Java Review

Adam Porter
September 4, 2003

Administrivia

• Meet your TA
• Project 1 will be posted later today, due 9/17
• You should have received e-mail from me
  – And a class account
• Reading: Liskov ch. 1, 2, 4

Selected Java Tidbits

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

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

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

Code Reuse by Inheritance

• U “is a subclass of” T (notated \( U \text{ extends } T \))
  – When defining the class \( U \), variables and methods are inherited from class \( T \) “for free”
  – In Java, U extends T implies \( U \leq T \)
• 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

Java Interfaces

• Inheritance
  – Hierarchical code sharing
• Interfaces
  – For defining subtypes
    • (often for unrelated classes)

String

Inherits from Object

public int compareTo(Object o)

Comparable

Implements

Number
**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

```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 ...;
    }
}
```

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

**Interface Inheritance**

- Interfaces can extend other interfaces
  - Type reuse
  - As with classes, I2 extends I1 implies I2 ≤ I1
- Given two interfaces I1 and I2, where I2 ≤ I1
  - If C implements I2, then C ≤ I2 and C ≤ I1
- Classes can implement multiple interfaces

**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 ...;
    }
}
```

**No Multiple Inheritance**

- A class type can be a subtype of many other types (*implements*)
- But can only inherit implementations from one superclass (*extends*)
Invoking Methods

- Given
  
  \[ o.m(\text{arg}1, \text{arg}2, \ldots, \text{argn}); \]

- Question:
  - Which method \( m \) will actually get run?

- Answer:
  - It depends on the \textit{declared type} and the \textit{actual type} of \( o \)
  - And the method being called

Declared vs. Actual Types

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

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

  \begin{verbatim}
  class Parent {
      int cost;
      void add(int x) {
          cost += x;
      }
  }
  
  class Child extends Parent {
      void add(int x) {
          if (\( x > 0 \)) cost += x;
      }
  }
  \end{verbatim}

Overriding (cont’d)

- Method with same name and argument types in child class overrides method in parent class

- Arguments and result types must be \textit{identical}
  - otherwise you are \textit{overloading} the method
  - Must raise the same or fewer exceptions
  - Why not more?

- Can override/hide instance variables
  - both variables will exist, but don’t do it, it’s confusing
**Dynamic Dispatch**

- Let B be a subclass of A, and suppose we have
  \[ A \ a = \text{new} \ B(); \]  
  *Declared type A*
  
- Then
  - *instance methods* invoked on a will get the methods for
    *actual type B* (in C++, virtual functions)

**Why? Allows *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 ≤ 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
- You can reference class variables and methods either through class name or through object ref
  - Clearer to reference via the class name
Static Method Dispatch

• Let $B$ be a subclass of $A$, and suppose we have $A a = new B();$. $\textbf{Declared type } A$

• Then
  – class (static) methods invoked on $a$ will get the methods for the $\textbf{declared type } A$

• Invoking class methods via objects strongly discouraged; invoke through the class instead ($A.m()$ instead of $a.m()$)

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!

Simple Method Dispatch: Example 1

```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

```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);
 Prints -1
```
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 from methods from Object
  - Default implementations may not be the ones you want

  ```
  public boolean equals (Object that)
  public String toString()
  public int hashCode()
  public void finalize()
  ```

  - 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`
      - Why is it this way?
  - 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

```java
class DumbSet {
    public void insert(Object o) {...o.equals(x)...}
    public bool member(Object o) {...o.equals(x)...}
    public Object any() {...} // return any Object in set
}
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

```java
class MyInt {
    int x;
    public MyInt(int x) {
        this.x = x;
    }
}
class Foo {
    void inc(int x) {
        System.out.println(x);
    }
    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);
System.out.println(x);
System.out.println(o);
```

Equality

- **Object .equals()** 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)
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

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 regardless of whether an exception */
    /* was thrown or a return taken */
}
```

Example Application

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

public class Echo {
    public static void main(String args[]) {
        BufferedReader in = ...;
        try {
            while((s = in.readLine()) != null)
                System.out.println(s);
            } catch(EOFException e) {
                e.printStackTrace();
            } finally {
            } /* stuff to do regardless of whether an exception */
        } /* was thrown or a return taken */
    }
}
```

Exception Hierarchy

```
Throwable
    ↓
  Error
  ↓
Exception
  ↓
RuntimeException
```

CMCS 433, Fall 2003 – Adam Porter
Unchecked Exceptions

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

- Currently used for things like
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `VirtualMachineError`