Last time - Java

- Arrays
  - are objects, with special syntax
  - space allocated with new

- Classes
  - can extend only one class
  - can implement many interfaces

- Methods
  - can be overloaded, and overridden

Administrivia

- C++ project
  - private drivers posted soon
  - other student’s code will be sent to you for commentary
  - 2 projects, about a paragraph for each

- Java project
  - web server is up and running for testing

- More Java suggested readings
  - I/O – Chapter 11, pages 573-605
  - RTTI – Chapter 12
  - Distributed Computing – Chapter 15, pages 903-923

Instance variable / method modifiers

- Visibility/access
  - public – visible everywhere
  - protected – visible within same package or in subclass
  - package (default) – visible within same package
  - private – visible only within this class
  - static – a class method or variable

Instance variable modifiers

- transient – not stored when object serialized
- volatile – don’t assume the variable hasn’t changed since the last time it was accessed
  - might be modified by another thread that doesn’t have a lock on the object
- final – can’t be changed; must be initialized in declaration or in constructor

Method modifiers

- abstract – no implementation provided
  - class must be abstract
- final – this method cannot be overridden
  - useful for security
  - allows compiler to inline method
- native – implemented in another language
- synchronized
  - locks object before method is executed
  - lock released after method finishes
Method arguments

- Only pass-by-value
  - but object parameters are references to heap objects that can be changed
- Only arguments used to distinguish methods
  - not return types
- Syntax same as C/C++

Overriding

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

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

Overloading

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

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

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

Simple Dynamic Dispatch Example

```java
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 class B extends A {
    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());
    }
}
```

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; }
}

A a = new A();  A ab = new B();  B b = new B();
System.out.println( a.f(a) + a.f(ab) + a.f(b) );
System.out.println( ab.f(a) + ab.f(ab) + ab.f(b) );
System.out.println( b.f(a) + b.f(ab) + b.f(b) );
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)/A.f(A)  B.f(A)/A.f(A)  B.f(B)/A.f(B)
System.out.println( a.g(a) + a.g(ab) + a.g(b) );
System.out.println( ab.g(a) + ab.g(ab) + ab.g(b) );
System.out.println( b.g(a) + b.g(ab) + b.g(b) );
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
System.out.println( a.h + "/" + a.getH() );
System.out.println( ab.h + "/" + ab.getH() );
System.out.println( b.h + "/" + b.getH() );
// A.h  A.getH():A.h
// 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() gives the B version of h
- Use of super in class B to reach A version of methods/variables
- super not allowed in static methods

### Static class components

- They belong to 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
  - don’t reference via object references!

### Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can/should be this(args) or super(args)
  - if these 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

### Interfaces

- An interface is an object type – no associated code or instance variables
  - only describes methods supported by interface
- A class can implement (be a subtype of) many interfaces
- Interfaces may have final static variables
  - to define a set of constants (like enum in C++)
Interface example

```java
public interface Comparable {
    public int compareTo(Object o);
}
public class Util {
    public static void sort(Comparable[] options) {
        // ... 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

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

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`

Administrivia

- C++ project
  - private drivers posted soon
  - comments on other students projects due 3/7
  - submit project 11 (e.g. `submit 11 comment.txt`)
    - if you care about someone seeing your name on a project, tell me
    - you will get the comments back for your project
- Java project
  - use jikes for compiling, in ~ pugh/bin
    - faster than javac
- Out of town Tuesday, guest lecturer
  - no office hours Tuesday

CMSC 433, Alan Sussman, U. Maryland (via Bill Pugh)
Last time - Java

- Instance variable and method modifiers
  - visibility, final, abstract, etc.
- Overriding – dynamic dispatch
- Overloading
- Class/static variables and methods
- Interfaces – to specify method signatures
- Reflection – Class objects

Types

- A type describes a set of values that can be:
  - held in a variable
  - returned in an expression
- Types include:
  - primitive types – boolean, char, short, int, …
  - Reference types:
    - Class types
    - Array types
    - Interface types

Class types

- Using the name of a class as a type means a reference to an instance of that class or a subclass is a permitted value
  - a subclass has all the fields of its superclass
  - a subclass has all the methods of its superclass
- null is also an allowed value

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
- Performance penalty?
- Similar (and maybe worse) problems in C++

Example: Object[]

```
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]);
    }
}
```

Interface types

- Using the name of an interface as a type means
  - a reference to any instance of a class that implements the interface is a permitted value
  - null is also allowed
- Object referenced is guaranteed to support all the methods of the interface
  - invoking a method on an interface might be a bit less efficient
Object Obligations

- many operations have default implementations
  - which may not be the ones you want

  ```java
  public boolean equals(Object that) { … } // return this == that
  public String toString() { … } // returns print representation
  public int hashCode() { … } // key for accessing object
  // important that a.equals(b) implies a.hashCode()==b.hashCode()
  public void finalize() { … } // called before object garbage
  // collected, default is {}
  public Object clone() { … } // default is shallow bit-copy if class
  // implements Cloneable, throw CloneNotSupportedException
  // otherwise
  ```

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

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.println(...) 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 Programming Environments

Packages

- Classes grouped into packages
- Example: `java.awt.image`
  - avoids namespace clashes
- But no semantics to having a common prefix
  - e.g., between `java.awt` and `java.awt.image`
- Package names are an implicit or explicit part of a class name

Packages (cont.)

- Import makes a class or package name implicit
  - e.g., allows use of `ColorModel` instead of `java.awt.image.ColorModel`
  - to import all classes in a package, use *
    - e.g., `import java.awt.image::*`;
- Implicit at the beginning of every Java file
  - `import java.lang.*`;
- Import never required, just allows use of shorter names

Files – what goes where

- Each public class C must be in a file C.java
- If a class C is part of package P
  - package P; must be the first statement in C.java
  - which must be in a directory P
  - treats . in package name as subdirectories
- Reverse of domain name is reserved package name
  - `edu.umd.cs` is reserved for UMD CS department

Files (cont.)

- `CLASSPATH` gives list of places to look for class files
  - both directories and archive (jar) files
  - don’t need to specify location of system files
  - only need to set it for your own files
    - if they are part of a package
    - if they aren’t in the current directory (where the interpreter is run from)

java.lang

- Wrapper classes
- class `String`
- class `StringBuffer`
Wrapper classes

- To create `Integer`, `Boolean`, `Double`, …
  - that is a subclass of `Object`
  - useful/required for polymorphic methods
    - `HashMap`, `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

class String

- Cannot be changed/updated
- Automatically created for string constants
- + used for concatenation (arguments converted to `String` as needed)
- lots of methods, including:
  - `int length()`, `char charAt(int pos)`
  - `int compareTo(String otherString)`
  - `void getChars(int begin, int end, char[ ] dst, int dstBegin)`
  - `int indexOf(int ch)`  // why doesn’t take a char??
  - `String toUpperCase()`

class StringBuffer

- String contents can be changed
- Constructors
  - `StringBuffer()`
  - `StringBuffer(String s)`
  - `StringBuffer(int initialBufferSize)`
- Lots of methods, including
  - `StringBuffer append(String str)`
  - `StringBuffer insert(int offset, String str)`
  - both can actually take many types as argument, and convert as needed (e.g., `Object`, `int`, `float`, …)

StringBuffer Example

- Used to implement `String` concatenation

```java
String s = "(X, Y) = (" + x + ", " + y + ")";
// is compiled to:
String s = new StringBuffer("(X, Y) = (")
  .append(x).append(", ")
  .append(y).append("")
  .toString();
```