Lecture 21: Java & Eiffel

CSC 131

Kim Bruce

Java Design Goals

- Portability across platforms
- Reliability *(code run on another computer)*
- Safety *(no viruses!)*
- Dynamic Linking *(change on the fly)*
- Multithreaded execution *(not ad hoc, part of language)*
- Simplicity and Familiarity *(C syntax, alas!)*
- Efficiency *(least important)*

Portability

- Compiled to Java Byte code (JVML) and then run

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

⟹

JVML

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge      44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge      31
16:  iload_1
17:  iload_2
18:  irem
```

```
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic      #84; ...
34:  iload_1
35:  invokevirtual  #85; ...
38:  iinc    1, 1
41:  goto    2
44:  return
```

Java

- Original implementations slow
  - Compiled to JVML and then interpreted
  - Now JIT
  - Garbage collection
- Safety - 3 levels:
  - Strongly typed
  - JVML bytecode also checked before execution
  - Run-time checks for array bounds, etc.
- Other safety features:
  - No pointer arithmetic, unchecked type casts, etc.
  - Super constructor called at beginning of constructor

Exceptions & Subtyping

- All non-Runtime exceptions must be caught or declared in “throws” clauses
  - void method readFiles() throws IOException {...}
- Suppose m throws NewException.
- What are restrictions on throwing exceptions if m overridden in subclass? Masquerade!

Simplify from C++

- Purely OO language (except for primitives)
- All objects accessed through pointers
  - reference semantics
- No multiple inheritance -- trade for interfaces
- No operator overloading
- No manual memory management
- No automatic or unchecked conversions

Interfaces

- Originally introduced to replace multiple inheritance

```
interface Comparable {
    boolean equal(Object other);
    boolean lessThan(Object other);
 }
```

Interfaces

- Allows pure use of subtype polymorphism w/out confusing with implementation reuse.
- `public sort(Comparable[] elts) {...}`
- Slower access to methods as method order in vtable not guaranteed

Encapsulation

- Classes & interfaces can belong to packages:

  package MyPackage;

  public class C ...
- If no explicit package then in “default” package
- public, protected, private, “package” visibility
- Class-based privacy (not object-based):
  - If method has parameter of same type then get access to privates of parameter

Problems w/Packages

- Generally tied to directory structure.
- Anyone can add to package and get privileged access
- All classes/interfaces w/out named package in default package (so all have access to each other!)
- No explicit interface for package
- Abstraction barriers not possible for interfaces. Discourages use of interfaces for classes.

Abstraction barriers not monotonic

```
package A;
public class Fst {
   void m(int k){System.out.println("Fst m: "+k);}
   public void n(){System.out.print("Fst n: "); m(3);}
}

package B;
import A.*;
public class Snd extends Fst{
   public void m(int k){System.out.println("Snd m: "+k);}
   public void p(){System.out.print("Snd p: "); m(5);}
}

package A;
import B.*;
public class Third extends Snd{
   public void m(int k){System.out.println("Third m: "+k);}
}
```

Abstraction barriers not monotonic

```
import A.*;
import B.*;
public class Fourth{
    public static void main(String[] args){
        Fst fst = new Fst();
        fst.n();                              Fst n: Fst m: 3

        Snd snd = new Snd();
        snd.n();                              Fst n: Fst m: 3    // ????
        snd.m(5);                             Snd m: 5

        Third third = new Third();
        third.n();                            Fst n: Third m: 3
        third.m(7);                           Third m: 7
        third.p();                            Snd p: Third m: 5
    }
}
```

The method Snd.m(int) does not override the inherited method from Fst since it is private to a different package

Goals of Java 5

- Ease of Development
  - Increased Expressiveness
  - Increased Safety
- *Scalability and Performance*
- *Monitoring and Manageability*
- *Desktop client*
- Minimize Incompatibility
  - No changes to virtual machine
  - Only one new keyword (enum)

Java 5

- Generics
- Enhanced for loop (w/iterators)
- Auto-boxing and unboxing of primitive types
- Type-safe enumerated types
- Static Import
- Simpler I/O

Generics Finally Added

- Templates done well (unlike C++)
  - Type parameters to classes and methods.
  - Type-checked at compile time.
  - Allows clearer code and earlier detection of errors.
  - Biggest impact on Collection classes.
- *Limitations*
  - Virtual machine has not changed.
  - Translated into old code with casts
  - Casts and instanceof don't work correctly
  - Can't construct arrays involving variable type.

Constrained Genericity

- Introduced by Cardelli & Wegner 1985
- Quickly added to Eiffel
- Need to constrain type parameters

```
class List<T extends GraphicObject> {
   private T head;
   ...
   ... head.show() ...
   ... head.move(dx,dy) ...
}
```

- Guarantees presence of methods from GraphicObject in objects of type T.

Constrained Genericity

- Recall the way we constrained type params in Clu:

```
sorted_bag = cluster [t : type] is create,
insert, ...
  where t has
    lt, equal : proctype (t,t) returns (bool);
```

- How can we model this in Java 5?

Constraining Genericity

```
interface Comparable {
    boolean equal(Comparable other);
    boolean lessThan(Comparable other);
 }

 class BST<T extends Comparable> { ... }

  class OrderedRecord implements Comparable {
     ... // inst vble declarations
     boolean lessThan(Comparable other) {
        ???
     }
}
```

F-Bounded Quantification

- Mitchell *et al* introduced F-bounded quantification

```
interface Comparable<T> {
    boolean equal(T other);
    boolean lessThan(T other);
 }

 class BST<T extends Comparable<T>> { ... }

  class OrderedRecord
                implements Comparable<OrderedRecord> {
     boolean lessThan(OrderedRecord other) {
        if (...)...
     }
}
```

F-Bounded Problems!

- Seems to solve the problem, but sometimes too complex to write easily.

```
public class ComparableAssoc
    <Key extends Comparable<Key>, Value>
implements Comparable<ComparableAssoc<Key,Value>> {
```

- Not preserved by subclasses.
  - Suppose C extends Comparable<C> and D extends C
  - Then D extends Comparable<C> but not Comparable<D>
- See Bruce, “Some Challenging Typing Issues in Object-Oriented Languages” on my web pages under recent papers.