• Term paper
  • due Nov 18

• Exam
  • Nov 4
Binding vs. assignment

Binding:
• names refer to values
• bindings do not change
• OCaml:  let x = 3
• ML: val y = 4
• Scheme: (let (x 5) (f x))

Assignment:
• names refer to locations
• contents of a location may change
• Scala: var y = 4; ... y = 5;
• Java, C, C++, C#: int w = 5; ... w = 6;
Final variables

Are final fields in Java bound or assigned?

Are final local variables in Java bound or assigned?
class C {
    C() { init(); }
    void init() { println("I am C"); }
}

class D extends C {
    final int f = 1;
    D() { super(); }
    void init() { println("D.f is " + f); }
}

new D() // prints what?
Java constructors + final fields

class C {
    C() { init(); }
    void init() { println("I am C"); }
}

class D extends C {
    final int f = 1;
    D() { super(); }
    void init() { println("D.f is "+ f); }
}

new D()    // prints 0
Final variables

Are final fields in Java bound or assigned?
• assigned

Are final local variables in Java bound or assigned?
• Can think of it either way, but less confusing to think of as assigned.

    final int x;
    if (c)
        x = 1;
    else
        x = 0;
Bound and free variables

- Variables are either *bound* or *free* in a given expression.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x is free</td>
</tr>
<tr>
<td>fun x -&gt; x</td>
<td>x is bound</td>
</tr>
<tr>
<td>fun x -&gt; x + y</td>
<td>x is bound, y is free</td>
</tr>
</tbody>
</table>

- A function can *capture* a free variable in its scope.

  - fun x -> x + y captures y, but not x
  - fun y -> fun x -> x + y does not capture y or x
Scoping

Static vs. dynamic scope

To what environment are free variables in a function body bound?

- environment at lambda creation
- environment at lambda invocation
Static scope

Free variables bound at lambda creation

def add1() = {
    val y = 1
    (x: Int) => x + y  // captures y
}

def g() = {
    val add1 = add1()
    val y = 2
    add1(3)
}

g() --> 3+1 --> 4
Static scope

Free variables bound at lambda creation

```scala
def add1() = {
    val y = 1
    (x: Int) => x + y  // captures y
}

def g() = {
    val add1 = add1()
    val y = 2
    add1(3)
    add1(3)
}

```
g() --> 3+1 --> 4
Dynamic scope

Free variables bound at lambda invocation

def add1() = {
  val y = 1
  (x: Int) => x + y  // captures y
}

def g() = {
  val add1 = add1()
  val y = 2
  add1(3)
}

g() --> 3+2 --> 5
Dynamic scope

Free variables bound at lambda invocation

def add1() = {
    val y = 1
    (x: Int) => x + y  // captures y
}

def g() = {
    val add1 = add1()
    val y = 2
    add1(3)
}

g() --> 3+2 --> 5
Static vs. dynamic scope

Static scoping
- Free variables captured at definition

Dynamic scoping
- Free variables captured at use

Which is better?
Advantages of dynamic scope

Can redefine library functions used by other library functions

Hypothetical:

```c
// redefine malloc
void *malloc(size_t n) { ... }
...
p = makeBigDataStructure();
    // call a function that uses malloc
```
Disadvantages of dynamic scope

Hard to reason about locally
Function body depends on environment of caller

Code is *implicitly* parameterized on every name it uses
Not clear to the user what code depends on
Breaks encapsulation
• behavior changes depending on where function is used
Languages with dynamic scope

Perl – using `local` rather than `my`
JavaScript – `with`
C – macros
LaTeX
LISP
JavaScript with

```javascript
x.g = 0;

with (x) {
    print(g); // prints x.g
}
```

**with** adds fields of x into scope
function f() {
  var o = new Object();
  o.g = 1;
  return o;
}

var x = f();
var g = 999;

with (x) {
  print(g); // g refers to x.g
}
function f() {
    var o = new Object();
    o.h = 1;
    return o;
}

var x = f();
var g = 999;

with (x) {
    print(g);  // g refers to local var g
}
C macros

#define P printf("%d\n", x)

{
    int x = 0;
    P; // prints 0
}

{
    int x = 1;
    P; // prints 1
}
Perl

$x = 99;

sub f {
    print "$x\n";
}

sub g {
    local $x = 0;
    &f;          # prints 0
}

sub h {
    local $x = 1;
    &f;          # prints 1
}

sub i {
    my $x = 2;
    &f;          # prints 99
}
Languages with static scope

Most of ‘em:
• C, C++
• Java, C#, Scala
• SML, OCaml, F#
• Haskell
• Scheme

This is because static scoping is “saner”.
Scoping in Java

Java is statically scoped

Various constructs that complicate scoping rules
• inheritance
• access modifiers
• imports
• implicit targets for field and method access
• different name spaces for variables, methods, types
Fields

Field names are in scope throughout the class body.

class C {
    int x;
    int m() {
        return x; // refers to field x - implicitly this.x
    }
}

Locals

Local variables are in scope for the remainder of their enclosing block.

```java
{  
    int x = 0; // a different x than ...
    f(x);
}

{  
    int x = 1; // ... this x
    g(x);
}
```
Locals

Local variables are not allowed to shadow other locals.

```java
{
    int x = 0;
    {
        int x = 1;  // error!
    }
}
```

Note: this is allowed in Scala.
Local variables can shadow fields.

class C {
    int x;
    int m() {
        int x = 0;
        return x;  // refers to local variable x.
        // local x shadows the field x
    }
}
Inheritance

Members of a superclass are in scope within the body of a class

class C {
    int x;
    int m() {
        return x; // C.x
    }
}

class D extends C {
    int m() {
        return x; // C.x
    }
}
Inheritance

Subclasses can **hide** superclass fields.

class C {
    int x;
    int m() {
        return x;  // C.x
    }
}

class D extends C {
    int x;
    int m() {
        return x;  // D.x
    }
}
Nested classes

Members of enclosing class are visible to nested classes.

class C {
    int x;
    class D {
        int m() {
            return x; // implicitly C.this.x
        }
    }
}
Nested classes + inheritance

class B {
    int x;
}

class C {
    int x;
    class D extends B {
        int m() {
            return x; // enclosing C.this.x or inherited x?
        }
    }
}

Nested classes + inheritance

```java
class B {
    int x;
}

class C {
    int x;
    class D extends B {
        int m() {
            return x; // inherited x
        }
    }
}

Inheritance beats lexical nesting
```
Local classes in Java

interface I { int n(); }

I m() {
    final int x = 3;
    final int y = 4;
    class C implements I {
        int n() { return x * y; }
    }
    return new C();
}

x and y are captured by the local class C.

m().n() --> 12

anonymous classes also capture their environment

local and anonymous classes are closures
Implementation of local classes

```java
interface I {
    int n();
}

class C$1 implements I {
    int x$, y$;
    C$1(int x, int y) {
        x$ = x;
        y$ = y;
    }
    int n() {
        return this.x$ * this.y$;
    }
}

I m() {
    final int x = 3;
    final int y = 4;
    return new C$1(x, y);
}
```

Translate local classes into top-level classes by copying in the environment.

Called *lambda lifting*. 
In Java, can import classes from other packages

```java
import p.C; // imports class C in package p
    // the name C in this file refers to p.C
import p.*; // lazily imports members of p
    // if a name X is unbound, will try to load from p
    // if p.X exists, the name X will refer to p.X

Can also import member classes
    import C.D;

and static fields
    import static C.f;
```
Imports vs. inheritance

Inheritance beats import

File 1:

```java
class B {
    int x;
    class A {
    }
}
```

File 2:

```java
import p.A;
import static q.E.x;

class C extends B {
    Object m() {
        return new A(x);
    }
}
```
Binding time

*Binding*: association of values with names

Binding of names before program is run
- *static binding*
- *early binding*

Binding of names while program is running
- *dynamic binding*
- *late binding*
Methods

Late binding:
• Virtual method call
  • x.m()
  • associates name (m) with a value (the method’s body)

Early binding
• Static method call
  • T.m()
  • compiler knows which method body is associated with m
Dynamic libraries

Another form of late binding

Load libraries at run time rather than compile time

• native shared libraries, DLLs
• Java class loading
• Java native interface (JNI), JNA
• C# assemblies
Compilers, linkers, loaders

**Compiler:** takes source code, generates object code
- `cc foo.c --> foo.o`
- object code contains *symbolic* references to undefined symbols

**Linker:** takes several object files, resolves references, produces executable
- `ld foo.o, bar.o --> foobar.exe`
  or library
- `ld foo.o bar.o --> foobar.dylib (.dll, ...)`
linker may produce file with references to resolve at run time

**Loader:** part of the OS, starts an executable, loads dynamic libraries
- `./foobar.exe --> “hello, world”`
Dynamic loaded libraries

some libraries loaded at load time—before the process executes
• e.g., libc.so (the standard C library)

some libraries loaded on demand
• programmer explicitly writes code to load in a library
  • Unix: dlopen
• used to implement plugins
  • e.g., the Flash plugin in your web browser
Java class loading

Classes are loaded on demand when first referenced
• can get load-time errors if class not found or if modified in an incompatible way

class A {
    B x;
}
class B {
    static int f;
}

A a = new A(); // loads A, but not B
int x = B.f;    // loads B
Can also explicitly load a class:

```java
Class k = Object.class.getClassLoader().loadClass("C");
C c = (C) k.newInstance();
```
JNI

Java Native Interface

Can use to have Java code invoke C code and vice versa.

- Declare Java method native.
- Run `javah` to generate stub C code.
- Edit stub to implement methods.
- Compile with C compiler into a shared library.
- Be sure to call `System.loadLibrary("mylib")` before first call to the native method.
JNA

Java Native Access

Can use to access native libraries

**Much** easier to use than JNI (but slower, less powerful)
- don’t have to write any C code, can link to existing libraries
  - simplifies build process–use Java compiler
- JNA automatically translates Java types to C types and back
  - e.g., String <--> char *
  - including pointers, structures, enums
JNA example

class Math {
    static { Native.register("math"); }

    public static native float cos(float f);
    public static native float sin(float f);

    public static void main(String[] args) {
        System.out.println(sin(0)); // 0
        System.out.println(cos(0)); // 1
        System.out.println(sin(3.14/4)); // sqrt(2)/2, about
    }
}
Questions?