CSE 413 Winter 2001

Java Overview

References

- Concise overviews
  - *Java in a Nutshell* & *Java Examples in a Nutshell* (O'Reilly)
  - Budd, *Understanding Object-Oriented Programming with Java* (Addison-Wesley)
- Longer Tutorials and References
  - *The Java Tutorial* (A-W) (available online at www.java.sun.com)
  - Arnold & Gosling, *The Java Programming Language* (2nd ed, A-W)
  - *Learning Java* (O'Reilly)
  - *Core Java* (Prentice-Hall)
- For Language Lawyers
  - *The Java Language Specification* Gosling, et al (A-W)*
- Many online sources; see course web for links

Some History

- 1993 Oak project at Sun
- 1995 Oak becomes Java; every major web player announces support
- 1996 Java 1.0 available
- 1997 (March) Java 1.1 - some language changes and much larger library, including new event handling GUI model (AWT)
- 1997 (September) Java 1.2 beta - including Swing GUI package
- 1998 (October) Java 1.2 final
- 2000 (April) Java 1.3 final

Design Goals

- Support secure, high-performance, robust applications running as-is on multiple platforms and over networks
- "Architecture-neutral", portable, allow dynamic updates and adapt to new environments
- Look enough like C++ for programmer comfort
- Support object-oriented programming
- Support concurrency (multithreading)
- Simplicity

Hello World in Java

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


Classes

- Everything in Java is a member of some class
  - No external (global) functions or variables
- Classes may contain methods and data members
- Class members may be
  - non-static: one copy for each instance of the class (object)
  - static: single copy associated with the class, not with any specific instances.

"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."

Gosling & McGilton *Java White Paper*

Hello World Revisited

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

- Every class may have a main method
- Execution begins in main of a designated class
- Class Xyzzy should be in file Xyzzy.java

```
%javac HelloWorld.java
%java Helloworld
Hello World
```


Command Line Arguments

(if you like this sort of thing)

```
public class PrintArgs {
   public static void main (String [ ] args) {
        for (int k=0; k < args.length; k++)
          System.out.print(args[k] + " ");
        System.out.println();
   }
}
```

```
%javac PrintArgs.java
%java PrintArgs Testing one, two, three
Testing one, two, three
```


Primitive Data Types

- 2's complement signed integer
  - int (32 bits), byte (8), short (16), long (64)
  - constants are normally type int
- IEEE floating point
  - double (64 bits), float (32)
  - floating constants are normally type double
- Unicode characters: char (16 bits)
- Logical: boolean
  - constants are true, false
  - not interchangeable with int
- None of these are "implementation-defined" or "implementation-dependent"

Vars, Expressions & Assignment

- Almost same as C/C++

```
int k = 17;  boolean maybe; double x=42.0
k = 2 * k;  maybe = k > 17;
```

- Declaration initializers are optional. If omitted,
  - Fields in class instances initialized to 0, false, null.
  - Local vars in methods not initialized by default; compiler complains if use before initalize is possible
- Assignment does coercion if no information lost

```
double y = (k+6)/2;
```

- Assignment that could lose information requires explicit cast

```
k = (int) x * 1.3 / (x-2)
```


Basic statements

- if, while, for, and switch work as in C/C++
  - Use { } to create compound statements
  - Logical && and || are short-circuit
  - switch requires explicit break if fall-through to next case is not desired; if default case is not provided and no case label matches, execution silently proceeds with next statement.

```
if (x < y) { tmp=x; x=y; y=tmp;} else x=0;
while (k < n && a[k] != x)
     k++;
```


Class Definitions

- Basic use is to define template for instances

```
public class Blob {
  private int val;              // Blob state
  public int getVal( )          // access methods
    { return val; }
  public void setVal(int val)
    { this.val = val; }
  // yield string representation of this Blob
  public String toString( )
    { return "Blob: val = " + val; }
}
```

- toString( ) automatically used to cast object to String when used in context that requires String

Visibility

- Class members can be preceded by a qualifier to indicate accessibility
  - public - accessible anywhere the class can be accessed
  - private - only accessible inside the class
  - If nothing is specified, the field can be referenced anywhere in the same package (more later).
  - protected - same as package visibility, and also visible in classes that extend this class.

Instance Creation and References

- All variables that do not have a primitive type are references. Objects are only created by explicit allocation on the heap (with new).

```
Blob bob;              // no blob allocated yet
bob = new Blob( );     // Blob allocated here
bob.setVal(42);
int k = bob.getVal( );
System.out.println("bob is " + bob);
```


References and Methods

- Dot notation is used to select methods and fields; implicit dereference (no -> as in C/C++).
- No pointer arithmetic; no & operator to generate the address of arbitrary variable; can't create pointers from random bits.
  - "Java has no pointers"
- All method parameters are call-by-value (copy of primitive value or object reference)
- Methods can be overloaded (different methods with same name but different number or types of parameters).

Object Allocation

- A variable declared as class X has type "reference to X". No object is created by such a declaration.
- Declaration and object creation can be combined.

```
Blob bob = new Blob( );
```

- The constant null belongs to all reference types and refers to nothing.
- If reference r is null, then selecting a field from r (r.fieldname) throws a NullPointerException.
- Storage occupied by an object is dynamically reclaimed when the object is no longer accessible (automatic garbage collection).

Constructors

- Constructor(s) can be provided to initialize objects when they are created. Constructors can be overloaded and can call other constructors.

```
class Blob {
  int val;
  // constructors
  Blob (int initial) { val = initial; }
  Blob ( ) { this(17); }
}
```


Static Methods and Fields

- static class members are most commonly used for data and methods that are not naturally associated with a specific class instance.

```
class Math {          // standard Java Math class
  static double sqrt(double x) { … }
  static double sin(double x)  { … }
}
```

- Static methods are referenced via the class name

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

Symbolic Constants

- A class member may be qualified as final.
  - For data, it means the variable must be initialized when declared and cannot be changed after that.
  - For methods, it means the method cannot be overridden in a derived class.
  - In either case, the compiler can take advantage of this to inline the constant value or method code.

```
class Math {            // standard Java Math class
  static final double PI = 3.1415926535;
  static final double E = 2.71828182845;
}
…
area = Math.PI * r * r;
```


Arrays

- Arrays are dynamically allocated. Declaring an array variable only creates a reference variable; it does not actually allocate the array.

```
double[ ] a;
a = new double[6]
for (int k = 0; k < 6; k++)
  a[k] = 2*k;
```


Array Notess

- Arrays are 0-origin, as in C/C++
- Arrays are also objects, with one constant member
  - If a is an array, a.length is its length
- An IndexOutOfBoundsException is thrown if a subscript is < 0 or >= the array length.
- The brackets indicating an array type may also appear after the variable name, as in C/C++

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


2-D Arrays

- A 2-D array is really a 1-D array of references to 1-D array rows. The allocation

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

is really 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] = 0.0;
```


Arrays of Objects

- If the array elements have an object type, the objects must be created individually.

```
Blob [ ] list;
list = new Blob[10];
for (int k = 0; k < 10; k++)
  list[k] = new Blob;
```


Strings

- A character string “abc” is an instance of class String, and is a read-only constant.
  - Strings are objects; they are not arrays of chars.
  - There is no ‘\0’ byte at the end
  - If s is a string, s.length() is its length, and s.charAt(k) is the character in position k.
  - Class String contains many useful string processing functions.

Derived Classes

- A class definition may extend (be derived from) a single parent class (single inheritance).

```
class Point {
  int h, v;
}

class ColorPoint extends Point {
  Color c;
}
```


Derived Classes (cont.)

- All of the usual object-oriented notions are supported, including inheritance of fields and methods from superclasses and overriding.
- Inside a method, this refers to the current object; super refers to the current object viewed as an instance of the parent class.
- There is a single class Object at the root of the class hierarchy.
  - If a class declaration does not explicitly extend some class, it implicitly extends Object.
- A class may be declared abstract if it is an interface that must be extended to be used.
- A final class may not be extended further.

Wrapper Classes for Basic Types

- For each basic type (int, double, etc.) there is a corresponding class (Integer, Double, etc.) that is an object version of that type.
  - Integer(17) is an object representation of the int 17.
  - Particularly useful with container classes that can only hold objects (Vector, HashTable, etc.)
  - Wrapper classes also contain many useful utility functions and constants.
    - if (k < (Integer.MAX_VALUE/10)) …
    - if (Character.isLowerCase(ch)) …

Interfaces

- Interfaces allow specification of constants and methods independently of the class hierarchy.
- Interfaces may extend other interfaces, but since they are pure specification, no implementation is inherited.

```
interface AbsType {
  static final int one = 1;
  static final int two = 2;
  void f(int a, int b);
  double g( );
}
```


Interfaces (cont)

- A class may implement as many interfaces as desired.
  - Full implementation of all methods in the interface must be provided by the class or inherited from a parent class. Nothing is inherited from the interface.
  - Gives most of the useful effects of multiple inheritance
    - Allows otherwise unrelated classes to implement common behavior
  - Some interfaces are “markers” - identify classes that can be used in certain contexts
    - Widely used for event handling in the Java user interface (MouseMotionListener, ActionListener, many others)

Object Compare and Copy

- Default assignment and comparison only copies or compares references (shallow operations)

```
Blob b = new Blob( );
Blob c = new Blob( );
if (b==c) System.out.println(“Something wrong”);
c = b;
b.setVal(100);
System.out.println( c.getval( ) );
```

Deep Compare and Copy

- All classes inherit equals and clone from Object
  - Default versions do a shallow compare/copy
  - Override if a deep compare/copy is desired
  - To override clone, a class must also extend the Cloneable interface (this is purely a marker interface, has no methods or constants)
- Intended meaning of a.equals(b) is that a and b are "equal" in whatever sense is appropriate for the class of a and b.
- b.clone should create a new "copy" of b and return a reference to it.

Exceptions

- Java has an extensive exception handling mechanism. Basic idea

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

- If an exception happens, a

```
throw new anExceptionClass(parameters);
```

statement will cause the call chain to unwind until a catch clause that matches the thrown object is found.

Exceptions (cont)

- Multiple catch clauses can be used to selectively handle exceptions

```
try {
  tryToReadData(x,y,z);
} catch (IOException e) {
  <deal with I/O problem>
} catch (Exception e) {
  <deal with other exceptions>
}
```

- If a method does something that might generate an exception, it must either handle it, or declare that it might throw that exception (throws clause).

Packages

- Packages provide a way to partition the global class namespace.
- A class is placed in a package by including at the beginning of class source file

```
package widget;
```

- A class in another package can use items from a package by explicitly qualifying the item name

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

or by importing names from the package

```
import widget.*;
…
Blob b = new Blob( );
```


Packages (cont)

- Package names are grouped into hierarchies by using package names with embedded dots
  - java.util, java.awt, java.awt.image
- Parts of a package hierarchy can be selectively imported.
- import is not transitive (unlike C/C++ #include)
- If a class definition does not include a package statement, that class is part of a default anonymous package.
  - Useful for small projects

Streams

- Stream = flow of data (bytes or characters)
- Can be associated with files, communication links, keyboard/screen/printer
- Many stream classes; most are designed to be used as wrappers that accept data and transform or filter it before passing it along
- Java 1.0: Byte streams with a few wrappers to handle ASCII text
- Java 1.1: Added text stream classes to handle Unicode text properly

Stream Classes (1)

- **`InputStream/OutputStream`** - abstract classes defining basic raw byte stream operations
- **`Reader/Writer`** - abstract classes defining basic text stream operations

All Java stream classes are built on top of these

- **`InputStreamReader/OutputStreamWriter`** - basic conversion between bytes and characters (in both directions)

Stream Classes (2)

- **`BufferedInputStream/ BufferedOutputStream BufferedReader/BufferedWriter`** - versions of streams that add buffering and additional input/output methods
- **`PrintWriter`** - Text stream with methods for printing **`String`**s and primitive types as text output.

Stream Classes (3)

- **`DataInputStream/DataOutputStream`** - Filter streams that can read/write simple types including **`String`** and primitive numeric types as binary byte streams.
- **`FileInputStream/FileOutputStream FileReader/FileWriter`** - byte and text streams that read and write from/to the local file system.

Ex: Read a byte from Keyboard

- **`System.in`** is an **`InputStream`**. At the lowest level, we can read bytes. As in C, the basic **`read()`** operation returns an **`int`**, with -1 indicating end of stream.

```
try {
  int nibble = System.in.read();
} catch (IOException e) { … }
```


Ex: Read Line from Keyboard

- To read lines of characters, convert **`System.in`** to a character stream, and wrap it in a **`BufferedReader`** to get **`readLine()`**.

```
try {
  InputStreamReader chars =
   new InputStreamReader(System.in);
  BufferedReader in =
   new BufferedReader(chars);
  String firstLine = in.readLine();
  …
} catch (IOException e) { … }
```


File I/O

- The file stream classes have constructors that take a filename as an argument and open the file.

```
Try {
  FileReader theFile =
     new FileReader("input.dat");
  BufferedReader input =
     new BufferedReader(theFile);
  String line = input.readLine();
  System.out.println(line);
} catch (IOException e) { … }
```

- Gotcha: File names depend on the underlying file system -- hard to be completely "platform independent".