Programming Paradigms Lecture 5: Names, Scopes, and Bindings (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Wake-up Exercise

What does the following Java code print?

```
Integer foo; Integer bar;

foo = 1000; bar = 1000;
System.out.println(foo <= bar);
System.out.println(foo == bar);

foo = 42; bar = 42;
System.out.println(foo <= bar);
System.out.println(foo == bar);
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Java code print?

```
Integer foo; Integer bar;

foo = 1000; bar = 1000;
System.out.println(foo <= bar);
System.out.println(foo == bar);

foo = 42; bar = 42;
System.out.println(foo <= bar);
System.out.println(foo == bar);
```

Result: true, false, true, true

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Java code print?

```
Integer foo; Integer bar;

foo = 1000; bar = 1000;
System.out.println(foo <= bar);
System.out.println(foo == bar);

foo = 42; bar = 42;
System.out.println(foo <= bar);
System.out.println(foo == bar);
```

Integer objects that wrap a primitive value

Result: true, false, true, true

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Java code print?

```
Integer foo; Integer bar;

foo = 1000; bar = 1000;
System.out.println(foo <= bar);
System.out.println(foo == bar);

foo = 42; bar = 42;
System.out.println(foo <= bar);
System.out.println(foo == bar);
```

Result: true, false, true, true

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Java code print?

```
Integer foo; Integer bar;

foo = 1000; bar = 1000;
System.out.println(foo <= bar);
System.out.println(foo == bar);

foo = 42; bar = 42;
System.out.println(foo <= bar);
System.out.println(foo == bar);
```

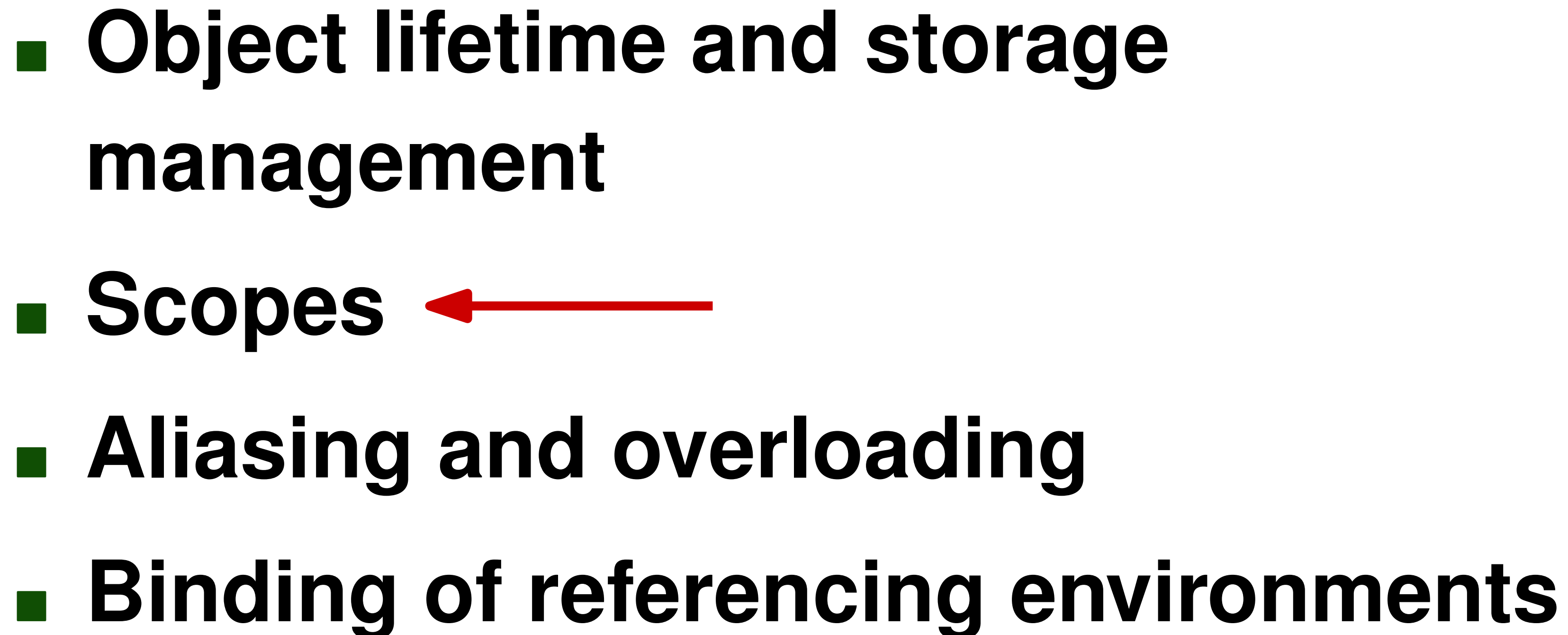
Compares the objects:

- **Different objects, hence false**
- **But: Java caches wrapped integers between -128 and 128, hence true**

Result: true, false, true, true

https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview



- Object lifetime and storage management
- Scopes
- Aliasing and overloading
- Binding of referencing environments

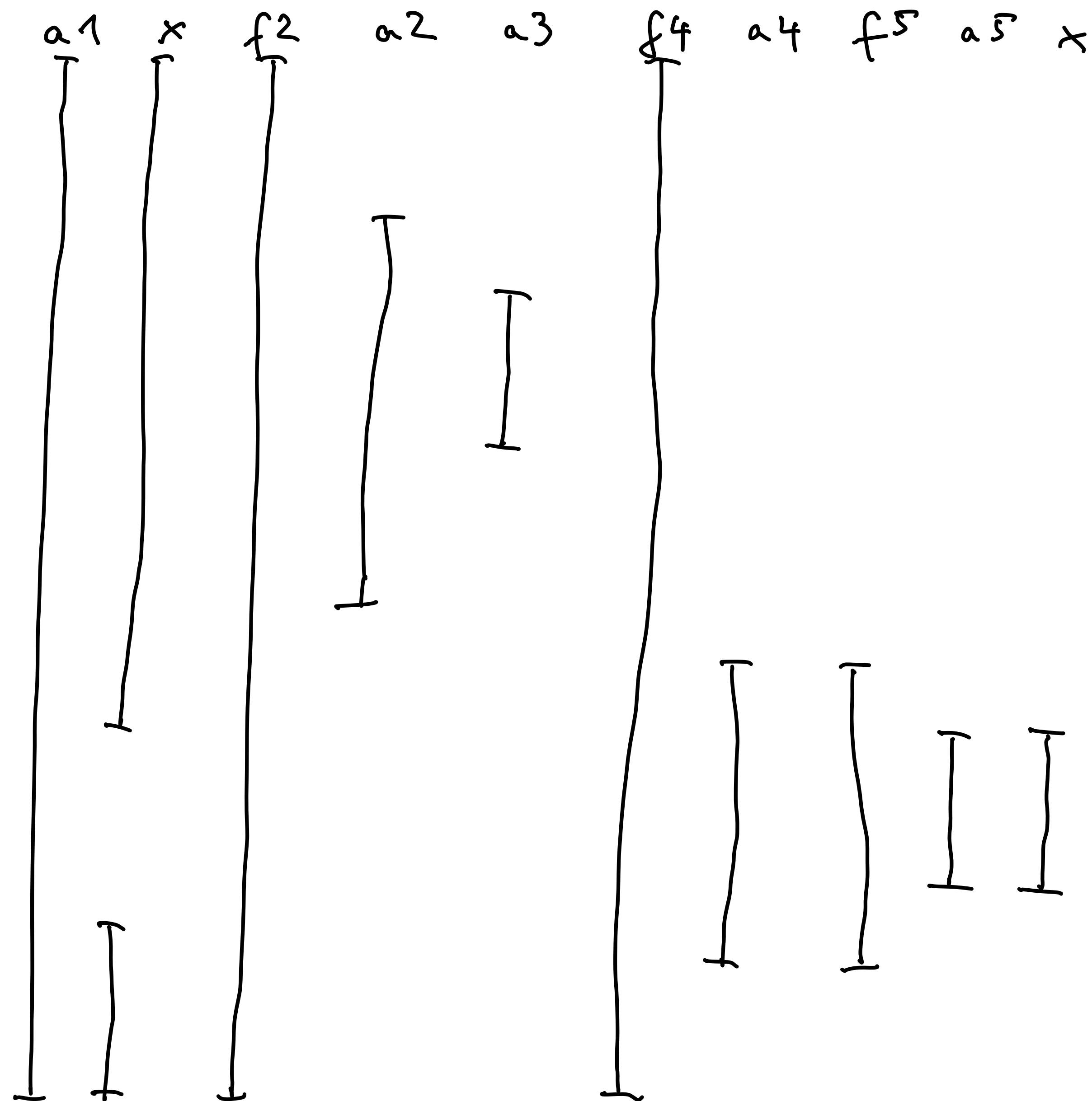
Scoping Rules

- **Scoping rules: Define which bindings are active**
  - I.e., what's the meaning of a name at a given program point?
- **Each PL defines its scoping rules**
  - E.g., Basic has only one scope
  - Most PLs have nested scopes for subroutines

Nested Scopes

- **Common for nested subroutines**
- **Each subroutine has its own scope**
- **Closest nested scope rule**
  - Name is known in scope where it is declared and all internally nested scopes
  - Inner scopes can hide names from outer scopes

```
fun f1 (a1) {
    var x
    fun f2 (a2) {
        fun f3 (a3) {
            :
        }
        :
    }
    fun f4 (a4) {
        fun f5 (a5) {
            var x
            :
        }
        :
    }
}
```



Static vs. Dynamic Scoping

Static scoping

- Binding of a name can be derived from program text
- Most common in today's PLs

Dynamic scoping

- Binding of a name depends on control flow
  - I.e., not known statically (in general)

Example

Pseudo code:

```
global x = 1
fun a() {
    local x = 3
    b()
}
fun b() {
    y = x
}
a()
```

Example

Pseudo code:

```
global x = 1
fun a() {
    local x = 3
    b()
}
fun b() {
    y = x
}
a()
```

Static scoping:

`y` gets value 1 because `b` doesn't have a local variable called `x` and the surrounding static scope provides the global variable `x`

Example

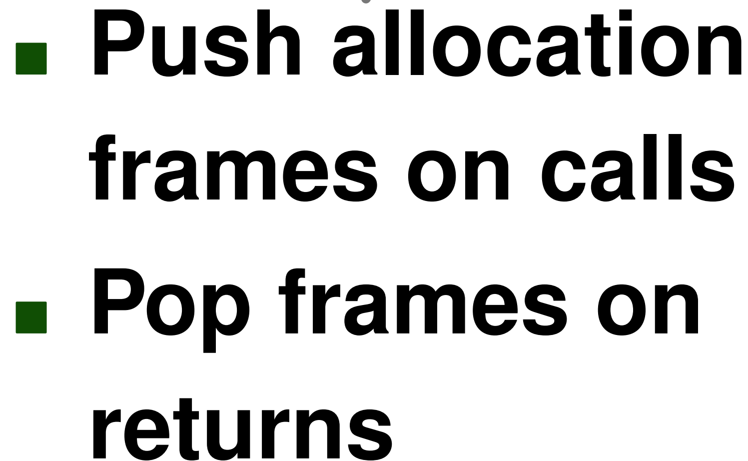
Pseudo code:

```
global x = 1
fun a() {
    local x = 3
    b()
}
fun b() {
    y = x
}
a()
```

Dynamic scoping:

`y` gets value 3 because `b` doesn't have a local variable called `x` and the dynamically closest scope provides the local variable `x` of `a`

Function Stack vs. Static Scopes



- **Push allocation frames on calls**
- **Pop frames on returns**

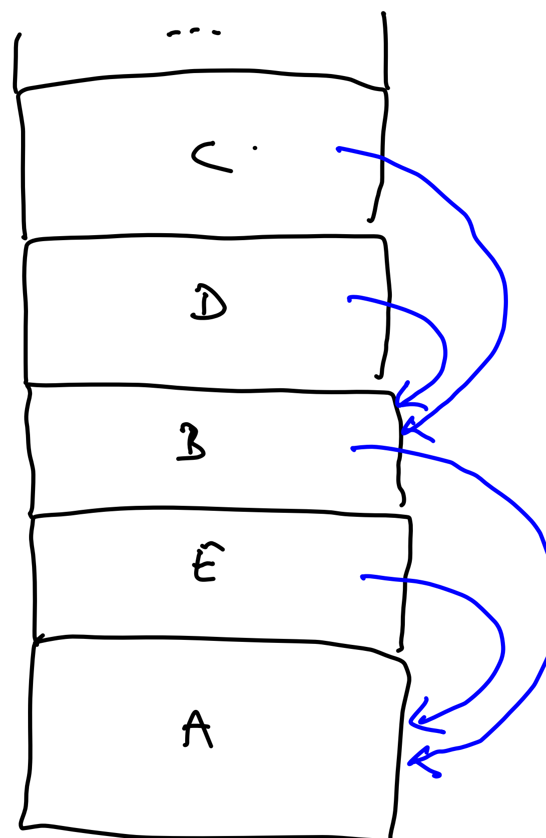
- **Not affected by which functions get called**

How to resolve bindings outside of current scope?

- Each allocation frame has a static link to its parent scope

Nested functions

Function stack



→ .. static link

Built-in Objects

Many PLs have built-in (or predefined) objects

- E.g., for built-in types and APIs
- Invisible, outer-most scope
- Accessible from all scopes, except if hidden

Example: Monkey Patching

[Demo: Monkey patching in JavaScript]

Quiz: Scopes

What does this Python code print?

(Hint: Python uses static scoping)

```
x = "a"
def f():
    def g():
        print(x)
    def h():
        g()
        x = "c"
        print(x)
    x = "b"
    h()
    print(x)
f()
print(x)
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Scopes

What does this Python code print?

(Hint: Python uses static scoping)

```
x = "a"
def f():
    def g():
        print(x) # (1) x in f    : "b"
    def h():
        g()
        x = "c"
        print(x) # (2) x in h    : "c"
    x = "b"
    h()
    print(x)     # (3) x in f    : "b"
f()
print(x)         # (4) x in main: "a"
```

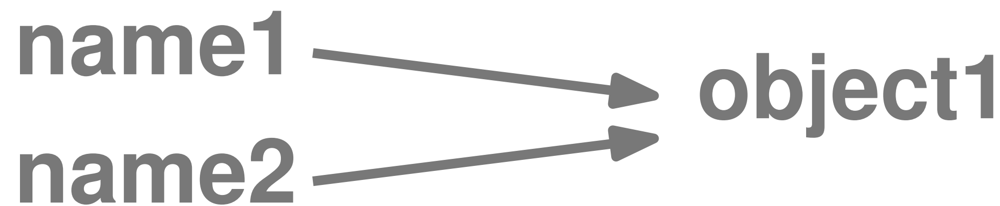
https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview

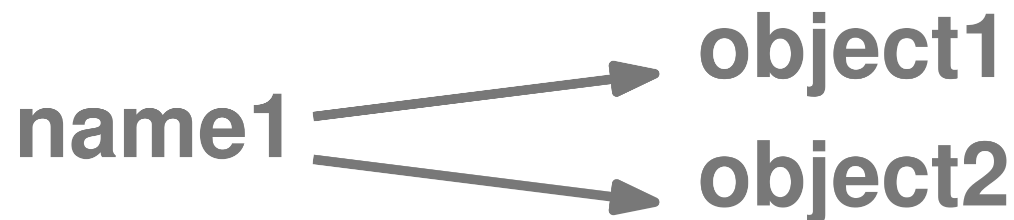
- **Object lifetime and storage management**
- **Scopes**
- **Aliasing and overloading**
- **Binding of referencing environments**

Aliasing and Overloading

Aliasing: Two more more names refer to the same object



Overloading: A name refers to two more objects



Aliasing: Example

```
#include <stdio.h>

void half(double& a)
{ // argument passed by reference
    a = a / 2;
}


int main( int argc, const char* argv[] )
{
    double n = 5.0;
    double *p = &n; // pointer to value stored in n

    half(n);
    half(*p);

    printf("%f\n", n);
}
```

Aliasing: Example

```
#include <stdio.h>

void half(double& a)
{ // argument passed by reference
    a = a / 2;
}


int main( int argc, const char* argv[] )
{
    double n = 5.0;
    double *p = &n; // pointer to value stored in n

    half(n);
    half(*p);

    printf("%f\n", n);
}
```

Aliases to same memory object

Result: 1.250000

Overloading: Example

```
class Overloading{
    void foo() {}
    void foo(int n) {}
    void foo(String s) {}

    public static void main(String[] args) {
        Overloading o = new Overloading();
        o.foo(...);
    }
}
```

Overloading: Example

```
class Overloading{
    void foo() {}
    void foo(int n) {}
    void foo(String s) {}

    public static void main(String[] args) {
        Overloading o = new Overloading();
        o.foo(...);
    }
}
```

Three methods, all with name “foo”

Resolution of name depends on arguments

Overview

- **Object lifetime and storage management**
- **Scopes**
- **Aliasing and overloading**
- **Binding of referencing environments** ←

Referencing Environment

Complete set of bindings at a point in the program

- Determined by scoping rules (e.g., static or dynamic scoping)

What if we create a reference to a function?

- When to apply the scoping rules?

Example

Pseudo code:

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

Example

Pseudo code:

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

Example

Pseudo code:

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

Reference to a function

Function called here

Example

Pseudo code:

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

What memory object is `x` bound to?

Shallow Binding

Referencing environment created when function is called

- Common in languages with dynamic scoping

Shallow Binding

Referencing environment created **when function is called**

- Common in languages with dynamic scoping

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

Shallow Binding

Referencing environment created **when function is called**

- Common in languages with dynamic scoping

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

x bound to the global variable initialized to 5; code prints 5

Deep Binding

Referencing environment created when the reference to the function is created

- Common in languages with static scoping

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

Deep Binding

Referencing environment created when the reference to the function is created

- Common in languages with static scoping

```
function a() {
  var x = 23;
  function b() {
    console.log(x);
  }
  x = 42;
  return b;
}
b = a();
var x = 5;
b();
```

x bound to the local variable initialized to 23; code prints 42, as this is the most recent value of x

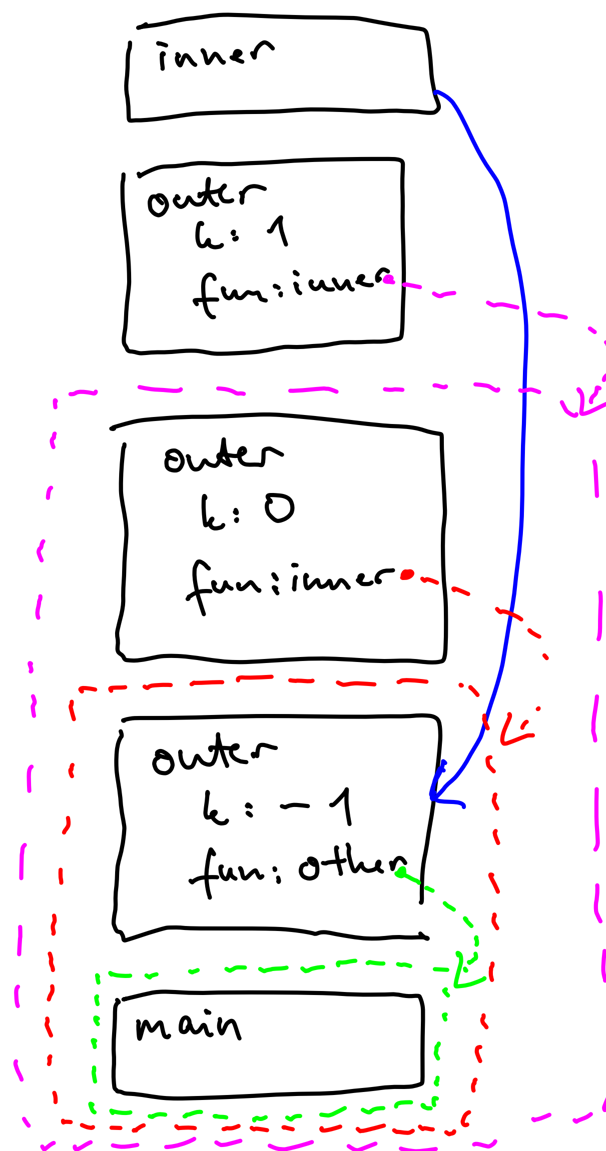
Closure

- **Implementation of deep binding**
- **Closure = Representation of referencing environment + function itself**
- **When creating reference to function, closure it created**

Example: Closures

```
function outer(k, fun) {
    function inner() {
        console.log(k);
    }

    if (k > 0)
        fun();
    else
        outer(k + 1, inner)
}

function other() {}

outer(-1, other);
```

```
function outer (k, fun) {
    function inner () {
        console.log (k)
    }
    if (k > 0)
        fun()
    else
        outer (k+1, inner)
}
function other () {}

outer (-1, other)
```



→ .. static link

⇢ } .. referencing environments captured by closures

prints 0

Quiz: Scopes and Bindings

Which of the following statements is true?

- Scoping rules (e.g., static or dynamic) determine how names are bound to memory objects.
- Built-in objects can be thought of as an invisible, outer scope.
- Languages with pointers don't have any aliasing.
- Closures implement referencing environments created via shallow binding.

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Scopes and Bindings

Which of the following statements is true?

- Scoping rules (e.g., static or dynamic) determine how names are bound to memory objects.
- Built-in objects can be thought of as an invisible, outer scope.
- ~~Languages with pointers don’t have any aliasing.~~
- ~~Closures implement referencing environments created via shallow binding.~~

https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview

- **Object lifetime and storage management**
- **Scopes**
- **Aliasing and overloading**
- **Binding of referencing environments**