Lecture 23: OO Languages: Java & Eiffel

CSC 131
Spring, 2019
Kim Bruce

Papers

- Growing a Language by Guy Steele
  - Not too big, not too small, but growable
- Design Principles behind Smalltalk

Java Design Goals

- Portability across platforms
- Reliability
- Safety (no viruses!)
- Dynamic Linking
- Multithreaded execution
- Simplicity and Familiarity
- Efficiency

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-Runnable 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
- Allows pure use of subtype polymorphism w/out confusing with implementation reuse.
- Slower access to methods as method order 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

```java
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);} }
p
```

```java
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);} }
p
```

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

```java
import A.*;
import B.*;
public class Fourth{
    public static void main(String[] args){
        Fst fst = new Fst();
        Fst n: Fst m: 3
        fst.n();
        Snd snd = new Snd();
        Snd n: Fst m: 3 // ????
        snd.n();
        Snd m: 5
        snd.m(5);
        Third third = new Third();
        Third n: Third m: 3
        third.n();
        Third m: 7
        third.m(7);
        Snd p: Third m: 5
        third.p();
    }
}
```
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

- Recall the way we constrained type params in Clu:

```java
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

```java
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

```java
text
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) {
        ...
    }
}
```

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

```java
text
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.
```

**Also Polymorphic Methods**

```java
text
interface Visitor<T> {
    T visitNumber(int n);
    T visitSum(T left, T right);
}
abstract class Expr {
    public <T> T accept(Visitor<T> v);
}
class Number extends Expr {
    private int n;
    public Number(int n) { this.n = n; }
    public <T> T accept(Visitor<T> v) {
        return v.visitNumber(this.n); }
}
```

**Java Wild Cards**

- Four ways to specify type parameters:
  - T: fixed type
  - ? extends T: some extension of T,
  - ? super T: some type that T extends,
  - ?: any type

- Examples:
  - C<? extends T>: can be C<U> for any U extending T.
  - C<? super T>: can be C<U> for any U s.t. T extends U.
  - C<?>: can be C<U> for any U.
Example

- In class TreeSet<E>:
  - boolean addAll(Collection<? extends E> c)
  - constructor: TreeSet(Comparator<? super E> c)
  - Comparator <? super E> comparator()
  - where interface Comparator<T> has method
    int compare(T o1, T o2)

In libraries almost all occurrences are of form ? extends E or just ?, and are in parameter position.

What do wildcards mean?

\[ C<? \text{ extends } T > \equiv \exists (t : T). C<t> \]
\[ C<? \text{ super } T > \equiv \exists (t :> T). C<t> \]
\[ C<> \equiv \exists t. C<t> \]

Compare with
\[ C<t \text{ extends } T > \equiv \forall (t : T). C<t> \]

Wildcard Restricts Usage

- If \( ds : \text{List}? \text{ extends } T > \)
  \[ \equiv \exists t \text{ extends } T. \text{List}<t> \]
  then can access elements, but not insert.
- More carefully, if \( \text{List}<T> \) has methods
  \[
  \text{get: } () \rightarrow T, \text{set: } T \rightarrow \text{void}
  \]
  then
  \[ ds.\text{get}() \text{ will return value of type } T, \text{ but} \]
  \[ ds.\text{set}(o) \text{ always illegal, no matter what type of } o. \]
  I.e., \( ds \) is read-only

Covariant occurrences of \( T \) are OK, contravariant are not!

Restrictions Confusing

- ?s are not equal to each other or even itself:
  \[
  \text{public void twiddle(Stack<<?> s) {}
    \text{if } (s.\text{empty()})
    \text{s.}\text{push}(s.\text{pop()});
  }\]
- \text{Illegal}, because type of s.\text{pop()} not recognized
  as same as argument type of s.\text{push(...)}.
- Can’t even write swap!
- Can fix by calling polymorphic method where
  type given a name.
Avoiding Wildcards

- Recall from logic, if B does not contain t then
  \[ \forall t. (A(t) \rightarrow B) = (\exists t. A(t)) \rightarrow B \]
- Thus by “Curry-Howard equivalence”
  \[ \langle T \text{ extends } C \rangle \text{ void } m(\text{List}<T> \text{ aList})\{...\} \]
  \[ \text{is equivalent to} \]
  \[ \text{void } m(\text{List}<? \text{ extends } C> \text{ aList})\{...\} \]
- However, there is no equivalent for return type or types of fields.

Are Wild-Cards Worth It?

- They show up in all of the Collection classes:
  
  ```java
  public ArrayList<Collection<? extends E>> c
  public void addAll(Collection<? extends E> c)
  public void removeAll(Collection<?> c)
  ```
- Can be replaced by similar:
  
  ```java
  public ArrayList<T extends E>(Collection<T> c)
  public <T extends E> void addAll(Collection<T> c)
  public <T> void removeAll(Collection<T> c)
  ```
- Java with wildcards has undecidable & unsound type system (can convert any type into any other type)

Java Verifier

- Many checks involving format, legality of names, correctness of final declarations, etc.
- Most important is bytecode verifier
- Why verify?
  - How was code constructed?
- Class file contains version info, constant pool, info about class & superclasses, info about fields and methods, debugging info.
Bytecode Verifier

- Ensures that at any point in code, no matter how got there:
  - Stack is always same type and contains same types of objects.
  - No register accessed unless known to contain value of appropriate type
  - Methods are called w/appropriate arguments
  - Fields are modified w/values of appropriate type
  - All opcodes have appropriate type args on stack & in registers.

Java Verifier

- Originally had holes, but now has been given formal specification which has been proven correct.
- New version for Java 6 allows speedier verification as type info can be provided in .class file
- Unfortunately browser plug-ins compromised:
  - https://www.makeuseof.com/tag/web-just-became-secure-google-drops-support-java/
Design by Contract

- Treat method calls as contractual obligations
  - Client must ensure that preconditions of the method are met when sending a message.
  - If client meets the preconditions then the routine guarantees that the postconditions will hold on exit.
  - Both parties may also guarantee that certain properties (the class invariant) hold on entrance to methods and again on exit.

Class Definition

class HELLO_WORLD
create
make
feature
make
do
   print("Hello, world!\n")
end
-- other method defs
invariant
   -- class invariant
end

Method Definition

class connect_to_server (server: SOCKET)
   -- Connect to a server or give up after 10 attempts.
   require
      server /= Void and then server.address /= Void
   local
      attempts: INTEGER
   do
      server.connect
   ensure
      connected: server.is_connected
   rescue
      if attempts < 10 then
         attempts := attempts + 1
      retry
done
end