INF3110 – Exam from 2015 Exercise 1, Runtime systems, scoping and types (40 %)

Eyvind W. Axelsen

eyvinda@ifi.uio.no | @eyvindwa

http://eyvinda.at.ifi.uio.no

Exercise 1a – Consider the following stack implementation

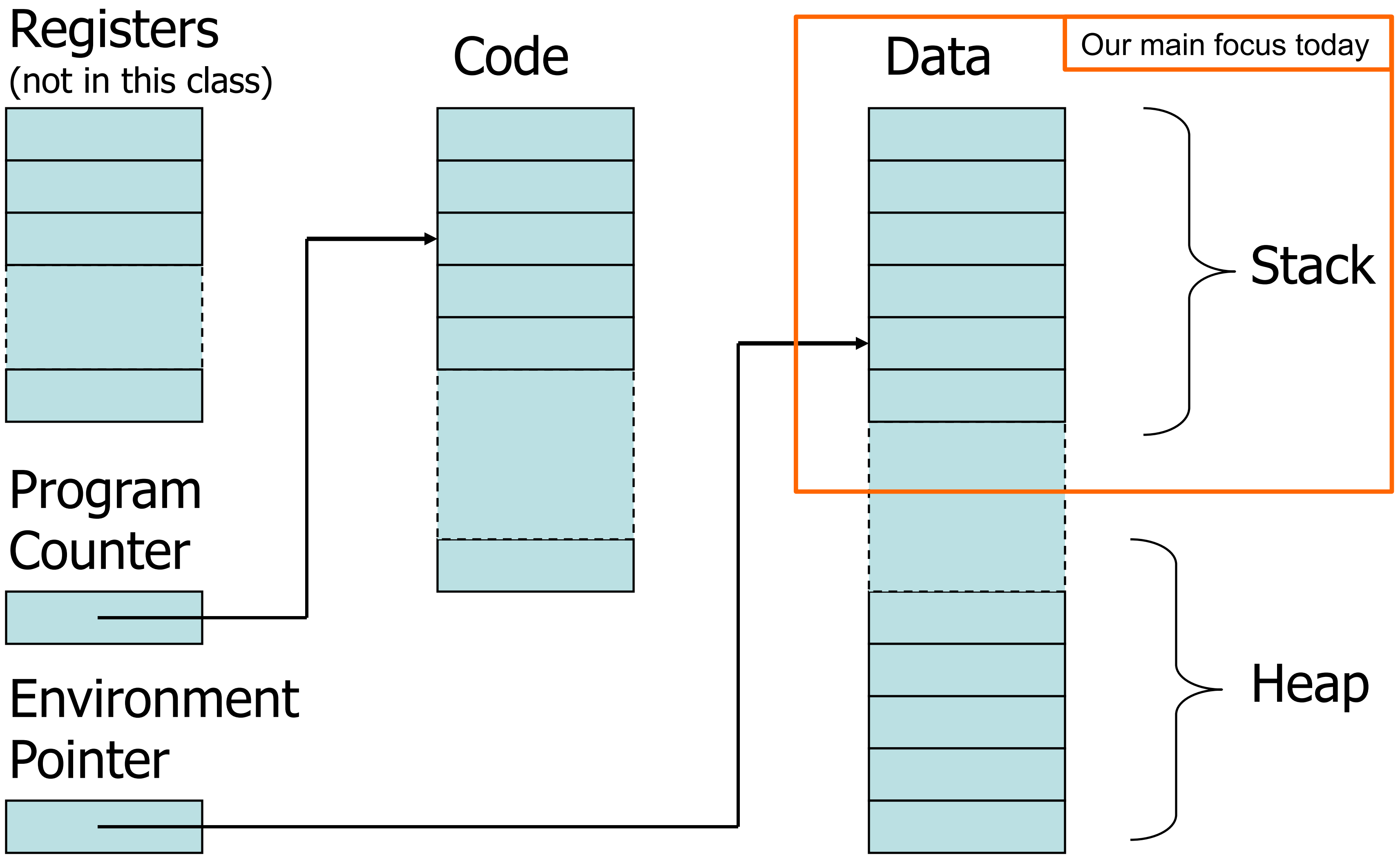
```
public class MyStack<T> {
   int maxSize;
   Object[] stackArray;
   int top = -1;

   public MyStack(int maxSize) {
      this.maxSize = maxSize;
      stackArray = new Object[maxSize];
   }

   public void push(T element) {
      stackArray[++top] = element;
      // HERE, see text below
   }
   public T pop() {
      return (T) stackArray[top--];
   }
   public boolean isEmpty() {
      return top < 0
   }

}
```

```
class Person {
   public String name;
   public Person(String name) { this.name = name; }
   /* some more content here */
}

class Student extends Person {
   public Student(String name) { super(name); }
   /* some more content here */
}
```

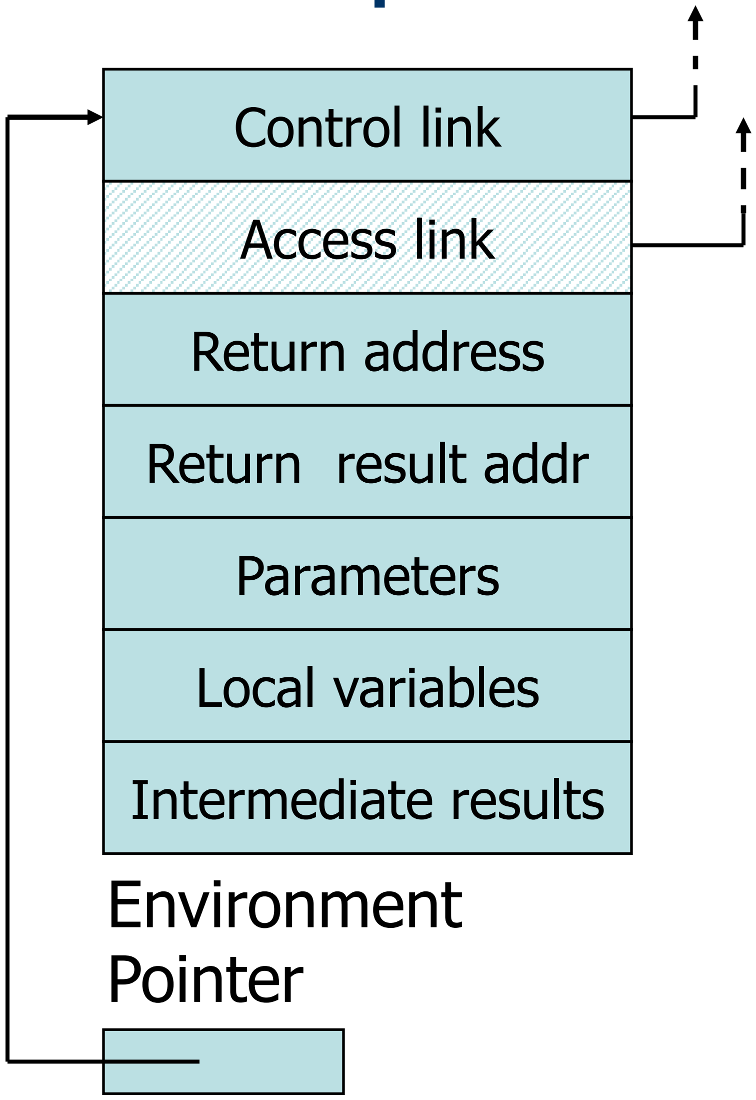
```
class Program {
   public static void main(String[] args) {
      MyStack<Student> s = new MyStack<Student>(10);
      s.push(new Student("Volker"));
      s.push(new Student("Eyvind"));
   }

}
```

Draw the runtime stack with activation blocks and objects (including static and dynamic links, using `this` for static links to objects, and local variables), at the point when the call to `s.push(new Student("Eyvind"))` has just been made, and the execution is at the point labeled "// `HERE`" in the code. You may assume that arrays are implemented as objects with an appropriate number of slots for their elements.

Lecture Runtime org. 1 - Simplified Reference Model of a Machine - used to understand memory management



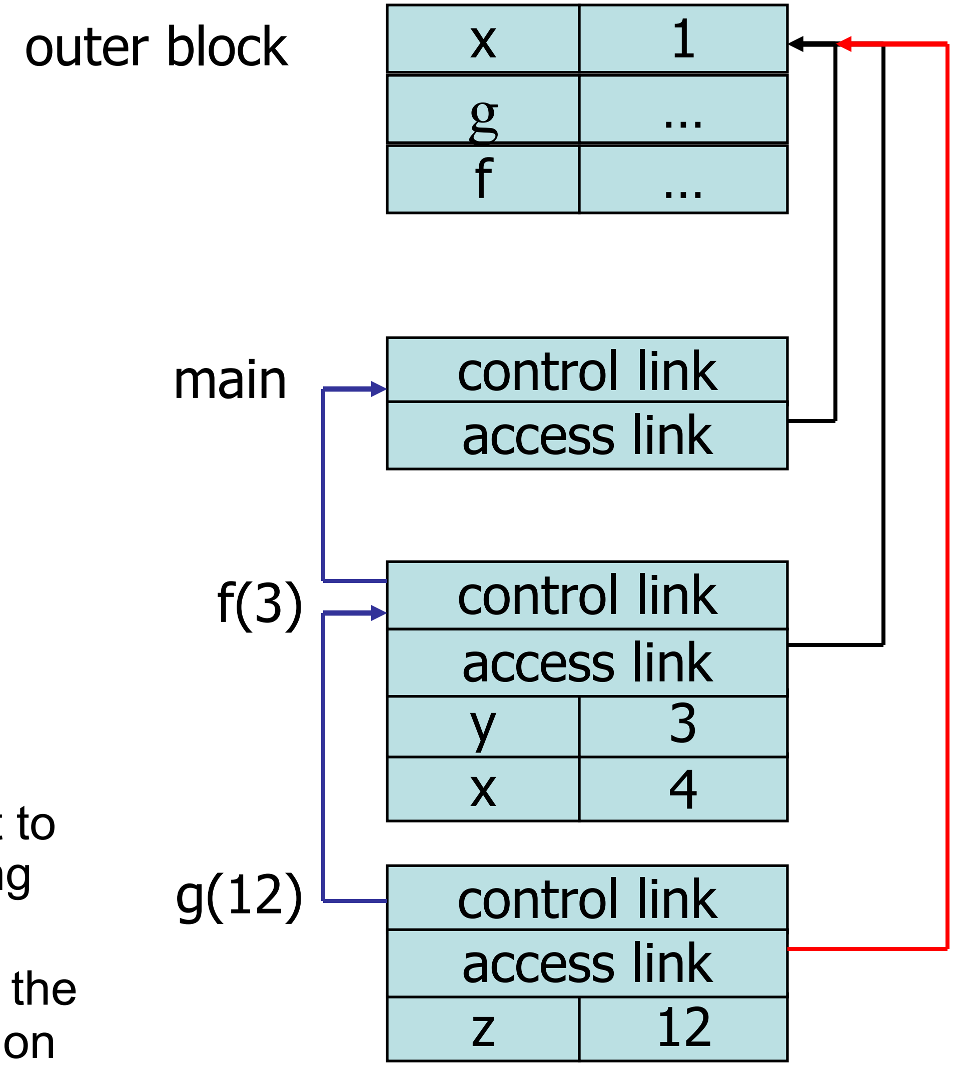
Lecture Runtime org. 1 - Activation record for static scope



- Control link (dynamic link)
  - Link to activation record of previous (calling) block
- 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

Lecture Runtime org. 1 - Static scope with access links (C-like notation)

```
{
  int x = 1;

  int function g(z) { return x+z };

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

  main() {
    f(3);
  }
}
```



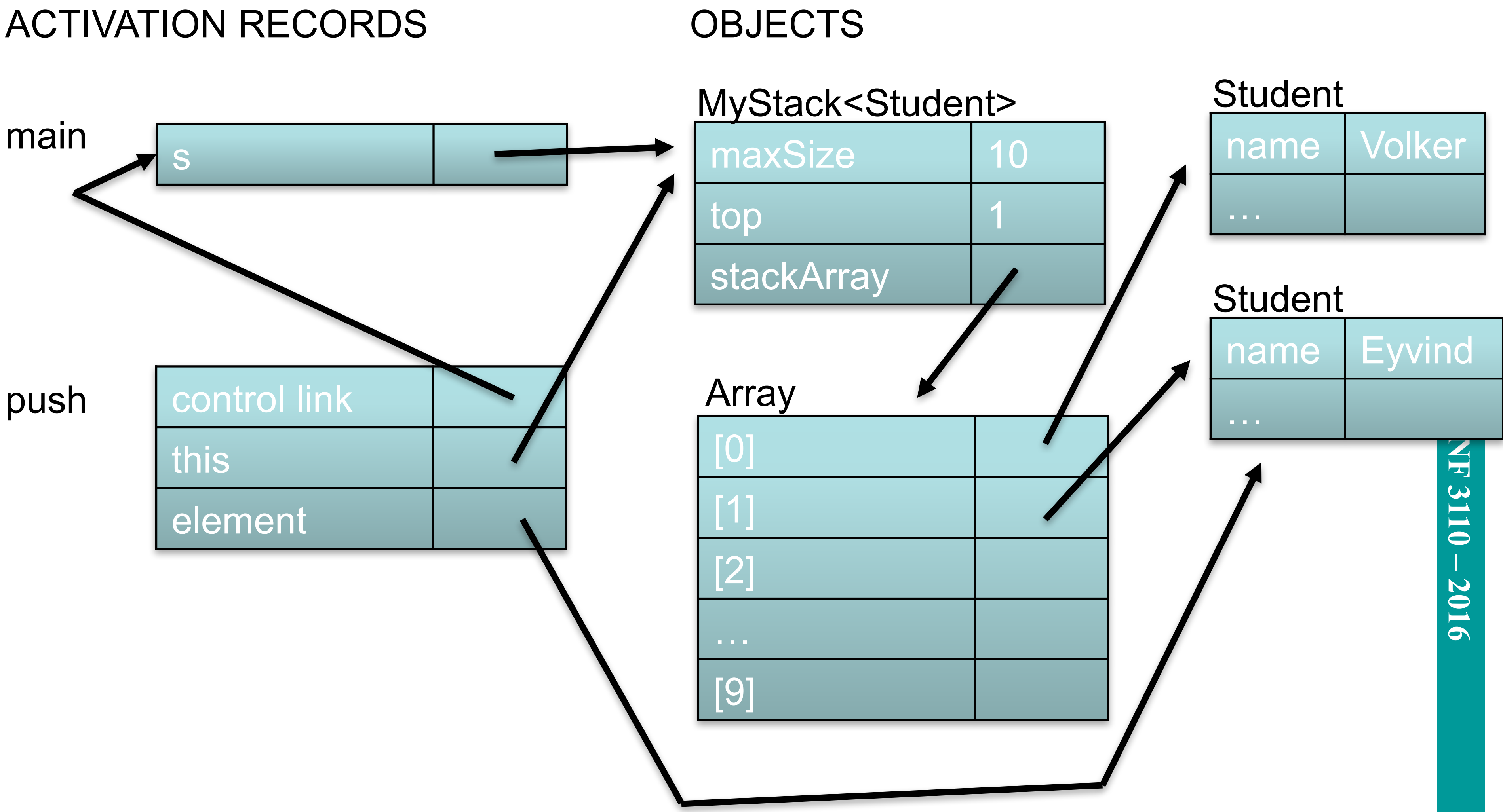
Use access link to find global variable:

- Access link is always set to frame of closest enclosing *lexical* block
- For function body, this is the block that contains function declaration

```java
public class MyStack<T> {
    int maxSize;
    Object[] stackArray;
    int top = -1;

    public MyStack(int maxSize) {
        this.maxSize = maxSize;
        stackArray = new Object[maxSize];
    }

    public void push(T element) {
        stackArray[++top] = element;
        // HERE, see text below
    }
    public T pop() {
        return (T) stackArray[top--];
    }
    public boolean isEmpty() {
        return top < 0
    }
}
```

```java
class Person {
    public String name;
    public Person(String name) {
this.name = name; }
    /* some more content here */
}

class Student extends Person {
    public Student(String name)
{ super(name); }
    /* some more content here */
}
```

```java
class Program {
    public static void
main(String[] args) {
        MyStack<Student> s = new
MyStack<Student>(10);
        s.push(new
Student("Volker"));
        s.push(new
Student("Eyvind"));
    }
}
```



Draw the runtime stack with activation blocks and objects (including static and dynamic links, using `this` for static links to objects, and local variables), at the point when the call to `s.push(new Student("Eyvind"))` has just been made, and the execution is at the point labeled "// `HERE`" in the code. You may assume that arrays are implemented as objects with an appropriate number of slots for their elements.

1b

Consider now the following program fragment, which uses the stack implementation from **1a**:

```
MyStack<Object> myStack = new MyStack<Object>(10);
myStack.push("Hello INF3110!");
myStack.push(new Object());
myStack.push(123);
```

Does this program fragment work (i.e., does it compile and run without any errors, provided it is wrapped in a suitable method and class declaration)? If yes, explain briefly how and why it works. If not, explain briefly what is wrong with it.

Solution: the stack is generic, but the generic type is Object, which means that you can put anything that extends Object into it. That works for strings and new Object(). The int 123 will be boxed into a new Integer, and thus that is OK too.

1c

Suppose now that we replace the first line of the program fragment from **1b** with the following code:

```
MyStack<Object> myStack = new MyStack<String>(10);
```

Explain how the fragment now differs from the one in **1b**. Will the compiler react differently to it? If it compiles correctly, will the runtime behavior of the fragment be different?

```
MyStack<Object> myStack = new MyStack<String>(10);
myStack.push("Hello INF3110!");
myStack.push(new Object());
myStack.push(123);
```

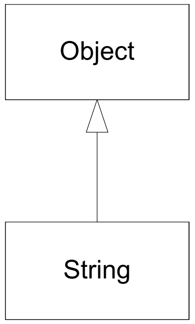
Lecture OO II - Generics and subtyping

- String subtype of Object $\cancel{\Rightarrow}$ List<String> subtype of List<Object> ?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Object());
String s = ls.get(0);
```

compile-time error

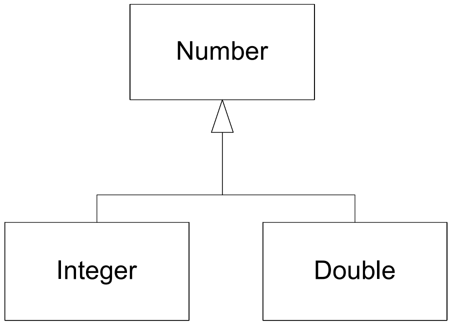
attempts to assign an Object to a String



- Integer subtype of Number $\cancel{\Rightarrow}$ List<Integer> subtype of List<Number> ?

```
List<Integer> ints = Arrays.asList(1,2);
List<Number> nums = ints;
nums.add(3.14);
```

compile-time error



1c

Suppose now that we replace the first line of the program fragment from **1b** with the following code:

```
MyStack<Object> myStack = new MyStack<String>(10);
```

Explain how the fragment now differs from the one in **1b**. Will the compiler react differently to it? If it compiles correctly, will the runtime behavior of the fragment be different?

```
MyStack<Object> myStack = new MyStack<String>(10);
myStack.push("Hello INF3110!");
myStack.push(new Object());
myStack.push(123);
```

Solution: This is unsafe, and it will not compile.

MyStack<String> is **not** a subtype of MyStack<Object>, even though String is a subtype of Object.

1d

Returning now to persons and students, assume that we want to write a method that can process stacks of persons and stacks of students, e.g. like this:

```
public static < … > void processPersons(... persons) {
   while(!persons.isEmpty()) {
       System.out.println(persons.pop().name);
   }

}
```

Replace the ellipses (…) two places in the method signature above with the appropriate generic declarations to make the following calls to `processPersons` work:

```
MyStack<Person> persons =    // some initialization code here,
MyStack<Student> students = // you do not need to provide this code

processPersons(persons);  // this call and the next should work
processPersons(students);
```

Lecture OO II - Bounded polymorhpism - Wildcards - II

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}

public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) { ... }
}

public class Canvas {
  public void draw(Shape s) { s.draw(this);}
}
```

Write code to draw a list of any kind of shape →

Lecture OO II - Bounded polymorhpism - Wildcards - III

```
// in class Canvas:
public void drawAll(List<Shape> shapes) {
  for (Shape s: shapes)
     s.draw(this);
}


public void drawAll(List<? extends Shape> shapes) {
  for (Shape s: shapes)
     s.draw(this);
}
```

- `List<S>` subtype of `List<? extends Shape >` for every `S` being a subtype of the (concrete) type `Shape`

- `List<S>` subtype of `List<? extends T >` for every `S` being a subtype of (the generic parameter) `T`

1d

Returning now to persons and students, assume that we want to write a method that can process stacks of persons and stacks of students, e.g. like this:

```
public static < … > void processPersons(... persons) {
   while(!persons.isEmpty()) {
       System.out.println(persons.pop().name);
   }
}
```

Replace the ellipses (…) two places in the method signature above with the appropriate generic declarations to make the following calls to `processPersons` work:

```
MyStack<Person> persons =    // some initialization code here,
MyStack<Student> students = // you do not need to provide this code

processPersons(persons);  // this call and the next should work
processPersons(students);
```

Solution:

```
public static <T extends Person> void processPersons(MyStack<T> persons)
{
    while(!persons.isEmpty()) {
         System.out.println(persons.pop().name);

} } }
```

1e Java 8 allows the usage of anonymous functions (or “lambdas”). For instance, we could define `processPersons` to have an argument that is a predicate filtering which person’s `name` to print:

```
public static void processPersons(Stack<Person> persons, Predicate<Person>
predicate) {
   while(!persons.isEmpty()) {
       Person person = persons.pop();
       if(predicate.test(person))
           System.out.println(person.name); // HERE
}}
```

`Predicate<T>`, as used in the method above, is a built-in interface in Java 8 that has a Boolean method `test` that can be implemented by an anonymous function. Thus, the `processPersons` method could now be called for instance like the following, to only print “`Volker`”:

```
public static void main(String[] args) {
   MyStack<Person> persons = new MyStack<Person>(10);
   persons.push(new Student("Volker"));
   persons.push(new Student("Eyvind"));
   String filter = "V";

   Predicate<Person> predicate = p -> p.name.startsWith(filter);
   processPersons(persons, predicate);
}
```

Draw the call stack as it is in the call to `processPersons` when the execution has reached the point marked “// `HERE`” in the code above.

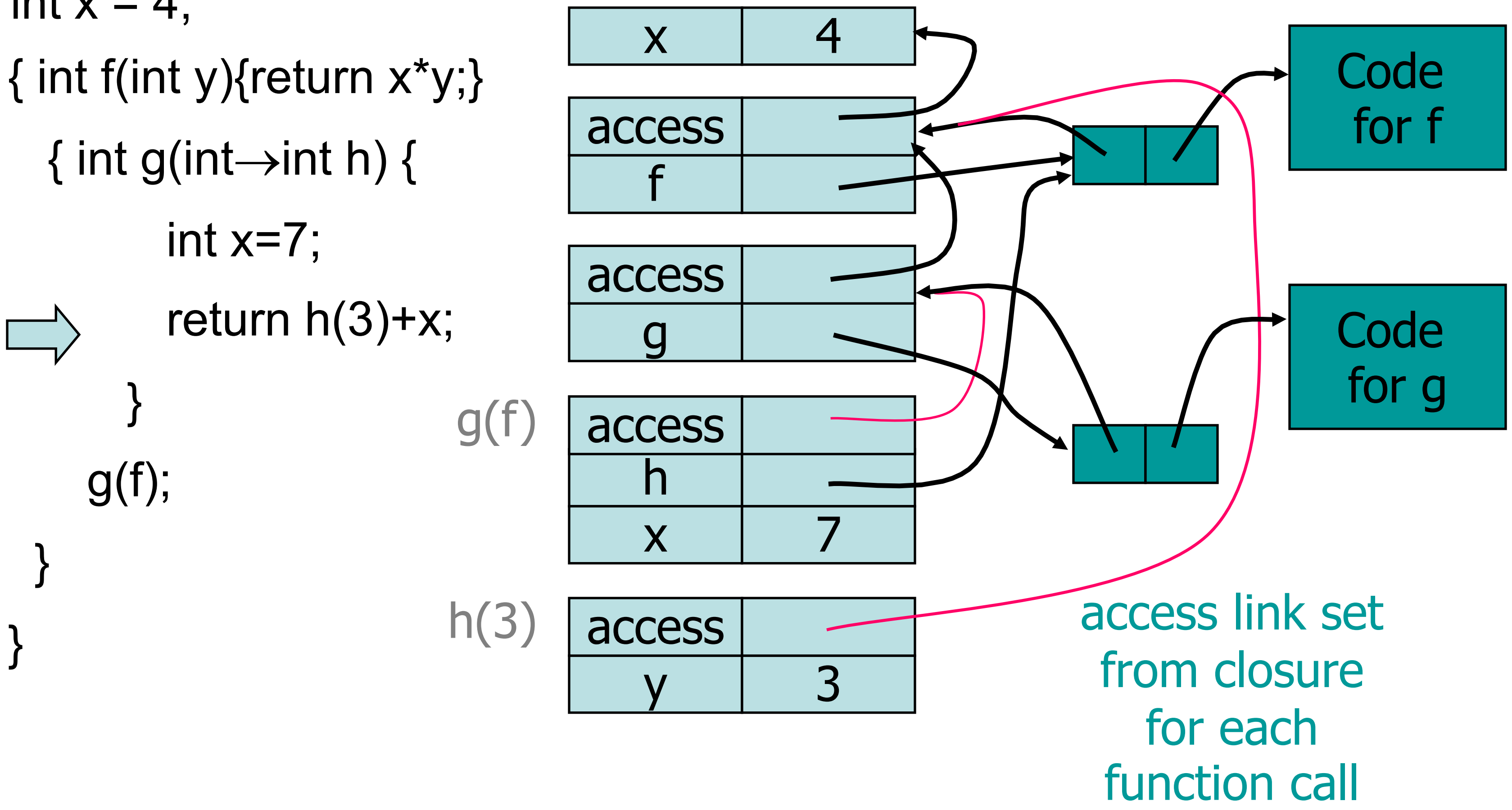
Lecture OO II - Closures

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

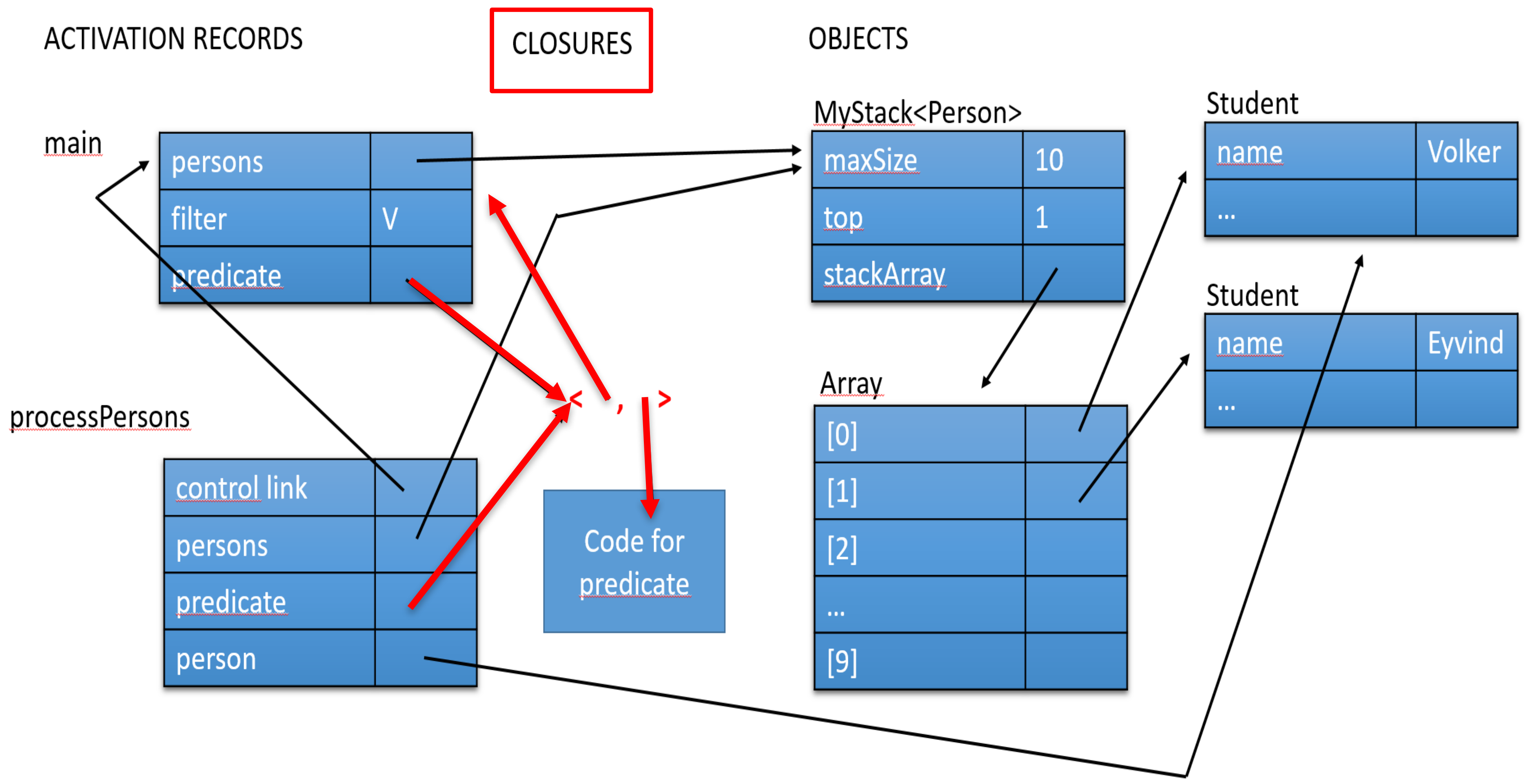
Lecture OO II - Function Argument and Closures

Run-time stack with access links

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



```
public static void processPersons(Stack<Person> persons,
Predicate<Person> predicate) {
   while(!persons.isEmpty()) {
       Person person = persons.pop();
       if(predicate.test(person))
           System.out.println(person.name); // Draw stack here
}}


public static void main(String[] args) {
   MyStack<Person> persons = new MyStack<Person>(10);
   persons.push(new Student("Volker"));
   persons.push(new Student("Eyvind"));
   String filter = "V";

   Predicate<Person> predicate = p-> p.name.startsWith(filter);
   processPersons(persons, predicate);
}
```


ACTIVATION RECORDS
CLOSURES
OBJECTS
main
persons
filter
V
predicate
processPersons
control link
persons
predicate
person
< , >
Code for predicate
MyStack<Person>
maxSize
10
top
1
stackArray
Array
[0]
[1]
[2]
…
[9]
Student
name
Volker
…
Student
name
Eyvind
…

Summing up

- The upcoming exam:
  - December 12, 14:30 (4 hours).
  - All written and printed material allowed
- Some main topics of today's (2015) exam task
  - Runtime stacks and activation records
  - Generics, variance
  - Functions/methods as parameters
- Remember: there are more topics in this course!
  - Syntax/semantics
  - Types and scopes
  - Object orientation
  - And of course SML, Prolog and all the rest of Volker's lectures
  - More exams (with solutions) on the course page: http://www.uio.no/studier/emner/matnat/ifi/INF3110/h16/undervisningsmateriale/
- Thank you, and GOOD LUCK!