Java Review

Michael Hicks
September 4, 2003

Administrivia

• Project 1 will be posted today, due Sep 17

• You should have received e-mail from the TA
  – And a class account

• Reading: (today) Liskov ch. 1, 2 (Tues) ch. 4
  – Supplemental: Eckel ch. 1, 7, 8 (first half), 9
Selected Java Tidbits

- Code reuse with subtyping and inheritance
- Every object is an **Object**
- Methods can be overloaded and overridden
- Object variables are references
- Exceptions

Java Classes and Code Reuse

- Each object is an instance of a class
  - Even an array is an object
- Classes can be reused in two ways
  - Subtyping
  - Extension (inheritance)
Code Reuse by Subtyping

• U “is a subtype of” T (notated U ≤ T)
  – When requiring an object of type T, an object of type U can be used, assuming U ≤ T
  – In Java, for all types T, T ≤ Object
• Permits reusing classes that manipulate objects
  – E.g. any method which expects an Object can be given an arbitrary T instead

Code Reuse by Inheritance

• U “is a subclass of” T (notated U extends T)
  – When defining the class U, code and methods are inherited from class T “for free”
  – In Java, U extends T implies U ≤ T
• Permits reusing classes to define new objects
  – Can define the behavior of the new object in terms of the old one, e.g. Point and ColorPoint
### Beware! Inheritance ? Subtyping

- $U$ extends $T$ implies $U \leq T$
- $U \leq T$ does not imply $U$ extends $T$
- Why?
  - Subtyping of primitives, e.g. `char \leq int`
  - Subtyping of interfaces
- Try to keep these ideas separate in your head

### Java Interfaces

- **Inheritance**
  - Hierarchical code sharing (“is-a”)
- **Interfaces**
  - For defining subtypes

![Object Inherits from Number](Inherits from)

**Comparable**

- `public int compareTo(Object o)`
- `Implements`
Interfaces

• An interface lists supported (public) methods
  – No constructors or implementations allowed
  – Can have final static variables
• A class can implement (be a subtype of) zero or more interfaces
• Given some interface I, declaring I x = ... means
  – x must refer to an instance of a class that implements I, or else null

Interface example

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

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

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

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

- Interfaces can extend other interfaces
  - Type reuse
  - As with classes, $I_2$ extends $I_1$ implies $I_2 \leq I_1$
- Given two interfaces $I_1$ and $I_2$, where $I_2 \leq I_1$
  - If $C$ implements $I_2$, then $C \leq I_2$ and $C \leq I_1$
- Since a class can implement multiple interfaces, interface extensions are less needed

No Multiple Inheritance

- A class type can be a subtype of many other types (implements)
- But can only inherit 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
Invoking Methods

• Given
  \[ o.m(\text{arg1, arg2, \ldots, argn}); \]

• Question:
  – Which method m will actually get run?

• Answer:
  – It depends on the \textit{declared type} and the \textit{actual type} of o

Declared vs. Actual Types

• The \textit{actual type} of an object is its allocated type
  – \texttt{Integer o = new Integer(1)};

• A declared type is a type at which an object is being viewed
  – \texttt{Object o = new Integer(1)};
  – class Foo { void m(\textbf{Object} o) { return; } }

• Each object always has \textit{one actual type}, but can have \textit{many declared types}
Overriding

• Define a method also defined by a superclass

```java
class Parent {
    int cost;
    void add(int x) {
        cost += x;
    }
}

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

Overriding (cont’d)

• Method with same name and argument types in child class overrides method in parent class
• Arguments and result types must be identical
  – otherwise you are overloading the method (e.g., equals from last time)
• Must raise the same or fewer exceptions
• Can override/hide instance variables
  – both variables will exist, but don’t do it
Dynamic Dispatch

• Let \( B \) be a subclass of \( A \), and suppose we have
  \[ \text{A a = new B();} \quad \text{Declared type A} \]
  \[ \text{Actual type B} \]

• Then
  – \textit{instance methods} invoked on \( a \) will get the methods for
    \textit{actual type B} (in C++, virtual functions)

Why? Allows \textit{Container Reuse}

• Say a class \( C \) manipulates objects of some type \( T \)
  – \( C \) should behave properly for subtypes of \( T \)
  – That is, when \( C \) invokes a \( T \) method \( m \), if the
    actual object has type \( U \leq T \), then the \( U \) version
    of \( m \) should be used
      • The \( m \) defined in the “actual” object

• Java class hierarchy set up with this in mind
Instance vs. *static*

- **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
Static Method Dispatch

- Let B be a subclass of A, and suppose we have
  \[ \text{A a = new B;} \]
  \text{Declared type A}
  \text{Actual type B}

- Then
  - class (static) methods invoked on a will get the methods for the declared type A
    - Invoking class methods via objects strongly discouraged; invoke through the class instead (A.m() instead of a.m())

Simple Method Dispatch Example

```java
public class A {
    String f() { return "A.f() "; }
    static String g() { return "A.g() "; }
}
public 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());
    }
}
```
java B generates:
B.f() A.g() B.f() B.g()
Overloading

- Methods with the same name, but different parameters (count or declared types) are overloaded
- Be careful: you may inadvertently overload a method you meant to override!

```java
class Parent {
    int cost;
    void add (int x) {
        cost += x;
    }
    void add(Object s) throws NumberFormatException {
        cost += Integer.parseInt((String)s);
    }
}
class Child extends Parent {
    void add(String s) throws NumberFormatException {
        if (x > 0) cost += Integer.parseInt(s);
    }
}
Child c = new Child();
c.add((Object)"-1");
System.out.println(c.cost);
```
Java Inheritance Hierarchy

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

```
Object
  ↓
Number  Thread
  ↓
Integer ...
```

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

Objects have methods

- All objects, therefore, inherit them
  - Default implementations may not be the ones you want

```
public boolean equals(Object that)  /* conceptual” equality
public String toString()  /* returns print representation
public int hashCode()  /* key for hash table
public void finalize()  /* called when object garbage-collected
```

- And others …
Subtype Polymorphism

- A data structure **Set** that implements sets of **Objects**
  - can summarily hold **Strings**
  - or images
  - or … anything!
- The trick is getting them back out:
  - When given an **Object**, you have to *downcast* it

Downcasting

- **(Foo) o**
  - If o has declared type **U**, actual type **T ≤ U**
    - Compile-time error if **Foo** is not a subtype of **U**
    - Cast succeeds when **T ≤ Foo**
    - Run-time exception if **Foo ≤ T** and **T ≠ Foo**
  - No run-time effect on success
    - Just treats the result as if it were of type **Foo**
- **o instanceof Foo**
  - Predicate: true if cast (**Foo**)o would succeed
Example

class DumbSet {
    public void insert(Object o) { ..o.equals(x).. }
    public bool member(Object o) { ..o.equals(x).. }
    public Object any() { .. }
}

class MyProgram {
    public static void main(String[] args) {
        DumbSet set = new DumbSet();
        String s1 = "foo";
        String s2 = "bar";
        set.insert(s1);
        set.insert(s2);
        System.out.println(s1+" in set?"+set.member(s1));
        String s = (String)set.any(); // downcast
        System.out.println("got "+s);
    }
}

Objects and references

• All variables of non-primitive type are references
  – Pointers to objects, not the objects themselves
  – Or null
• All objects allocated on the heap with new()
  – No stack allocation
• Objects no longer usable are reclaimed automatically (garbage collection)
  – No free()
References and call-by-value

class MyInt {
  int x;
  public MyInt(int x) {
    this.x = x;
  }
}
class Foo {
  void inc(int x) {
    x = x+1;
  }
  void inc(MyInt o) {
    o.x = o.x+1;
  }
}

MyInt o = new MyInt(5);
int x = 5;
Foo f = new Foo();
f.inc(x);
f.inc(o);

Prints:
System.out.println(x); 5
System.out.println(o); 6

Passing object expressions to methods copies the reference, not the object

Equality

- **Object .equals(Object) method**
  - Structural ("conceptual") equality
- **==** operator (**!=** as well)
  - Operates on references, not objects
  - True if arguments refer to *same runtime object*
  - o == p implies o.equals(p)
Overriding Equals

class Foo {
    bool equals(Foo f) { ... } // wrong!
}

class Foo {
    bool equals(Object o) { ... } // right!
}

The first case creates an overloaded method, while the second overrides the parent (Object) method.

Preconditions

• Functions often have requirements on their inputs

    // Return maximum element in A[i..j]
    int findMax(int[] A, int i, int j) { ... }

    – A is non-empty
    – i and j must be non-negative
    – i and j must be less than A.length
    – i < j (maybe)

• These are called preconditions or requires clauses
Dealing with Errors

- What do you do if a precondition isn’t met?

- What do you do if something unexpected happens?
  - Try to open a file that doesn’t exist
  - Try to write to a full disk

Throwing an Exception

- Signals a programming error
- Create a Exception object, and throw it
  
  ```java
  if (i > 0 && i < a.length)
  return (a[i]);
  else throw new ArrayIndexOutOfBoundsException();
  ```

- Exceptions thrown are declared as part of the return type
  - An overriding method cannot throw more exceptions than its parent’s version
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 (masked by) the method
  – includes exceptions thrown by called methods
  – certain built-in exceptions excluded

Exception Handling

• An exception of type T gets caught by first `catch` with declared type U, where T ≤ U
• `finally` is always executed

```java
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("a[] out of bounds");}
catch (MyOwnException e) {
    System.out.println("Caught my error");}
catch (Exception e) {
    System.out.println("Caught" + e.toString());
    throw e;
}
finally { /* stuff to do whether an exception */
    /* was thrown or a return taken */  }
```
Masking Exceptions

- Handle exception and continue

```java
while ((s = ...) != null) {
    try {
        FileInputStream f =
            new FileInputStream(s);
        ...
    }
    catch (FileNotFoundException e) {
        System.out.println(s + " not found");
    }
}
```

Reflecting Exceptions

- Pass exception up to higher level
  - Automatic support for throwing same exception
  - Sometimes useful to throw different exception

```java
try {
    ... a[0] ...
}
catch (IndexOutOfBoundsException e) {
    throw new EmptyException("Arrays.min");
}
```
Exception Chaining

• Indicate the cause of a thrown exception
  – Specify the exception that caused this one
  – Shows cause chain in stack trace

```java
try {
    ... a[0] ...
}
catch (IndexOutOfBoundsException e) {
    Exception e2=new EmptyException("Arrays.min");
    e2.initCause(e);
    throw e2;
}
```

Exception Hierarchy

![Exception Hierarchy Diagram](image)
Unchecked Exceptions

• Subclasses of `RuntimeException` and `Error` are unchecked
  – Need not be listed in method specifications

• Currently used for things like
  – `NullPointerException`
  – `IndexOutOfBoundsException`
  – `VirtualMachineError`

• Is this a good design?