Imperative Programming 2:
Polymorphism 2

Hongseok Yang
University of Oxford

Generic class/trait

- You saw them already.

Generic class/trait

- You saw them already.

```
class Queue[T] {
   ....
   def put(x: T): Unit ...
   def get(): T ...
   ...
}
```

Generic class/trait

- You saw them already.

```
class Queue[T] {
   trait Ordered[T] {
     def compare(that: T): Int ...
     def <(that: T): Boolean ...
     def >(that: T): Boolean ...
     def <=(that: T): Boolean ...
     ...
   }
}
```

Generic class/trait

- You saw them already.

```
class Queue[T] {
  trait Ordered[T] {
    scala> val x = new Array[Int](3)
    x: Array[Int] = Array(0, 0, 0)

}
    scala> val x: Array[Int] = new Array(3)
    x: Array[Int] = Array(0, 0, 0)
  }
```

Generic class/trait

- You saw them already. How would you define?

```
class Queue[T] {
  trait Ordered[T] {
    scala> val x = new Array[Int](3)
    x: Array[Int] = Array(0, 0, 0)

}   scala> val x: Array[Int] = new Array(3)
    x: Array[Int] = Array(0, 0, 0)
  }
```

Generic class/trait

- You saw them already. How would you define?
- Experts' usage.

Generic class/trait

- You saw them already. How
- Experts' usage.

(1) +B, This <: ...

Implementation of immutable map

```
trait MapLike[A, +B, +This <: MapLike[A, B, This] with Map[A, B]]
  extends scala.collection.MapLike[A, B, This]
     with Parallelizable[(A, B), ParMap[A, B]]
```

Generic class/trait

- You saw them already. How
- Experts' usage.

(1) +B, This <: ...
(2) -T2

Implementation of immutable map

Implementation of functions with three arguments

```
trait Function3[-T1, -T2, -T3, +R] extends AnyRef { self =>
```

Generic class/trait

- You saw them already. How
- Experts' usage.

(1) +B, This <: ...
(2) -T2
(3) CC[...]

Implementation of immutable map

Implementation of functions with three arguments

Implementation of set

```
abstract class SetFactory[CC[X] <: Set[X] with SetLike[X, CC[X]]]
  extends GenSetFactory[CC] with GenericSeqCompanion[CC]
```

Generic class/trait

Interaction between generics and subtyping

(1) +B, This <: ...
(2) -T2
(3) CC[...]

Type-constructor parameter

Learning outcome

- Can explain what a generic class/trait/method is and how it interacts with subtyping.
- Can develop generic classes with covariant or contravariant type variables.

Basics of generics

Generic class

- A class or trait taking type parameters.
- Syntax:

```
class C[T] {..}          new List[Double]
```

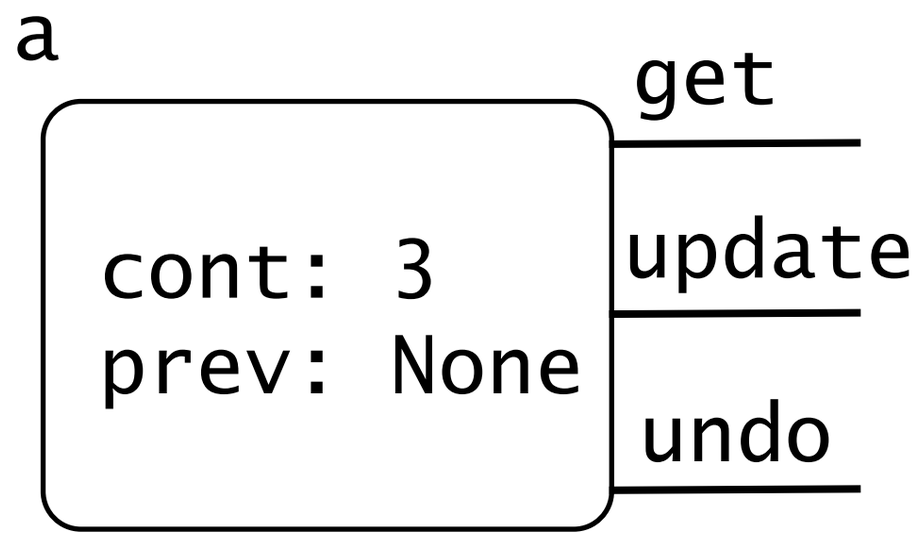
- Intuition 1 -- A type constructor.
- Intuition 2 -- No computation in a generic class/trait depends on specifics of type parameters.

Programming task: Generic cell with undo

- Create a class for immutable cell objects, which store values of unspecified type T.
- Cell objects should support `get`, `update` and undo operations.

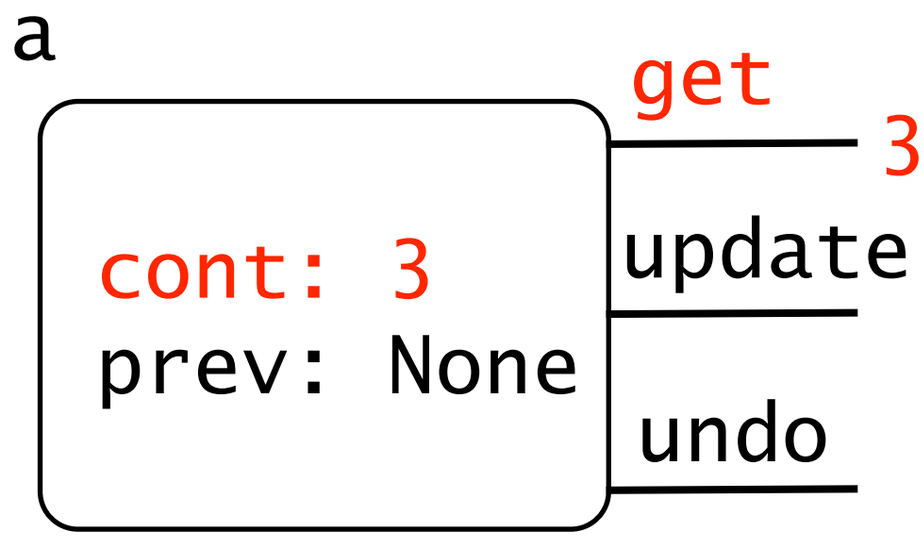
Data structure

```
val a = new Cell[Int](3, None)
```



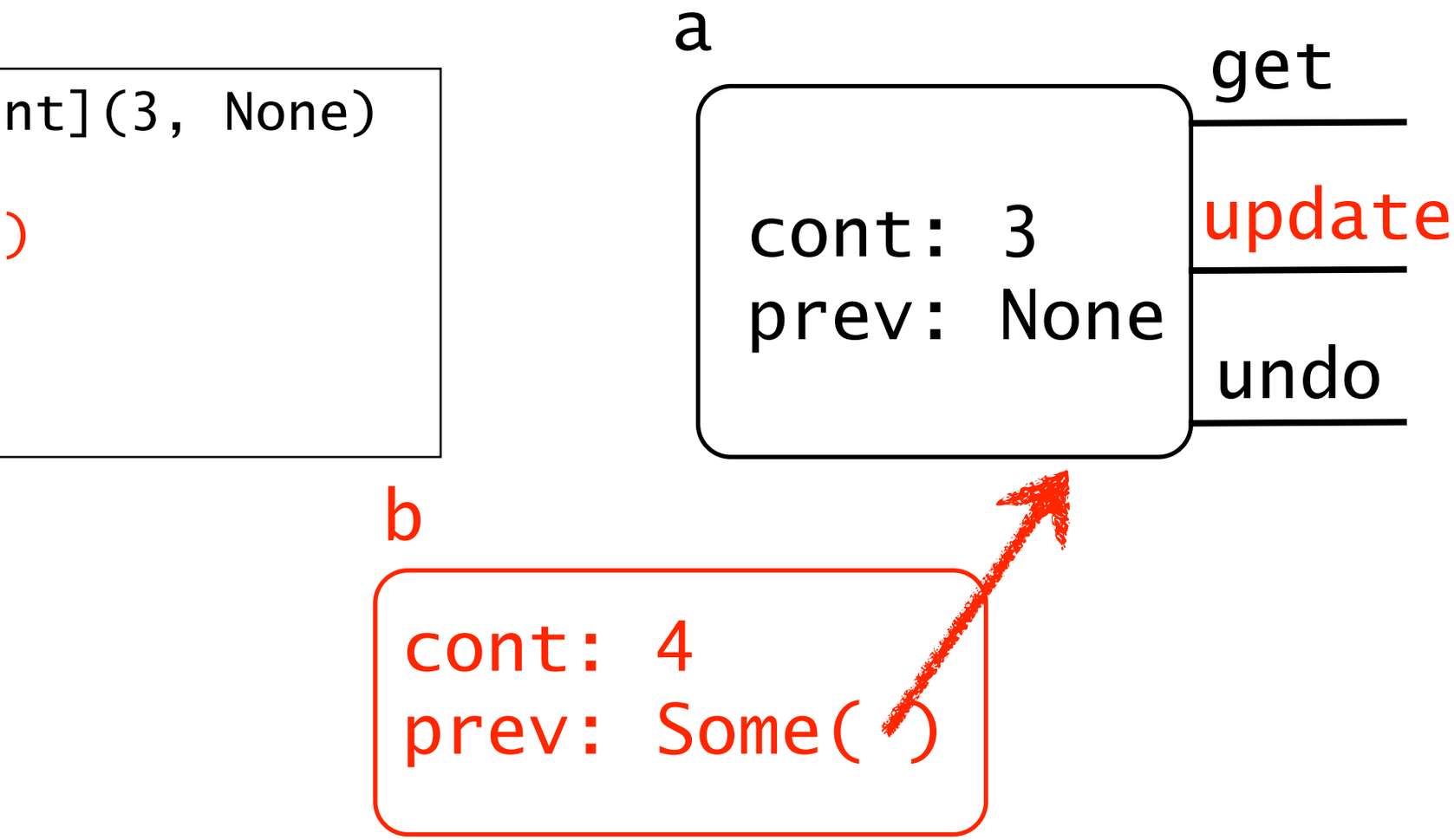
Data structure

```
val a = new Cell[Int](3, None)
println(a.get)
```



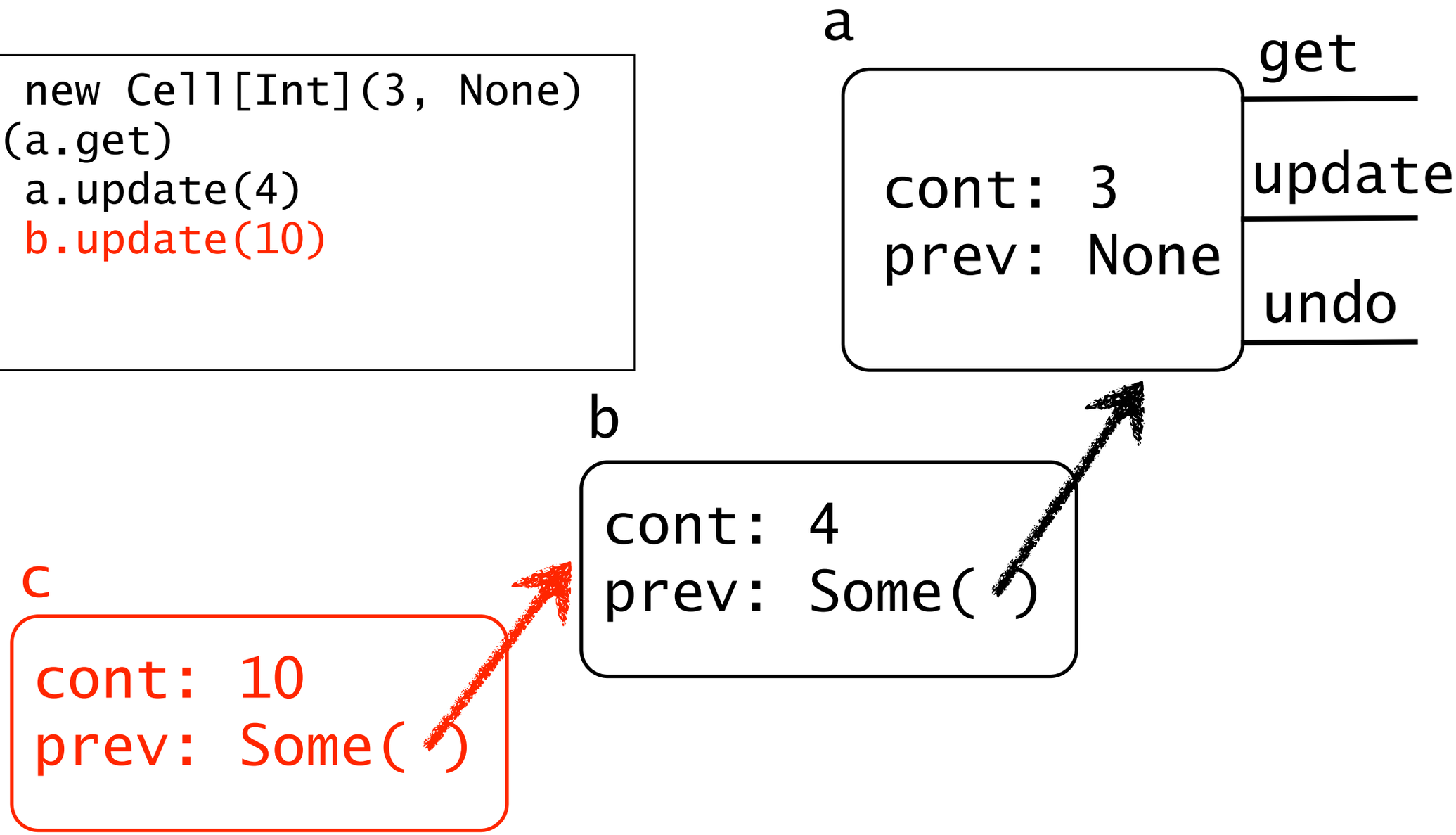
Data structure

```
val a = new Cell[Int](3, None)
println(a.get)
val b = a.update(4)
```



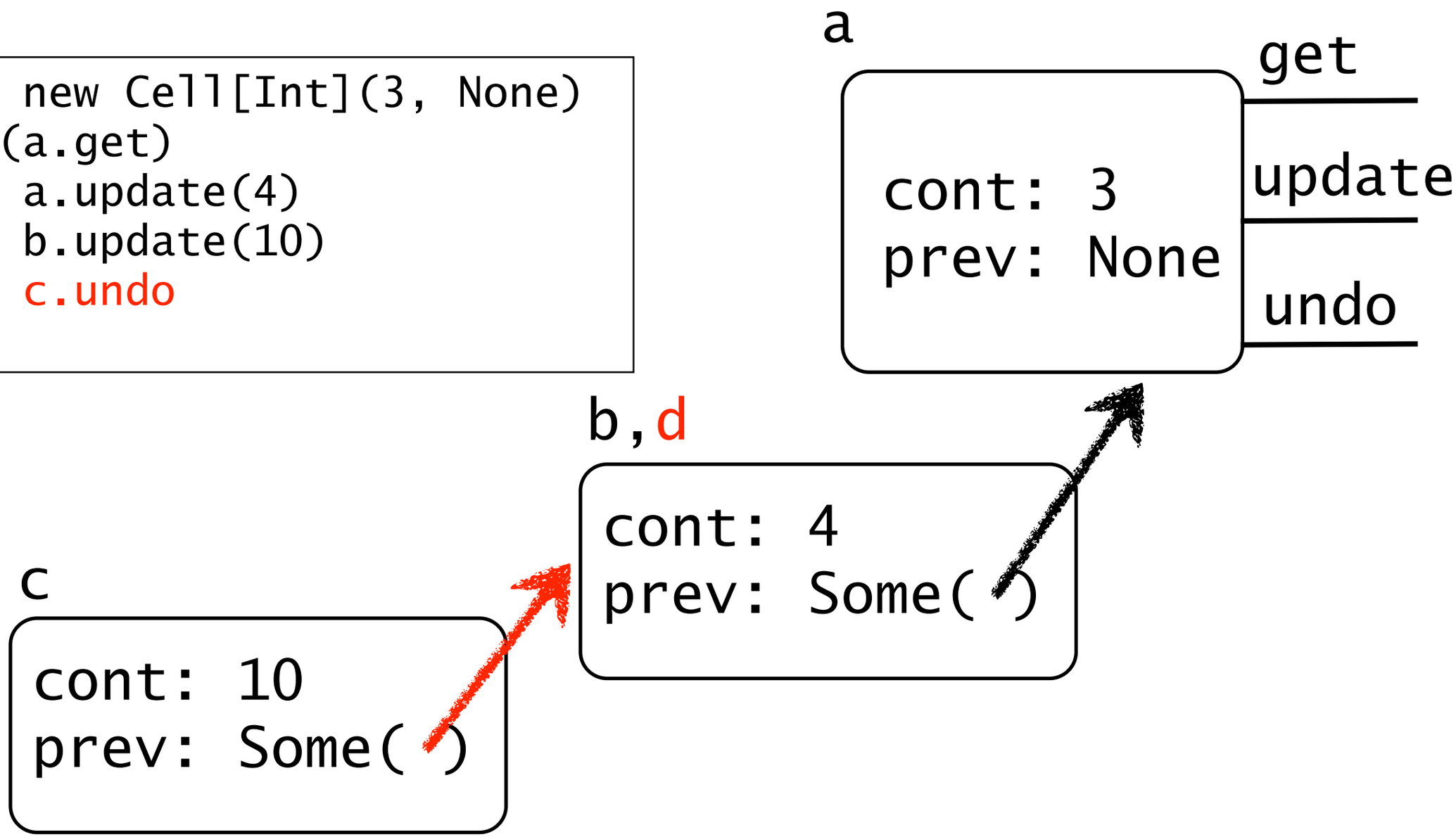
Data structure

```
val a = new Cell[Int](3, None)
println(a.get)
val b = a.update(4)
val c = b.update(10)
```



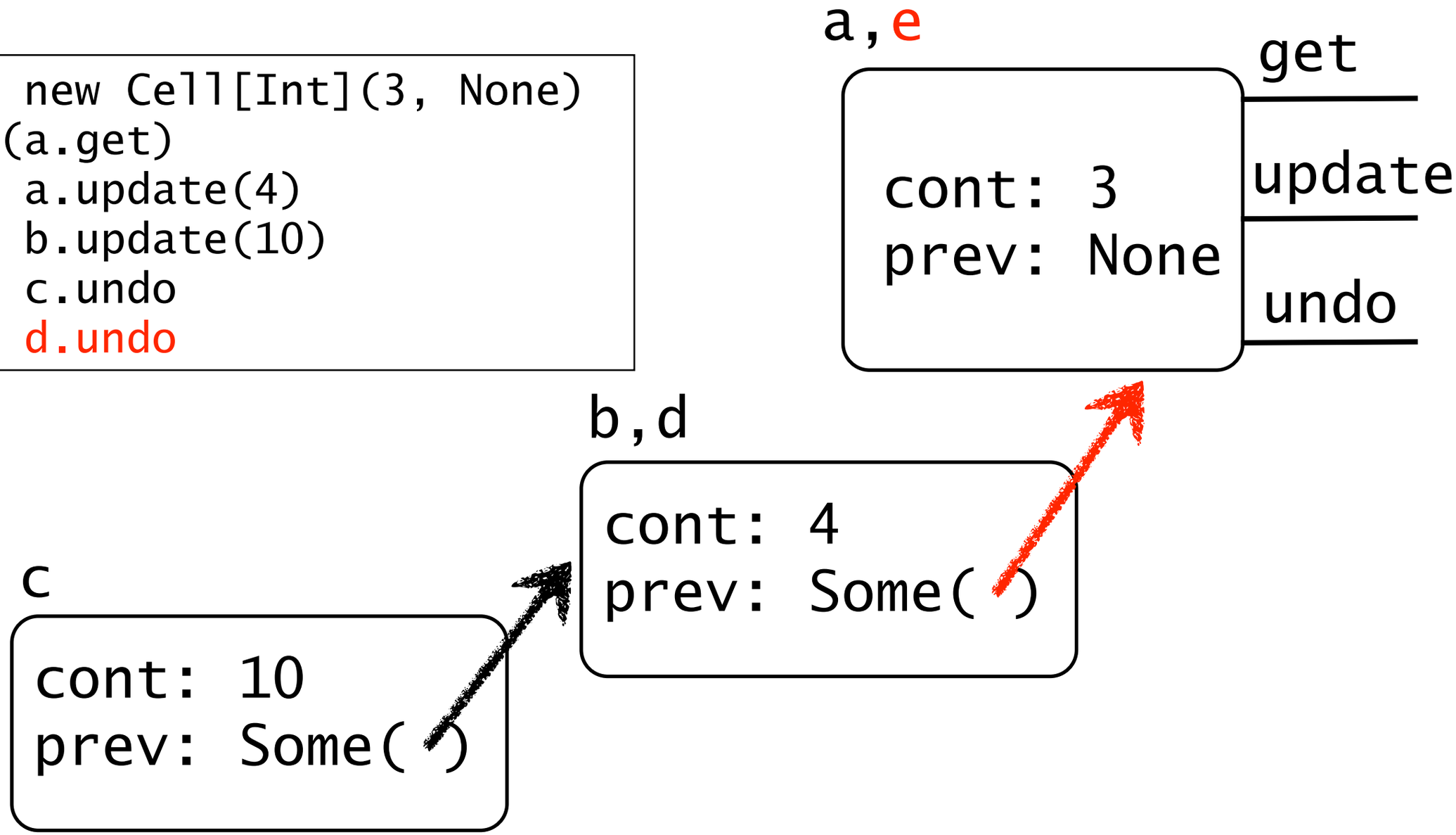
Data structure

```
val a = new Cell[Int](3, None)
println(a.get)
val b = a.update(4)
val c = b.update(10)
val d = c.undo
```



Data structure

```
val a = new Cell[Int](3, None)
println(a.get)
val b = a.update(4)
val c = b.update(10)
val d = c.undo
val e = d.undo
```



Exercise:
Complete the implementation

```
class Cell[T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T =
  def update(x: T): Cell[T] =
  def undo: Cell[T] =
}
```

Exercise:
Complete the implementation

```
class Cell[T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell[T](x,Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Exercise:
Complete the implementation

```
class Cell[T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x,Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Exercise:
Complete the implementation

```
class Cell[T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x,Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Little or actually no assumption on T made here.
A distinguishing characteristic of a generic class.

Client' response:

- Your `Cell` class is broken.

Client' response:

- Your `Cell` class is broken.
- The compiler rejects my code below.

```
class Person { var birth = 0 }
class Banker extends Person { var bonus = 0 }

def predict(p: Cell[Person]) =
  if (p.get.birth < 22) “Success“ else “Fail“

val b = new Banker; b.bonus = 100; b.birth = 30
val cellB = new Cell[Banker](b, None)
predict(cellB)
```

Client' response:

- Your `Cell` class is broken.
- The compiler rejects my code below.

```
class Person { var birth = 0 }
class Banker extends Person { var bonus = 0 }

def predict(p: Cell[Person]) =
  if (p.get.birth < 22) “Success“ else “Fail“

val b = new Banker; b.bonus = 100; b.birth = 30
val cellB = new Cell[Banker](b, None)
predict(cellB)
```

Why does the compiler complain?

Client' response:

- Your `Cell` class is broken.
- The compiler rejects my code below.

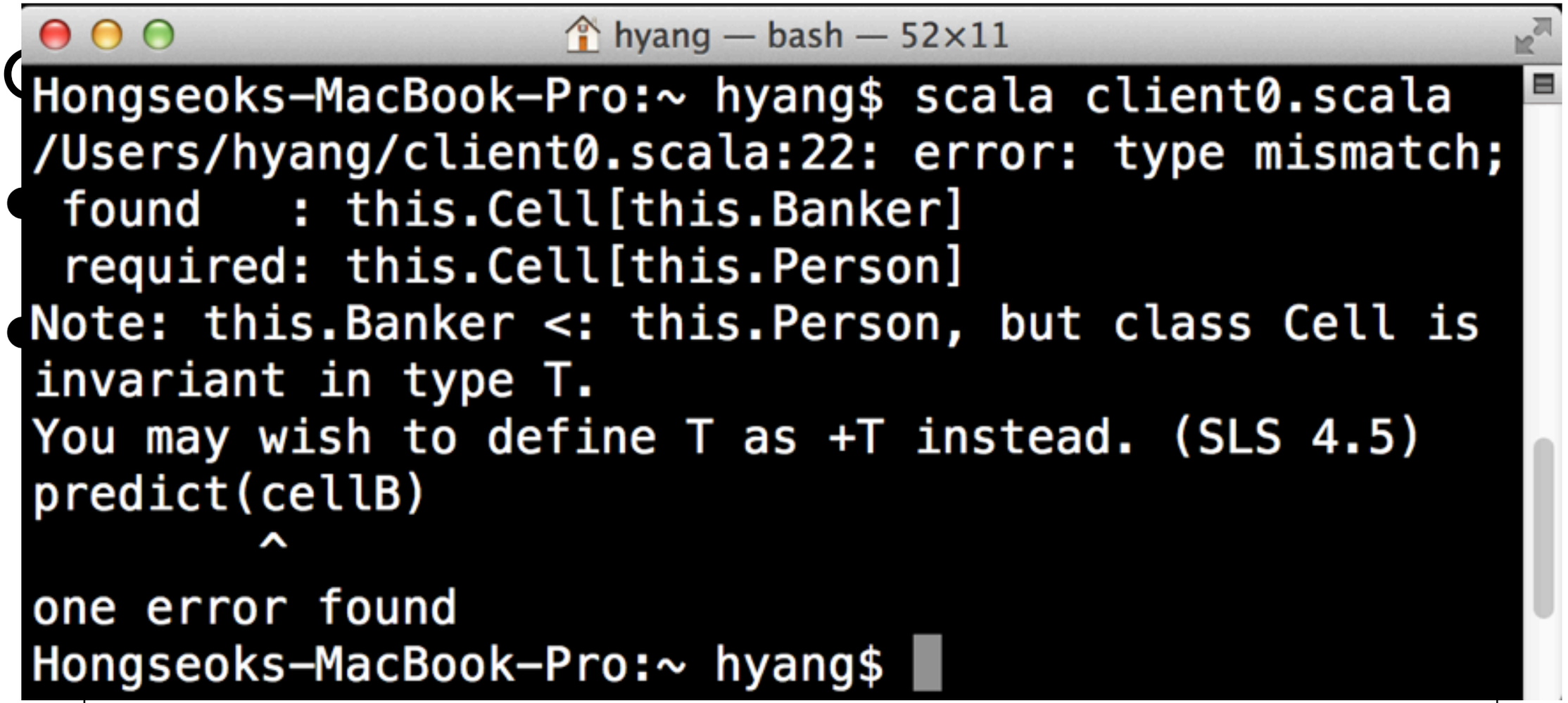
```
class Person { var birth = 0 }
class Banker extends Person { var bonus = 0 }

def predict(p: Cell[Person]) =
  if (p.get.birth < 22) “Success“ else “Fail“

val b = new Banker; b.bonus = 100; b.birth = 30
val cellB = new Cell[Banker](b, None)
predict(cellB)
```

Why does the compiler complain?

- We don't have `Cell[Banker]<:Cell[Person]`.
- That is, `Cell[..]` doesn't preserve subtyping.




```
val b = new Banker; b.bonus = 100; b.birth = 30
val cellB = new Cell[Banker](b, None)
predict(cellB)
```

Why does the compiler complain?

- We don't have `Cell[Banker]<:Cell[Person]`.
- That is, `Cell[..]` doesn't preserve subtyping.

Generics and subtyping

Subtyping C <: D

- Intuition 1 (convertibility) -- We can convert an object of type C to that of type D.
- Intuition 2 -- An object of type C supports all methods in type D.

Variance

- Variance describes how a generic class/trait C[..] interacts with subtyping.
- Covariant: $A <: B \implies C[A] <: C[B]$.
- Contravariant: $A <: B \implies C[B] <: C[A]$.
- Invariant: No subtyping relationships between C[A] and C[B] for different A,B.

Question:
Variance of Option[T]?

- Co.: A <: B $\implies$ Option[A] <: Option[B].
- Contra.: A <: B $\implies$ Option[B] <: Option[A].
- Invariant: None of the above.

Question:
Variance of Option[T]?

- Co.: A <: B $\implies$ Option[A] <: Option[B].
- Contra.: A <: B $\implies$ Option[B] <: Option[A].
- Invariant: None of the above.
- Answer: Covariant.

Question:
Variance of List[T]?

- Co.: A <: B $\implies$ List[A] <: List[B].
- Contra.: A <: B $\implies$ List[B] <: List[A].
- Invariant: None of the above.

Question:
Variance of List[T]?

- Co.: A <: B $\implies$ List[A] <: List[B].
- Contra.: A <: B $\implies$ List[B] <: List[A].
- Invariant: None of the above.
- Answer: Covariant.

Question:
Variance of U=>V for U?

- Co.: A <: B $\Longrightarrow$ (A=>V) <: (B=>V).
- Contra.: A <: B $\Longrightarrow$ (B=>V) <: (A=>V).
- Invariant: None of the above.

Question:
Variance of U=>V for U?

- Co.: A <: B $\Longrightarrow$ (A=>V) <: (B=>V).
- Contra.: A <: B $\Longrightarrow$ (B=>V) <: (A=>V).
- Invariant: None of the above.
- Answer: Contravariant for U.

Question:
Variance of U=>V for U?

- Co.: A <: B $\Longrightarrow$ (A=>V) <: (B=>V).
- Contra.: A <: B $\Longrightarrow$ (B=>V) <: (A=>V).
- Invariant: None of the above.
- Answer: Contravariant for U.
- What about the variance of U=>V for V?

Question:
Variance of U=>V for U?

- Co.: A <: B $\Longrightarrow$ (A=>V) <: (B=>V).
- Contra.: A <: B $\Longrightarrow$ (B=>V) <: (A=>V).
- Invariant: None of the above.
- Answer: Contravariant for U.
- What about the variance of U=>V for V?
- Answer: Covariant for V.

Question:
Variance of Array[T]?

- Co.: A <: B $\Longrightarrow$ Array[A] <: Array[B].
- Contra.: A <: B $\Longrightarrow$ Array[B] <: Array[A].
- Invariant: None of the above.

Question: Variance of Array[T]?

- Co.: A <: B $\Longrightarrow$ Array[A] <: Array[B].
- Contra.: A <: B $\Longrightarrow$ Array[B] <: Array[A].
- Invariant: None of the above.
- Answer: Invariant. Because of mutable state.

```
class Person { var birth = 0 }
class Banker extends Person { var bonus = 0 }

val x: Array[Banker] = new Array[Banker](1)
val y: Array[Person] = x; y(0) = new Person
x(0).bonus = 1
```

Defining covariant and contravariant generic classes

```
class C[+T,-U,V] ...
```

- Put + for covariance, - for contravariance, and nothing for invariance.
- Check whether C meets variance specs.
- Done by checking the positions of type variables.
  - Covariant ones only in positive positions.
  - Contravariant ones only in negative positions.

```
class Cell[T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

- Put + for covariance.

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

- Put + for covariance.
- Check: A <: B $\implies$ Cell[A] <: Cell[B].
  - Can we convert?
  - Are all methods supported?

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont:  ,
  private val prev:                    
) {
  def get:   = cont
  def update(x:  ):            = new Cell(x, Some(this))
  def undo:            =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont:  ,
  private val prev:
) {
  def get:  = cont
  def update(x:  ):          = new Cell(x, Some(this))
  def undo:           =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Positive

1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont:
  private val prev:
) {
  def get:  = cont
  def update(x:  ):          = new Cell(x, Some(this))
  def undo:
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Positive

Negative

1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont:
  private val prev: Option[
) {
  def get:  = cont
  def update(x:  ): Cell[ ] = new Cell(x, Some(this))
  def undo: Cell[ ] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Positive

Negative

1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Positive

Negative

1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x:  ): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Positive

Negative

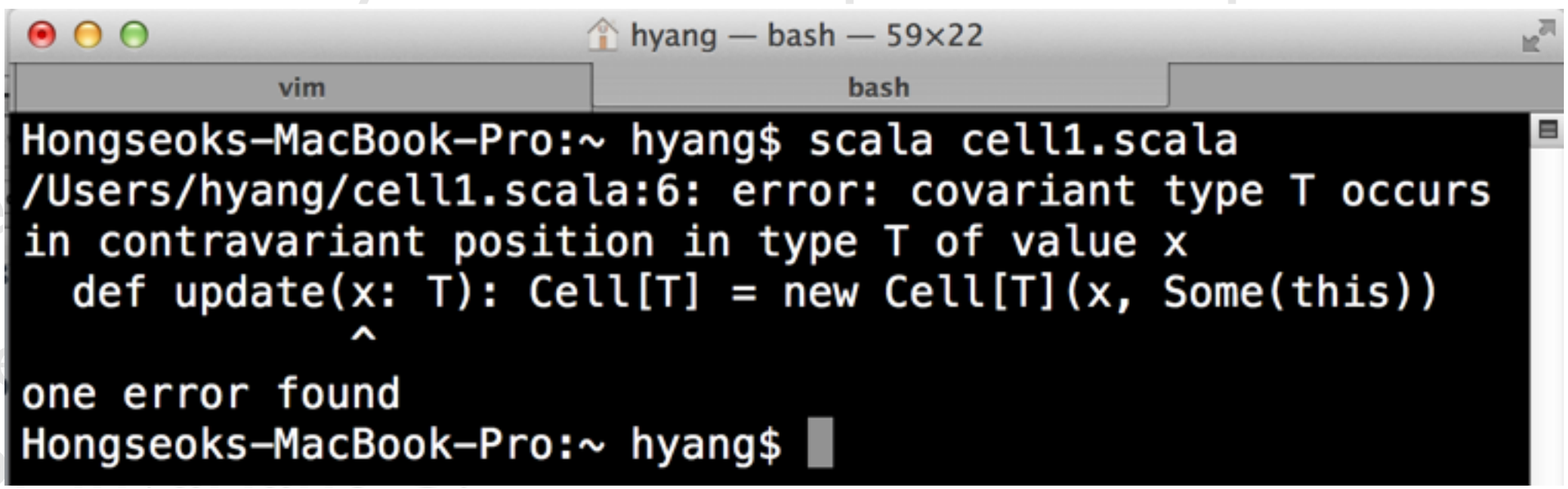
1. Going through code top-down from decl. to def'n.
2. Initially and usually the current position is positive.
3. Flipped at value parameter positions, and type parameter positions C[T] for contravariant C.
4. Becomes invariant at type parameter positions C[T] for invariant C.

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update(x: T): Cell[T] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Positive

Negative

The type variable T appears in a negative position. The compiler complains.



Lower and upper bounds

```
class C[T>:List[U]]
class D[T<:Ordered[T]]
```

```
def f[T>:List[U]]
def g[T<:Ordered[U]]
```

- `T>:List[U]` -- `T` is a superclass of `List[U]`.
- `T<:Ordered[U]` -- `T` is a subclass of `Ordered[U]`.
- Positive and negative positions are flipped at the right side of the lower bound (e.g., `List[U]` above).

Covariant generic cell

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Covariant generic cell

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Assuming A <: B, a method of type

```
def update[U>:A](x: U): Cell[U]
```

can be used as one of the following type instead:

```
def update[U>:B](x: U): Cell[U]
```

Covariant generic cell

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Flip once

Covariant generic cell

Flip again

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Flip once

Covariant generic cell

Flip again

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
}
```

Flip once

Hence, the type checker also says yes.

Exercise:
Implement copyCell

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
  def copyCell ...

}
```

```
val c1 = new Cell[Int](3,None)
val c2 = new Cell[String](“Hongseok“, None)
val c3 = c1.copyCell(c2)
println(c3.get); println(c3.undo.get)
```

```
Hongseok
3
```

Exercise:
Implement copyCell

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
  def copyCell[U>:T](c: Cell[U]): Cell[U] =
    new Cell(c.get, Some(this))
}
```

```
val c1 = new Cell[Int](3,None)
val c2 = new Cell[String](“Hongseok“, None)
val c3 = c1.copyCell(c2)
println(c3.get); println(c3.undo.get)
```

```
Hongseok
3
```

Exercise:
Implement copyCell

```
class Cell[+T](
  private val cont: T,
  private val prev: Option[Cell[T]]
) {
  def get: T = cont
  def update[U>:T](x: U): Cell[U] = new Cell(x, Some(this))
  def undo: Cell[T] =
    prev match {
      case None => throw new IllegalArgumentException
      case Some(c) => c
    }
  def copyCell[U>:T](c: Cell[U]): Cell[U] =
    new Cell(c.get, Some(this))
}
```

Type of c3?

```
val c1 = new Cell[Int](3,None)
val c2 = new Cell[String](“Hongseok“, None)
val c3 = c1.copyCell(c2)
println(c3.get); println(c3.undo.get)
```

```
Hongseok
3
```

Summary

- Generic class and trait.
- Covariant, contravariant and invariant type parameters.
- Type-checking algorithm.
- Type bounds and idiom of using a lower bound.
- Read Chap 19.