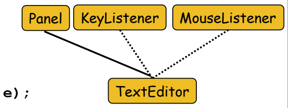
Java Part II

CSCI 334
Stephen Freund

Interfaces




```
interface KeyListener {
  void keyPressed(KeyEvent e);
  void keyReleased(KeyEvent e);
  void keyTyped(KeyEvent e);
}

interface MouseListener {
  void buttonClicked(MouseEvent e);
}

class TextEditor extends Panel
                 implements KeyListener,MouseListener {
  void keyPressed(KeyEvent e)   { /* code */ }
  void keyReleased(KeyEvent e)  { /* code */ }
  void keyTyped(KeyEvent e)     { /* code */ }
  void buttonClicked(MouseEvent e) { /* code */ }
}
```

Scala Traits

- Completely Abstract

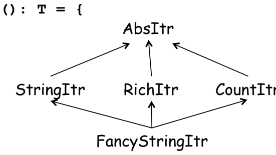
```
trait AbsIterator[T] {
  def hasNext(): boolean;
  def next(): T;
}
```

- Partially Implemented

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

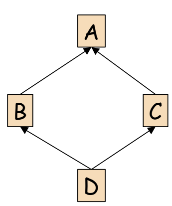
Scala Traits

```
trait CountingIterator[T] extends AbsIterator[T] {
  var count = 0;
  abstract override def next(): T = {
    count = count + 1;
    super.next();
  }
  def count() = count;
}
```




```
class FancyStringIterator(s: String)
      extends StringIterator(s)
      with RichIterator[Char]
      with CountingIterator[Char] { ... }
```

Name Resolution via Linearization

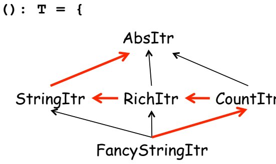



```
trait A { }
trait B extends A { }
trait C extends A { }
class D extends B with C { }
```

Right-first depth-first search: [D,C,A,B,A]
Eliminate all but last occurrence: [D,C,B,A]

Scala Traits

```
trait CountingIterator[T] extends AbsIterator[T] {
  var count = 0;
  abstract override def next(): T = {
    count = count + 1;
    super.next();
  }
  def count() = count;
}
```




```
class FancyStringIterator(s: String)
      extends StringIterator(s)
      with RichIterator[Char]
      with CountingIterator[Char] { ... }
```

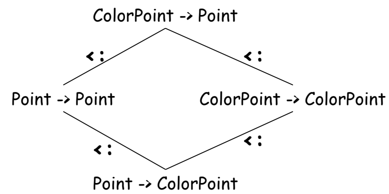
http://en.wikipedia.org/wiki/Mixin

Mixins at Allocation Time

```
val iter = new StringIterator("moo")
       with RichIterator[Char]
       with CountingIterator[Char] { ... }

iter.foreach(println(_));
println(iter.count);
```

Function Subtyping Revisited



- Return Types are *Covariant*
- Argument Types are *Contravariant*

Java Array Covariant Subtyping Rule

```
class Point { ... }
class ColorPoint extends Point { ... }

Point pts[] = new Point [100];


pts[0] = new Point (10,10);
pts[1] = new Point (20,20);
```

Java Array Covariant Subtyping Rule

```
class Point { ... }
class ColorPoint extends Point { ... }

ColorPoint cpts = new ColorPoint[100];
Point pts[] = cpts;

pts[0] = new Point (10,10);
pts[1] = new Point (20,20);

 ...
 cpts[0].setColor(RED);
```

Why Did They Add It?

```
static void arrayCopy(Point src[], Point dst[]) {
  for (int i = 0; i < src.length; i++)
    dst[i] = src[i];
}


Point p[];
Point q[];
arrayCopy(p,q);

String s[];
String t[];
arrayCopy(s,t);
```

General arrayCopy Operation

```
static void arrayCopy(Object src[], Object dst[]) {
  for (int i = 0; i < src.length; i++)
    dst[i] = src[i];
}


Point p[];
Point q[];
arrayCopy(p,q);

String s[];
String t[];
arrayCopy(s,t);
```

From Bill Joy (Sun Cofounder)

Date: Fri, 09 Oct 1998 09:41:05 -0600
From: bill joy
Subject: ...[discussion about java genericity]

actually, java array covariance was done for less noble reasons ...: it made some generic "bcopy" (memory copy) and like operations much easier to write...

I proposed to take this out in 95, but it was too late (...).

i think it is unfortunate that it wasn't taken out...

it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.

wnj

Variance in Scala

```
Array[ColorPoint] <: Array[Point] ?

List[ColorPoint] <: List[Point]?

PartialFunction[Point,ColorPoint]
              <:
  PartialFunction[Point,Point] ?
```

Variance Annotations in Scala

- Class defined with covariant parameter T:

```
class List[+T] { ... }
```

- So `List[ColorPoint] <: List[Point]`
- Which of these is type-safe?

```
class List[+T] {
  def add(t : T) = ...
}
```

```
class List[+T] {
  def get() : T = ...
}
```

- Function Types:

```
class PartialFunction[-A,+R] { ... }
```

Variance Annotations in Scala

- Function Types:

```
class PartialFunction[-A,+R] { ... }
```

- Immutable Maps:

```
class Map[Key,+Value] extends PartialFunction[Key,Value]

Map[String,ColorPoint] <: Map[String,Point]
```

- Mutable Maps:

```
class Map[Key,Value] extends PartialFunction[Key,Value]

Map[String,ColorPoint] <: Map[String,Point]
```

Java 1.5

The new language features all have one thing in common: they take some common idiom and provide linguistic support for it. In other words, they shift the responsibility for writing the boilerplate code from the programmer to the compiler.

- Joshua Bloch, Sun Microsystems

Java 1.5

I've had the good fortune to play with early prototypes of the new language features, and I find them a joy to use.

- Joshua Bloch, Sun Microsystems

New Features

- Return-type Covariance
- Generic Classes
- Enhanced for Loop
- Autoboxing/Unboxing
- Typesafe Enums
- Varargs
- Static Imports
- Metadata (Annotations)

For Loops

```
void print(Collection<T> c) {
  Iterator<T> i = c.iterator();
  while (i.hasNext()) {
    System.out.println(i.next());
  }
}
```

```
void print(Collection<T> c) {
   for (T t: c) {
     System.out.println(t);
   }
}
```

For Loops (II)

```
// sum elems in array a
int sum(int a[]) {
  int result = 0;
  for (int i: a) {
    result += i;
  }
  return result;
}
```

Autoboxing/Unboxing

- **Java 1.4:**

```
Vector v = new Vector();
v.add(new Integer(3));
System.out.println(
    ((Integer)v.get(0)).intValue() + 4);
```

Autoboxing/Unboxing

- **Java 1.5:**

```
Vector<Integer> v = new Vector<Integer>();
v.add(new Integer(3));
System.out.println(
    (~~(Integer)~~v.get(0)).intValue() + 4);
```

Autoboxing/Unboxing

- **Java 1.5:**

```
Vector<Integer> v = new Vector<Integer>();
v.add(3);
System.out.println(v.get(0) + 4);
```

- special case???

```
v = new Vector<Integer>();
v.add(null);
System.out.println(v.get(0) + 4);
```

Java 1.0 (Subtype Poly.) vs Java 1.5 (Parametric Poly.)

```
class Stack {
  void push(Object o) {...}
  Object pop() {...}
  ...
}

String s = "Hello";
Stack st = new Stack();


st.push(s);

String t = (String)st.pop();
```

```
class Stack<T> {
  void push(T o) {...}
  T pop() {...}
  ...
}

String s = "Hello";
Stack<String> st =
  new Stack<String>();

st.push(s);

String t = st.pop();
```

Type Checking

```
interface Printable {
  void print();
}

class PrintableStack<T> implements Printable {
  void push(T o) {...}
  T pop() {...}

  void print() {
    ...
    for (T t : elems) {
      t.print();
    }
  }
}
```

```
PrintableStack<X> st = ..
st.push(x);
st.print();
```

Type Checking & Bounded Polymorphism

```
interface Printable {
  void print();
}

class PrintableStack<T implements Printable>
             implements Printable {
  void push(T o) {...}
  T pop() {...}

  void print() {
    ...
    for (T t : elems) {
      t.print();
    }
  }
```

```
PrintableStack<X> st = ..
st.push(x);
st.print();
```

Compilation

```
class Stack<T implements
              Printable> {
  void push(T o) {...}
  T pop() {...}

  void print() {
    ...
    for (T t : elems) {
      t.print();
    }
  }
}
```

```
class Stack {
  void push(Printable o) {..}
  Printable pop() {...}

  void print() {
    ...
    for (Printable t : elems)
      t.print();
    }
  }
}
```

F-Bounded Polymorphism

```
interface Comparable {
  int compareTo(Object other);
}

class Point implements Comparable {
  int compareTo(Object other) {
     Point otherAsPoint = (Point)other;
     return x == otherAsPoint.x && y == otherAsPoint.y;
  }
}

class PriorityQueue<T implements Comparable> {
  void add(T t) {
    T o = ...;
    ... t.compareTo(o) ...
```

F-Bounded Polymorphism

```
interface Comparable<T> {
  int compareTo(T other);
}

class Point implements Comparable<Point> {
  int compareTo(Point other) {
    return x == other.x && y == other.y;
  }
}

class PriorityQueue<T implements Comparable<T>> {
  void add(T t) {
    T o = ...;
    ... t.compareTo(o) ...
  }
```

In Scala...

```
trait Comparable[T] {
  def compareTo(other : T): Boolean;
}

class Point extends Comparable[Point] {
  def compareTo(other : Point) = {
    x == other.x && y == other.y;
  }
}

class PriorityQueue[T <: Comparable[T]] {
  def add(t : T) = {
    val o : T = ...;
    ... t.compareTo(o) ...
  }
```

Wildcards

```
void printElements(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}

Collection<String> cs;
Collection<Integer> ci;
printElements(cs);
printElements(ci);
```

Wildcards

```
void printElements(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}

Collection<String> cs;
Collection<Integer> ci;
printElements(cs);
printElements(ci);
```

Wildcards

```
void movePoints(Collection<?> c) {
  for (Point p : c) {
    p.move(10,10);
  }
}

Collection<Point> pts;
Collection<ColorPoint> cpts;
printElements(pts);
printElements(cpts);
```

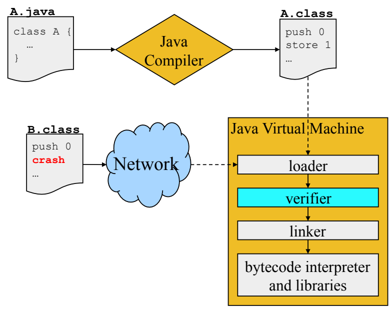
Wildcards

```
void movePoints(Collection<? extends Point> c) {
  for (Point p : c) {
    p.move(10,10);
  }
}

Collection<Point> pts;
Collection<ColorPoint> cpts;
printElements(pts);
printElements(cpts);
```

[Weak form of existential types.
Will explore in Scala on HW...]

JVM Architecture



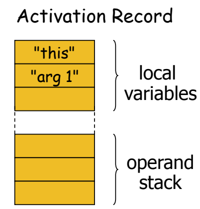
Java Bytecodes

- Java:

```
class A {
   int i;
   void f(int val) { i = val+1; }
}
```

- Bytecode:

```
Method void f(int)
  0: aload 0        // push "this"
  1: iload_1        // push arg
  2: iconst_1       // push 1
  3: iadd
  4: putfield "int A.i"
  5: return
```



Java Bytecodes

- Java:

```
Point p;
...
p.getX();
...
```

- Bytecode:

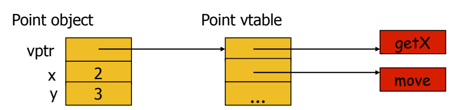
```
4: aload_1      // push p onto stack
5: invoke_virtual "int Point.getX()"
```

Java Run-Time Representation

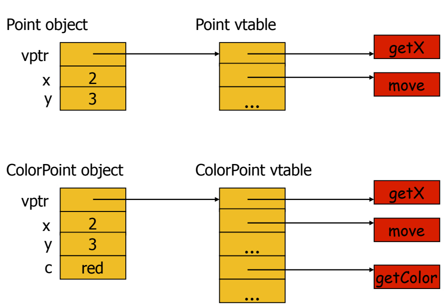
Map for symbolic names to offsets

Constant Pool for Point

field	"int x"	4
field	"int y"	8
method	"int getX()"	0
method	"move(int,int)"	1
…	"…"	…



Java Run-Time Representation



Method Invocation – Symbolic Linking

```
Point p;
...
p.getX()
```

```
...
aload 1    // push p
invoke_virtual "int Point.getX()"
```

```
...
aload_1
invoke_virtual_quick 0
```

Constant Pool for Point

field	"int x"	4
field	"int y"	8
method	"int getX()"	0
method	"move(int,int)"	1
…	"…"	…

1. Look up string description in constant pool
2. Find offset into vtable
3. Change instruction to "fast form"

Field Access

```
class Point {
  int getX() {
    return this.x;
  }
}
```

```
int getX() {
  aload 0
  getfield "int Point.x"
  ireturn
}
```

```
int getX() {
  aload 0
  getfield_quick 4
  ireturn
}
```

Constant Pool for Point

field	"int x"	4
field	"int y"	8
method	"int getX()"	0
method	"move(int,int)"	1
...	"..."	...

1. Look up string description in constant pool
2. Find offset into object record
3. Change instruction to "fast form"

Interface Method Invocation

```
MouseListener m;
m = ...;
m.buttonClicked(e)
```

```
aload 1  /* push m */
aload 2  /* push e */
invoke_interface
  "void MouseListener.
      buttonClicked(MouseEvent)"
```

Constant Pool for TextEditor

field	...	...
field	...	...
method	...	...
method	"buttonClick..."	3
...	"..."	...

Search for method in constant pool

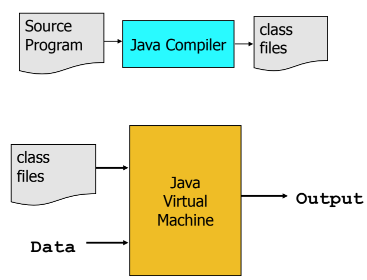
Summary of Lookup Operations

- invoke_virtual, get_field
  - first time: must resolve symbolic name to offset
  - after that: constant offsets used
- invoke_interface
  - must search for method entry every time
  - cache most recent (object, vtable-offset) pair:

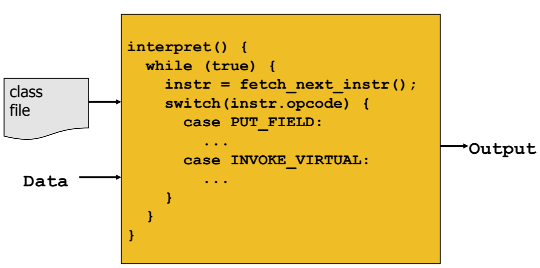
```
for (int i = 0; i < 100; i++) {
  obj.interfaceMethod();
}
```

[Polymorphic Inline Caching, Holzle, Chambers, Ungar, *ECOOP '91*]

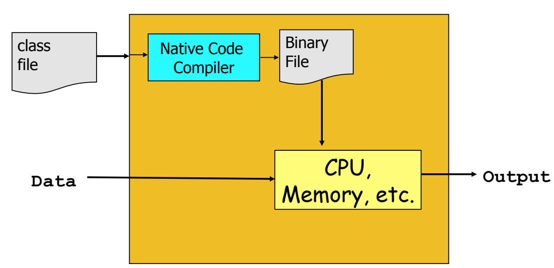
Java Compiler and Interpreter



Standard Interpreter



Interpreter with JIT Compiler

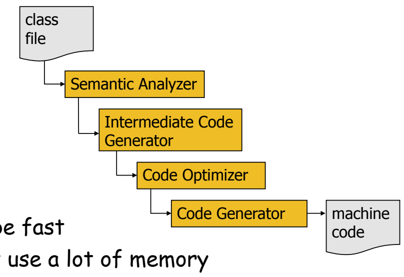


Example Translation

```
void f() {
  this.x = this.moo() + 1;
}
```

```
aload 0                            mov (r0), r1
invoke_virtual "Method moo"        call 4(r1)
aload 1                            add 1, (r30)
iadd                               mov (r30), 8(r0)
aload 0
putfield "Field int x"
```

JIT



- Must be fast
- Cannot use a lot of memory
- Semantic analysis must consider dynamic loading

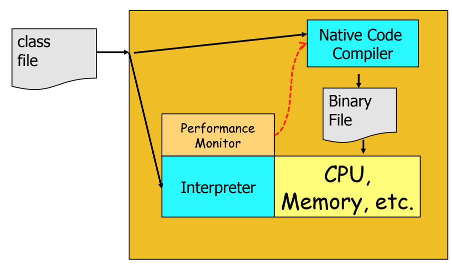
Example

```
class Cow {                         class Cow {
  int x = 2;                          int x = 2;
  public void moo() { x++; }          public void moo() { x++;
}                                   }

Cow c = ...;                        Cow c = ...;
...                                 ...
c.moo();                            c.x++;
```

- If no subclasses of `Cow` are loaded:
  - No need to perform virtual method lookup.
  - (Can even inline method body...)
- Must undo optimization if a subclass of `Cow` is loaded after `Cow` is JIT-compiled

Interpreter with JIT Compiler



HotSpot, SpiderMonkey, ...

JIT Example

```
 > java -XX:+PrintCompilation RSG 20 < Haiku.g
  1       java.lang.String::hashCode (60 bytes)
  2       java.lang.String::charAt (33 bytes)
...
  9       sun.security.provider.SHA::implCompress (491 bytes)
Grammar is: ...
 rising far away/ Buddhist over clouds Kyushu/ purple in day tree

...

 rushing purple cow/ red yakitori falling/ drifting far away

  19      sky java.lang.String::equals (88 bytes)
...

 rushing mountains/ red city to hell blue clouds/ faces sky falling

...
```