Structured Programming

COMP1110/6710

Image Courtesy NASA/JPL-Caltech.

Australian National University

Revisiting min-difference

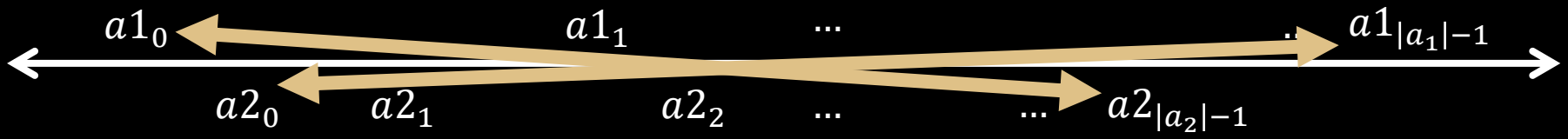
[Distinction-Level Content]

Australian National University

Two-list max-difference

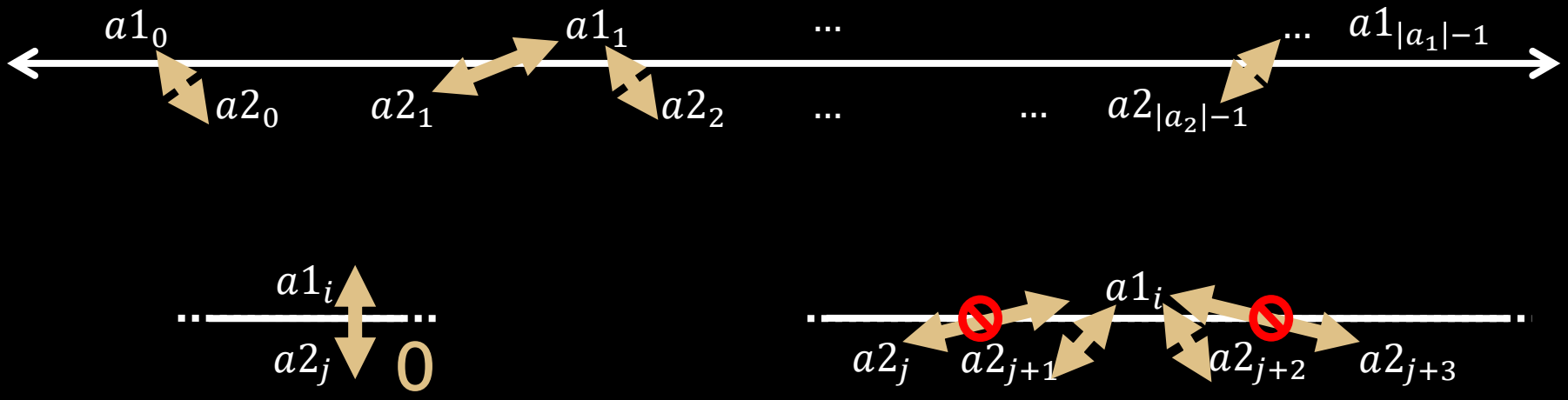
```
int maxDifference(int[] a1, int[] a2) {
    Arrays.sort(a1);
    Arrays.sort(a2);
    return Math.max(Math.abs(a1[0]-a2[a2.length-1]),
                    Math.abs(a2[0]-a1[a1.length-1]));
}
```

Two-list max-difference




```
int maxDifference(int[] a1, int[] a2) {
    Arrays.sort(a1);
    Arrays.sort(a2);
    return Math.max(Math.abs(a1[0]-a2[a2.length-1]),
                    Math.abs(a2[0]-a1[a1.length-1]));
}
```

Two-list min-difference



Code in demonstrations-repo

min-difference (Exercise)

Back to the original problem:
You have one list, and want to know the minimum difference between any two elements in it.

- Can you use the two-list version?
- How would a specialized one-list version work?

Today's Theme: Stack vs. Heap

Also: more on Types

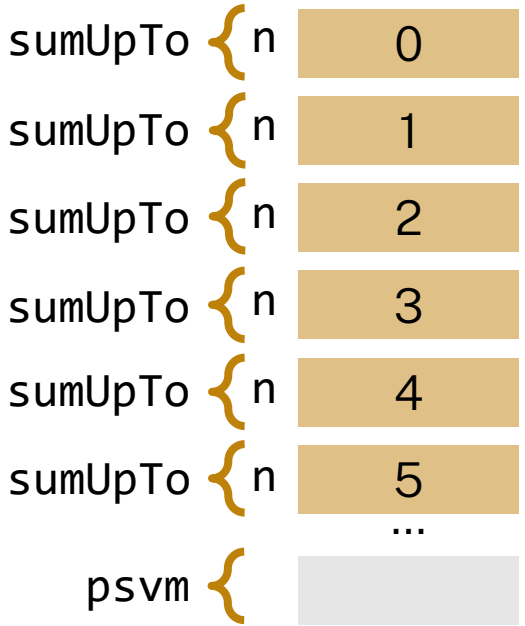
Australian National University

Recall: Stack & Heap

Stack

- A data structure where you can only remove/add things at the top

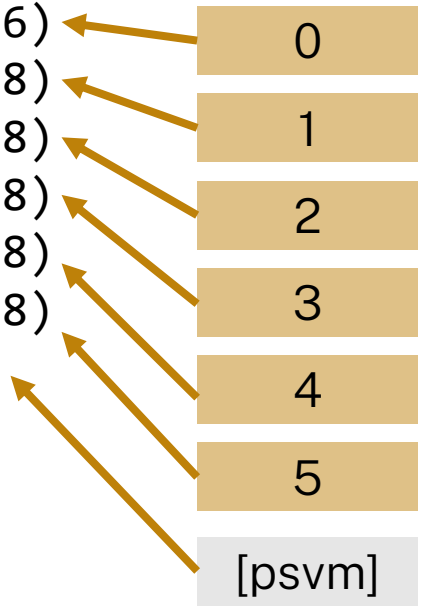
```
public static int sumUpTo(int n) {
  if(n<=0) { return n/n; }
  else {
    return n + sumUpTo(n-1);
  }
}
```



Recall: Stack & Heap

A Stack Trace

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at ws9a.SumUpToBugged.sumUpTo(SumUpToBugged.java:6)
  at ws9a.SumUpToBugged.sumUpTo(SumUpToBugged.java:8)
  at ws9a.SumUpToBugged.sumUpTo(SumUpToBugged.java:8)
  at ws9a.SumUpToBugged.sumUpTo(SumUpToBugged.java:8)
  at ws9a.SumUpToBugged.sumUpTo(SumUpToBugged.java:8)
  at ws9a.SumUpToBugged.sumUpTo(SumUpToBugged.java:8)
  at ws9a.SumUpToBugged.main(SumUpToBugged.java:13)

Process finished with exit code 1
```



Recall: Stack & Heap

Values on the Stack

n | 0

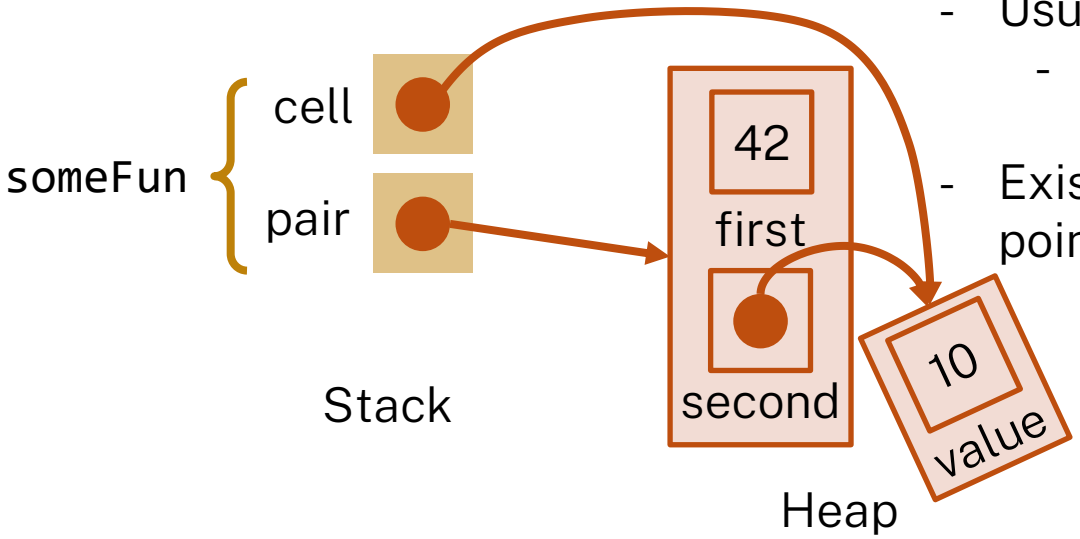
These boxes can contain one of:

- A primitive value (of a primitive type)
  - boolean
  - byte
  - char
  - short
  - int
  - long
  - float
  - double
- A pointer to an object on the Heap (of a reference type)

Boxes (and values in them) only exist for duration of method call.

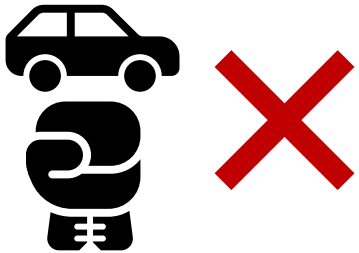
Recall: Stack & Heap

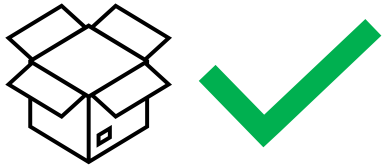
Heap Values



- Bigger boxes that can contain multiple smaller boxes
- Arrays or regular Objects
  - Both inherit from Object anyway
- Usually created with “new”
  - But also String constants, autoboxing (next)
- Exist as long as something points to them

Autoboxing





Recall: Type Erasure

```
public class Cell<T> {
    T value;
}
```

is really

```
public class Cell {
    Object value;
}
```

This means `T` has to be a reference type – primitive types don't have methods, including `equals` and `toString`

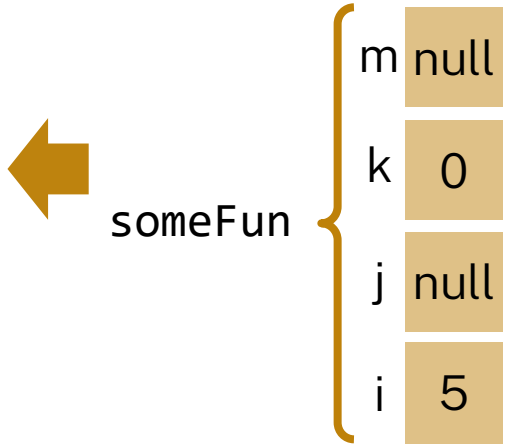
Cell<Integer> ✓

Cell<int> ✗

Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

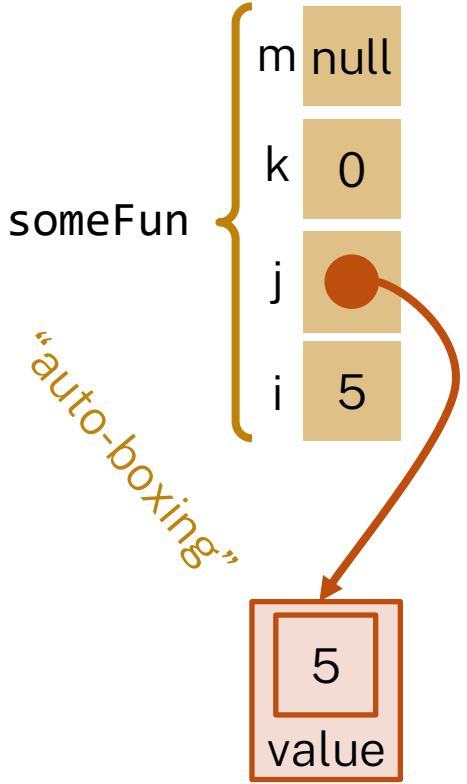
```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

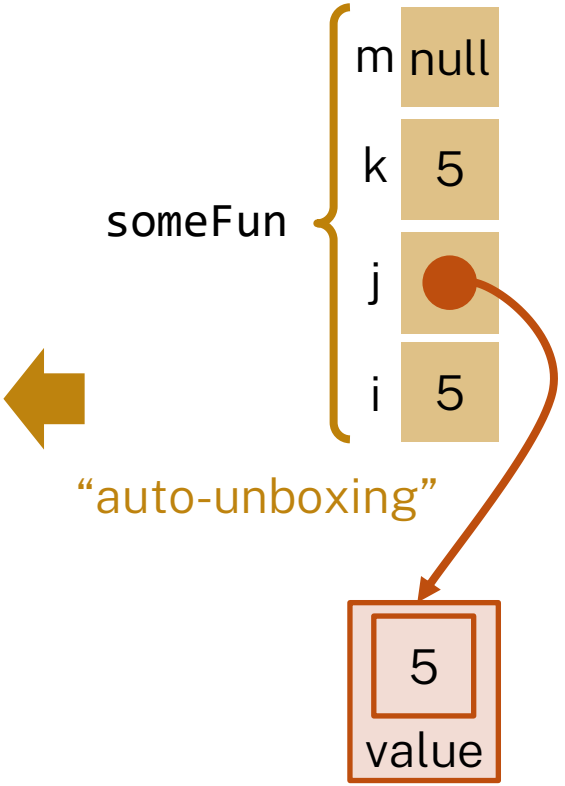
```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

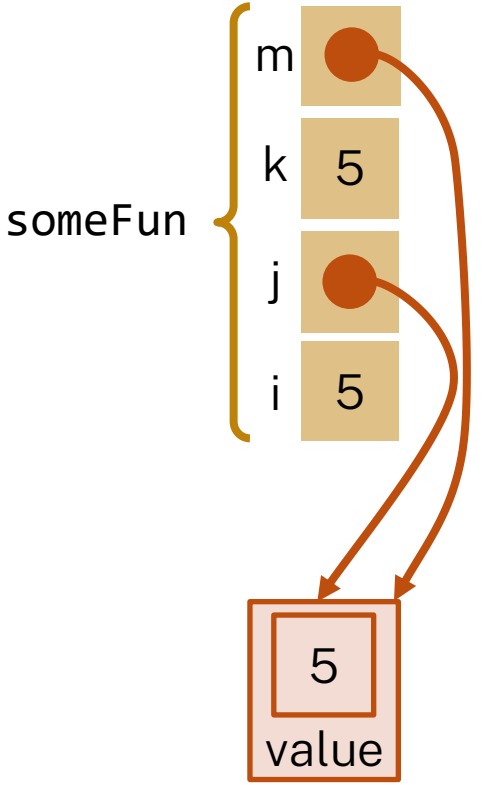
```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

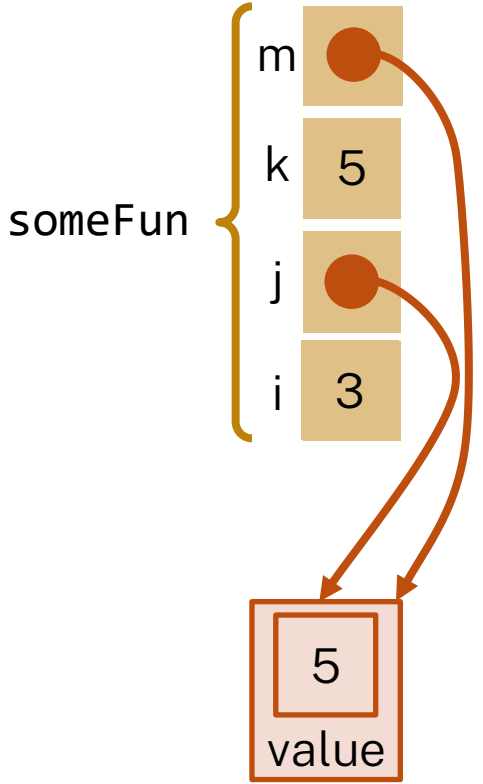
```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

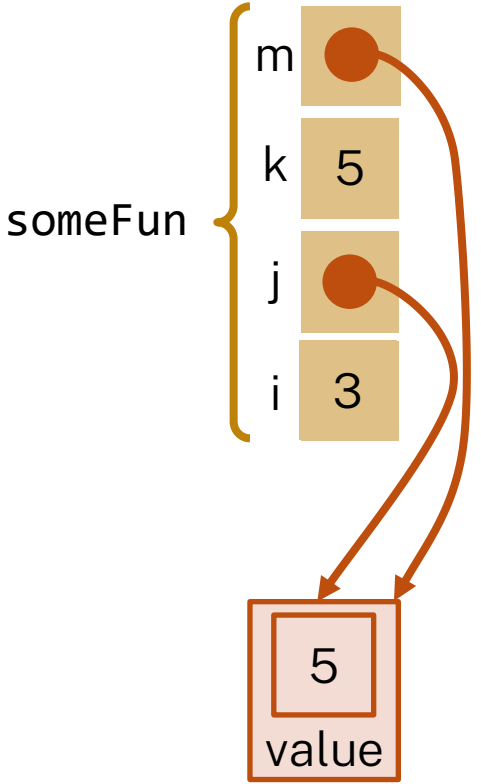
```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```

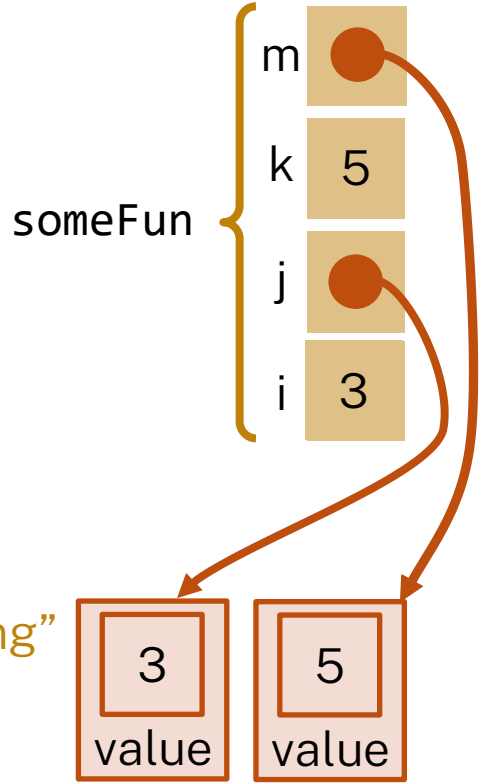


Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```

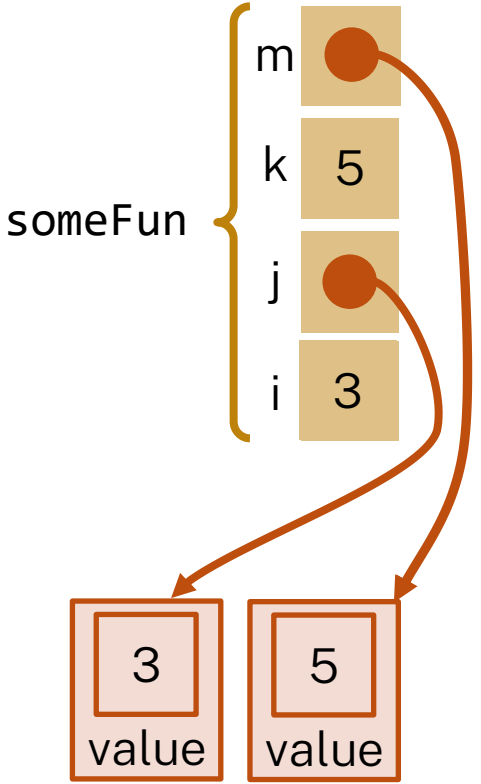
“auto-boxing”



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

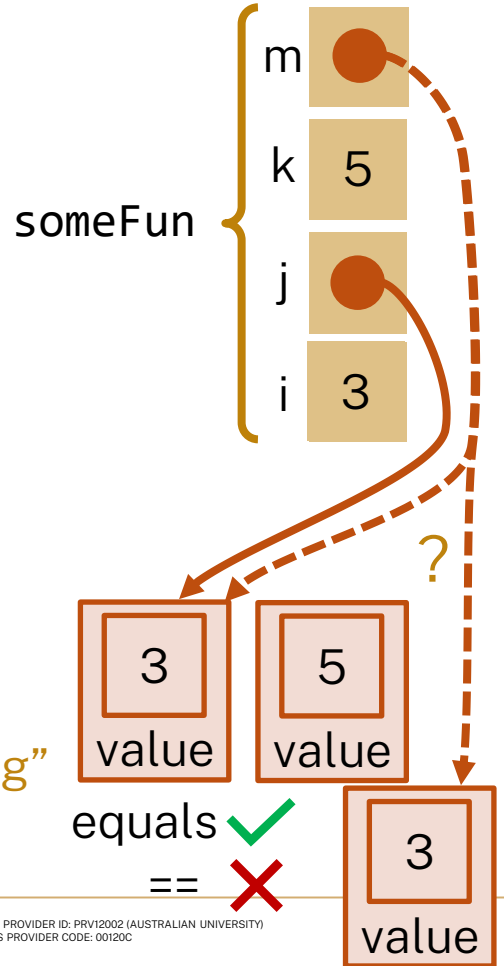
```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Wrapper Types

Primitive Type	Reference Type
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int i = 5;

Integer j = i;

int k = j;

Integer m = j;

i = 3;

j = i;

m = i;
```



Autoboxing - Summary

Boxing – creating a heap-value that stores a primitive value

Unboxing – retrieving a primitive value from a wrapper value

Auto-Boxing – the idea that Java does this for you automatically

Just because its automated does not mean you don’t have to be aware, see: equals vs ==

Lambdas & Functional Interfaces

Or: why there is a difference between `.apply` and `.test`

Australian National University

Function Types

What we've seen so far

"Functional Interfaces"

- `Function<T,R>`
- `BiFunction<T,U,R>`
- `Predicate<T>`
- `BiPredicate<T,U>`

```
public interface Function<T,R> {
   R apply(T t);
}

public interface BiFunction<T,U,R> {
   R apply(T t, U u);
}

public interface Predicate<T> {
   boolean test(T t);
}
```

Functional Interfaces

Are interfaces with a single abstract instance method (i.e. not default, not static).

Java comes with more than those four we saw (see java.util.function), and you can write your own.

Why `Predicate<T>` and not `Function<T, Boolean>` ?

➔ Boxing: `Predicate<T>.test` has return type `boolean`

```
public interface Function<T,R> {
   R apply(T t);
}
```

```
public interface Predicate<T> {
   boolean test(T t);
}
```

Functional Interfaces

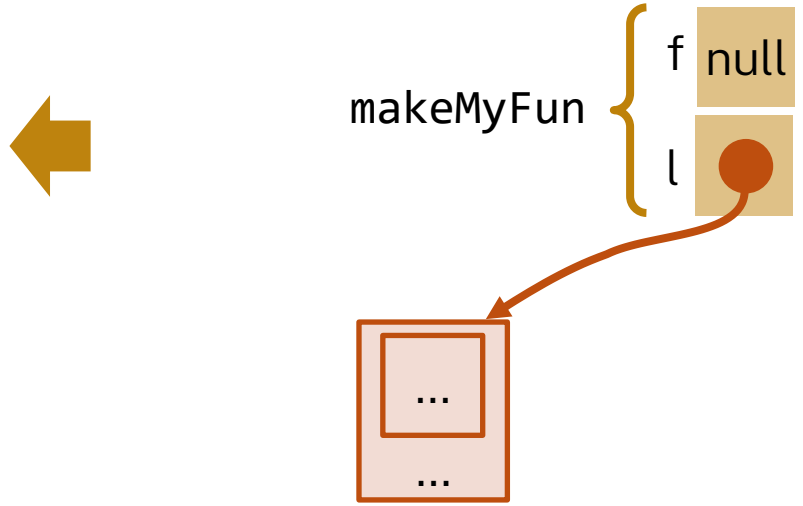
```
public interface MyFun<T> {
  List<T> append(T element);
}

…
List<String> foo(MyFun<String> fun) {
  return fun.append(“Hello!”);
}
```


Functional Interfaces & Closures

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
List<String> foo(MyFun<String> fun) {
  return fun.append("Hello!");
}
```

```
public MyFun<String> makeMyFun() {
    List<String> l = new ArrayList<>();
    MyFun<String> f = str -> {
      l.add(str);
      return l;
    };
    return f;
}
```

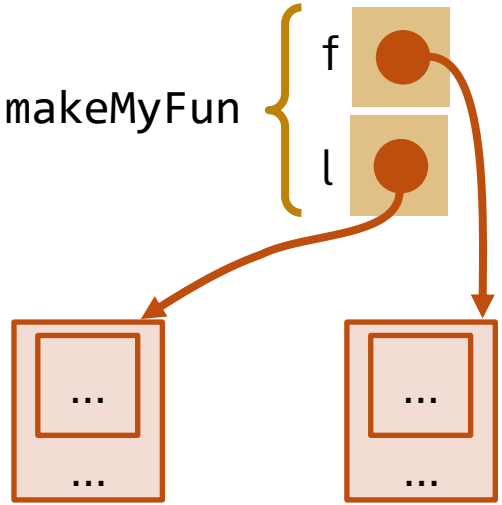


Functional Interfaces & Closures

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
public MyFun<String> makeMyFun() {
    List<String> l = new ArrayList<>();
    MyFun<String> f = str -> {
      l.add(str);
      return l;
    };
    return f;
}
```

```
List<String> foo(MyFun<String> fun) {
  return fun.append("Hello!");
}
```

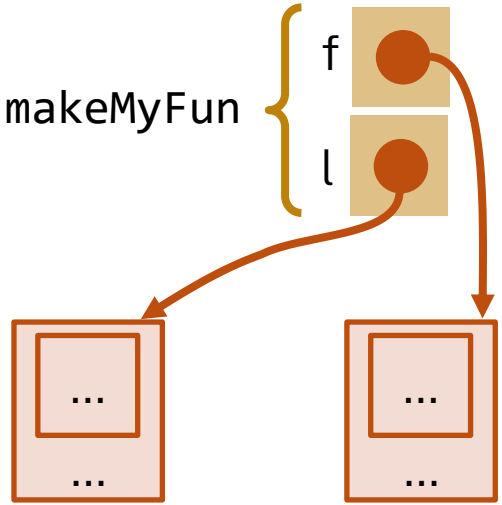


Functional Interfaces & Closures

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
public MyFun<String> makeMyFun() {
    List<String> l = new ArrayList<>();
    MyFun<String> f = str -> {
      l.add(str);
      return l;
    };
    return f;
}
```

```
List<String> foo(MyFun<String> fun) {
  return fun.append("Hello!");
}
```



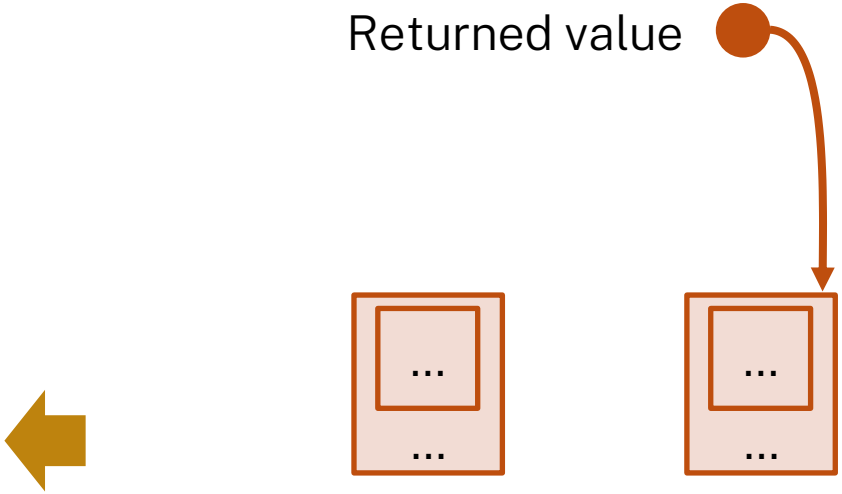
Functional Interfaces & Closures

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
List<String> foo(MyFun<String> fun) {
  return fun.append("Hello!");
}
```

```
public MyFun<String> makeMyFun() {
    List<String> l = new ArrayList<>();
    MyFun<String> f = str -> {
      l.add(str);
      return l;
    };
    return f;
}
```

How do we get to l?



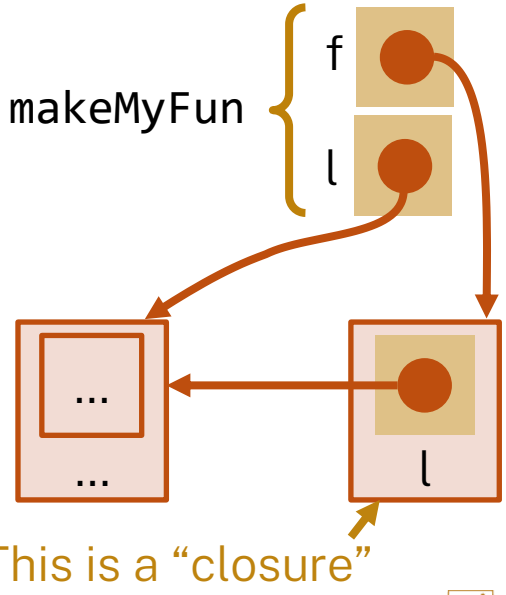
Functional Interfaces & Closures

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
List<String> foo(MyFun<String> fun) {
  return fun.append("Hello!");
}
```

```
public MyFun<String> makeMyFun() {
    List<String> l = new ArrayList<>();
    MyFun<String> f = str -> {
      l.add(str);
      return l;
    };
    return f;
}
```

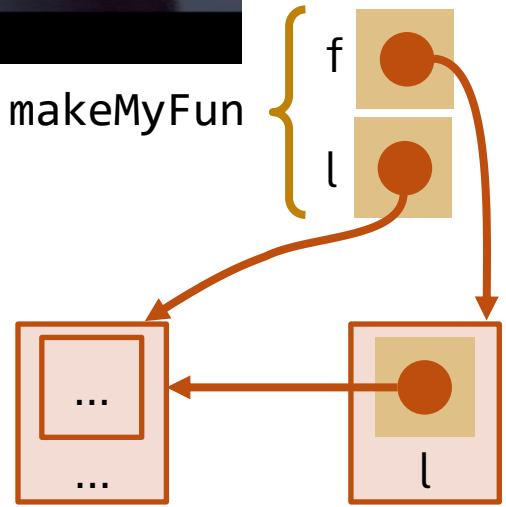
"Closures" capture part of the stack on the heap, to make it possible to use later.



Functional Interfa

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
fun) {
```



What we've said before:

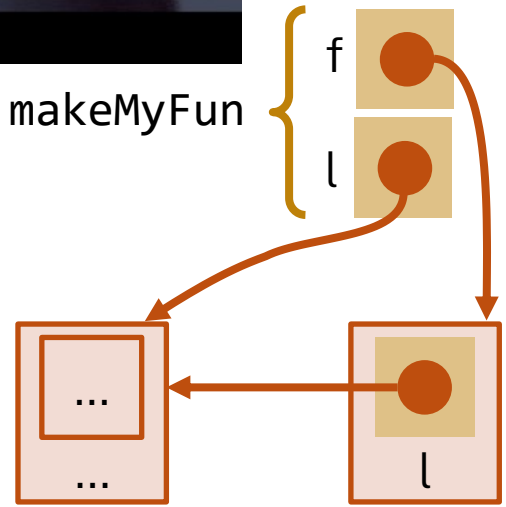
Imperative Programming

Each variable represents a single **slot** whose content can be accessed and changed throughout its scope

Functional Interfa

```
public interface MyFun<T> {
  List<T> append(T element);
}
```

```
fun) {
```



Java's Solution

When there are multiple slots representing the same variable, they need to be "final" or "effectively final", meaning you can only assign the variable once.

Closures - Summary

Closures capture part of the stack, by effectively copying the relevant boxes when the closure is created

To make this work out, the boxes need to be “read-only”, which Java calls “final” or “effectively” final.

In a wider sense, closures capture any sort of state apart from the arguments of a function ➔ Objects are big closures, too.