Programming Paradigms

Names, Scopes, and Bindings

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Names in PLs

Abstraction in two dimensions

- **From hardware**
  - Variable names abstract away how exactly values are stored
- **From implemented functionality**
  - Function names abstract from the implemented behavior

Binding

- **Association between entities and their names, e.g.,**
  - A variable bound to a memory object
  - A function bound to the code implementing the function
- **Different languages have different rules**
  - E.g., static vs. dynamic binding

Scope

- **Scope of a binding**: Textual region where binding is active
- **Scope**: Maximal region where no bindings change

Example (Python):

```
x = 1
def f():
    x = 2
    y = x
```

Scope

- **Scope of a binding**: Textual region where binding is active
- **Scope**: Maximal region where no bindings change

Example (Python):

```
x = 1
def f():
    x = 2
    y = x
```

Outer scope

Scope of function

Overview

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

Object Lifetime

Every memory object has a lifetime

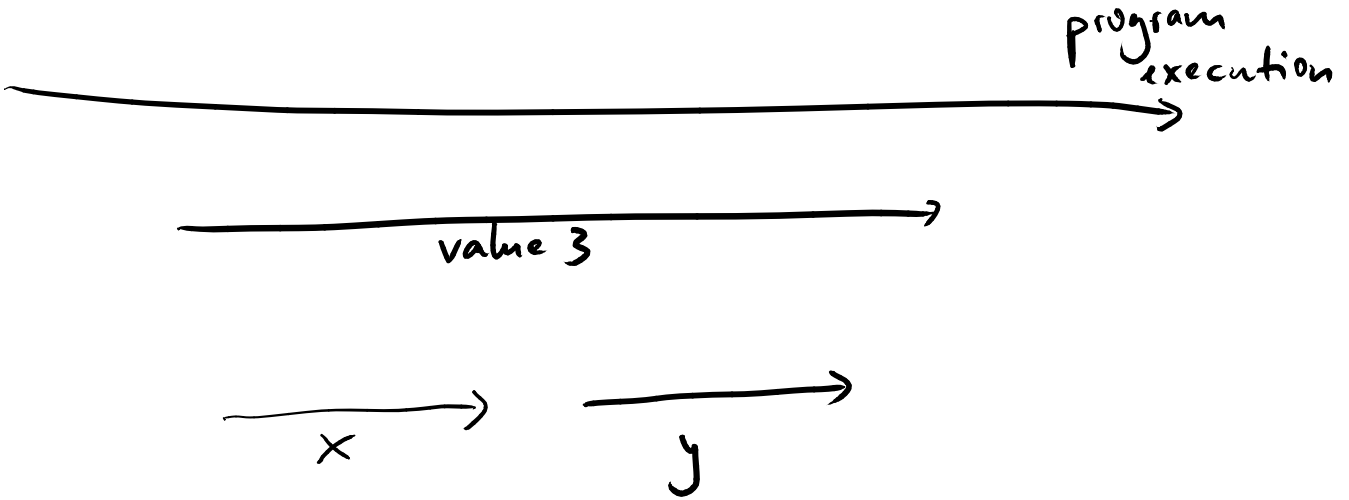
- Global variables: Entire program execution
- Local variables: Function execution

Object lifetime vs. binding lifetime

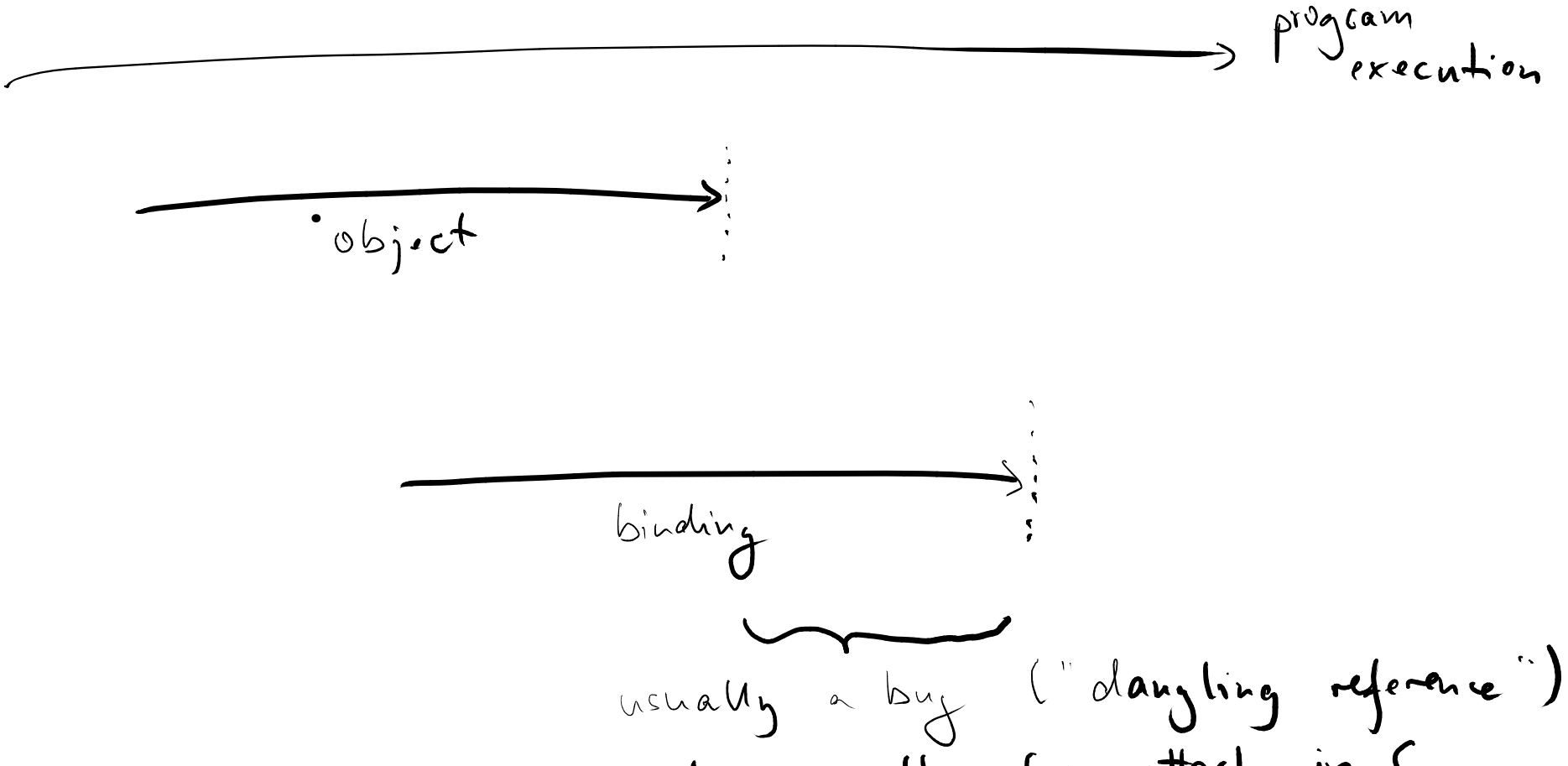
- A single object may be bound to multiple names
- Bindings may be concurrent

Example 1

```
fun f() {
    x = 3
    return x
}
y = f()
```



Example 2


program execution
object
binding
usually a bug ("dangling reference")
↳ use-after-free attack in C

Storage Allocation

Three kinds of memory objects

- Static
  - Absolute address retained throughout execution
- Stack
  - Usually within subroutines
  - Allocation/deallocation on call/return
- Heap
  - Allocation and deallocation at arbitrary times

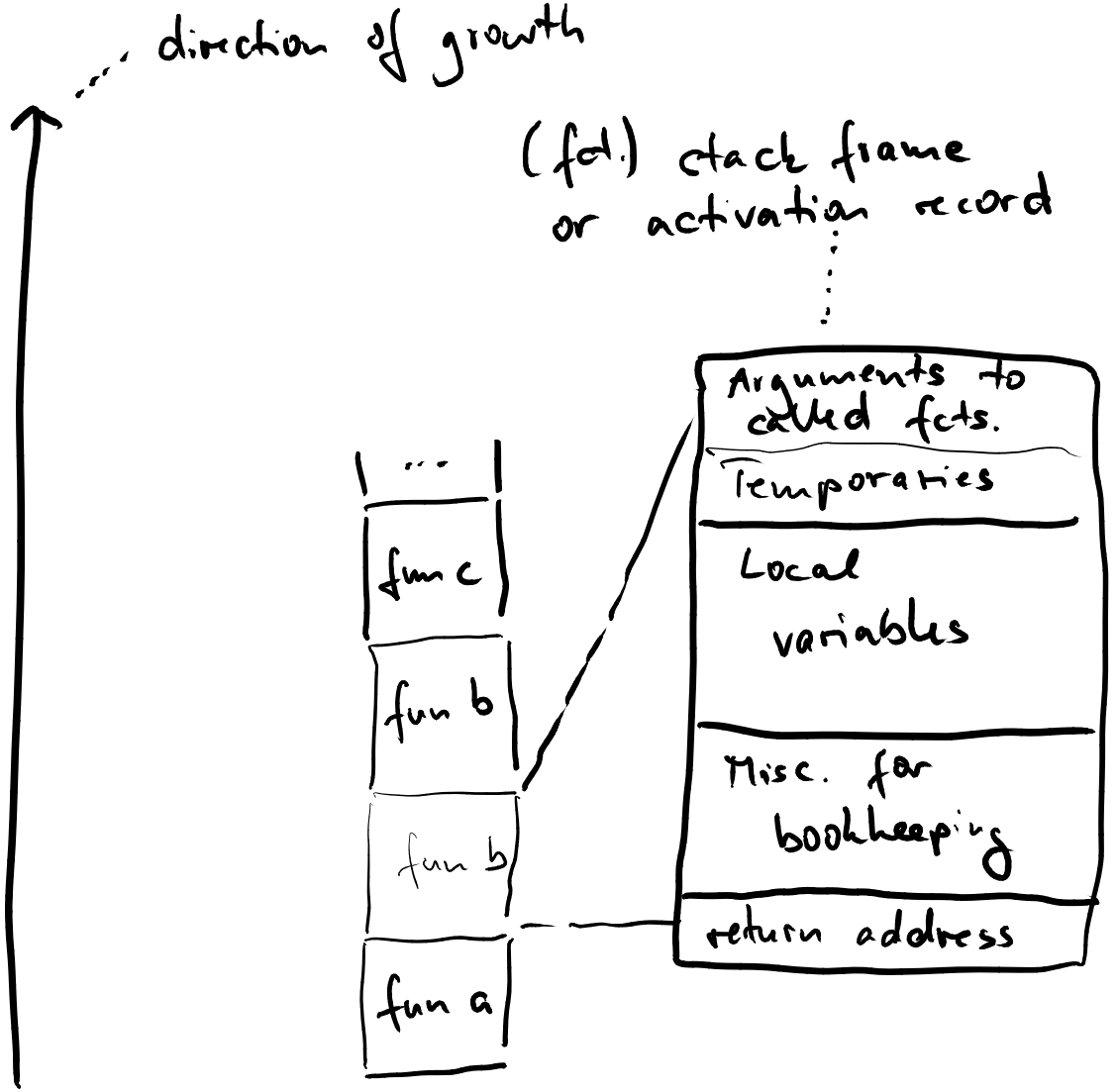
Statically Allocated Memory

Depending on the PL, used, e.g., for

- Global variables
- Constant literals
- Symbol tables
- Program code itself
- Compile-time constants
  - Even if local to function

Stack-based Allocation

```
fun c() {
    ...
}
fun b() {
    if ...
        b()
    else
        c()
}
fun a() {
    b()
}
// main
a()
```



Heap-based Allocation

- **For dynamically allocated data structures and objects whose size is statically unknown**
  - E.g., objects in Java
- **Some PLs: Managed memory**
  - Unreachable objects: Implicitly deallocated
    - Unreachable = No active binding
  - Less control but fewer bugs
    - E.g., no use-after-free

Quiz: Memory Allocation

```
class Course {
    String name;
    int credits;

    // constructor
}

public class App {
    public static void main(String[] args) {
        String name = "PP";
        int credits = 3+1;

        Course c = null;
        c = new Course(name, credits);
    }
}
```

Where are the following data objects stored (in Java)?

- The integer 4
- The reference variable `c`
- The `Course` object

Quiz: Memory Allocation

```
class Course {
    String name;
    int credits;

    // constructor
}

public class App {
    public static void main(String[] args) {
        String name = "PP";
        int credits = 3+1;

        Course c = null;
        c = new Course(name, credits);
    }
}
```

Stack (in allocation frame of `main`)

Quiz: Memory Allocation

```
class Course {
    String name;
    int credits;

    // constructor
}

public class App {
    public static void main(String[] args) {
        String name = "PP";
        int credits = 3+1;

        Course c = null;
        c = new Course(name, credits);
    }
}
```

Quiz: Memory Allocation

```
class Course {
    String name;
    int credits;

    // constructor
}

public class App {
    public static void main(String[] args) {
        String name = "PP";
        int credits = 3+1;

        Course c = null;
        c = new Course(name, credits);
    }
}
```

Bonus: Where is the string stored?

Quiz: Memory Allocation

```
class Course {
    String name;
    int credits;

    // constructor
}

public class App {
    public static void main(String[] args) {
        String name = "PP";
        int credits = 3+1;

        Course c = null;
        c = new Course(name, credits);
    }
}
```

Bonus: Where is the string stored? String pool in heap space

Overview

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

Scoping Rules

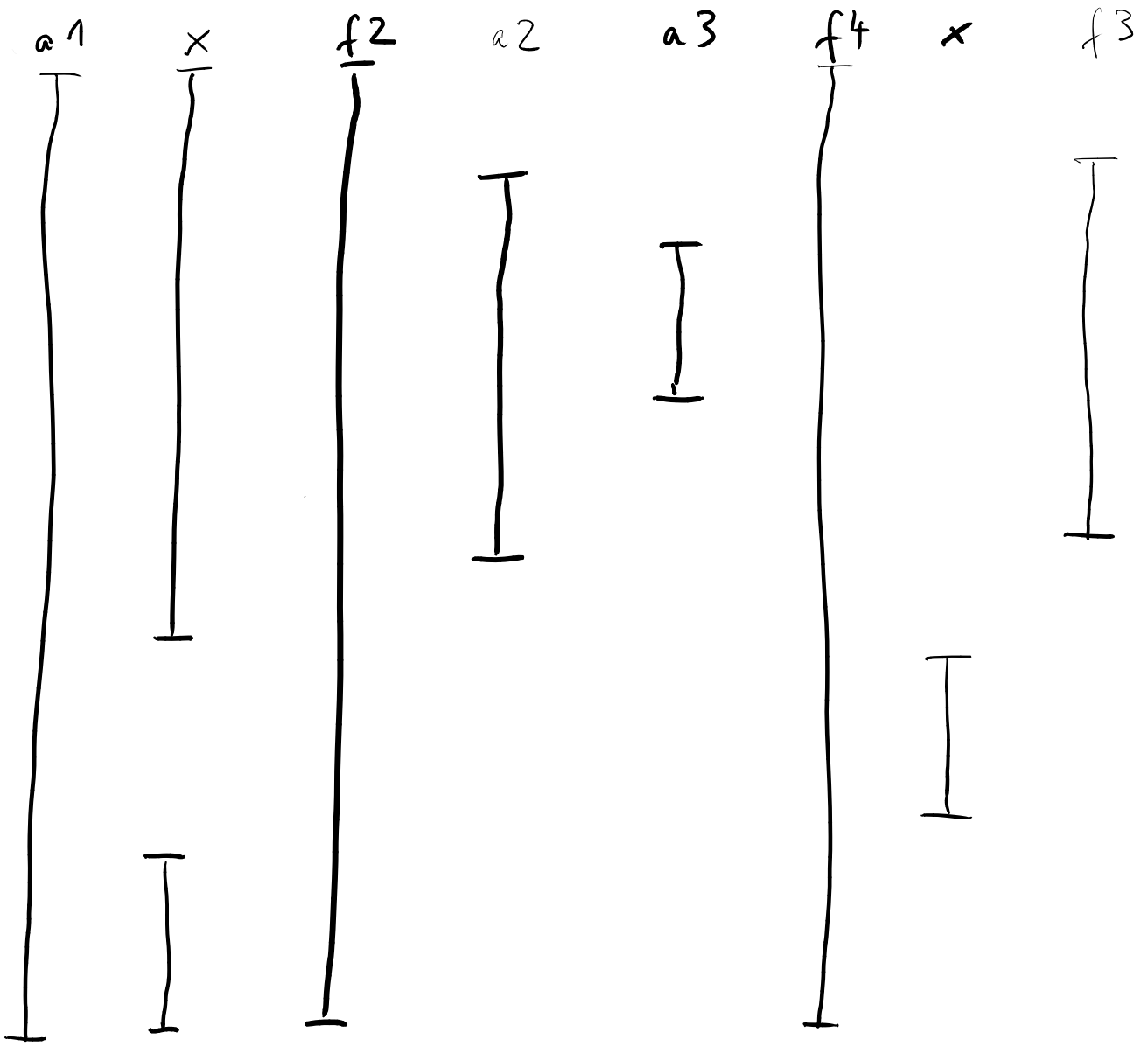
- **Scoping rules: Define which bindings are active**
  - I.e., what's the meaning of a name at a given program point?
- **Each PL defines its scoping rules**
  - E.g., Basic has only one scope
  - Most PLs have nested scopes for subroutines

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 scopes nested within
  - Inner scopes can hide names from outer scopes

Example

```
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`
- 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`
- the dynamically closest scope provides the local variable `x` of `a`

Quiz: Dynamic Scoping

What does this Perl code print?

(Hint: Perl uses dynamic scoping for `local` variables)

```
$b = 5;
sub foo {
  return $b;
}
sub bar {
  local $b = 3;
  return foo();
}
print bar();
```

Quiz: Dynamic Scoping

What does this Perl code print?

(Hint: Perl uses dynamic scoping for `local` variables)

```
$b = 5;
sub foo {
  return $b;
}
sub bar {
  local $b = 3;
  return foo();
}
print bar();
```


Scope of `local $b` dynamically extends into invocation of `foo`

Quiz: Dynamic Scoping

What does this Perl code print?

(Hint: Perl uses dynamic scoping for `local` variables)

```
$b = 5;
sub foo {
  return $b;
}
sub bar {
  local $b = 3;
  return foo();
}
print bar();
```


Scope of `local $b` dynamically extends into invocation of `foo`

Answer: 3

Quiz: Static Scoping

What does this Python code print?

(Hint: Python uses static scoping)

```
x = "1"
def f():
    def g():
        print(x)
    def h():
        g()
        x = "s"
        print(x)
    x = "e"
    h()
    print(x)
f()
print(x)
```

Quiz: Static Scoping

What does this Python code print?

(Hint: Python uses static scoping)

```
x = "1"
def f():
    def g():
        print(x) # (1) x in f    : "e"
    def h():
        g()
        x = "s"
        print(x) # (2) x in h    : "s"
    x = "e"
    h()
    print(x)     # (3) x in f    : "e"
f()
print(x)         # (4) x in main: "1"
```

Function Stack vs. Static Scopes

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

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

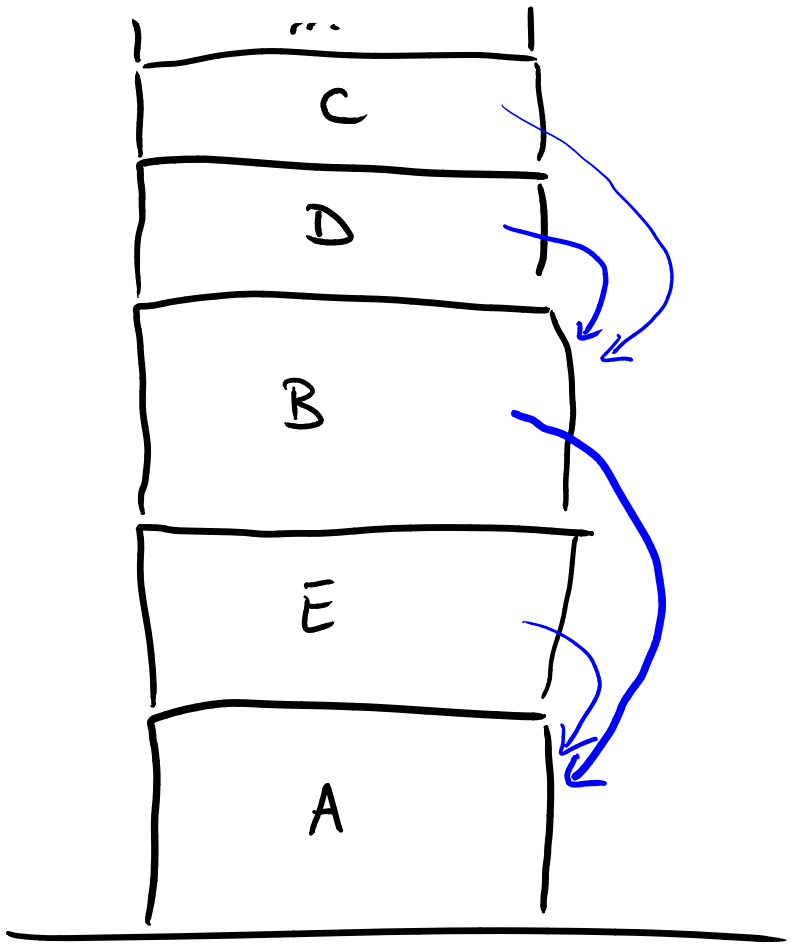
How to resolve bindings outside of current scope?

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

Example

Nested functions

Function stack



→ ... static link

Built-in Objects

Many PLs have built-in (or predefined) objects

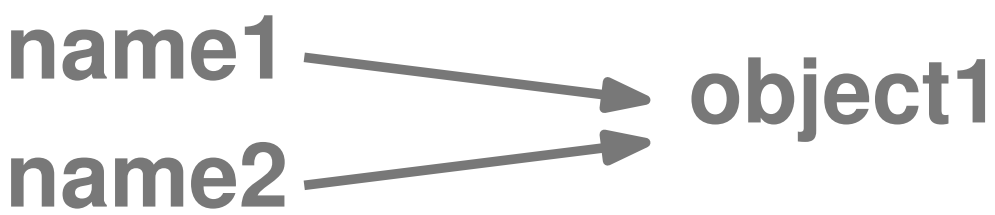
- E.g., for built-in types and APIs
- Invisible, outer-most scope
- Accessible from all scopes, except if hidden

Overview

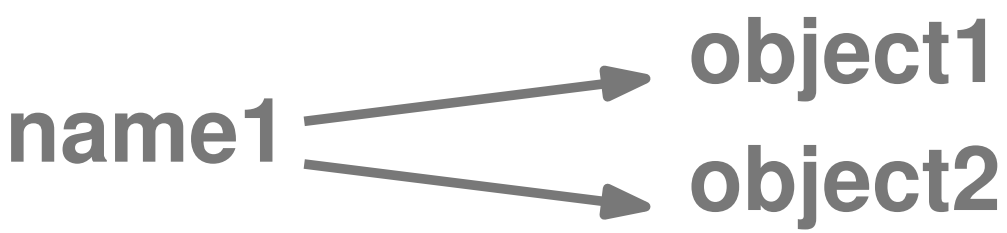
- Object lifetime and storage management
- Scopes
- Aliasing and overloading 
- Binding of referencing environments

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();
```

Reference to a function

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

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 is created

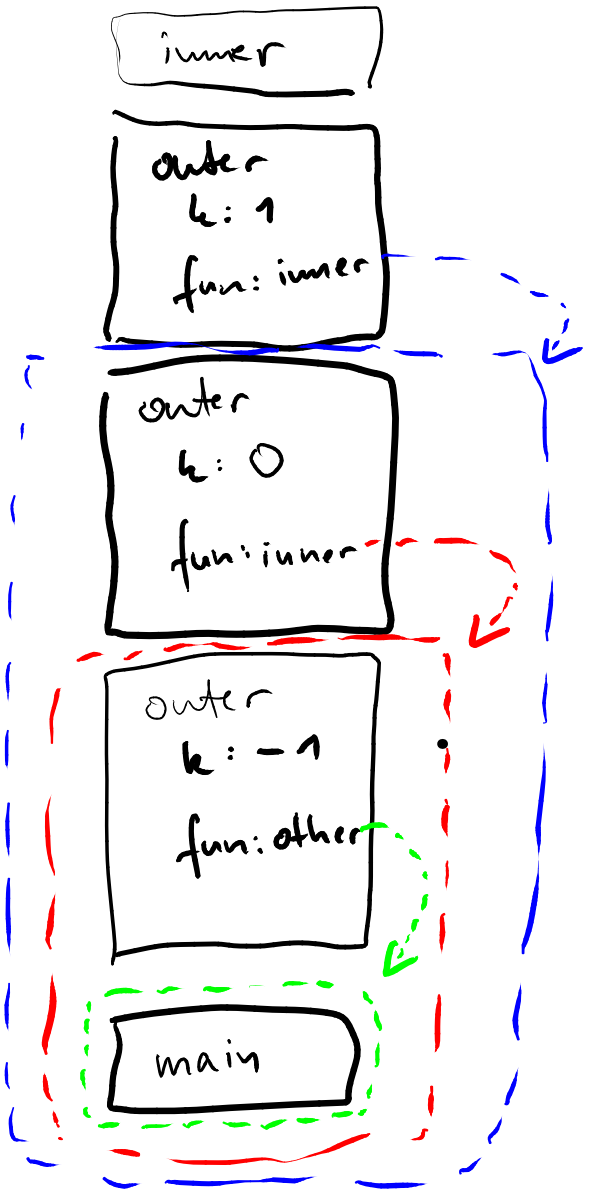
Example: Closures

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

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

function other() {}

outer(-1, other);
```



referencing environments captured by closures

prints 0

Quiz: Scopes and Bindings

Which of the following statements is true?

- Heap-allocated memory is freed at the end of each function call.
- A memory object is bound to at most one name.
- Built-in objects are available in an invisible, outer-most scope.
- With shallow binding, the referencing environment is created when a function is called.

Quiz: Scopes and Bindings

Which of the following statements is true?

- ~~Heap-allocated memory is freed at the end of each function call.~~
- ~~A memory object is bound to at most one name.~~
- Built-in objects are available in an invisible, outer-most scope.
- With shallow binding, the referencing environment is created when a function is called.

Overview

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