Lecture 34: Java & Eiffel

CSCI 131
Spring, 2011

Kim Bruce

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<? extends T>` ≡ ∃`(t<:T). C<t>`

`C<? super T>` ≡ ∃`(t:>T). C<t>`

`C<?>` ≡ ∃`t. C<t>`

Compare with

`C<t extends T>` ≡ ∀`(t<:T). C<t>`

Wildcard Restricts Usage

- If `ds : List<? extends T>`
  ≡ ∃`t extends T.List<t>`
  then can access elements, but not insert.
- More carefully, if `List<T>` has methods
  `get: () → T, set: T → void`
  then
  `ds.get()` will return value of type `T`, but
  `ds.set(o)` *always illegal*, no matter what type of `o`.
  I.e., ds is *read-only*

Restrictions Confusing

- ?s are not equal to each other or even itself:

```
public void twiddle(Stack<?> s) {
   if (!s.empty())
      s.push(s.pop());
}
```

- Illegal, because type of `s.pop()` not recognized as same as argument type of `s.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
  ∀t.(A(t) → B) ≡ (∃t.A(t))→ B
- Thus by "Curry-Howard equivalence"

```
<T extends C> void m(List<T> aList){...}
```

is equivalent to

```
void m(List<? extends C> 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:

```
public ArrayList( Collection<? extends E> c )
public void addAll( Collection<? extends E> c )
public void removeAll( Collection<?> c )
```

- Can be replaced by similar:

```
public ArrayList<T extends E>( Collection<T> c )
public <T extends E> void addAll( Collection<T> c )
public <T> void removeAll( Collection<T> c )
```

- Provides more information: *Can write swap!*

Eiffel

- Introduced in 1985 by Bertrand Meyer
- Design goals:
  - Promote clear and elegant programming.
  - Support object-oriented design, including "design-by-contract"
- Design-by-contract is most important impact

Features

- Purely object-oriented
- Multiple inheritance
- Automatic memory management
- Assertions integral part of language
- Static typing (*but not type-safe, alas*)

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

```
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
    end
  end
```

Inheritance & Assertions

- What changes can you make in preconditions and postconditions of method when override?
- Need to maintain contract as masquerades.
- Can
  - weaken preconditions
  - strengthen postconditions

Static Typing Issues

- In subclass, can
  - specialize type of instance variables
  - specialize return type of methods
  - specialize parameter type of methods
- First & third lead to errors
- Several proposals made to fix, including whole-program analysis
  - None appear to have been implemented

like Current

```
class LINKABLE [G]

feature
   item: G;
   right: like Current;

   putRight (other: like Current) is
      do
         right := other
      ensure
         chained: right = other
      end;

end -- class LINKABLE
```

Type *like Current* is type of class

```
class BILINKABLE [G] inherit LINKABLE [G]
           redefine
              putRight
           end

feature
   left: like Current;   -- Left neighbor

   putRight (other: like Current) is
         -- Put `other' to right of current cell.
      do
         right := other;
         if (other /= Void) then
            other.simplePutLeft (Current)
         end
       end;

    putLeft (other: like Current) is ...
```

Very Flexible

- Define
  - class LINKEDLIST[NODE -> LINKABLE] ... *like Java extends*
  - Can instantiate with LINKABLE to get singly-linked list or BILINKABLE to get doubly-linked list.
- Can't do in Java or C++!
  - Why?
- Type Unsafe
  - *See this week's homework – implicit change of parameter type*
  - Subclass, but not subtype

Scala's Mixins & Traits

Approximating Multiple Inheritance

- No interfaces in Scala
- Traits are new construct -- more powerful
- Purely abstract trait is like interface
- But traits can contain implementations

Iterator Traits

```
trait AbsIterator[T] {
   def hasNext: boolean
   def next: T
}
```

Can also contain behavior:

```
trait RichIterator[T] extends
                        AbsIterator[T] {
   def foreach(f: T => unit): unit =
      while (hasNext) f(next)
}
```

Traits can be used as Classes

- But traits have no constructors
- Can also be used as "mixins"
- Class can extend at most one class or trait
  - but can mixin many traits

Trait as Superclass

```
class StringIterator(s: String) extends AbsIterator[char] {
  private var i = 0
  def hasNext = i < s.length
  def next = { val x = s charAt i; i = i + 1; x }
}
```

Traits applied in Order

```
object Test {
  def main(args: Array[String]): unit = {
    class Iter extends StringIterator(args(0))
                with RichIterator[char]
    val iter = new Iter
    iter foreach System.out.println
  }
}
```

- Solves multiple inheritance problems because linearize order
  - last definition wins
  - super refers to previous trait/superclass
  - Can override in body or in any trait

Approximating Multiple Inheritance

- No interfaces in Scala
- Traits are new construct -- more powerful
- Purely abstract trait is like interface
- But traits can contain implementations

Iterator Traits

```
trait AbsIterator[T] {
   def hasNext: boolean
   def next: T
}
```

Can also contain behavior:

```
trait RichIterator[T] extends
                        AbsIterator[T] {
   def foreach(f: T => unit): unit =
      while (hasNext) f(next)
}
```

Traits can be used as Classes

- But traits have no constructors
- Can also be used as "mixins"
- Class can extend at most one class or trait
  - but can mixin many traits

Trait as Superclass

```
class StringIterator(s: String) extends AbsIterator[char] {
  private var i = 0
  def hasNext = i < s.length
  def next = { val x = s charAt i; i = i + 1; x }
}
```

Traits applied in Order

```
object Test {
  def main(args: Array[String]): unit = {
    class Iter extends StringIterator(args(0))
                with RichIterator[char]
    val iter = new Iter
    iter foreach System.out.println
  }
}
```

- Solves multiple inheritance problems because linearize order
  - last definition wins
  - super refers to previous trait/superclass
  - Can override in body or in any trait

Another Example

```
trait SyncIterator[T] extends AbsIterator[T] {
   abstract override def hasNext: boolean =
                        synchronized(super.hasNext)
   abstract override def next: T =
                        synchronized(super.next)
}
```

abstract because not implemented in AbsIterator

```
StringIterator(someString) with RichIterator[char]
                           with SyncIterator[char]
```

- Order not matter here, but could in other examples
- Super calls go to StringIterator, not AbsIterator