` is automatically reclaimed when the object is no longer accessible (automatic garbage collection). For large objects, it's sometimes a good idea to overwrite any references to them when they are no longer needed to allow the storage to be reclaimed.

  ```
  b = null;
  ```


Constructors

If nothing special is done, all of the data members of a newly allocated object are given default initial values (lots of 0's). Constructors can be included in a class to specify code to be executed when an object is created. Constructors, like other functions, can be overloaded, i.e., several definitions can be given differing in number and/or types of parameters. The actual parameter list is used to determine which one to call.

```
class Blob {
   int val;     // Blob value

   // constructors
   Blob( ) {  val = 42; }

   Blob(int initialVal) {
      val = initialVal;
   }
   ...
}

Blob b = new Blob( );  // b.val = ____ ?

Blob c = new Blob(17); // c.val = ____ ?
```


`static` Methods

Conceptually, every object has a copy of each method declared in a class. But there are some methods (functions) that aren't naturally associated with objects — the most obvious ones are the basic math functions (`sin`, `cos`, `sqrt`, …).

Such methods still must be declared in a class, but they are declared `static` to indicate that there is only one copy of the method and it is associated with the class. For example, the basic math functions are static members of class `Math`.

```
class Math {     // part of java.lang
   static double sqrt(double x) {  ... }
   static double sin (double x) {  ... }
...
}

...

d = Math.sqrt(x*x + y*y);
```

"Hello World" Revisited

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

Execution of a program always begins in method `main` of some class. It must be `static` and publicly visible.

The standard class `System` provides basic I/O and other services. `System.out.println` writes its string argument to the console.

Note: In Java, every class may have a method `main`. Among other things, this provides a useful place to keep test programs for individual classes.

Implementation note: You may need to set some preferences or options to specify which class's `main` method should be used to begin execution.

`static` Fields

Normally, every object of a class has its own copy of each field declared in the class. Sometimes, however, we only want a single copy of a variable to be shared among all instances of a class. This is also specified by `static`.

Example: `Blob`s with sequential serial numbers.

```
Class Blob {
   int val;     // value of this blob
   int serial;  // serial number of this blob

   static int nextSerial = 0; // next serial #

   // constructor
   Blob ( ) {
      val = 17;
      serial = nextSerial;
      nextSerial++;
   }
   ...
}
```


Static data member example, cont.

```
Blob b = new Blob( );

Blob ob = new Blob( );

b.setVal(17);

ob.setVal(b.getVal( )++);
```


Symbolic Constants

A class variable (but not a local variable in Java 1.0) may be qualified with the keyword `final`, meaning that the variable must be initialized when declared and can't be changed later. If a `final` variable is not `static`, then every object of the class has a separate copy, possibly with different values. A `static final` variable is a symbolic constant associated with the class.

Example: Symbolic constants provided in standard class `Math`.

```
class Math {
   static final double PI = 3.14159265359;
   static final double E  = 2.71828182845;
   ...
}
...

double a = Math.PI * r * r;
```

Arrays

Arrays, like everything in Java that isn't a primitive data type (`int`, `double`, `char`, etc.), are dynamically allocated with `new`. Unlike most other languages, declaring an array in Java doesn't actually create it.

```
double[ ] a;
a = new double[6];
for (int k = 0; k < 6; k++)
   a[k] = k;
```

Notes:

- Arrays are 0-origin, as in C/C++.
- If `a` is an array, `a.length` is the number of elements in it.
- An `IndexOutOfBoundsException` is thrown if a subscript is out of range.
- The array brackets can be placed after the variable, as in C/C++ (but why would anyone want to?).

```
int a[ ] = new int[30];
```


2-D Arrays

A 2-dimensional array is really an array of references to array rows. The allocation

```
double[ ][ ] matrix = new double[10][20];
```

is shorthand for

```
double[ ][ ] matrix = new double[10][ ];
for (int k = 0; k < 10; k++)
   matrix[k] = new double[20];
```

Array elements are accessed in the usual way.

```
for (int r = 0; r < 10; r++)
   for (int c = 0; c < 20; c++)
      matrix[r][c] = r*c;
```


Arrays of Objects

If the elements of the array are not primitive types, they must be allocated individually.

```
class Point {    // point on cartesian plane
   public double x;
   public double y;
}
...

// declare Point array
Point[ ] pList;

// allocate Point array
pList = new Point[4];

// allocate array Points
for (int k=0; k<4; k++)
   pList[k] = new Point( );
```


Strings

Class `String` describes read-only string objects; the `StringBuffer` class provides string objects that can be modified (are "mutable").

```
String s = "String Constants are written ";
String t = "with the usual notation.\ n"
System.out.println(s + t +
   "The + operator indicates string " +
   "concatenation.\ n");
```

Notes:

- A `String` is an object — it is not the same as an array of characters. There's no `'\0'` byte at the end.
- String elements are Unicode characters.
- If `s` is a String, `s.length( )` is its length and `s.charAt(k)` is the `char` in position `k`. Class `String` contains many useful string-processing functions.
- Most classes include a `toString( )` method that is executed automatically when an object is used in a context where a string is expected.

Derived Classes

Java supports single inheritance to derive new classes from one parent class. A derived class is said to extend its superclass. Example:

```
// Point in 3-D space
class Point3D extends Point {
   public double z;    // z coordinate
}
```

Class `Object` is at the root of the inheritance hierarchy. Any class that does not explicitly extend another class implicitly extends `Object`. These are equivalent:

```
class Point {  ... }
class Point extends Object {  ...}
```

All of the usual object-oriented notions are supported, including `this` and `super` to refer to members of the current class and superclass, abstract classes, protected access to parent class members, etc.

Wrapper Classes for Basic Types

Everything in Java extends `Object` except for the basic numeric, character, and boolean types. For situations where one needs to use data of these types when an `Object` is required, Java provides wrapper classes `Integer`, `Double`, `Boolean`, `Char`, etc.

`Integer(17)` is an object representation of the `int` 17. If `I` is an `Integer`, `I.intValue( )` is its `int` value. The other wrapper classes work similarly.

These classes are also used as a convenient place to stash useful static utility functions and constants. Examples:

```
Integer.MAX_VALUE   // largest int
Double.MIN_VALUE    // smallest double > 0
Character.isLowerCase(ch) //== ch is lowercase
Double(“ 123.45  ”) // double value of string
```


Interfaces

Restricting Java to single inheritance simplifies the language, but rules out some sensible designs. For example, a random access file should be usable as both an input file and an output file, i.e., it should be derived from both classes. Java interfaces provide most of the necessary features.

An interface is a specification of constants and abstract methods. It contains no code and no objects can be created from it. It can extend other interfaces.

A class may implement as many interfaces as desired. The full implementation must be provided — no code is "inherited" from an interface.

```
interface DataInput {  ... }  // stream input
interface DataOutput {  ... } // stream output

class DataInputStream
   extends FilterInputStream
   implements DataInput {  ... }

class RamdomAccessFile extends Object
   implements DataInput, DataOutput {  ... }
```


Packages

Java provides packages to group related classes and interfaces and to avoid name clashes between packages developed independently. To incorporate a source file in, for example, package `widget`, the first non-comment line in the file must be

```
package widget;
```

Items in a package normally have access to classes and methods in all files that are part of the package. Files in other packages must import it. There are several forms; the basic one imports all visible names in the package.

```
import widget.*;
```

This should be placed after any `package` declaration and before anything else.

Files with no package declaration are grouped in an "unnamed package". For most of your projects, it's probably easiest to omit package declarations.

Standard Libraries

Java includes a ~~large~~ huge class library grouped into many packages. Some of the basic ones are listed below. Everything in `java.lang` is imported automatically (including things like `Math`, the `Integer`-like wrapper classes, etc). An `import` declaration is needed to use routines in other packages.

- `java.lang` — standard system types and classes.
- `java.io` — streams and random access files.
- `java.net` — TCP/IP sockets, telnet, URLs.
- `java.util` — basic container and utility classes: dictionaries, hash tables, Date, Time, …
- `java.awt` — Abstract Windowing Toolkit. Version 1.0 lasted less than a year. Version 1.1 has a much-improved event-handling model, and is supported by current (Sp99) web browsers. ~~Java 1.2~~ Java 2 includes a new user interface toolkit, Swing, that is independent of the underlying OS GUI, but is not widely available yet.

Comparing and Copying Objects

For objects, the equality operators `==` and `!=` determine whether two references point to the same object. If `b` and `c` are both `Blob`s, then `b == c` is true iff `b` and `c` both refer to the same `Blob`. Similarly, assignment `(=)`, only copies references. Example:

```
Blob b = new Blob(17);
Blob c = new Blob(42);

if (b == c)
   System.err.println("Something's broken");
else
   System.err.println("no problem");

c = b;
b.setVal(100);
System.out.println("b = " + b.getVal( ) +
                   "c = " + c.getVal( ));
```


Deep Copy/Compare

Class `Object` (and therefore every class) includes two methods that are intended do a deep copy or comparison. Normally (assuming that this makes sense for a class), a class author would include implementations for `equals` and `clone` so that

`a.equals(b)` is true iff `a` and `b` have the "same value".

`b.clone( )` creates a new "copy" of `b` and returns a reference to the new copy.

Technicality: Any class can define `equals` with no further formalities. In Java 1.0, to override (implement) `clone`, the class had to explicitly implement the `Cloneable` interface.

```
class ClonableBlob extends Blob
                   implements Cloneable {
   ... definitions of clone and equals
}
```

All array "objects" are `Cloneable` by default.
[To check: Is this still needed in Java 1.1, 1.2?]

Exceptions

Java has an extensive (and somewhat intrusive) exception mechanism. The basic idea is to surround a section of code with an exception handler.

```
try {
   thisMightExplode(x,y);
}  catch (Exception e) {
   <deal with problem>
}
```

If something goes wrong during execution of method `thisMightExplode` (or any method that it calls) the method detecting the error will execute

```
throw new someExceptionClass(parameters);
```

to create an exception object (of a class derived from `Exception`) and unwind the call chain until some routine catches the exception or terminates the program.

A method that calls a method that might throw an exception must either contain catch blocks for any exceptions that might be thrown, or include a "`throws`" clause in its heading to show that it might propagate the exception.

Output

There is an extensive stream library in `java.io`. It provides fairly reasonable support for simple text output.

`System.out` and `System.err` are the standard output and error output stream objects. These normally write to a console window. The basic methods are:

```
System.out.print( ... );   // print as text
System.out.flush( );     // flush buffered data
System.out.println( ... ); // print as text +
                           // newline & flush
```

These methods are overloaded for all of the basic types.

```
System.out.print(k);
System.out.println(" is a number");
```

But since most classes include `toString( )` to produce a string representation when needed, usually several things can be written at once.

```
System.out.println(k + " is a number");
System.out.println("The string representation"
   + " b is " + b)  // calls b.toString()
```


Input

Text input was (to say the least) very awkward in Java 1.0. A stream `System.in` is available, but it only allows for character input. Class `StreamTokenizer` is available to parse input into numbers and character strings, but the interface is fairly complicated. The basic object wrapper classes like `Integer` include methods to parse character strings into values of the appropriate type.

The text classes in Java 1.1 and later provide reasonably complete formatting capabilities, but are somewhat verbose and clumsy to use. The basic idea is fairly simple, however. The stream classes provide streams of bytes to and from various places (console, files, network connections). The text classes filter the raw bytes from streams and provide formatted input and output with appropriate conversions between data values and strings.

File I/O

`java.io` includes an extensive set of classes and methods that provide dialog boxes to select files and open files as input and/or output streams. Once a file is open, it works the same as any other I/O stream.

In Java 1.1 and earlier, Java Applets (executing in web browsers) are not allowed to read and write files. Java applications are allowed to do so, because they come from presumably trusted sources.

Java 1.2 has a much more flexible security model that allows "trusted" applets to access local resources like files.

Graphics

Package `java.awt` includes a fairly extensive 2-D graphics library. Here's a simple application that creates a window and draws a circle and label in it.

```
import java.awt.*;

// Class Drawing: a simple graphics window.
public class Drawing extends Frame {
   public void paint(Graphics g) {
      g.setColor(Color.black);
      g.drawOval(15,10,30,30);
      g.setColor(Color.red);
      g.drawString("circle",15,60);
   }
}

// Main program.  Create and display drawing
public class GraphicsApplication {
   public static void main(String args[]) {
      Drawing d = new Drawing();
      d.resize(200,150);
      d.setTitle("Drawing");
      d.show();
      d.toFront();
   }
}
```

Notes:

Class `Drawing` extends `Frame`, which ultimately derived from `Component`. A `Frame` is a simple window with a title bar and not much else.

Everything in a Java window — pictures, dialog buttons, text fields — is derived directly or indirectly from `Component`.

Method `paint` is called whenever an object becomes visible on the screen. For our drawing window, we define a simple `paint` to draw a circle and label it.

(Warning: `paint` is called whenever the underlying window manager needs to redraw the window. If you've got an interactive debugger and you're debugging a drawing window, be sure the debugger's windows don't overlap the drawing. If it does, you'll get lots of extra calls to `paint`.)