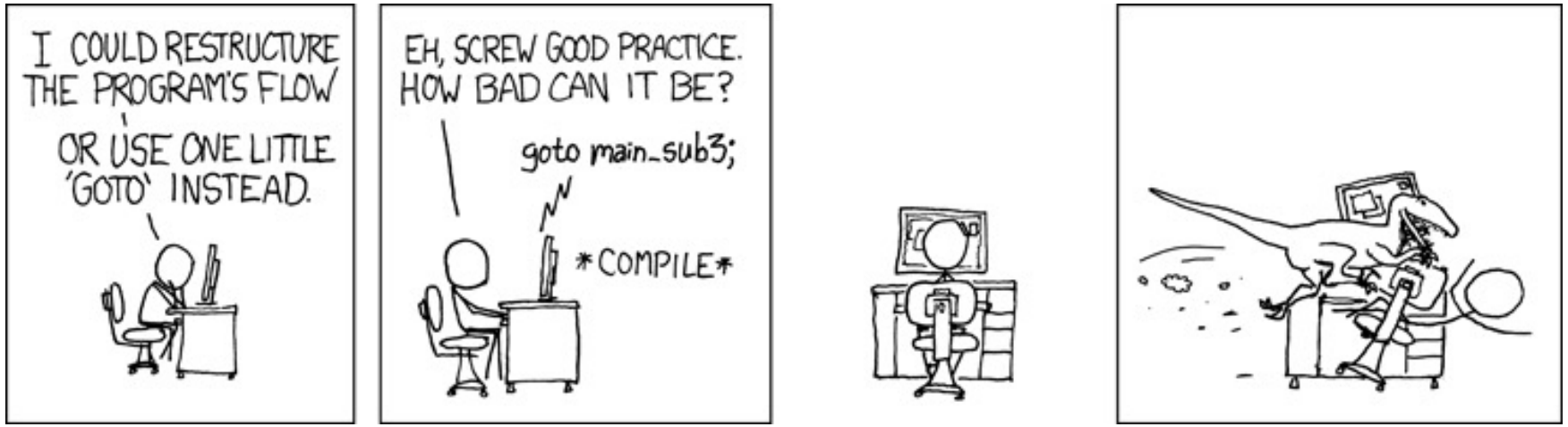
IN3040 – Programming Languages
Runtime Organization part II

Eyvind W. Axelsen (eyvinda@ifi.uio.no)



Slides adapted from previous years' material by Birger Møller-Pedersen (birger@ifi.uio.no)

Today: Higher-Order Functions, and Objects at runtime

- Higher-order ("first class") functions:
  - Functions passed as arguments
  - Functions that return functions from nested blocks
  - Need to maintain environment of function
- Simpler case
  - Function passed as argument
  - Need pointer to activation record "higher up" in stack
- More complicated case
  - Function returned as result of function call
  - Need to keep activation record of returning function

- Objects at runtime
  - Which activation blocks do we use?

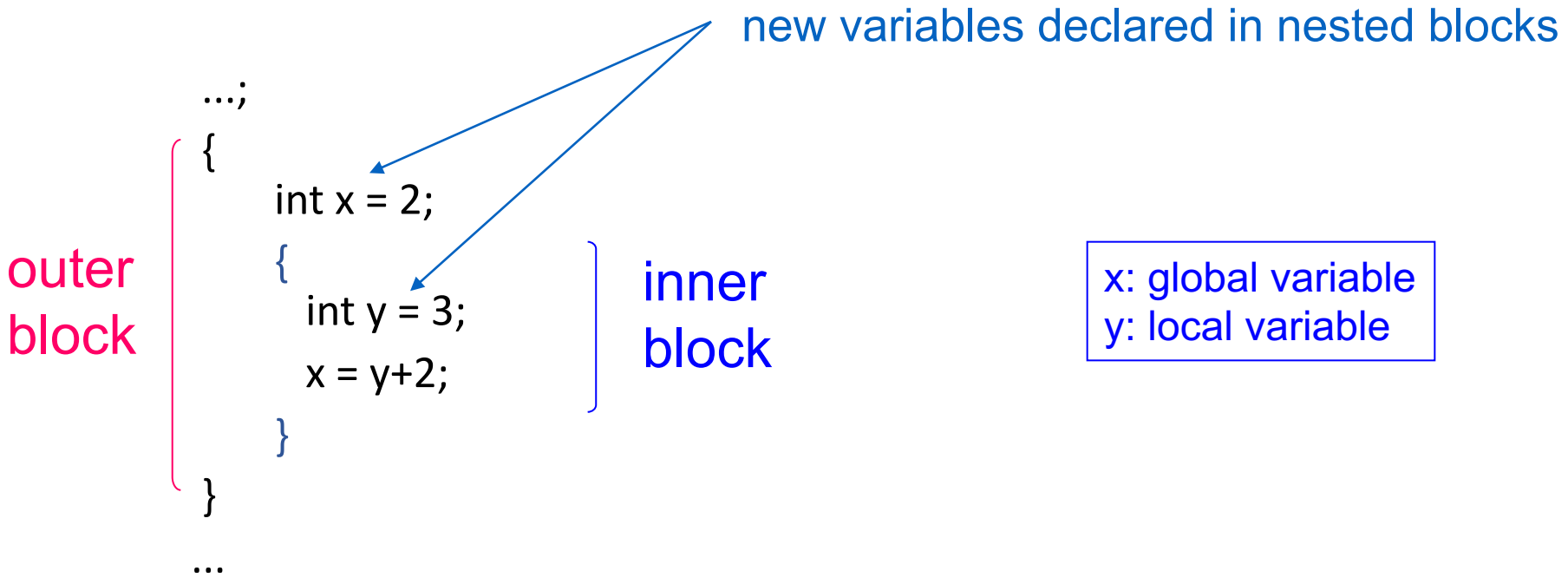
- Fun with Javascript
  - The worlds most popular(?) programming language!
  - Heavy use of higher-order functions

Repetition from last time



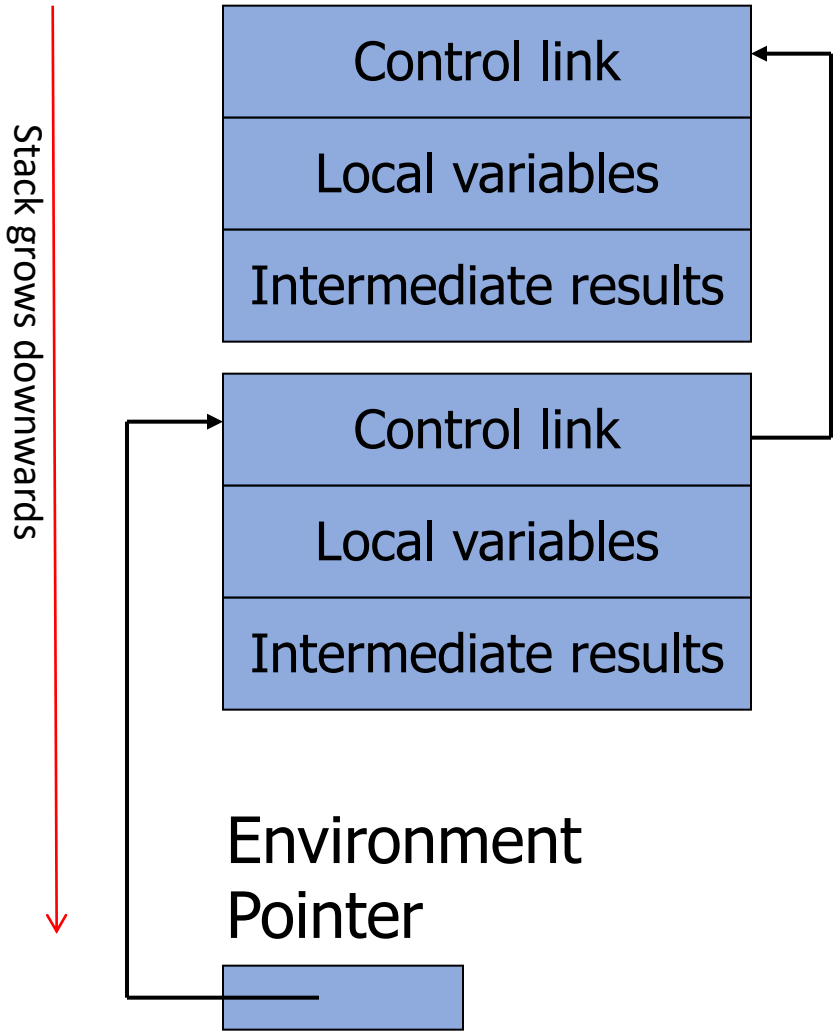
Block-Structured Languages

- Blocks are *syntactical* structures
- Can be nested within each other



- Storage management – memory representation
  - Enter block: allocate space for variables
  - Exits block: space *may* be de-allocated

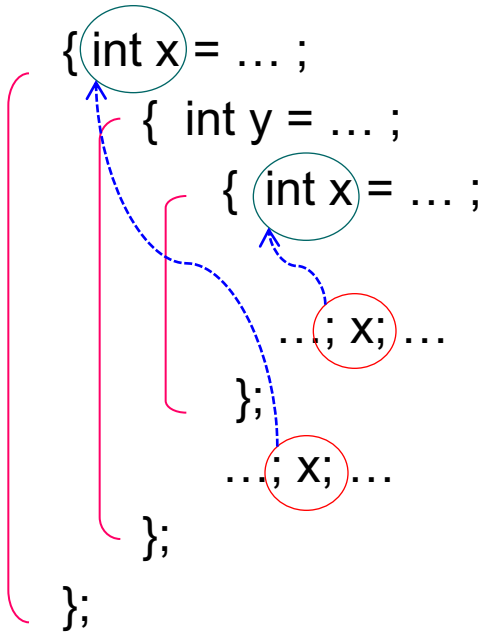
Activation record for in-line block



- Environment pointer
  - Pointer to current record on stack
- Control link (dynamic link)
  - Pointer to previous record on stack
- Push record on stack
  - Set new control link (in new record) to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer
  - (No need to actively blank memory)

Some basic concepts

- Declaration
  - Specifies properties (name, type, kind) of an identifier
- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated

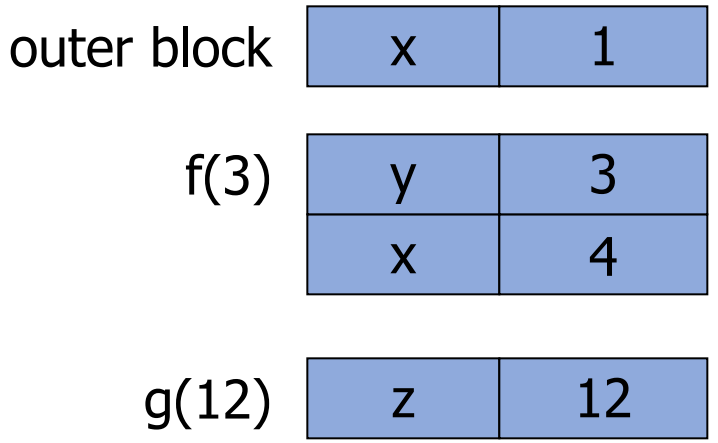


- Inner declaration of x hides outer one.
- Called “hole in scope”
- Lifetime of outer x includes time when inner block is executed
- Lifetime ≠ scope

Access to global variables

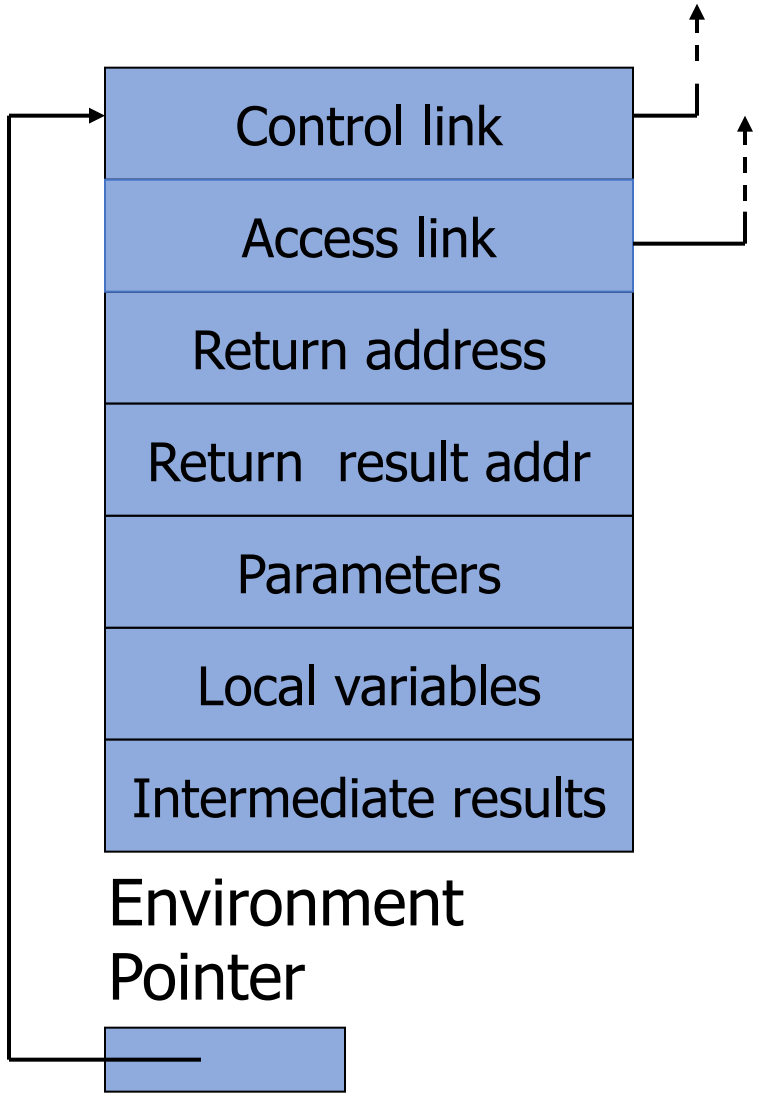
- Two possible scoping conventions
  - Static scope: refer to closest enclosing block (syntactically)
  - Dynamic scope: most recent activation record on stack
- Example

```
int x = 1;
function g(z) = x+z;
function f(y) =
    {
      int x = y+1;
      return g(y*x)
    };
f(3);
```

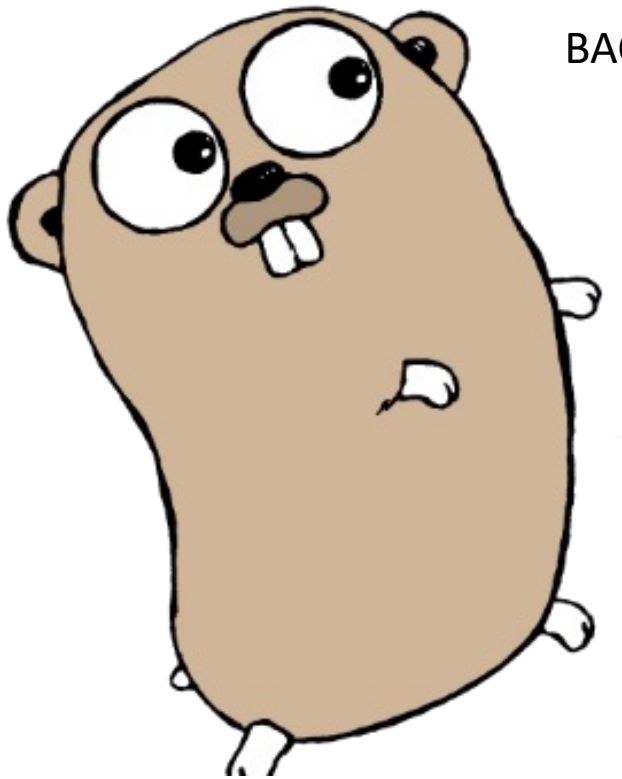


Which x is used for expression x+z?

Activation record for static scope



- Control link (dynamic link)
  - Link to activation record of previous (calling) block
  - Why is it called *dynamic?*
- Access link (static link)
  - Link to activation record corresponding to the closest enclosing block in program text
  - Why is it called *static*?
- Difference
  - Control link depends on dynamic behavior of program
  - Access link depends on static form of program text



BACK ON TRACK

First Class Functions

Gopher by Renee french

Functions as parameters: why?

Given a Person class and a list of Persons

Can you write a function that finds all persons that
- Are female?
- Are older than 50 years?
- Like drinking beer?

Functions as parameters: why?

Given a Person class and a list of Persons

Can you write a function that finds all persons that
- Are female?
- Are older than 50 years?
- Like drinking beer?

A first attempt in Java:

```
List<Person> findPersonsThatAreFemale(List<Person> persons) {
    List<Person> filteredList = new List<Person>();
    for(Person p in persons) {
        if(p.gender == ”female”) {
            filteredList.Add(p);
        }
    }
    return filteredList;
}
```

Functions as parameters: why?

Given a Person class and a list of Persons

Can you write a function that finds all persons that
- Are female?
- Are older than 50 years?
- Like drinking beer?
- …

A first attempt in Java:

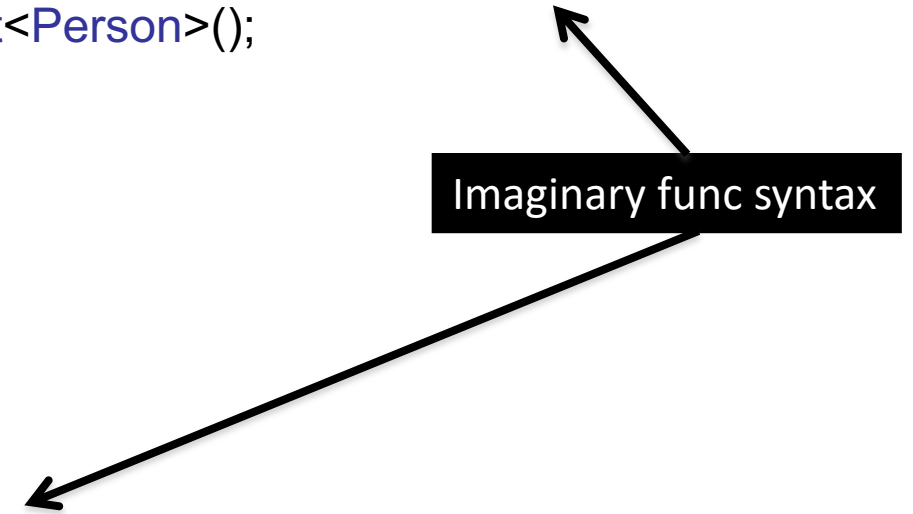
```
List<Person> findPersonsThatAreOlderThan50(List<Person> persons) {
    List<Person> filteredList = new List<Person>();
    for(Person p in persons) {
        if(p.age > 50) {
            filteredList.Add(p);
        }
    }
    return filteredList;
}
```

Functions as parameters: why? - DRY

```
List<Person> filterPersons(List<Person> persons, (Person → Boolean) filter) {
    List<Person> filteredList = new List<Person>();
    for(Person p in persons) {
        if(filter(p)) {
            filteredList.Add(p);
        }
    }
    return filteredList;
}

filterPersons( persons, function(Person p) { return p.age > 50 } );

filterPersons( persons, function(Person p) { return p.gender == ”female” } );
```



Is this in use in languages today?

Traditional OO approach: make a class out of it!

E.g. in Java (pre v8):

```
Collections.sort(list, new Comparator<MyClass>() {
    public int compare(MyClass a, MyClass b)
    {
        // compare objects here
    }
});
```

What is going on here?

- Comparator<T> is an interface, and T is a type parameter
- This interface has one method signature, int compare(T a, T b);
- Starting from the first {, we have an anonymous class implementing this interface

Is this in use in languages today?

```
Collections.sort(list, new Comparator<MyClass>() {
   public int compare(MyClass a, MyClass b)
   {
      return a.isGreaterThan(b);
   }
});
```

Functional approach:

- Java (v8 and up):

```
list.sort((a, b) -> a.isGreaterThan(b));
```

- C#:

```
myList.Where(a => a.Number > 42).OrderBy(a => a.Number)
      .ThenBy(a => a.FooBar);
```

- Python:

```
Celsius = [39.2, 36.5, 37.3, 37.8]
Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
```

- JavaScript (Node):

```
app.get('/somepath/:date', function (req, res)
 res.setHeader('Content-Type', 'application/json');
 fetchStuff({ date: req.params.date}, function (error, result) {
    if (error) console.log(error);
    res.end(result);
  });
});
```

Pass function as argument

There are two declarations of x

Which one is in scope for each usage of x?

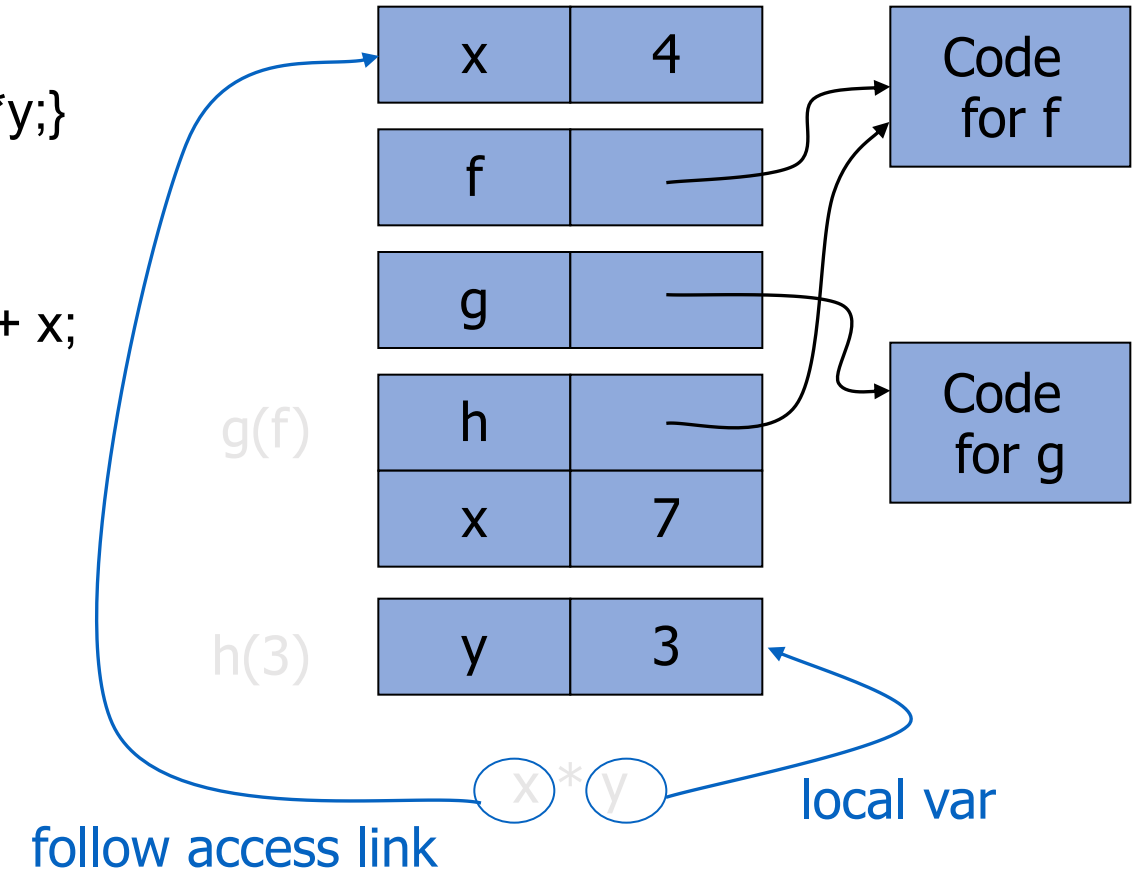
```
{ int x = 4;
   { int f(int y) {return x*y;}
      { int g(int→int h) {
            int x=7;
            return h(3) + x;
         }
        g(f);
      }
   }
}
```

Formal function parameter

Actual function parameter

Static Scope for Function Argument

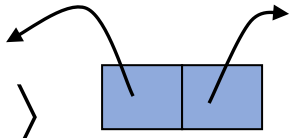
```
{ int x = 4;
   { int f(int y) {return x*y;}
      { int g(int→int h) {
            int x=7;
            return h(3) + x;
         }
         g(f);
      }
   }
}
```



How is access link for h(3) set? → Next slides

Closures

- Function value is pair *closure* = ⟨*env*, *code* ⟩



- A closure “closes” around its environment
  - A function value needs to take its environment along:

```
var createGreeter = function() {
    var msg = "Hello World";

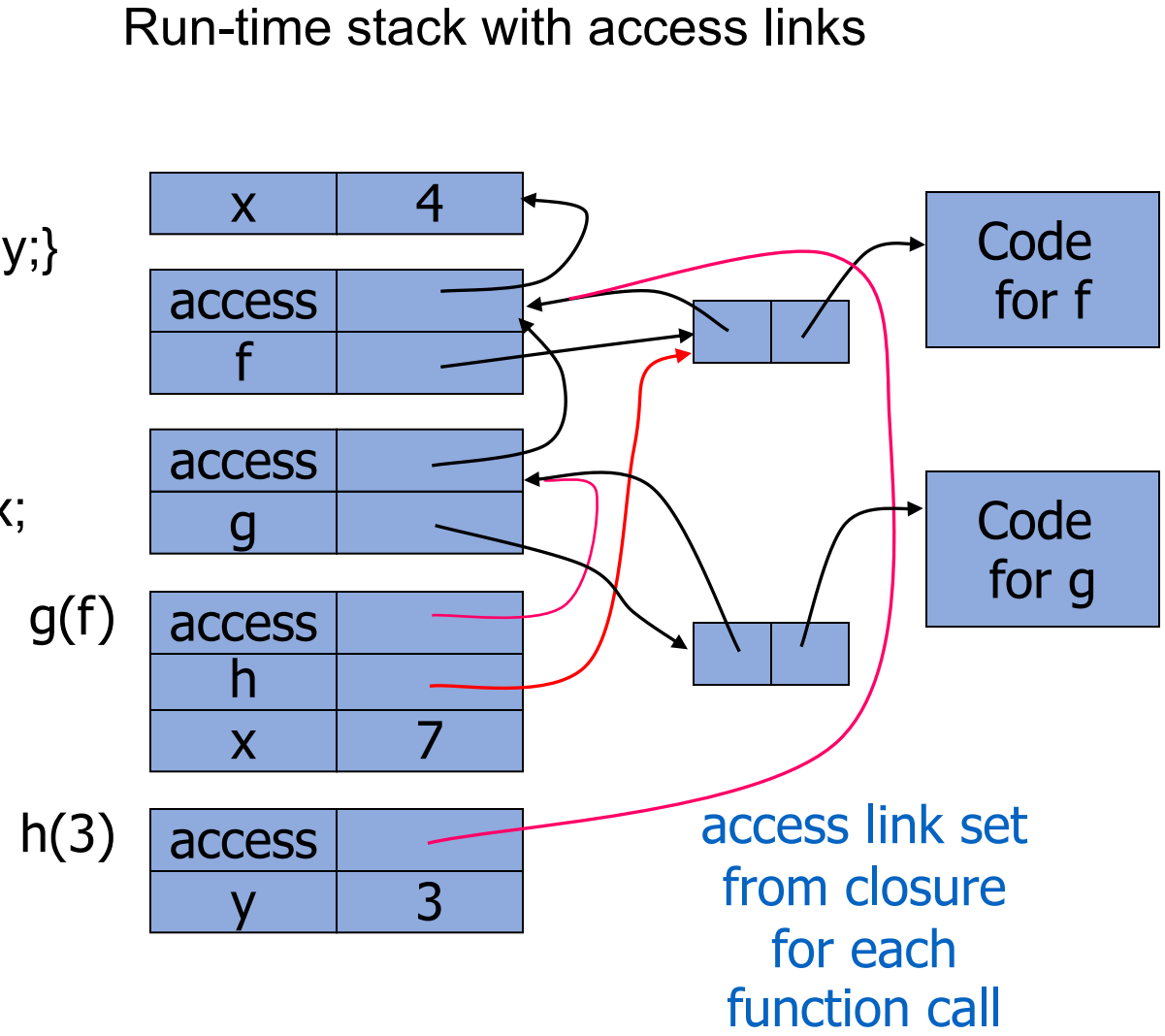
    var greeter = function() {
        console.log(msg);
    }

    return greeter;
}
```

- When a function represented by a closure is called
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure

Function Argument and Closures

```
{ int x = 4;
  { int f(int y){return x*y;}
    { int g(int→int h) {
          int x=7;
          return h(3)+x;
      }
    g(f);
  }
}
```



Return Function as Result

- Language feature
  - Functions that return “new” functions
  - Need to maintain environment of function

- Function “created” dynamically
  - function value is closure = ⟨env, code⟩
  - code *not* compiled dynamically (in most languages)

Example: Return function with private state

Function “make counter” returns a closure

```
int→int make_counter (int init) {
      int count = init;
      int counter(int inc) {
         return count += inc;
      }
      return counter; // we return a function
}
int→int c  = make_counter(1);
print c(2) + c(2);
```

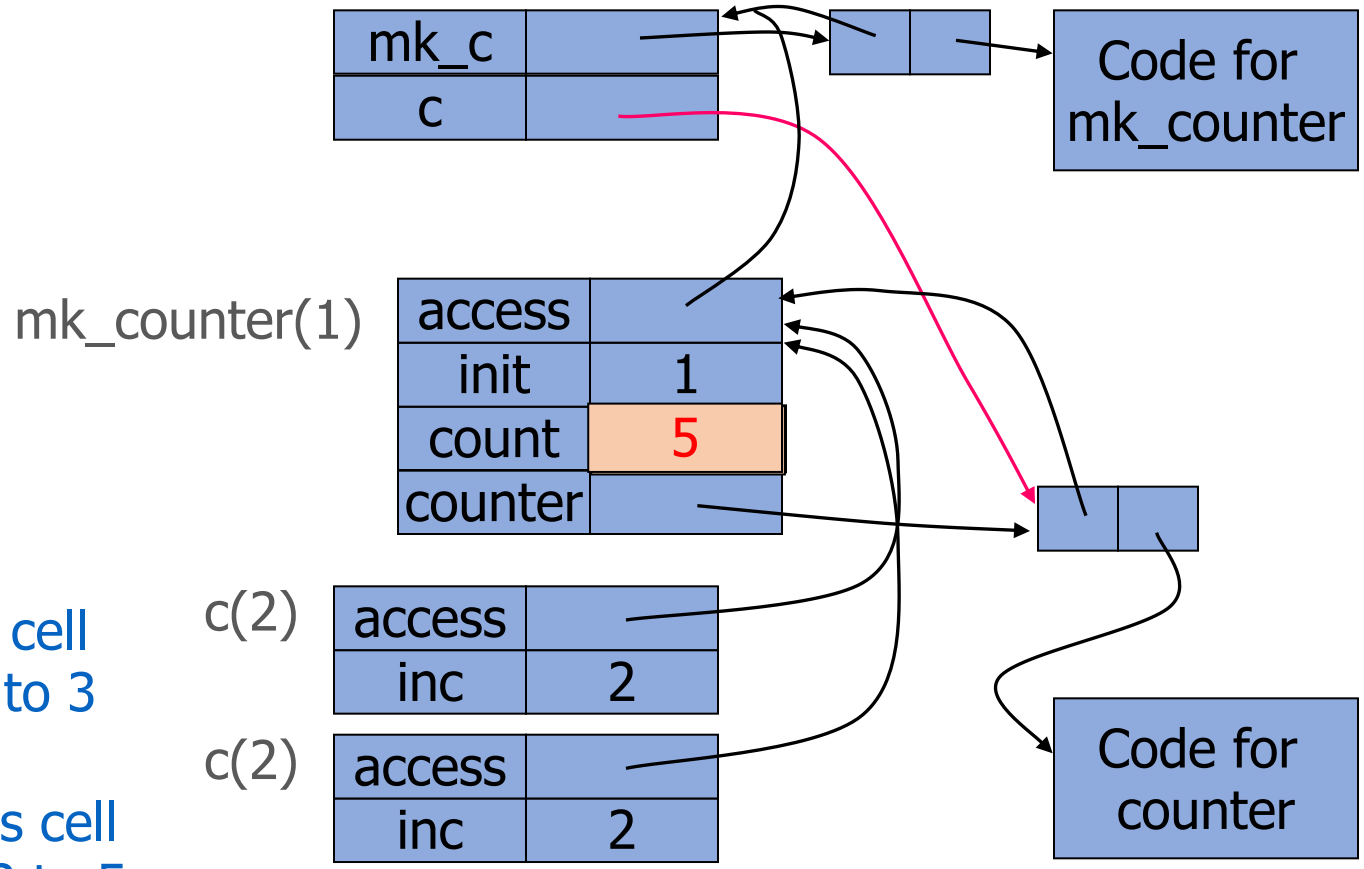
How is correct value of count determined in call c(2) ? Next slide →

Function Results and Closures

```
int→int mk_counter (int init) {
    int count = init;
    int counter(int inc) {
        return count += inc;
    }
    return counter;
}
int→int c  = mk_counter(1);
print c(2) + c(2);
```

Call changes cell value from 1 to 3

Call changes cell value from 3 to 5



Activation record associated with returned function cannot be deallocated upon function return



Regular functions

Call functions, return values from functions



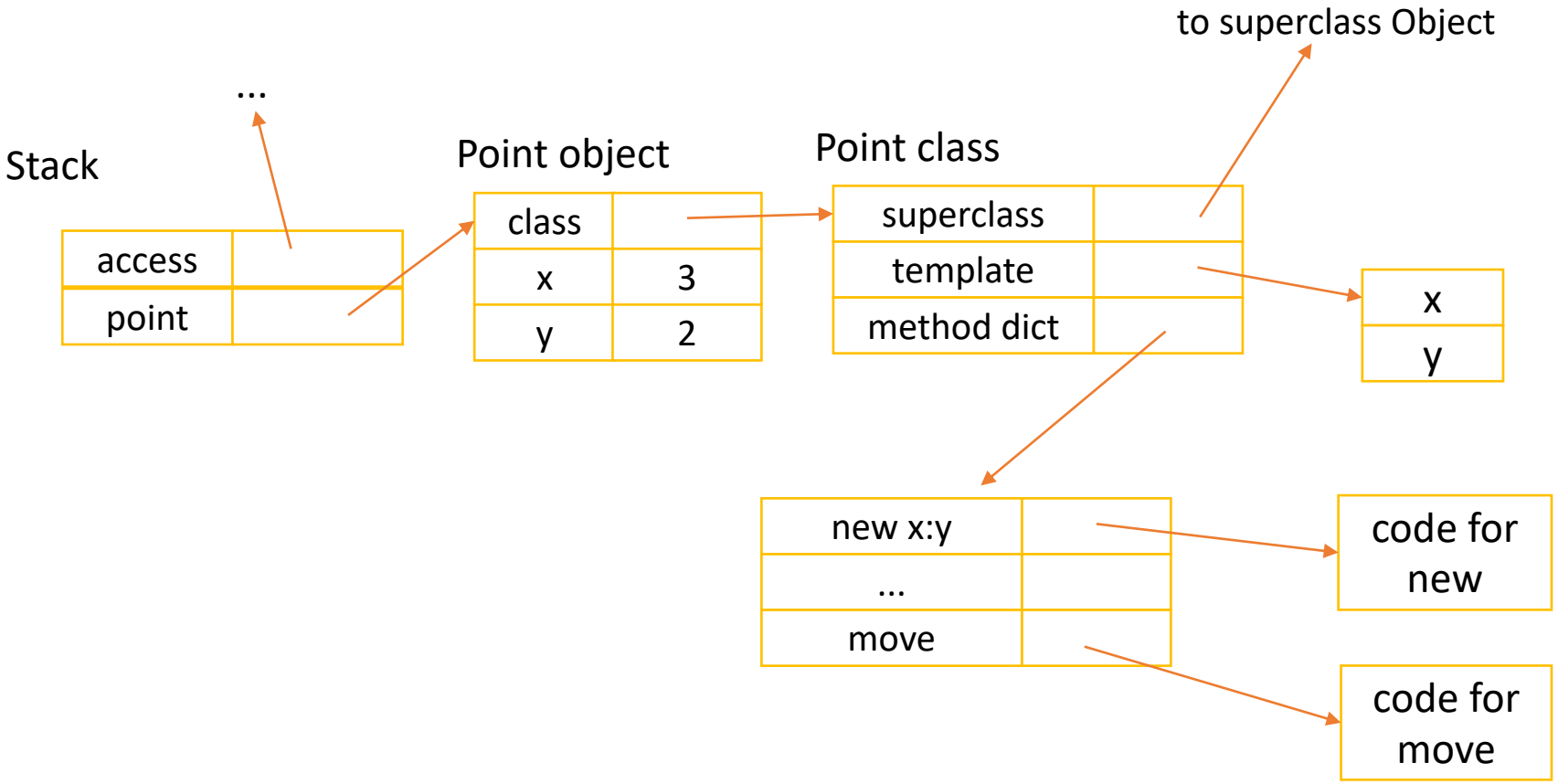
First-class functions

Use functions as ordinary ("first-class») values;
send as actual parameter, return as result

Classes and objects at runtime

- Pointers to objects on the stack
  - In “normal” activation bloks
- The objects themselves are typically not stored on the stack
  - Separate location called the heap
- Data for each object stored with the object
  - E.g. x and y coordinates for a point
- Common functionality stored in shared location
  - Methods, static variables

Smalltalk – Point object and class



Smalltalk – runtime support for inheritance

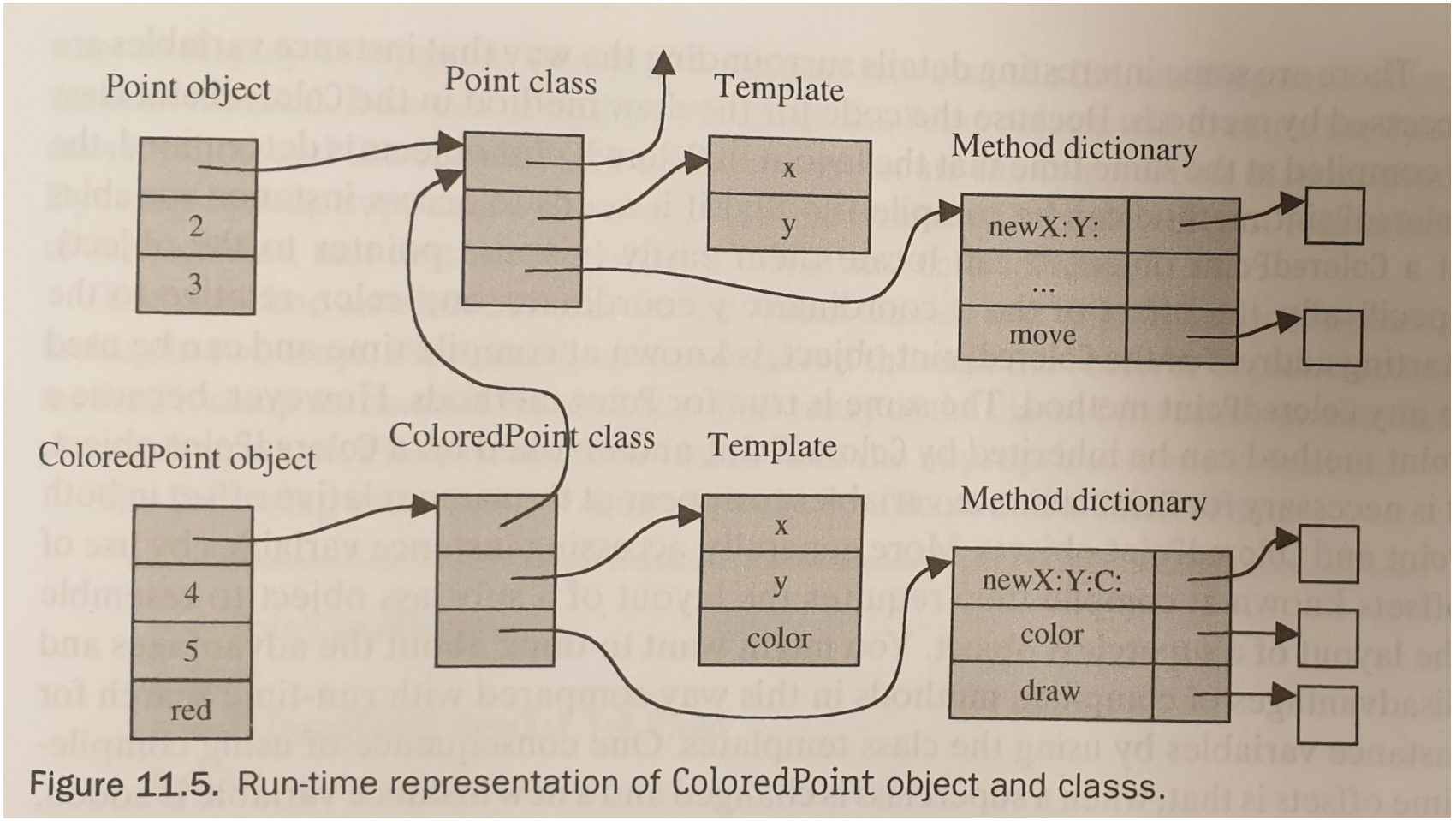


Figure 11.5. Run-time representation of ColoredPoint object and classs.

Mitchell p. 317

Aside: not all scopes are equal




```
this.value = 42; //Global variable

var obj = {
    value: 0, //Local field in object
    increment: function() {
        this.value++;
        alert(this.value);

        var innerFunction = function() {
            alert(this.value);
        }

        innerFunction(); // Function call
    }
}
obj.increment(); // Method call
```

What will be shown on screen?

Try it out yourselves: http://jsfiddle.net/7jxw1r9v/1/

How do we fix this?

```
this.value = 42; //Global variable

var obj = {
    value: 0, //Local field in object
    increment: function() {
        this.value++;
        alert(this.value);
        var that = this;

        var innerFunction = function() {
            alert(that.value);
        }

        innerFunction(); //Function call
    }
}
obj.increment(); //Method call
```

Why does this help?

Because this function is represented by a *closure* that captures the «that» variable

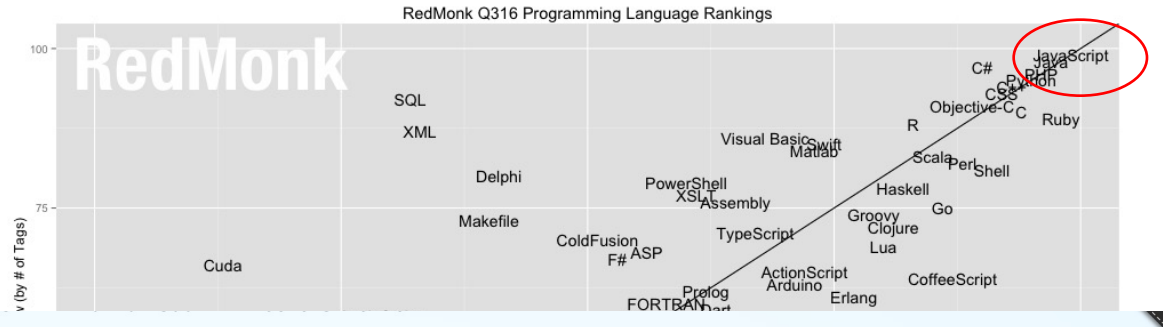
Everyone loves Javascript!

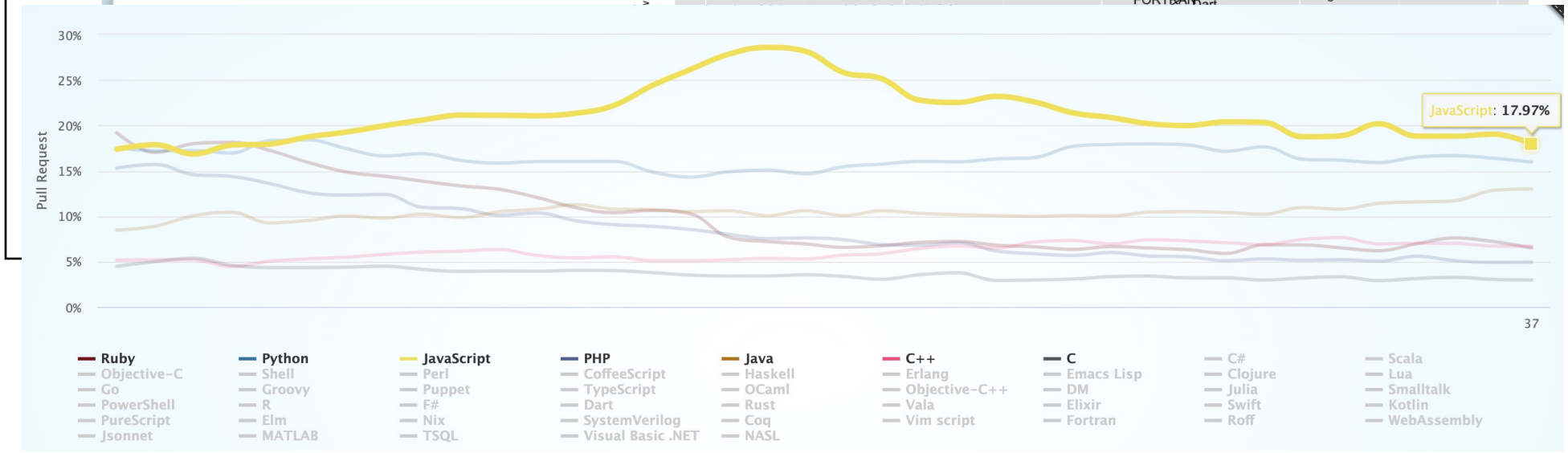
www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Programming Language

The hall of fame listing all "Programming Language of the Yea
language that has the highest rise in ratings in a year.

Year	Winner
2014	JavaScript





https://www.destroyallsoftware.com/talks/wat

But WHY all these WATs?

- JavaScript has automatic type coercion
  - It will try to convert types into something that matches the operand!
- [ ] + [ ] = ""
  - The + operator cannot operate on arrays, so the array is *coerced* to its string representation, which is a toString() of all the elements joined by commas.
- { } + [ ] = 0
  - The first is recognized as an empty code block.
  - The plus is thus unary, and [ ] is coerced to an empty string, which is in turn coerced to 0.
- { } + { } = NaN
  - The first is again an empty code block
  - The second { } is an empty object, which is coerced to [object Object], which is the toString() repr of objects
  - Which is again not a number, or NaN

More fun: scoping and blocks

Java:

```
void main() {
    Integer x = 1;
    System.out.println(x);
    if (true) {
        Integer x = 2;
        System.out.println(x);
    }
    System.out.println(x);
}
```

Output: «1», «2», «1»

JavaScript:

```
function main() {
    var x = 1;
    console.log(x);
    if (true) {
        var x = 2;
        console.log(x);
    }
    console.log(x);
}
```

Output: «1», «2», «2»!

- JavaScript has blocks, but (traditionally) not block scope!
- Declarations are always «hoisted» to the top of the function

More fun: scoping and blocks

Java:

```
void main() {
    Integer x = 1;
    System.out.println(x);
    if (true) {
        Integer x = 2;
        System.out.println(x);
    }
    System.out.println(x);
}
```

Output: «1», «2», «1»

JavaScript, explicit hoisting:

```
function main() {
   var x;
   var x;
   x = 1;
   console.log(x);
   if (true) {
      x = 2;
      console.log(x);
   }
   console.log(x);
}
```

Output: «1», «2», «2»!

- JavaScript has blocks, but (traditionally) not block scope!
- Declarations are always «hoisted» to the top of the function

More fun: scoping and blocks

Java:

```
void main() {
    Integer x = 1;
    System.out.println(x);
    if (true) {
        Integer x = 2;
        System.out.println(x);
    }
    System.out.println(x);
}
```

Output: «1», «2», «1»

JavaScript/*EcmaScript 6*+:

```
function main() {
    let x = 1;
    console.log(x);
    if (true) {
        let x = 2;
        console.log(x);
    }
    console.log(x);
}
```

Output: «1», «2», «1»!

- EcmaScript 6 has blocks *and* block scope, if you use ”let”!

Upcoming

- *Fun*ctional programming with SML/Haskell
- Types and object orientation lecture part II
- Summary

