CMSC 433, Spring 2003
Programming Language Technology and Paradigms
Basic Java

Jeff Foster
January 30, 2003

Administrivia

• Meet your TA
• Project 1 posted, due February 12, 2003
• You should have received e-mail from me
  – And a class account
• Reading: Liskov ch. 1 and 2

Outline

• Object oriented programming principles
  – How Java realizes them
  – How Java differs from C++
• Useful information on Java (not covered in class)
  – Using the compiler
  – I/O libraries
  – Container classes

Software Engineering Goals

• Reliability (it works!)
• Performance
• Reusability (write-once, then reuse)
• Maintainability
  – Easy to modify/extend
  – Easy to understand
• Quick development time

And One More Goal…

• Security
  – Write code that is as secure as possible
  – We will not discuss much in class
  – ...but note some security concerns in project 1

Basic Engineering Techniques

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

CMSC 433, Jeff Foster, U. Maryland (via M. Hicks, W. Pugh, A. Sussman, and A, Porter)
Example: Abstraction

```java
... int sum(int[] A) {
    int s = 0;
    for (i = 0; i < A.length; i++)
        s += A[i];
    return s;
}
...
```

Example: Encapsulation

```java
class Unique {
    private int x;
    Unique() { x = 0; }
    int getUnique() {
        return x++;
    }
}
```

Also:

Collection

Map

SortedMap

Going Overboard

- Don’t abstract everything in sight
  - Only apply abstractions if they will help you meet your goals
  - E.g., don’t create a function used only once
- Don’t encapsulate and then expose
  - E.g., if you write get and set methods for a field, why is it private?

Object Orientation

- Combining data and behavior
  - objects, not developers, decide how to carry out operations
- Sharing via abstraction and inheritance
  - 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

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 inherits from Object*
  - Allows sharing, generics, and more

```
Object
  Number
  Thread
  Integer
  ... *
```

* Well, almost: there are primitive int, long, float, etc.
Java Design

- **Inheritance**
  - Hierarchical code sharing (“is-a”)
- **Interfaces**
  - For “mixins” & non-hierarchical frameworks

```
Object
  \|-- Comparable
    \|-- Number
```

Java Design

- Security and reliability
  - **Strong type system**
    - Object o = (Object)27; not allowed!
  - Garbage collection
    - No free()
  - Exceptions
    - Separation of error-handling from algorithm

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, …)
- Some “low-level” types
  - structs and unions
  - enumerated types
  - bit-fields
- Some function features
  - variable-length argument lists
  - operator overloading
- Some class features
  - multiple inheritance (of implementation)
  - templates/parameterized types (but now in 1.4!)

Naming conventions

- Classes/Interfaces start with a capital letter
  - `Object`, `Number`, `Thread`, …
- packages/methods/variables start lowercase
  - `Thread` myThread = new `Thread`();
  - `java.lang`, `org.xml.sax`
- Capitalize multi-word names (no underscores)
  - `SortedList`, `compareTo`, `toBinaryString`
- CONSTANTS all in uppercase (use underscores)
  - `PI`, `E`, `MAX_VALUE`

Object-oriented programming in Java
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

Objects have methods

- All objects, therefore, inherit them
  - Default implementations may not be the ones you want
    - The object.equals() method
    - returns print representation
    - key for hash table
    - called when object garbage-collected
    - And others …

Objects and references

- All objects allocated on the heap with new()
  - All variables of non-primitive type are references to an object or null; assignment (=) copies references
  - 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

- String example

Mutability

- An object is mutable if its state can change
- An object is immutable if its state never changes
  - Once its been initialized
- Why was sharing from substring safe?
  - Strings are not mutable

Equality

- Object .equals() method
  - Structural (“conceptual”) equality
- == operator
  - true if arguments reference the same object
  - o == p [ ] o.equals(p)
class Complex – a toy example

public class Complex {
private double r, i;
public Complex(double r, double i) {
this.r = r;
this.i = i;
}
public String toString() {
return "(" + r + "," + i + ")";
}
}

public Complex plus(Complex that) {
return new Complex(
this.r + that.r,
this.i + that.i);
}

Using Complex

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): false
c == d: false

Adding equals to Complex

public class Complex {
... public boolean equals(Complex c) {
return (c.r == this.r && c.i == this.i);
}
}

This is wrong!

Using Complex again

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

Adding equals to Complex

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;
}
}
}

Runtime test to determine the object’s actual class

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)
  – no effect at run-time; just treats the result as if it were of type Bar
• o instanceof Foo returns true iff o is an instance of a subclass of Foo
Different from C++

• No malloc()
  – Only new
• No free()
  – 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

Visibility Modifiers

• Indicate visibility of
  – Classes
  – Methods
  – Fields
• Support abstraction
  – Clients unaffected by change in implementation
• Support encapsulation
  – Prevents leaking of information to clients

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

Instance vs. static

Class definition

Public class Foo {
  int foo;
  static int bar;
}

Class implementation

Foo
int bar;
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