CMSC 433, Fall 2002
Programming Language
Technology and Paradigms
Basic Java

Adam Porter
Sep 5, 2002

Administrivia
• Project 1 posted today
• Java readings from Thinking in Java
  – we’ll do parts of many chapters, but definitely read Chapter 1 for an overview
  – I’ll add some suggestions on Web page, but use it as a reference

Outline
• Object oriented programming principles
  – How Java realizes them
  – How Java differs from C++
• Useful information on Java
  – Using the compiler
  – I/O libraries
  – Container classes

Object Orientation
• Abstraction
  – focus on essential properties, ignore unimportant details
• Encapsulation
  – separate external, visible behavior from internal, hidden behavior

Object Orientation (cont’d)
• Combining data and behavior
  – objects, not developers, decide how to carry out operations
• Sharing
  – similar operations and structures are implemented once
• Emphasis on object-structure rather than procedure structure
  – behavior more stable than implementation
  – … but procedure structure still useful

And one more …
• Security and Reliability
  – Write code that works, and is not insecure
Java

• Similar to C++, but with “unsafe” features removed, and others added
• Fully specified, compiles to virtual machine
  – machine-independent
• Secure
  – bytecode verification (“type-safe”)
  – security manager

Java design

• Everything is an Object
  – Allows sharing, generics, and more
  – There are primitive int, long, float values, etc.
• Interfaces and Inheritance
  – Allows role sharing and code sharing
  – Simulates multiple inheritance
• Security and reliability-conscious
  – Strong type system
  – Garbage collection
  – Exceptions

Wrapper classes

• To create Integer, Boolean, Double, …
  – that is a subclass of Object
  – useful/required for polymorphic methods
  • HashTable, LinkedList, …
  – used in reflection classes
• Include many utility functions
  – e.g., convert to/from String
• Number: superclass of Byte, Short, Integer, Long, Float, Double
  – allows conversion to any other numeric primitive type

Java libraries and features

• Utilities
  – collection classes, Zip files, internationalization
• GUIs, graphics and media
• Networking
  – sockets, URLs, RMI, CORBA
• Threads
• Databases
• Cryptography/security

Some of what’s missing from C++

• preprocessor (#include, #define, …)
• structs and unions
• enumerated types
• bit-fields
• variable-length argument lists
• multiple inheritance (of implementation)
• operator overloading
• templates/parameterized types
  – GJ (generic Java)

Naming conventions

• Classes/interfaces start with a capital letter
• packages/methods/variables start with a lowercase letter
• for names with multiple words, CapitalizeFirstLetterOfEachWord
• don’t use underscores
• CONSTANTS all in uppercase
Object-oriented programming in Java

Using Complex

```java
public static void main(String[] args) {
    Complex a = new Complex(5.5, 9.2);
    Complex b = new Complex(2.3, -5.1);
    Complex c, d;
    c = a.plus(b);
    d = a.plus(b);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c.equals(d): " + c.equals(d));
    System.out.println("c == d: " + (c == d));
}
```

prints:
```
a = (5.5,9.2)
b = (2.3,-5.1)
c.equals(d): false
c == d: false
```

Java Classes and Objects

• Each object is an instance of a class
  – an array is an object
• Each class extends one superclass
  – Object if not specified
  – class Object has no superclass

instance operations

• = assignment
  – for object references: copies reference, not object
• == equality test
  – for object references: true if references to same object

Some Object operations

• o.equals(bar)
  – intended to compare contents of objects
• o.toString()
  – returns String representation of the object, can/should be overridden
Adding equals to Complex

```java
public class Complex {
    ...
    public boolean equals(Object o) {
        if (o instanceof Complex) {
            Complex c = (Complex) o;
            return (c.r == this.r &&
                    c.i == this.i);
        } else {
            return false;
        }
    }
}
```

Using Complex again

```java
public static void main(String[] args) {
    Complex a = new Complex(5.5, 9.2);
    Complex b = new Complex(2.3, -5.1);
    Complex c, d;
    c = a.plus(b);
    d = a.plus(b);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("c.equals(d): " + (c.equals(d)));
    System.out.println("c == d: " + (c == d));
}
```

prints:
```
a = (5.5, 9.2)
b = (2.3, -5.1)
c = (7.8, 4.1)
c.equals(d): true
c == d: false
```

Objects and references

- All objects allocated on the heap with `new()`
  - All variables of non-primitive type are references to an object or `null`
  - No object can “contain” another object
  - No objects stack-allocated (only references there)
- Reference is like a C++ pointer, except
  - can only point to start of heap-allocated object
  - no pointer arithmetic allowed
  - use . instead of -> to access fields/methods

String example

```java
public class String {
    char[] data;
    int length;
    public String(String s) {
        data = s.toCharArray();
        length = s.length();
    }
    public int getLength() { return length; }
    public boolean equals(String s) { ... }
}
```

Different from C++

- No malloc()
  - Only `new()`
- No free()
  - Uses garbage collection
- No pointer operations: *, &, ->, +, ++, etc.
  - Simplifies usage and implementation
- Method parameters pass-by-value
  - but object parameters are references to heap objects that can be changed

More Object Obligations

- many operations have default implementations
  - which may not be the ones you want
  ```java
  public boolean equals(Object that) { ... }
  public String toString() { ... }
  public int hashCode() { ... }
  ```
  - important that a.equals(b) implies a.hashCode()==b.hashCode() 
  - public void finalize() { ... } // called before object garbage 
  - public Object clone() { ... } // default is shallow bit-copy if class 
    implements Cloneable, throw CloneNotSupportedException 
  - otherwise
**Class modifiers**

- **public** – class visible outside package
- **final** – no other class can extend this class
- **abstract** – no instances of this class can be created
  - only instances of extensions of the class
- No modifier implies *package*-level scope

**Variable / method visibility**

- **public** – visible everywhere
- **private** – visible only within this class
- **protected** – visible within same package or in subclass
- **package** (default) – visible within same package

**Instance vs. static variables**

- **static** – the data is stored “with the class”
  - static variables allocated once, no matter how many objects created
  - static methods are not specific to any class instance, so can’t refer to *this* or *super*
- Can reference class variables and methods through either class name or an object ref
  - Clearer to reference via the class name

**Examples**

- **public static** void main(String args[]) { … }
- public class Math {
  public final static PI = 3.14159…;
}
- public class System {
  public static PrintStream out = …;
}

**Instance variable modifiers**

- **final** – can’t be changed; must be initialized in declaration or in constructor

- **transient, volatile**
  - will cover later

**Method modifiers**

- **final** – this method cannot be overridden
  - useful for security
  - allows compiler to inline method
- **abstract** – no implementation provided
  - class must be abstract
- **native, synchronized**
  - will cover later
Method invocation

- Method names can be overloaded
  - method invoked is determined by both its name and the types of the parameters
  - resolved at compile-time, based on compile-time types
- Methods can also be overridden
  - define a method also defined by a superclass
  - arguments and result types must be identical
  - resolved at run-time, based on type of object method is invoked on

Overloading

- Methods with the same name, but different parameters (count or types) are overloaded

```
class Parent {
    int cost;
    void add(int x) {
        cost += x;
    }
}
class Child extends Parent {
    void add(String s) throws NumberFormatException {
        cost += Integer.parseInt(s);
    }
}
```

Overriding

- Overriding
  - methods with same name and argument types in child class override method in parent class
  - you can override/hide instance variables
    - both variables will exist, but don’t do it

```
class Parent {
    int cost;
    void add(int x) {
        cost += x;
    }
}
class Child extends Parent {
    void add(int x) {
        if (x > 0) cost += x;
    }
}
```

Dynamic Method Dispatch

- If you have a ref a of type A to an object that is actually of type B (a subclass of A)
  - instance methods invoked on a will get the methods for class B (like C++ virtual functions)
  - class methods invoked on a will get the methods for class A
    - invoking class methods on objects strongly discouraged

```
class A {
    String f() {return “A.f()”; }
    static String g() {return “A.g()”; }
}
class B extends A {
    String f() {return “B.f()”; }
    static String g() {return “B.g()”; }
    public static void main(String args[]) {
        A a = new B();
        B b = new B();
        System.out.println(a.f() + a.g() + b.f() + b.g());
    }
}
```

Simple Dynamic Dispatch Example

```
java B generates:
```

Self reference

- this refers to the object the method is invoked on
- super refers to the same object as this
  - but used to access methods/variables in superclass
- Like C++
Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can be this(args) or super(args)
  - if those are omitted, super() is called
  - must be very first statement, even before variable declarations
- not used for type conversions or assignments
- void constructor generated if no constructors given

Garbage collection

- Objects that are no longer accessible can be garbage collected
- Method void finalize() called when an object is collected
  - best to avoid using it, since no way to tell when it will get called
- Garbage collection not a major performance bottleneck
  - new/delete in C++ can be expensive too

Detailed Example

- Shows
  - polymorphism for both method receiver and arguments
  - static vs. instance methods
  - overriding instance variables

Source code for classes

class A {
    String f(A x) { return "A.f(A) "; }
    String f(B x) { return "A.f(B) "; }
    static String g(A x) { return "A.g(A) "; }
    static String g(B x) { return "A.g(B) "; }
    String h = "A.h";
    String getH() {return "A.getH(): " + h; }
}
class B extends A {
    String f(A x) { return "B.f(A)/ " + super.f(x); }
    String f(B x) { return "B.f(B)/ " + super.f(x); }
    static String g(A x) { return "B.g(A) "; }
    static String g(B x) { return "B.g(B) "; }
    String h = "B.h";
    String getH() {return "B.getH(): " + h + " / " + super.h; }
}

Invocation and results

A a = new A();  A ab = new B();  B b = new B();
System.out.println( a.f(a) + a.f(ab) + a.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)  B.f(A)  B.f(B)
System.out.println( ab.f(ab) + ab.f(ab) + ab.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)  B.f(A)  B.f(B)
System.out.println( ab.f(ab) + ab.f(ab) + ab.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)  B.f(A)  B.f(B)
System.out.println( ab.f(ab) + ab.f(ab) + ab.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)  B.f(A)  B.f(B)
System.out.println( ab.f(ab) + ab.f(ab) + ab.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)  B.f(A)  B.f(B)
System.out.println( a.g(a) + a.g(ab) + a.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( ab.g(ab) + ab.g(ab) + ab.getH());
// A.h  A.getH():A.h
System.out.println( ab.h + ab.getH());
// A.h  B.getH():A.h
System.out.println( ab.h + ab.getH());
// B.h  B.getH():B.h/A.h

What to notice

- Invoking ab.f(ab) invokes B.f(A)
  - run-time type of object determines method invoked
    - compile-time type of arguments used
- ab.h gives the A version of h
- ab.getH() (in super) method invoked
  - in B.getH(), h gives B version of h
- Use of super in class B to reach A version of methods/variables
- super not allowed in static methods
Interfaces

- An interface lists supported methods
  - No constructors or implementations allowed
  - Can have final static variables
- A class can implement (be a subtype of) one or more interfaces
- Using the name of an interface as a type (i.e. to declare a variable) means
  - a reference to any instance of a class that implements the interface is a permitted value
  - null is also allowed

Interface example

```java
public interface Comparable {
    public int compareTo(Object o);
}

public class Util {
    public static void sort(Comparable[] a) {
        // ... implementation...
    }
}

public class Choices implements Comparable {
    public int compareTo(Object o) {
        return ...;
    }
}

Choices[] options = ...;
Util.sort(options);
```

No multiple inheritance

- A class type can be a subtype of many other types (implements)
- But can only inherit method implementations from one superclass (extends)
- Not a big deal
  - multiple inheritance rarely, if ever, necessary and often badly used
- And it’s complicated to implement well

Poor man’s polymorphism

- Every object is an Object
- An Object[] can hold references to any objects
- E.g., for a data structure Set that holds a set of Object
  - can use it for a set of String
  - or a set of images
  - or a set of anything
- Java’s container classes are all containers of Object
  - when you get a value out, have to downcast it

Downcasting

- (Bar) foo
  - run-time exception if object reference by foo is not a subclass of Bar
  - compile-time error if Bar is not a subtype of foo (i.e. it always throws an exception)
  - doesn’t transform anything, just allows treating the result as if it were of type Bar
- o instanceof Foo returns true iff o is an instance of a subclass of Foo

Array types

- If S is a subtype of T
  - S[] is a subtype of T[]
- Object[] is a supertype of all arrays of reference types
- Storing into an array generates a run-time check that the type stored is a subtype of the declared type of the array elements
Example: Object[]

```java
public class TestArrayTypes {
    public static void reverseArray(Object [] A) {
        for(int i=0, j=A.length-1; i<j; i++,j--) {
            Object tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
        }
    }
    public static void main(String [] args) {
        reverseArray(args);
        for(int i=0; i < A.length; i++)
            System.out.println(args[i]);
    }
}
```

Class Objects

- For each class, there is an object of type `Class`
- Describes the class as a whole
  - used extensively in Reflection package
- `Class.forName("MyClass")`
  - returns class object for `MyClass`
  - will load `MyClass` if needed
- `Class.forName("MyClass").newInstance()`
  - creates a new instance of `MyClass`
- `MyClass.class` gives the `Class` object for `MyClass`

Exceptions

- On an error condition, we `throw` an exception
- At some point up the call chain, the exception is `caught` and the error is handled
- Separates normal from error-handling code
- A form of non-local control-flow
  - Like goto, but structured

Throwing an Exception

- Create a new object of the class Exception, and `throw` it
- Exceptions thrown are part of return type
  - when overriding a method in a superclass
  - can’t throw anything that would surprise a superclass object

Method throws declarations

- A method declares the exceptions it might throw
  - `public void openNext() throws UnknownHostException, EmptyStackException`
  - `...`
- Must declare any exception the method `might` throw
  - unless it is caught in the method
  - includes exceptions thrown by called methods

Exception Handling

- All exceptions eventually get caught
- First `catch` with supertype of the throwable catches it
- `finally` is always executed
java.lang.Throwable

- Exception is a subclass of Throwable
- Many objects of class Throwable have a message
  - specified when constructed, as String
- String getMessage() returns the message
- String toString()
- void printStackTrace()
- void printStackTrace(PrintWriter s)

Example Application

Public class BufferedReader {
    public String readLine() throws IOException {
        ... } ...
}

public class Echo {
    BufferedReader in = ...
    try {
        while((s = in.readLine()) != null)
            System.out.println(s);
    } catch(Exception e) {
        System.out.println(e.stackTrace());
    }
}

Creating New Exceptions

- User-defined exception is just a class that is a subclass of Exception

class MyOwnException extends Exception {}
class MyClass {
    void oops() throws MyOwnException {
        if (some_error_occurred) {
            throw new MyOwnException();
        }
    }
}

Interacting with External Environment

Applications and I/O

- Java external interface is a public class
- via public static void main(String [] args)
- args[0] is first argument
  - unlike C/C++
- System.out and System.err are PrintStreams’s
  - should be PrintWriter’s, but would break 1.0 code
  - System.out.print(...) prints a string
  - System.out.println(...) prints a string with a newline
- System.in is an InputStream
  - not quite so easy to use

Input (JDK 1.1 and higher)

- Wrap System.in in an InputStreamReader
  - converts from bytes to characters
- Wrap the result in a BufferedReader
  - makes input operations efficient
  - supports readline() interface
- readline() returns a string
  - returns null if at EOF
Example Echo Application

```java
import java.io.*;
public class Echo {
    public static void main(String[] args) {
        String s;
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int i = 1;
        try {
            while((s = in.readLine()) != null)
                System.out.println((i++) + " : " + s);
        } catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

Java Libraries

I/O and Utility Libraries

You should familiarize yourself

- Packages
  - java.lang
  - java.util
  - java.net
  - java.io
- Read the documentation on line

I/O Classes

- **OutputStream** – byte stream going out
- **Writer** – character stream going out
- **InputStream** – byte stream coming in
- **Reader** – character stream coming in

OutputStream - bytes

- base types
  - `ByteArrayOutputStream`
  - `FileOutputStream` – goes to file
  - `PipedOutputStream` – goes to `PipedInputStream`
  - `SocketOutputStream` (not public) – goes to TCP socket
- Filters – wrapped around an `OutputStream`
  - `BufferedOutputStream`
  - `ObjectOutputStream` (should implement `FilterOutputStream`) – serialization of object graph
**Writer - characters**

- **OutputStreamWriter**
  - wraps around OutputStream to get a Writer
  - takes characters, converts to bytes
  - can specify encoding used to convert
- **CharArrayWriter**
- **StringWriter**
- **Filters**
  - **PrintWriter** – supports print, println
  - **BufferedWriter**
- **Convenience writers**
  - wrap OutputStreamWriter around an OutputStream
  - **FileWriter** and **PipedWriter**

**InputStream - bytes**

- **base types**
  - **ByteArrayInputStream**
  - **FileInputStream**
  - **PipedInputStream**
    - **SocketInputStream** (not public) – comes from TCP socket
  - Filters – wrapped around **InputStream**
    - **BufferedInputStream**
    - **PushedBackInputStream**
    - **ObjectInputStream**
- **SequenceInputStream** - concatenate

**Reader - characters**

- **InputStreamReader**
  - wrap around InputStream to get a Reader
  - takes bytes, converts to characters
  - can specify encoding used to convert
- **CharArrayReader**
- **StringReader**
- **Filters**
  - **BufferedReader** – efficient, supports readLine()
    - **LineNumberReader** – reports line numbers
  - **PushBackReader**
- **Convenience Readers**
  - wrap InputStreamReader around InputStream
  - **FileReader** and **PipedReader**

**java.util**

- Lots of stuff
  - **Vector**
  - **Dictionaries**
  - **Enumerations and Bitsets**
  - **Collection classes**
- We will focus on Collection classes

**Other libraries**

- **java.lang.Math**
  - abstract final class – only static members
  - includes constants $e$ and $\pi$
  - includes static methods for trig, exponentiation, min, max, …
- **java.text**
  - text formatting tools
    - class **MessageFormat** provides printf/scanf functionality
  - lots of facilities for internationalization

**Java Container Classes**

- A unified architecture for representing and manipulating collections of objects
- Container classes contain three things:
  - Interfaces: abstract data types representing collections of objects
  - Implementations: concrete implementations of the collection interfaces
  - Algorithms: methods that perform computations on objects that implement collection interfaces
Container class hierarchy

Collection Classes
- Collections contain groups of objects (elements)
- Collection interface is not implemented in Java.
  Subinterfaces implemented
  - Set: unordered, can’t contain duplicate elements
    - HashSet: unordered, no duplicates
    - TreeSet: ordered, no duplicates
  - List: ordered, can contain duplicate elements
    - LinkedList: unordered, dynamic size, add/delete quick
    - ArrayList: unordered, dynamic size, random access

Map Classes
- A Map is an object that contains key:value pairs
- Maps cannot contain duplicate keys:
  - Each key can map to at most one value
- Map not implemented. Subinterfaces implemented
  - HashMap, entries stored in a hash table
  - TreeMap, entries maintained in sorted order
- Variants
  - Ordered/unordered (e.g., map vs. sorted map)

Object Ordering
- Two ways to order objects:
  - Comparable interface provides automatic
    natural order on classes that implement it
  - Comparator interface gives the programmer
    complete control over object ordering

compareTo Interface
- public int compareTo(Object o)
- The natural comparison method (i.e., default)
  - Returns a negative integer, zero, or a positive integer as this object
    is less than, equal to, or greater than o
  - sgn(x.compareTo(y)) = sgn(y.compareTo(x))
  - (x.compareTo(y)>0 && y.compareTo(z)>0) => x.compareTo(z)>0.
  - x.compareTo(y)==0 => sgn(x.compareTo(z)) = sgn(y.compareTo(z))
- Recommended that (x.compareTo(y)==0) == (x.equals(y))

Comparator Interface
- When natural order isn’t acceptable
- public int compare(Object o1, Object o2)
  - Returns a negative integer, zero, or a positive integer as the first
    argument is less than, equal to, or greater than the second
  - sgn(compare(x, y)) = sgn(compare(y, x))
  - (compare(x, y)>0 && compare(y, z)>0) => compare(x, z)>0.
  - compare(x, y)==0 => sgn(compare(x,z))=sgn(compare(y,z))
  - recommended (compare(x, y)==0) == (x.equals(y))
- public boolean equals(Object obj)
  - Indicates whether some other object is “equal to” this Comparator.
Example 1

```java
import java.util.*;
import java.awt.*;
class MyPoint extends java.awt.Point implements Comparable {
    MyPoint(int x, int y) {super(x,y);}
    public int compareTo(Object o) {
        MyPoint p = (MyPoint)o;
        double d1 = Math.sqrt(x*x + y*y);
        double d2 = Math.sqrt(p.x*p.x + p.y*p.y);
        if (d1 < d2) {return -1;}
        else if (d2 < d1) {return 1;}
        return 0;
    }
}
```

```java
class Sort3 {
    public static void main(String[] args) {
        Random rnd = new Random();
        MyPoint[] points = new MyPoint[10];
        for (int i=0; i<points.length; i++) {
            points[i] = new MyPoint(rnd.nextInt(100), rnd.nextInt(100));
            System.out.println(points[i]);
        }
        System.out.println("-----------");
        Arrays.sort(points);
        //Print the points
        for (int i=0; i<points.length; i++) {
            System.out.println(points[i]);
        }
    }
}
```

Output

Sort2 …….. // after sort
MyPoint[x=1,y=95]
MyPoint[x=2,y=16]
MyPoint[x=3,y=26]
MyPoint[x=12,y=95]
MyPoint[x=22,y=55]
MyPoint[x=39,y=73]
MyPoint[x=31,y=42]
MyPoint[x=66,y=33]
MyPoint[x=70,y=33]
MyPoint[x=80,y=31]

Sort3 …….. // after sort
MyPoint[x=2,y=0]
MyPoint[x=0,y=15]
MyPoint[x=18,y=4]
MyPoint[x=39,y=13]
MyPoint[x=39,y=19]
MyPoint[x=42,y=23]
MyPoint[x=65,y=5]
MyPoint[x=38,y=74]
MyPoint[x=80,y=40]
MyPoint[x=87,y=62]