Software Verification (Autumn 2015) Lecture 17: Separation Logic for Object-Orientation

Chris Poskitt

Chair of Software Engineering

ETH zürich

(adapted from material by Stephan van Staden, Matthew Parkinson, and Gavin Bierman)

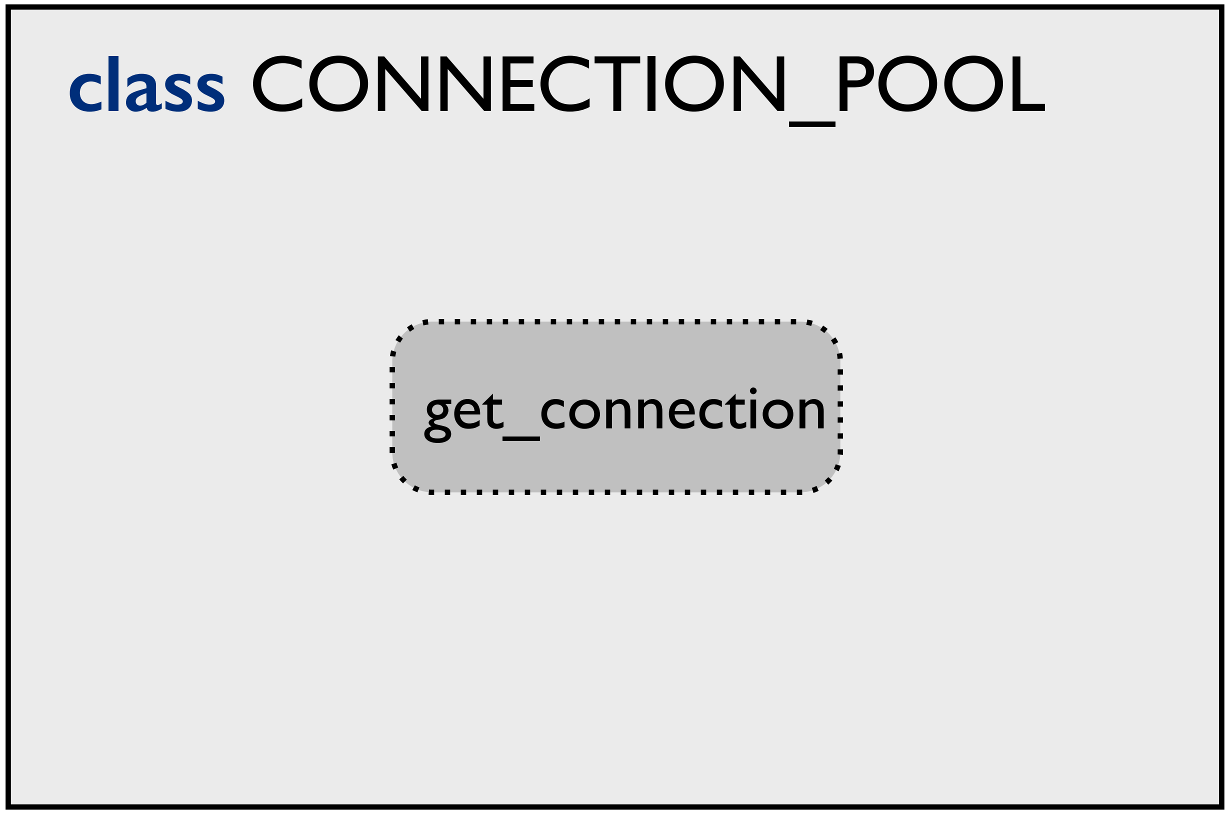
Verifying object-oriented programs

- object-oriented (O-O) languages are popular and widely used
  => *objects combine data with operations*
  => *clients don't need to know about internal representation*

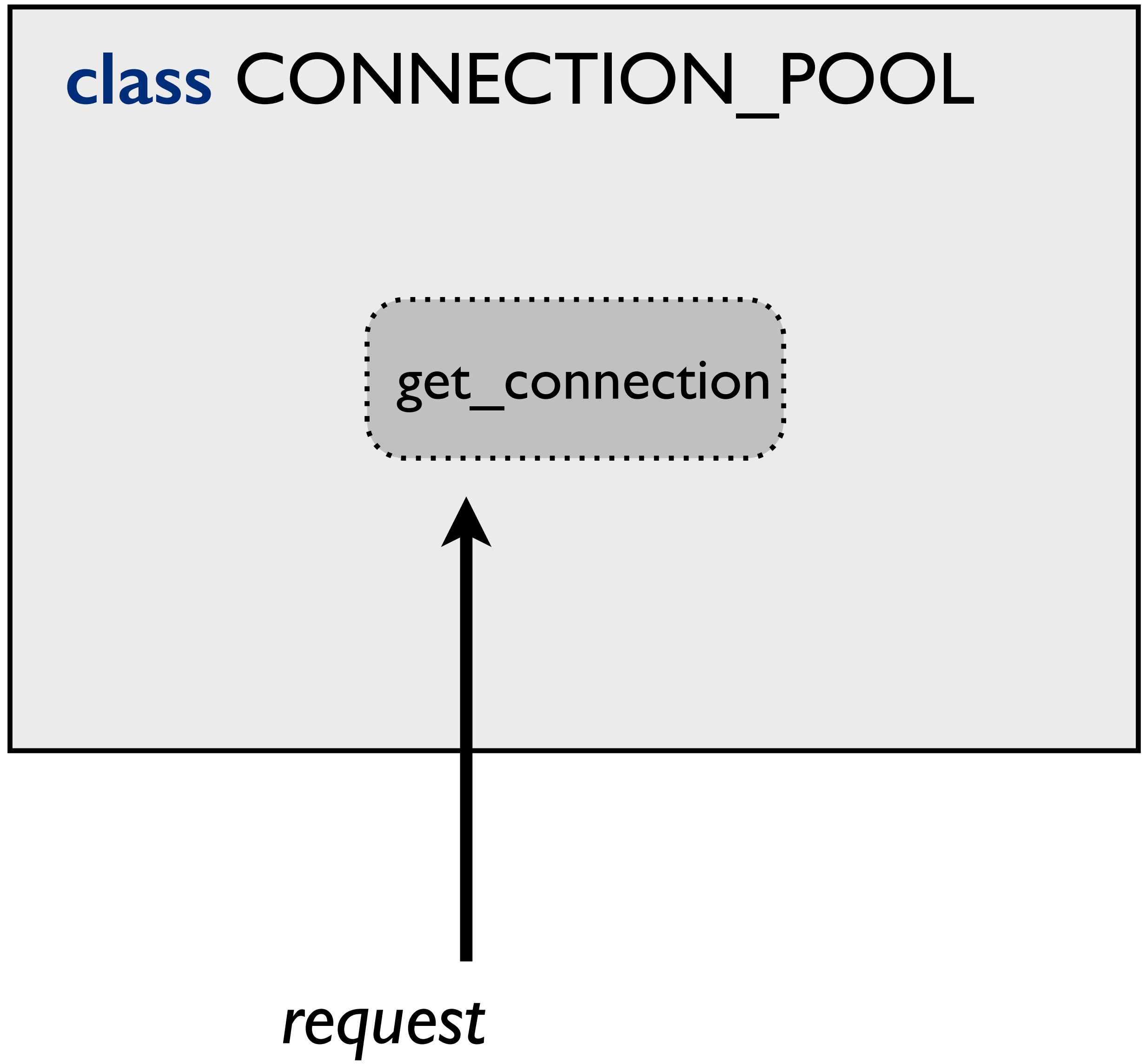
- encapsulation facilitates modular thinking

- reasoning about O-O programs is challenging
  => *shared mutable state*
  => *inheritance (i.e. subtyping and method overriding)*

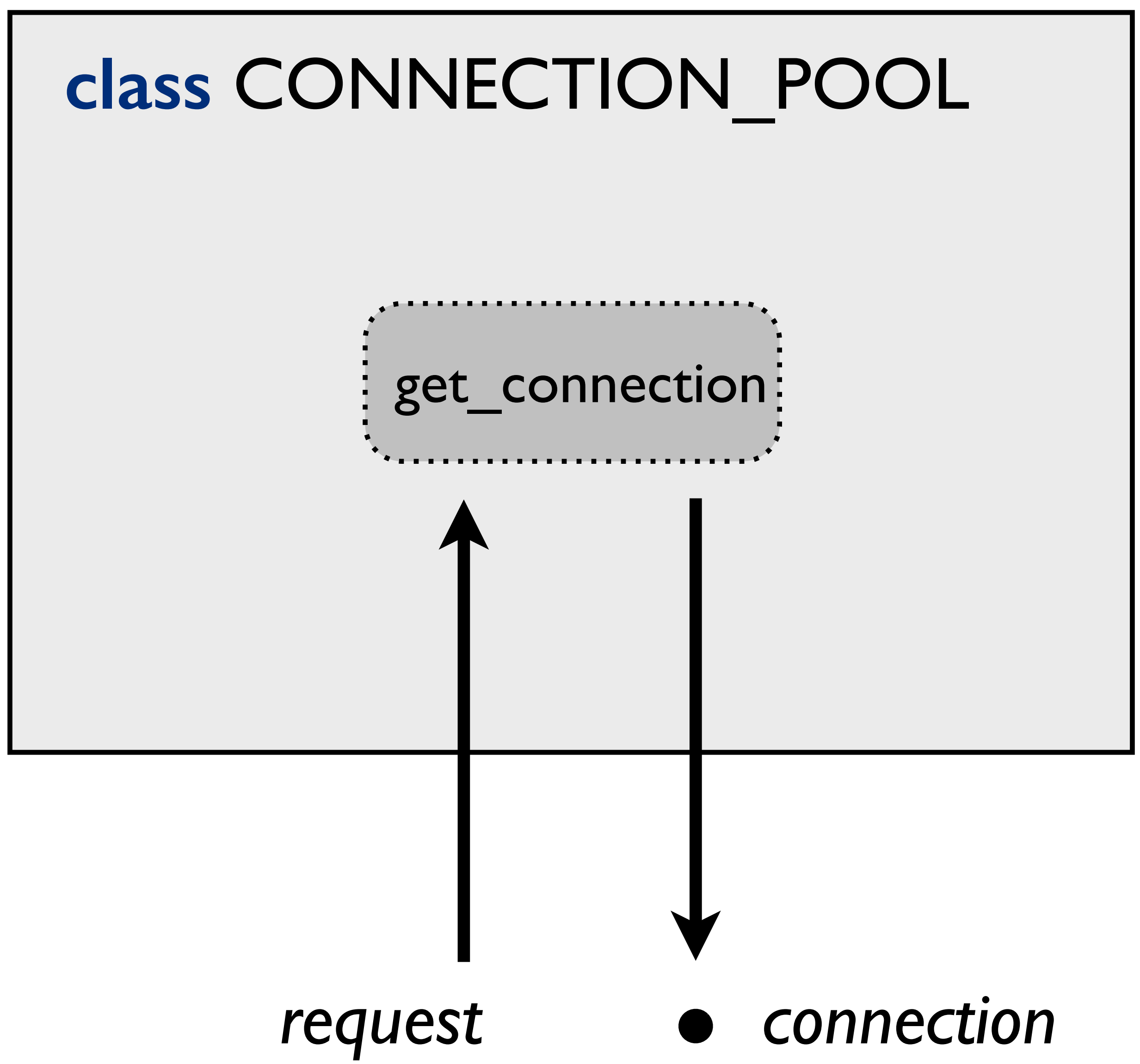
Shared mutable state



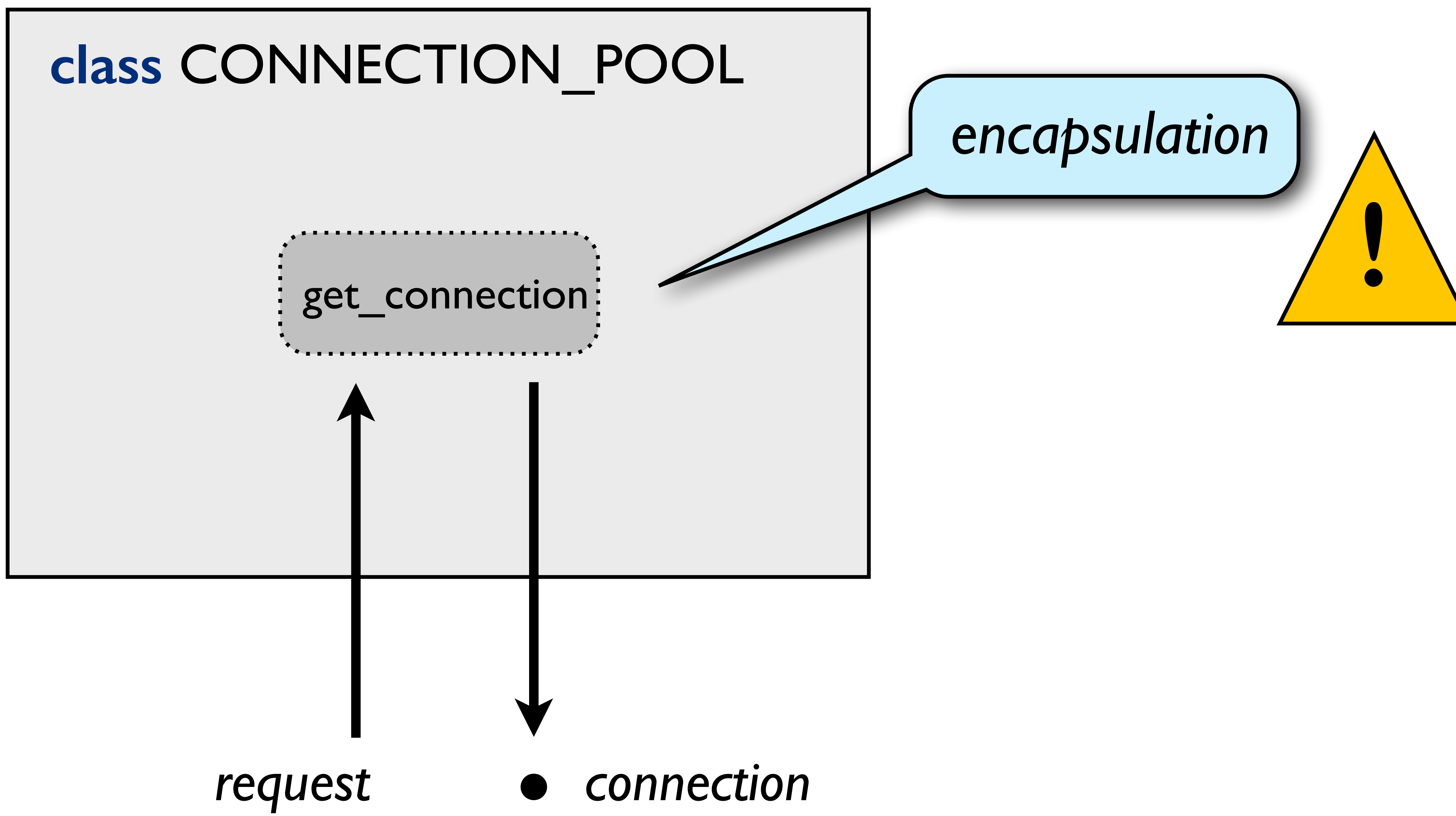
Shared mutable state



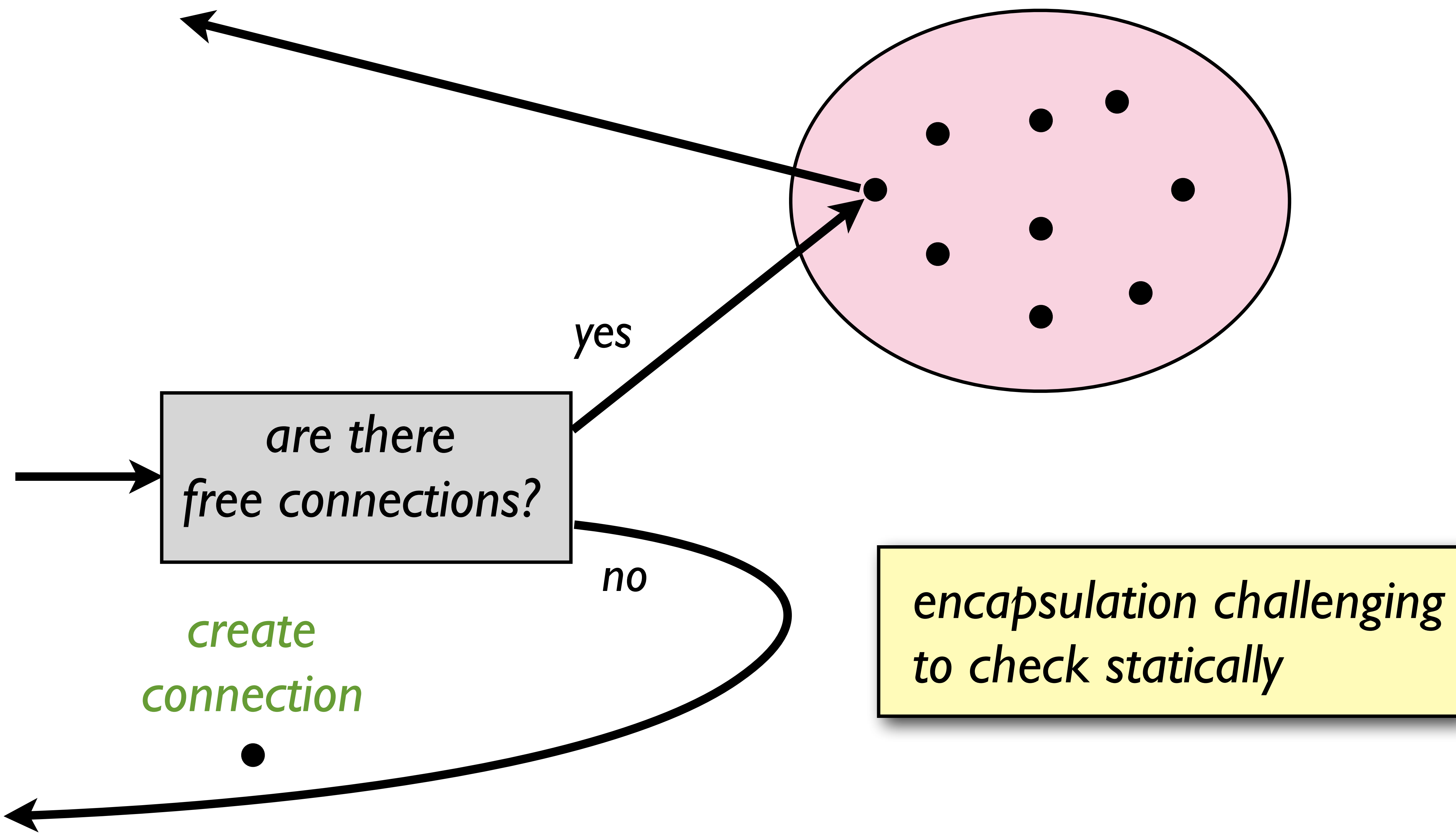
Shared mutable state



Shared mutable state



Shared mutable state



Inheritance

- inheritance allows specialisation and overriding
- determining what a method call actually does is difficult
- lookup scheme relies on dynamic information
  => *but we are interested in static reasoning and verification*

Inheritance

```
class CELL
{
   private int val;

   public virtual void set(int x)
   {
      this.val = x;
   }

  public virtual int get()
  {
      return this.val;
   }
}
```

Inheritance

```
class CELL
{
   private int val;

   public virtual void set(int x)
   {
      this.val = x;
   }

  public virtual int get()
  {
     return this.val;
  }
}
```

```
class RECELL: CELL
{
   private int bak;
   public override void set(int x)
   {
      this.bak = base.get();
      base.set(x);
   }
}
```

Inheritance

```
class CELL
{
   private int val;

   public virtual void set(int x)
   {
      this.val = x;
   }

  public virtual int get()
  {
     return this.val;
   }
}
```

```
class RECELL: CELL
{
   private int bak;
   public override void set(int x)
   {
      this.bak = base.get();
      base.set(x);
   }
}
```

```
class DCELL: CELL
{
   public override void set(int x)
   {
      base.set(2*x);
   }
}
```

Inheritance

```
class CELL
{
  private int val;

  public virtual void set(int x)
  {
    this.val = x;
  }

 public virtual int get()
 {
    return this.val;
  }
```

inheritance is not subtyping!

```
class RECELL: CELL
{
  private int bak;
  public override void set(int x)
  {
    this.bak = base.get();
    base.set(x);
  }
}

class DCELL: CELL
{
  public override void set(int x)
  {
    base.set(2*x);
  }
}
```

Motivating separation logic

- let’s first see how far we can go with classical Hoare logic
- consider the method java.awt.Rectangle.translate(int x, int y)

Motivating separation logic

- let's first see how far we can go with classical Hoare logic
- consider the method java.awt.Rectangle.translate(int x, int y)

$$\{\textbf{this}.x = X \wedge \textbf{this}.y = Y\}$$
$$\texttt{Rect::translate(x,y)}$$
$$\{\textbf{this}.x = X + x \wedge \textbf{this}.y = Y + y\}$$

Motivating separation logic

{**this**.x = *X* /\ **this**.y = *Y*}
    Rect::translate(x,y)
{**this**.x = *X* + x /\ **this**.y = *Y* + y}

```
this.x += x;
this.y += y;
```

Motivating separation logic

{**this**.x = *X* /\ **this**.y = *Y*}

    Rect::translate(x,y)

{**this**.x = *X* + x /\ **this**.y = *Y* + y}

```
this.x += x;
this.y += y;
```

```
this.x += x;
this.y += y;
this.h = 0;
```

```
this.x += x;
this.y += y;
if (this.parent != this)
    this.parent.x += x;
```

Motivating separation logic

{**this**.x = *X* /\ **this**.y = *Y*}

Rect::translate(x,y)

{**this**.x = *X* + x /\ **this**.y = *Y* + y}

framing?

```
this.x += x;
this.y += y;
```

```
this.x += x;
this.y += y;
this.h = 0;
```

```
this.x += x;
this.y += y;
if (this.parent != this)
   this.parent.x += x;
```

Motivating separation logic

- specifying what isn't modified is tedious:

$$\{(z \mathrel{/\!=} \mathbf{this} \;\backslash/\; f \notin \{x,y\}) \;/\backslash\; z.f = V\}$$

Rect::translate(x,y)

$$\{z.f = V\}$$

- can we just use modifies clauses?

Motivating separation logic

- specifying what isn't modified is tedious:

$$\{(z \mathbin{/=} \mathbf{this} \mathbin{\backslash/} f \notin \{x,y\}) \mathbin{/\backslash} z.f = V\}$$

Rect::translate(x,y)

$$\{z.f = V\}$$

- can we just use modifies clauses?

Rect::translate(x,y) modifies **this**.x, **this**.y

Motivating separation logic

- not when we have complex shapes in memory
- consider the method System.Collection.SortedList.Clear()

Motivating separation logic

- not when we have complex shapes in memory
- consider the method System.Collection.SortedList.Clear()

SortedList::Clear() modifies *.next

Motivating separation logic

- not when we have complex shapes in memory

- consider the method System.Collection.SortedList.Clear()

SortedList::Clear() modifies *.next

- not precise
- breaks abstraction
- doesn't work at all for interfaces

Motivating separation logic

- not when we have complex shapes in memory

- consider the method System.Collection.SortedList.Clear()

SortedList::Clear() modifies *.next

- not precise
- breaks abstraction
- doesn't work at all for interfaces

if not modifies clauses,
then what?
=> ownership, SL, ...

Motivating separation logic

- separation logic makes modifications implicit in the specification
  => *"anything not mentioned isn't changed"*

- supports assertions describing only the part of the memory being modified

- "natural" reasoning for O-O programs
  => *but need a new memory model*
  => *need to address encapsulation*
  => *and need to accommodate and control inheritance*

Next on the agenda

(1) motivation and challenges ✓

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification

Recap: the heaplet model

- the store: state of the local variables

$$\text{Variables} \to \text{Integers}$$

- the heap: state of dynamically-allocated objects

$$\text{Locations} \to \text{Integers}$$

where: $\text{Locations} \subset \text{Naturals}$

Recap: separating conjunction

$$\boxed{s, h \models p * q}$$

- informally: the heap *h* can be divided in two so that *p* is true of one partition and *q* of the other

Recap: separating conjunction

$$\boxed{s, h \models p * q}$$

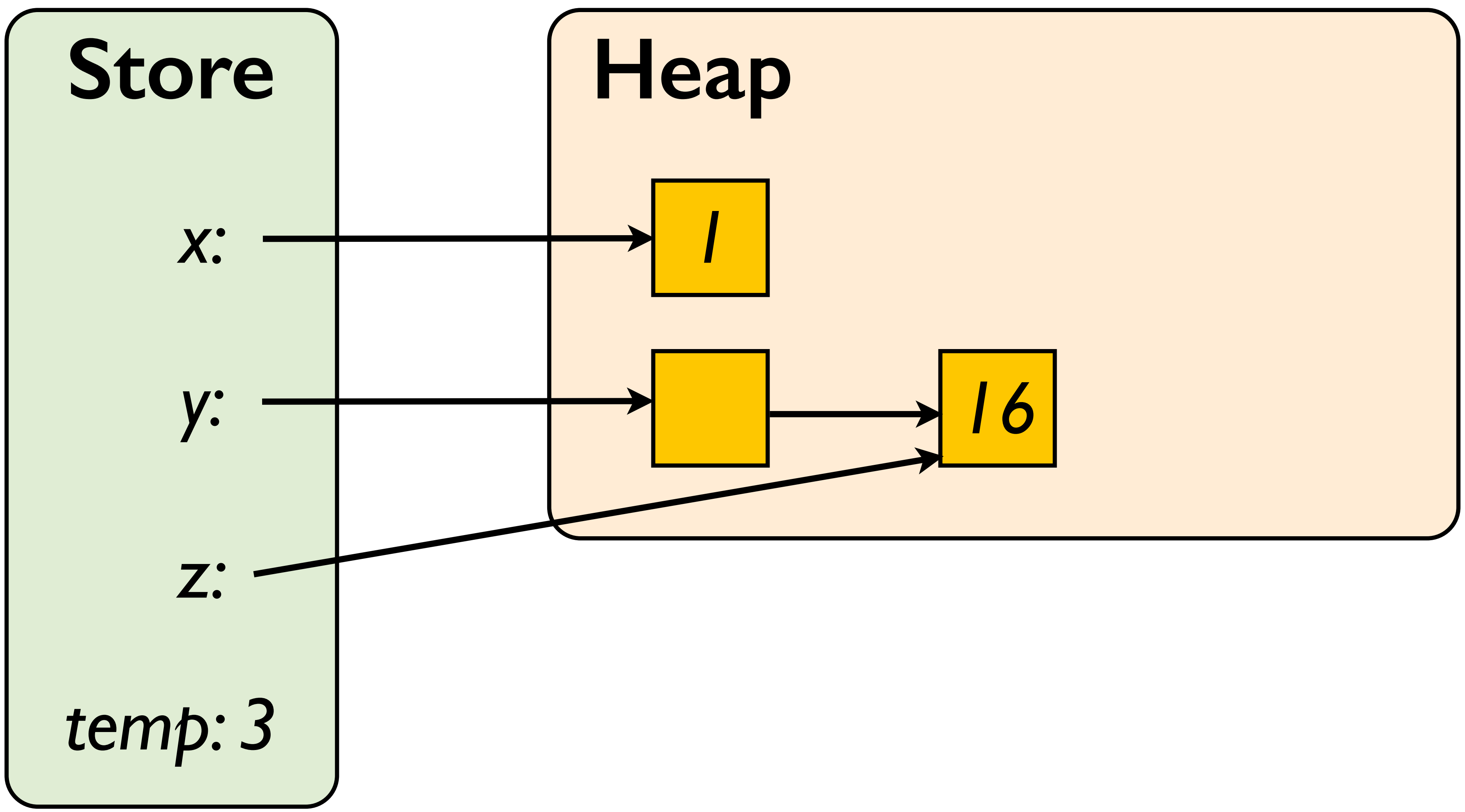
- informally: the heap *h* can be divided in two so that *p* is true of one partition and *q* of the other

disjoint domains of definition

disjoint function composition

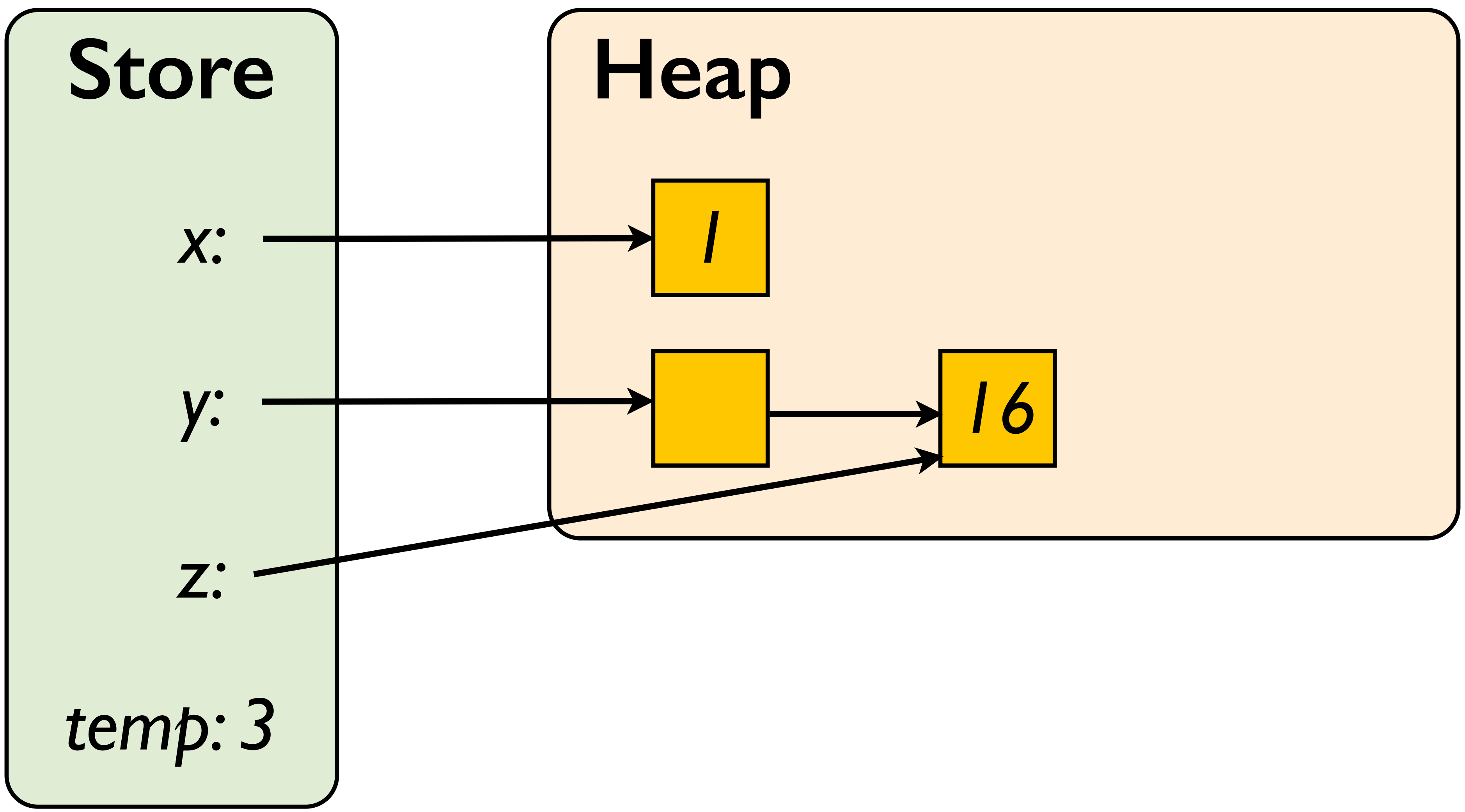
$$s, h \models p * q \quad \text{if} \quad \exists h_1, h_2.\ (h_1 \perp h_2),\ (h_1 \circ h_2 = h),$$
$$s, h_1 \models p \text{ and } s, h_2 \models q$$

Recap: example store and heap



Recap: example store and heap

$$x \mapsto 1 * y \mapsto z * z \mapsto 16 \wedge temp = 3$$



Extending the memory model

$$\boxed{s, h, d \models p}$$

- must now accommodate objects and dynamic types

Extending the memory model

$$\boxed{s, h, d \models p}$$

- must now accommodate objects and dynamic types

$$s : \text{Variables} \to \text{ObjectIDs} \cup \text{Integers}$$

Extending the memory model

$$\boxed{s, h, d \models p}$$

- must now accommodate objects and dynamic types

$$s : \mathrm{Variables} \rightarrow \mathrm{ObjectIDs} \cup \mathrm{Integers}$$

$$h : \mathrm{ObjectIDs} \times \mathrm{FieldNames} \rightarrow \mathrm{ObjectIDs} \cup \mathrm{Integers}$$

Extending the memory model

$$\boxed{s, h, d \models p}$$

- must now accommodate objects and dynamic types

$$s : \mathrm{Variables} \rightarrow \mathrm{ObjectIDs} \cup \mathrm{Integers}$$

$$h : \mathrm{ObjectIDs} \times \mathrm{FieldNames} \rightarrow \mathrm{ObjectIDs} \cup \mathrm{Integers}$$

$$d : \mathrm{ObjectIDs} \rightarrow \mathrm{ClassNames}$$

A partial semantics

- let se denote a "simple expression", i.e. one that does not access any fields or methods

$$s, h, d \models se_1.f \mapsto se_2$$

$$s, h, d \models se : C$$

$$s, h, d \models se_1 = se_2$$

$$s, h, d \models p * q$$

A partial semantics

- let se denote a "simple expression", i.e. one that does not access any fields or methods

$$s, h, d \models se_1.f \mapsto se_2$$ if h([|se1|]s, f) = [|se2|]s

$$s, h, d \models se : C$$

$$s, h, d \models se_1 = se_2$$

$$s, h, d \models p * q$$

A partial semantics

- let se denote a "simple expression", i.e. one that does not access any fields or methods

$s, h, d \models se_1.f \mapsto se_2$ if h([|se1|]s, f) = [|se2|]s

$s, h, d \models se : C$

- different to last week!

- contains "at least" (not "exactly")

$s, h, d \models se_1 = se_2$

$s, h, d \models p * q$

A partial semantics

- let se denote a "simple expression", i.e. one that does not access any fields or methods

$s, h, d \models se_1.f \mapsto se_2$ if h([|se1|]s, f) = [|se2|]s

$s, h, d \models se : C$ if d([|se|]s) = C

$s, h, d \models se_1 = se_2$

$s, h, d \models p * q$

A partial semantics

- let se denote a "simple expression", i.e. one that does not access any fields or methods

$s, h, d \models se_1.f \mapsto se_2$ if h([|se1|]s, f) = [|se2|]s

$s, h, d \models se : C$ if d([|se|]s) = C

$s, h, d \models se_1 = se_2$ if [|se1|]s = [|se2|]s

$s, h, d \models p * q$

A partial semantics

- let se denote a "simple expression", i.e. one that does not access any fields or methods

$s, h, d \models se_1.f \mapsto se_2$ if h([|se1|]s, f) = [|se2|]s

$s, h, d \models se : C$ if d([|se|]s) = C

$s, h, d \models se_1 = se_2$ if [|se1|]s = [|se2|]s

$s, h, d \models p * q$ if ∃ h1,h2 . h1 ⊥ h2 /\ h = h1 ∪ h2 /\ s,h1,d |= p /\ s,h2,d |= q

Separating conjunction example

- what kind of heap would satisfy the following?

$$s, h, d \models x_1.f \mapsto y * x_2.f \mapsto y$$

Separating conjunction example

- what kind of heap would satisfy the following?

$$s, h, d \models x_1.f \mapsto y * x_2.f \mapsto y$$

- ...and what about this?

$$s, h, d \models x_1.f \mapsto y \wedge x_2.f \mapsto y$$

Next on the agenda

(1) motivation and challenges ✓

(2) extending the memory model ✓

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification

Simple instructions and proof rules

- we start by building a separation logic for a simple object-oriented language
  => *field mutation, field lookup, ...*

- postpone method specification and verification
  => *avoid the complexity of method dispatch until later*

- reminder: tight interpretation of triples

$$\models \{pre\}\ P\ \{post\}$$

Field mutation and field lookup

$$\vdash \ \{x.f \mapsto \_\}\ x.f := y\ \{ \qquad\qquad\qquad \}$$

$$\vdash \ \{x.f \mapsto e\}\ y := x.f\ \{ \qquad\qquad\qquad\qquad\qquad \}$$

Field mutation and field lookup

$$\vdash \ \{x.f \mapsto \_\}\ x.f := y\ \{\text{x.f |-> y}\}$$

$$\vdash \ \{x.f \mapsto e\}\ y := x.f\ \{\qquad\qquad\qquad\qquad\}$$

Field mutation and field lookup

$$\vdash \ \{x.f \mapsto \_\}\ x.f := y\ \{$$x.f |-> y$$\}$$

$$\vdash \ \{x.f \mapsto e\}\ y := x.f\ \{$$x.f |-> e /\ y = e$$\}$$

provided y /= x and y not free in e

Field mutation and field lookup

$\vdash\ \{x.f \mapsto _\}\ x.f := y\ \{$x.f |-> y$\}$

$\vdash\ \{x.f \mapsto e\}\ y := x.f\ \{$x.f |-> e /\ y = e$\}$

provided y /= x and y not free in e

and if not...?

Structural rules

$$\frac{\{p\}\ \ C\ \ \{q\}}{\{p * r\}\ \ C\ \ \{q * r\}}$$

provided modifies(C) ∩ fv(r) = {}

$$\frac{\{p\}\ \ C\ \ \{q\}}{\{\exists v.\ p\}\ \ C\ \ \{\exists v.\ q\}}$$

provided v not free in C

Simple proof example

- consider the statement x := x.next
  => *e.g. from a linked list class*

- verify the following triple:

$$\{x.next \mapsto \_ * x = y\}\ x := x.next\ \{y.next \mapsto x\}$$

Simple proof example

{x.next |-> _ * x = y}

x := x.next;

Simple proof example

{x.next |-> _ * x = y}

{$\exists$n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old}}

    x := x.next;

Simple proof example

{x.next |-> _ * x = y}

{$\exists$n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old}}

{x.next |-> n * x = x^{old} /\ y = x^{old}}

x := x.next;

Simple proof example

{x.next |-> _ * x = y}

{∃n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old}}

{x.next |-> n * x = x^{old} /\ y = x^{old}}

x := x.next;

{x^{old}.next |-> n * x = n /\ y = x^{old}}

Simple proof example

{x.next |-> _ * x = y}

{$\exists$n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old}}

{x.next |-> n * x = x^{old} /\ y = x^{old}}

x := x.next;

{x^{old}.next |-> n * x = n /\ y = x^{old}}

{$\exists$n, x^{old}. x^{old}.next |-> n * x = n /\ y = x^{old}}

Simple proof example

{x.next |-> _ * x = y}

{∃n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old}}

{x.next |-> n * x = x^{old} /\ y = x^{old}}

x := x.next;

{x^{old}.next |-> n * x = n /\ y = x^{old}}

{∃n, x^{old}. x^{old}.next |-> n * x = n /\ y = x^{old}}

{y.next |-> x}

Next on the agenda

(1) motivation and challenges ✓

(2) extending the memory model ✓

(3) simple statements and proof rules ✓

(4) tackling inheritance: abstract predicate families

(5) method specification and verification

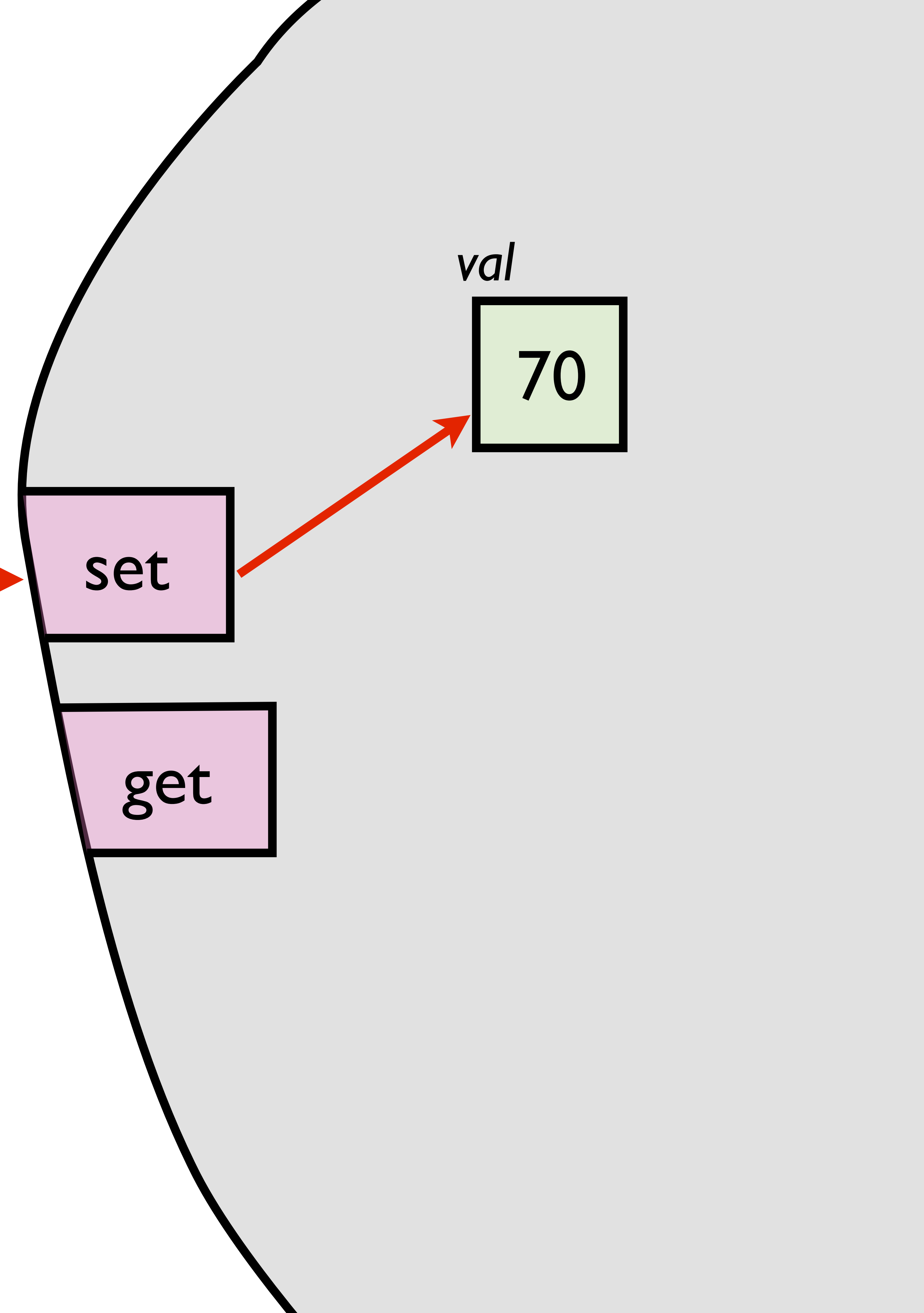
Recap: Cell, ReCell, and DCell

```
class CELL
{
  private int val;

  public virtual void set(int x)
  {
    this.val = x;
  }

  public virtual int get()
  {
    return this.val;
  }
}
```

```
class RECELL: CELL
{
  private int bak;
  public override void set(int x)
  {
    this.bak = base.get();
    base.set(x);
  }
}
```

```
class DCELL: CELL
{
  public override void set(int x)
  {
    base.set(2*x);
  }
}
```

```
{ ??? }
  x.set(5)
{ ??? }
  y := x.get + 5
{ ??? }
```



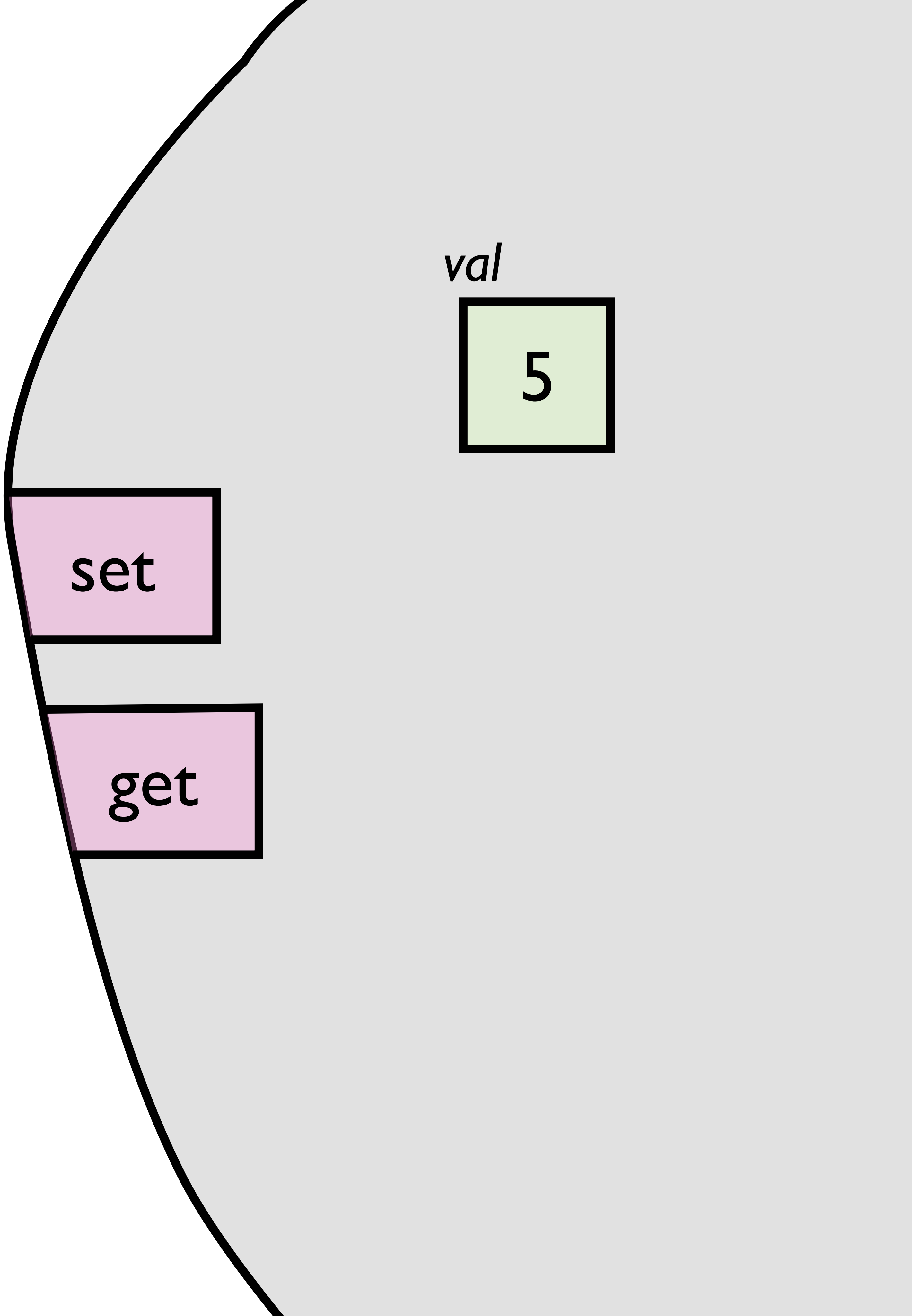
{ ??? }

    x.set(5)

{ ??? }

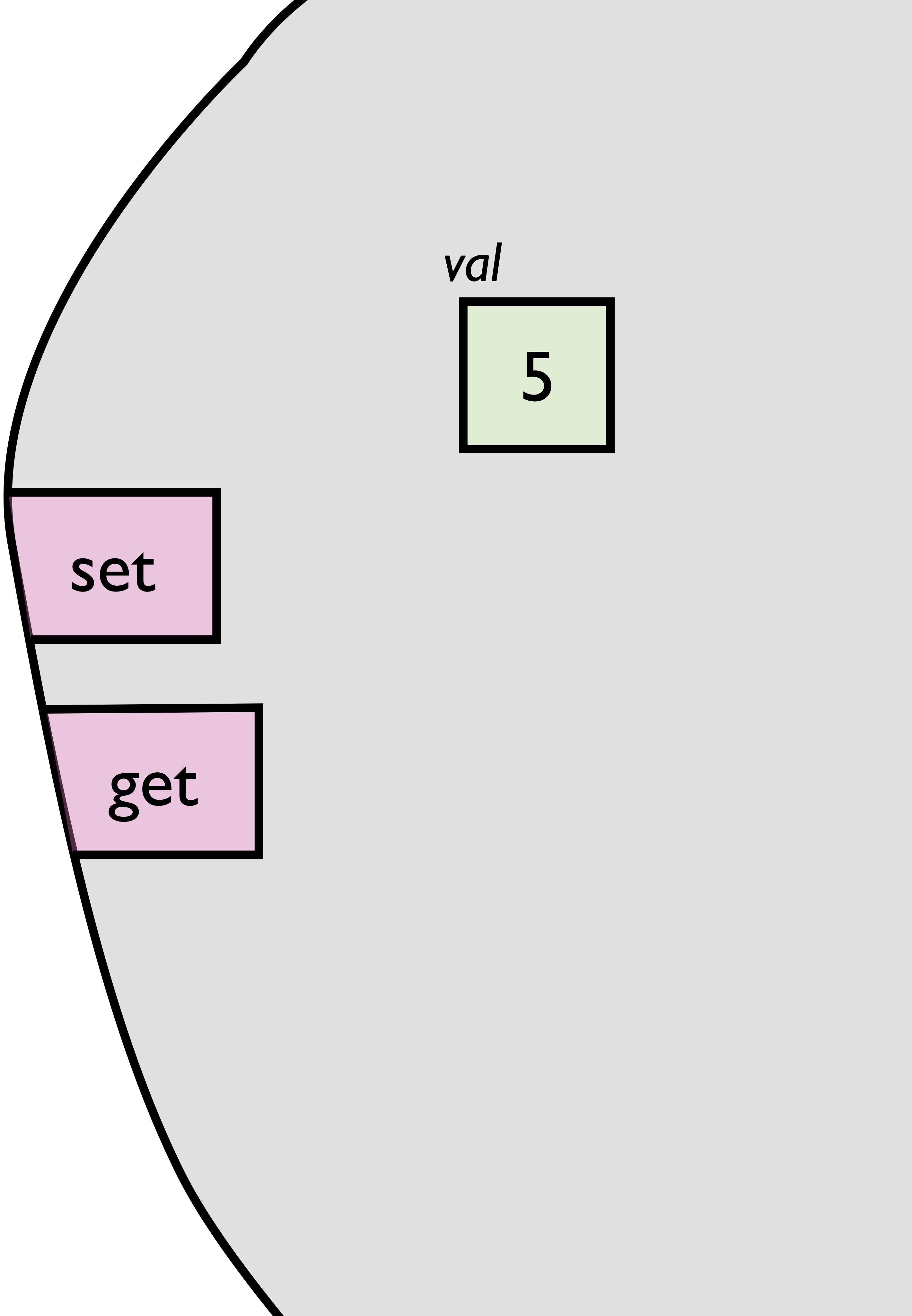
    y := x.get + 5

{ ??? }



{ x.val |-> _ }
  x.set(5)
{ x.val |-> 5 }
  y := x.get + 5
{ ??? }

breaks abstraction!



{ x.val |-> _ }

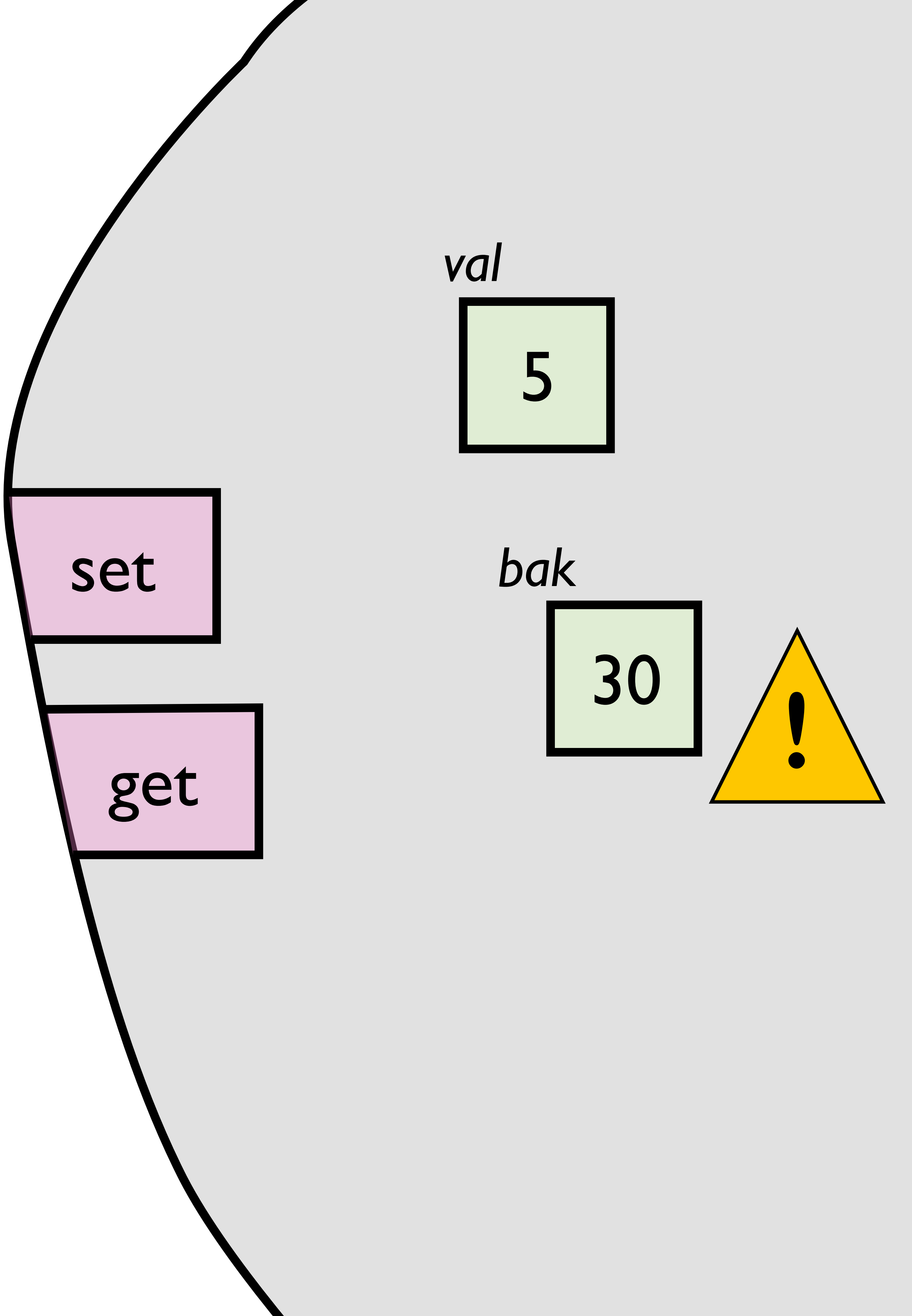
x.set(5)

{ x.val |-> 5 }

y := x.get + 5

{ ??? }

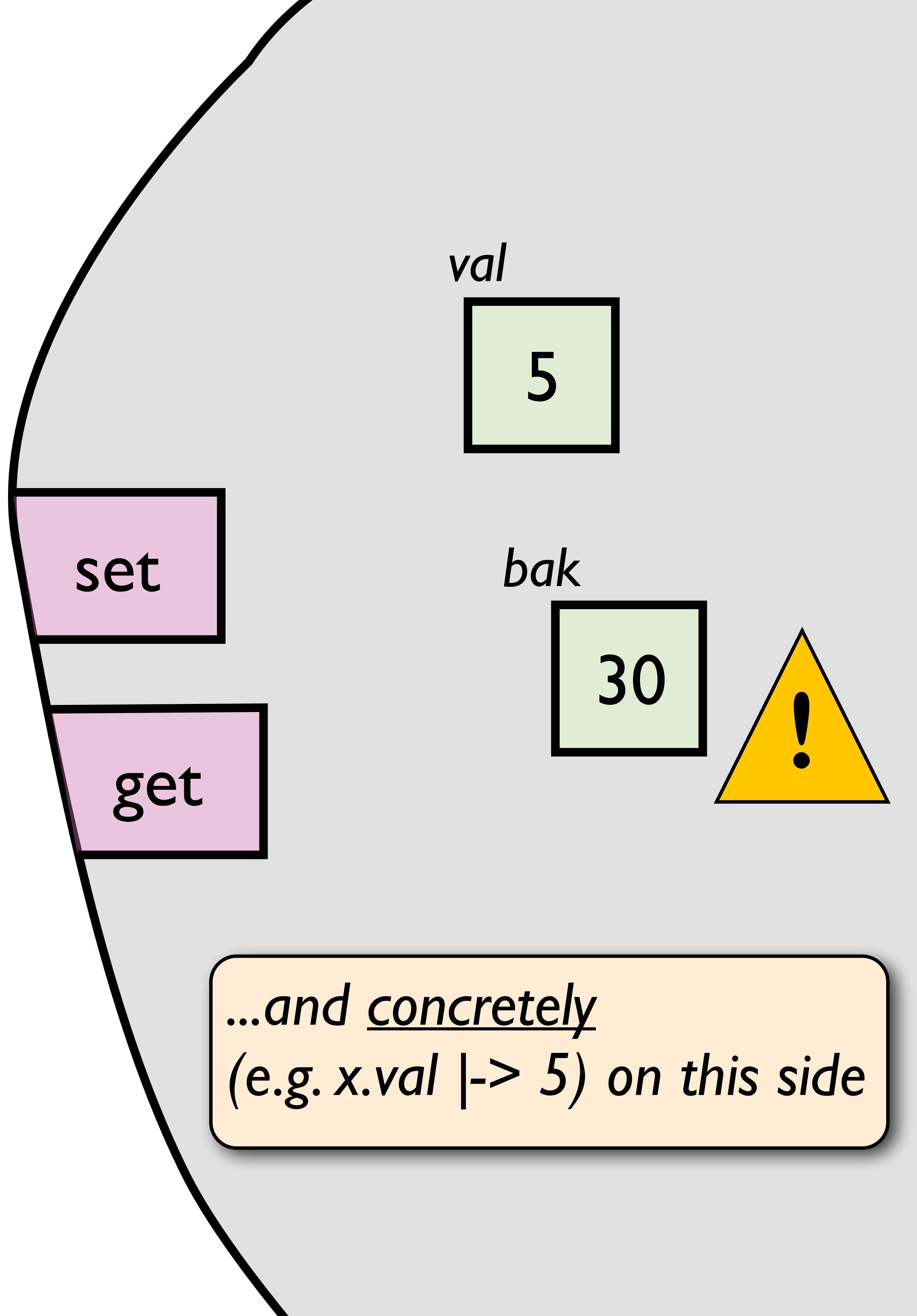
breaks abstraction!



{ x.val |-> _ }
  x.set(5)
{ x.val |-> 5 }
  y := x.get + 5
{ ??? }

breaks abstraction!

need to be able to reason <u>abstractly</u> on this side...



The boundary of abstraction

- there is a need for data-centred abstractions in our reasoning system
- reason, on the client side, about encapsulated state abstractly
- need to cope with inheritance and dynamic dispatch

Abstract predicates (Ap)

- annotate classes with abstract predicate (Ap) definitions

- an Ap consists of a name, definition, and scope
  => *for simplicity, scope here is a single class*

- within the scope, can freely change between the name and definition

- outside the scope, can only use the name

Abstract predicate example

```
class CELL
{
  // Ap definitions
  define x.Val_Cell(n) as x.val |-> n

  // field declarations
  private int val;

  // methods (i.e. set, get)
  ...
}
```

name?
definition?
scope?

Abstract predicate example

```
class CELL
{
   // Ap definitions
   define x.Val_Cell(n) as x.val |-> n

   // field declarations
   private int val;

   // methods (i.e. set, get)
   ...
}
```

name? x.Val$_{Cell}$(n)

definition? x.val |-> n

scope? CELL

Abstract predicate example

```
class CELL
{
   // Ap definitions
   define x.Val_Cell(n) as x.val |-> n

   // field declarations
   private int val;

   // methods (i.e. set, get)
   ...
}
```

name? x.Val_{Cell}(n)
definition? x.val |-> n
scope? CELL

{ ??? }
   x.set(5)
{ ??? }
   y := x.get + 5
{ ??? }

Abstract predicate example

```
class CELL
{
   // Ap definitions
   define x.Val_Cell(n) as x.val |-> n

   // field declarations
   private int val;

   // methods (i.e. set, get)
   ...
}
```

name? $x.Val_{Cell}(n)$

definition? x.val |-> n

scope? CELL

{ $x.Val_{Cell}(\_)$ }

x.set(5)

{ $x.Val_{Cell}(5)$ }

y := x.get + 5

{ ??? }

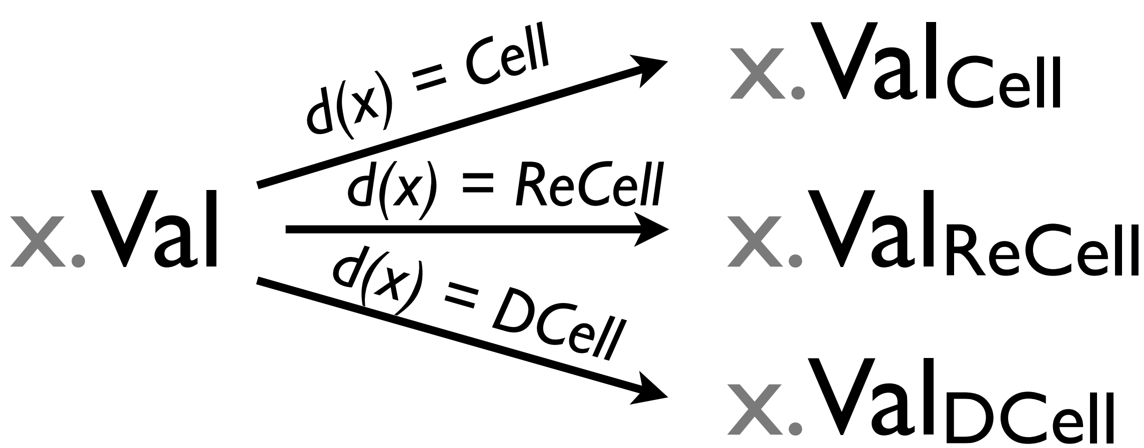
- *how do we prove {$x.Val_{Cell}(\_)$} x.set(5) {$x.Val_{Cell}(5)$} ?*
- *what if d(x) = ReCell ?*

Abstract predicate <u>families</u> (Apfs)

- different (sub)classes can have different Ap definitions
- abstract predicate families (Apfs) provide different definitions, or "entries", based on dynamic type information
  => *"dynamically dispatched predicates"*
- annotate classes with different Apf entries

Abstract predicate <u>families</u> (Apfs)

- different (sub)classes can have different Ap definitions
- abstract predicate families (Apfs) provide different definitions, or “entries”, based on dynamic type information
  => *“dynamically dispatched predicates”*
- annotate classes with different Apf entries



Abstract predicate family example

```
class CELL
{
  // Apf definitions
  define x.Val_Cell(n) as x.val |-> n

  // field declarations
  private int val;

  // methods (i.e. set, get)
  ...
}
```

```
class RECELL: CELL
{
  // Apf definitions
  ???


  // field declarations
   private int bak;

  // methods (i.e. override set)
  ...
}
```

Abstract predicate family example

```
class CELL
{
  // Apf definitions
  define x.Val_Cell(n) as x.val |-> n

  // field declarations
  private int val;

  // methods (i.e. set, get)
  ...
}
```

```
class RECELL: CELL
{
  // Apf definitions
  define x.Val_Recell(n,b)
  as x.Val_Cell(n) * x.bak |-> b

  // field declarations
   private int bak;

  // methods (i.e. override set)
  ...
}
```

Abstract predicate family example

```
class CELL
{
  // Apf definitions
  define x.Val_Cell(n) as x.val |-> n

  // field declarations
  private int val;

  // methods (i.e. set, get)
  ...
}
```

```
class RECELL: CELL
{
  // Apf definitions
  define x.Val_Recell(n,b)
  as x.Val_Cell(n) * x.bak |-> b

  // field declarations
   private int bak;

  // methods (i.e. override set)
  ...
}
```

ReCell adds an argument to the Apf Val

=> in the <u>scope</u> of ReCell, ∀x,n: x.Val(n) <=> x.Val(n,_)

Next on the agenda

(1) motivation and challenges

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification

Static vs. dynamic specifications

- two types of method calls in O-O languages
  => *<u>statically</u> dispatched, e.g. y.Cell::get()*
  => *<u>dynamically</u> dispatched, e.g. x.m(a)*

- annotate methods with static <u>and</u> dynamic specifications

describes in detail what the body does

more abstract: "idea" behind the method that subclasses must respect

Example

```
class CELL
{
  define x.Val_Cell(n) as x.val |-> n

  private int val;

  public virtual void set(int x)
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
   { this.val = x; }

  public virtual int get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
  { return this.val; }
}
```

Example

```
class CELL
{
  define x.Val_Cell(n) as x.val |-> n

  private int val;

  public virtual void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
  static { ??? } _ { ??? }
   { this.val = x; }

  public virtual int get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
  { return this.val; }
}
```

Example

```
class CELL
{
  define x.ValCell(n) as x.val |-> n

  private int val;

  public virtual void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
  static { this.ValCell(_) } _ { this.ValCell(x) }
   { this.val = x; }

  public virtual int get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
  { return this.val; }
}
```

Example

```
class CELL
{
  define x.ValCell(n) as x.val |-> n

  private int val;

  public virtual void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
  static { this.ValCell(_) } _ { this.ValCell(x) }
   { this.val = x; }

  public virtual int get()
  dynamic { this.Val(v) } _ { this.Val(v) /\ Res = v }
  static { ??? } _ { ??? }
  { return this.val; }
}
```

Example

```
class CELL
{
  define x.Val_Cell(n) as x.val |-> n

  private int val;

  public virtual void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
  static { this.Val_Cell(_) } _ { this.Val_Cell(x) }
   { this.val = x; }

  public virtual int get()
  dynamic { this.Val(v) } _ { this.Val(v) /\ Res = v }
  static { this.Val_Cell(v) } _ { this.Val_Cell(v) /\ Res = v }
  { return this.val; }
}
```

prove!

```
{ true }
 x := new Cell(3)
 y := new Cell(4)
 x.set(5)
 n := y.get()
{ x.Val(5) * y.Val(4)
  * n=4 }
```

Verifying a newly introduced method

- two proof obligations

- first, verify the method body against the static specification
  => e.g. $\{this.Val_{Cell}(\_)\}$ *this.val := x* $\{this.Val_{Cell}(x)\}$
  => *we are now "in scope" and can use the definition of* Val_{Cell}

- second, check the consistency of the static and dynamic specifications
  => e.g. $\{this.Val_{Cell}(\_)\}$ _ $\{this.Val_{Cell}(x)\}$
  implies
  $\{this{:}Cell * this.Val(\_)\}$ _ $\{this.Val(x)\}$

Subclassing

```
class RECELL: CELL
{
  // Apf definitions
  define x.ValRecell(n,b) as x.ValCell(n) * x.bak |-> b

  private int bak;

  public override void set(int x)
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
  { this.bak = base.get(); base.set(x); }

  inherit get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
}
```

Subclassing

```
class RECELL: CELL
{
  // Apf definitions
  define x.Val_Recell(n,b) as x.Val_Cell(n) * x.bak |-> b

  private int bak;

  public override void set(int x)
  dynamic { this.Val(v,_) } _ { this.Val(x,v) }
  static { ??? } _ { ??? }
  { this.bak = base.get(); base.set(x); }

  inherit get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
}
```

Subclassing

```
class RECELL: CELL
{
  // Apf definitions
  define x.ValRecell(n,b) as x.ValCell(n) * x.bak |-> b

  private int bak;

  public override void set(int x)
  dynamic { this.Val(v,_) } _ { this.Val(x,v) }
  static { this.ValReCell(v,_) } _ { this.ValReCell(x,v) }
  { this.bak = base.get(); base.set(x); }

  inherit get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
}
```

Subclassing

```
class RECELL: CELL
{
  // Apf definitions
  define x.Val_Recell(n,b) as x.Val_Cell(n) * x.bak |-> b

  private int bak;

  public override void set(int x)
  dynamic { this.Val(v,_) } _ { this.Val(x,v) }
  static { this.Val_ReCell(v,_) } _ { this.Val_ReCell(x,v) }
  { this.bak = base.get(); base.set(x); }

  inherit get()
  dynamic { this.Val(v,b) } _ { this.Val(v,b) /\ Res = v }
  static { ??? } _ { ??? }
}
```

Subclassing

```
class RECELL: CELL
{
  // Apf definitions
  define x.Val_Recell(n,b) as x.Val_Cell(n) * x.bak |-> b

  private int bak;

  public override void set(int x)
  dynamic { this.Val(v,_) } _ { this.Val(x,v) }
  static { this.Val_ReCell(v,_) } _ { this.Val_ReCell(x,v) }
  { this.bak = base.get(); base.set(x); }

  inherit get()
  dynamic { this.Val(v,b) } _ { this.Val(v,b) /\ Res = v }
  static { this.Val_ReCell(v,b) } _ { this.Val_ReCell(v,b) /\ Res = v }
}
```

Verifying an overridden method
(e.g. set)

- three proof obligations
- (1) body verification; (2) consistency checking; and
- (3) verify that the dynamic specification is stronger than the one in the parent class

Verifying an inherited method

(e.g. get)

- three proof obligations
- (1) body verification; (2) consistency checking; and
- (3) verify that the static specification follows from the one in the parent class

And what about this?

```
class DCELL: CELL
{
   public override void set(int x)
   {
      base.set(2*x);
   }
}
```

And what about this?

```
class DCELL: CELL
{
   define ??? as ???



   public override void set(int x)
   dynamic { ??? } _ { ??? }
   static { ??? } _ { ??? }
   { base.set(2*x); }

  public inherit get()
   dynamic { ??? } _ { ??? }
   static { ??? } _ { ??? }
}
```

And what about this?

```
class DCELL: CELL
{
   define x.Val_DCell(n) as false
   define x.DVal_DCell(n) as x.Val_Cell(n)

   public override void set(int x)
   dynamic { this.Val(_) } _ { ??? }
    also { this.DVal(_) } _ { ??? }
   static { ??? } _ { ??? }
   { base.set(2*x); }

  public inherit get()
  dynamic { this.Val(v) } _ { ??? }
    also { this.DVal(v) } _ { ??? }
   static { ??? } _ { ??? }
}
```

idea: ensure that <u>no</u> client will <u>ever</u> have a Val predicate for a DCell object

And what about this?

```
class DCELL: CELL
{
  define x.Val_DCell(n) as false
  define x.DVal_DCell(n) as x.Val_Cell(n)

  public override void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
    also { this.DVal(_) } _ { this.DVal(2*x) }
  static { ??? } _ { ??? }
  { base.set(2*x); }

  public inherit get()
  dynamic { this.Val(v) } _ { ??? }
    also { this.DVal(v) } _ { ??? }
  static { ??? } _ { ??? }
}
```

idea: ensure that no client will ever have a Val predicate for a DCell object

And what about this?

```
class DCELL: CELL
{
   define x.ValDCell(n) as false
   define x.DValDCell(n) as x.ValCell(n)

   public override void set(int x)
   dynamic { this.Val(_) } _ { this.Val(x) }
     also { this.DVal(_) } _ { this.DVal(2*x) }
   static { this.DValDCell(_) } _ { this.DValDCell(2*x) }
   { base.set(2*x); }

  public inherit get()
  dynamic { this.Val(v) } _ { ??? }
     also { this.DVal(v) } _ { ??? }
   static { ??? } _ { ??? }
}
```

idea: ensure that no client will ever have a Val predicate for a DCell object

And what about this?

```
class DCELL: CELL
{
   define x.Val_DCell(n) as false
   define x.DVal_DCell(n) as x.Val_Cell(n)

   public override void set(int x)
   dynamic { this.Val(_) } _ { this.Val(x) }
     also { this.DVal(_) } _ { this.DVal(2*x) }
   static { this.DVal_DCell(_) } _ { this.DVal_DCell(2*x) }
   { base.set(2*x); }

  public inherit get()
  dynamic { this.Val(v) } _ { this.Val(v) /\ Res = v }
     also { this.DVal(v) } _ { this.DVal(v) /\ Res = v }
   static { ??? } _ { ??? }
}
```

idea: ensure that no client will ever have a Val predicate for a DCell object

And what about this?

```
class DCELL: CELL
{
  define x.ValDCell(n) as false
  define x.DValDCell(n) as x.ValCell(n)

  public override void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
    also { this.DVal(_) } _ { this.DVal(2*x) }
  static { this.DValDCell(_) } _ { this.DValDCell(2*x) }
  { base.set(2*x); }

  public inherit get()
  dynamic { this.Val(v) } _ { this.Val(v) /\ Res = v }
    also { this.DVal(v) } _ { this.DVal(v) /\ Res = v }
  static { this.DValDCell(v) } _ { this.DValDCell(v) /\ Res = v }
}
```

idea: ensure that <u>no</u> client will <u>ever</u> have a Val predicate for a DCell object

Next on the agenda

(1) motivation and challenges ✓

(2) extending the memory model ✓

(3) simple statements and proof rules ✓

(4) tackling inheritance: abstract predicate families ✓

(5) method specification and verification ✓

Conclusion

- separation logic, for reasoning about shared mutable state, can be extended to object-oriented programs
- memory model extended to support objects and dynamic type information
- inheritance tackled with Apfs and static/dynamic specs
- implemented (e.g. jStar, VeriFast); can verify common design patterns
- only just the basics! See the papers for the full story

Main sources for these lectures

Parkinson and Bierman: *Separation Logic, Abstraction and Inheritance.*
In: POPL 2008

Parkinson and Bierman: *Separation Logic for Object-Oriented Programming.*
In: Aliasing in Object-Oriented Programming, 2013

Thank you! Questions?

Next unit:

- testing (with Bertrand Meyer and guest lecturers)