Imperative Programming 3

Lecture 10: Implementation

Peter Jeavons

Trinity Term 2019

Implementing OOP

Aim: understand how the main features of object-oriented languages are implemented and how programs are organised in memory

We'll first look at these features:

- Identity: each instance of a class has a distinct identity
- Polymorphism: classes with same *interface* can be used interchangeably
- Inheritance: class may be defined as extension of another

Identity

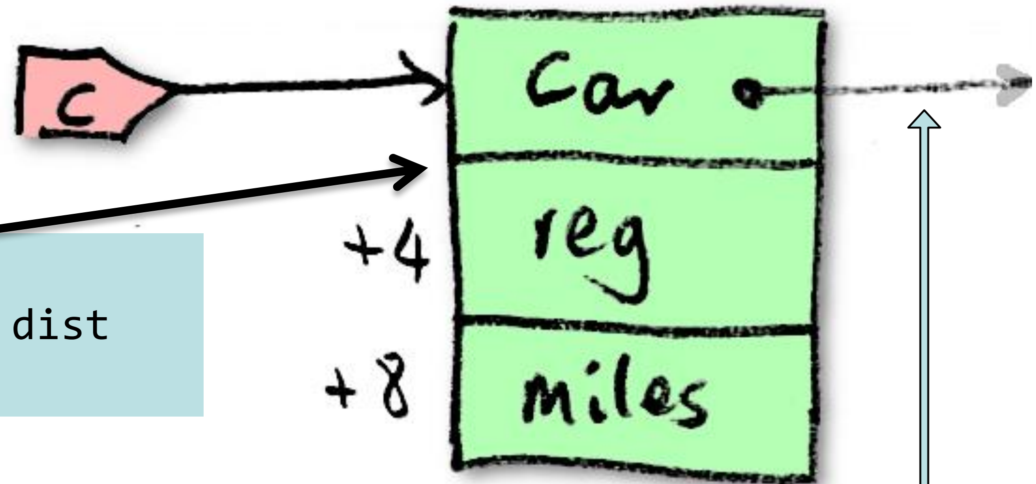
Object identity is simply the address of the chunk of memory allocated to the object

- Not to be confused with the hash code which is printed
- Good enough for C++ (user-driven memory management)
- In Scala/Java the garbage collector dynamically *moves objects around* to remove unused objects
  - Keeps the free memory contiguous
  - Objects *change addresses* dynamically
  - What if another object is using one which has moved?
- Hence Scala/Java make it impossible to get address

Objects in memory

```
class Car {
  private var reg: Int
  private var miles: Int

  ...

  def drive(dist: Int) {
     this.miles = this.miles + dist
  }
}
```

```
val c = new Car
```



- Each object has a (shared) pointer to a class descriptor
- Instance variables allocated at fixed offsets from base
- Each method has implicit parameter `this`
- The `miles` field has the address `this+8`

Polymorphism

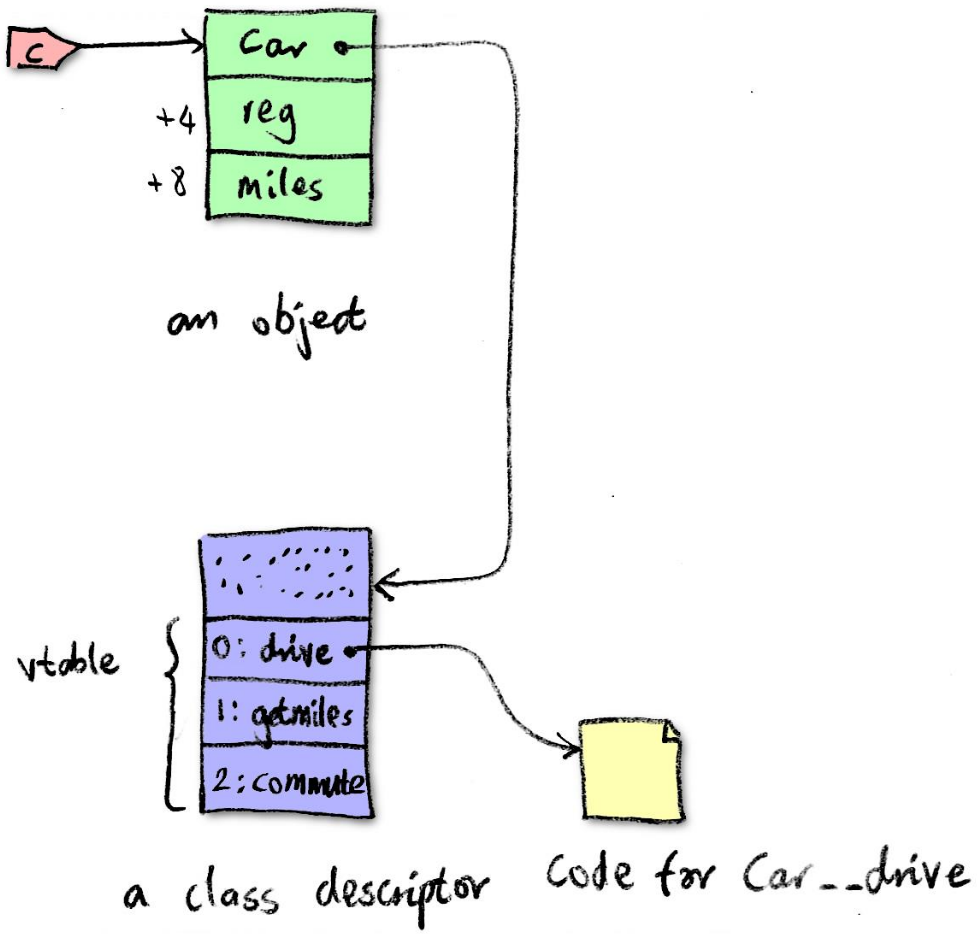
- Recall that which method is called on an object is determined by the *dynamic type* not the *static type*

```
var v: Vehicle
var c: Car
v=c
v.drive(100)
```

- Here the v is a parent *trait* (or class)
- Compiler verifies static type (i.e. that `Vehicle` has `drive`)
- Runtime uses the dynamic type (`drive` method in `Car`)

```
v.drive(100)
```

Method invocation



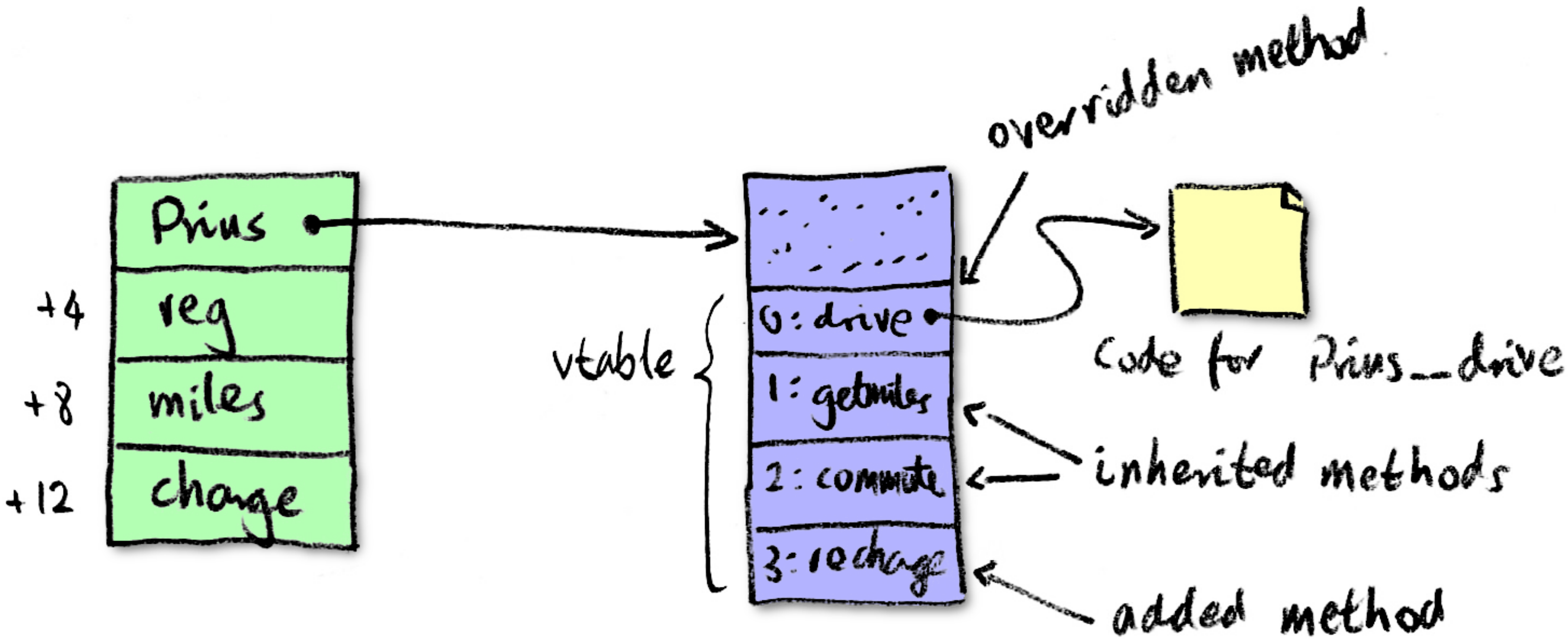

```
v.drive(100)
```

is shorthand for

```
v.class.vtable[0](v, 100)
```

Inheritance

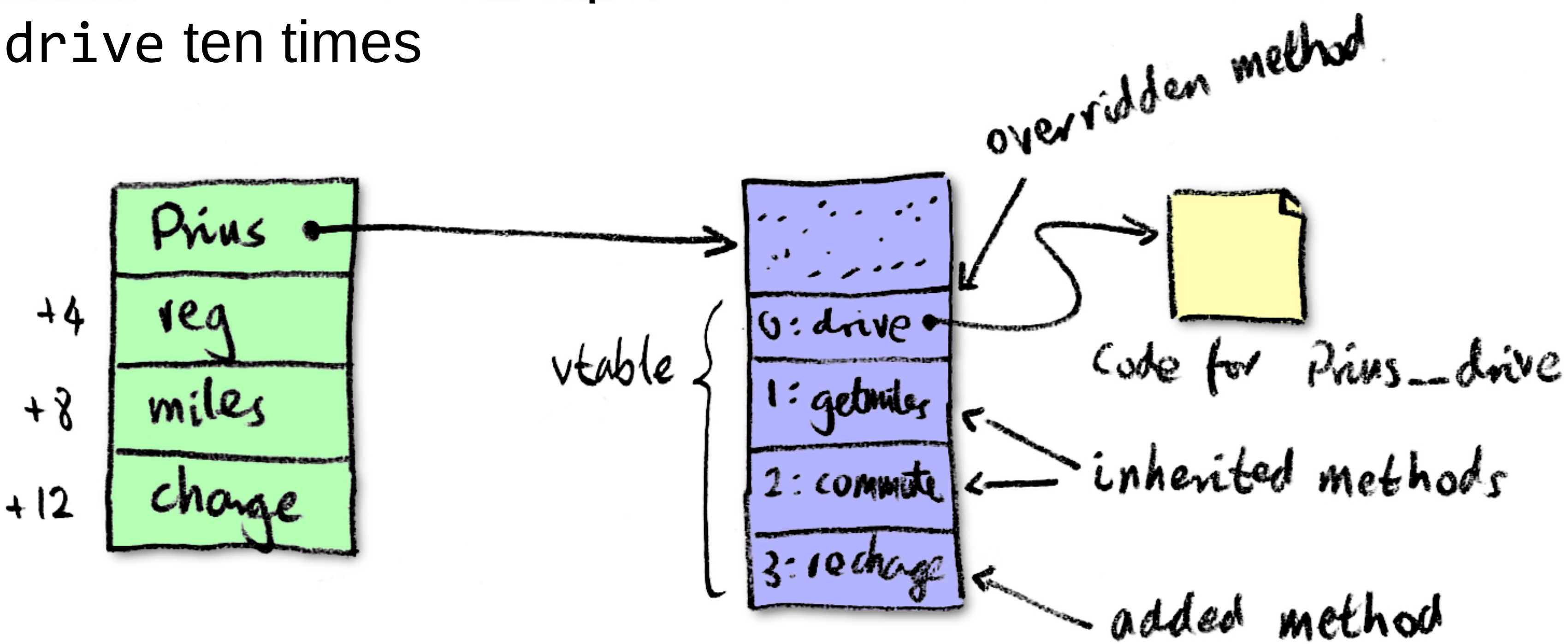
- In a subclass: extend superclass' object record and vtable
- Instance variables are inherited (and extended)
- Methods are inherited, extended or overridden
- Dynamic dispatch relies on consistent method indexing



Late binding of methods

```
// In class Car
  def commute() {
    for (i <- 1 to 10) this.drive(50)
  }
```

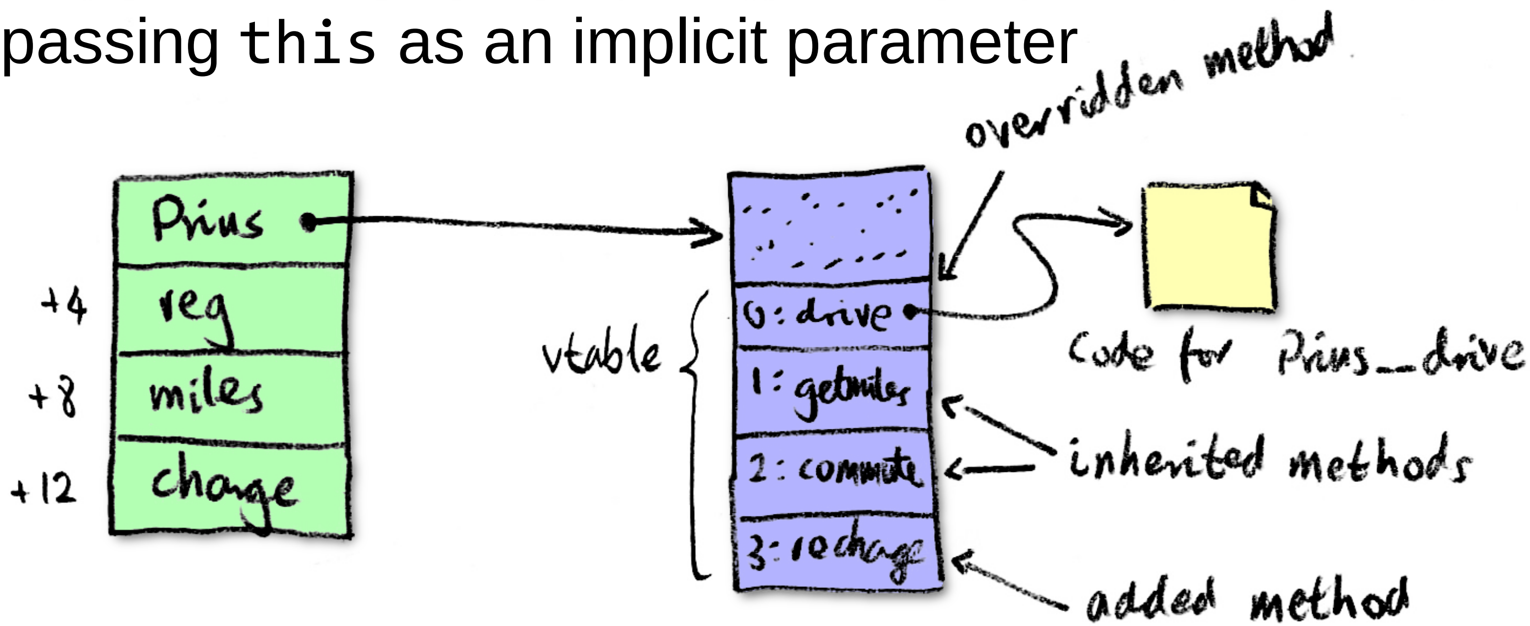
Consequence: calling `commute` on a `Prius` object will invoke `commute` in the `Car` superclass and call the `Prius` version of `drive` ten times



Super calls

```
// In class Prius
  override def drive(dist: Int) {
    super.drive(dist)
    this.charge = this.charge – dist/10
}
```

- This `drive` method behaves as before & reduces charge
- ‘Super call’ unambiguously refers to `Car` version of drive passing `this` as an implicit parameter



Some problems

- The *fragile binary interface problem*
  - If we change `Car` class then its record or `vtable` layout may change
  - Changes in offsets mean that all subclasses of `Car` should be recompiled
  - In Scala the Java Virtual Machine (JVM) delays layout until the program is loaded which adds an overhead
- The *virtual lookup problem*
  - Each virtual call requires a least one extra dereference which adds an overhead at runtime
  - Can be mitigated by an optimising compiler inlining some virtual calls (thus avoiding a `vtable` lookup)
  - Can also be mitigated by the programmer e.g. avoid overridden method calls inside tight loops and instead place the loop inside the method
- Main message: advanced language features have a runtime cost…
  - …but possibly the cost is not so high as some claim

Memory management

Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations.

- Memory-related bugs are among the most prevalent and difficult to catch of all software bugs
  - especially, in programs written in an unsafe language (C/C++)
  - but can also arise in Java or Scala programs

Storing Objects in Main Memory

- Objects must be stored somewhere
  - in main memory
  - storing objects in secondary storage is a separate problem
- Memory allocation strategy:
  - When/how is memory allocated/released?
  - Who takes care of these actions?
    - programmer vs. the “language”
- Three common types of memory allocation:
  - static
  - stack-based
  - heap-based

Static Allocation

- Memory is reserved at compile time
  - constants (e.g., arrays) are stored in the executable
  - memory is allocated when program starts
  - memory is released when program terminates
- Main benefit: efficiency
- Drawbacks:
  - program can use only a fixed amount of memory
  - inflexible for representing complex structures
  - prevent recursion
- Available in most languages
  - Fortran, C, C++, Ada

Static Memory Allocation in Java

- Class fields can be made `static`
  - they exist only once per class (AKA singletons)
  - i.e., different objects share the same field
- Example: want to count how many Region objects have been allocated

```
public class Region {
   protected static int regionsNo;
   static {
      regionsNo = 0;
   }

   public Region(...) {
      regionsNo++;
      ...
   }
}
```

Only one `regionsNo` field exists for all instances of Region.

Static initializer: code that is executed when a class is loaded to initialize the class.

Static Memory Allocation in Scala

- Companion objects - Scala's equivalent of `static`
- The companion object of a class
  - has the same name as the class
  - is defined in the same file as the class

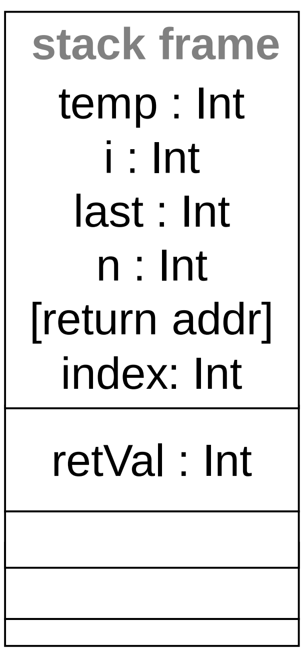
```
object Region {
  private var regionsNo = 0
  def count = regionsNo
}

class Region(...) {
  Region.regionsNo += 1
}
```

The object and class can access each other's private fields and methods.

Access to the fields and methods in the companion object must be qualified with the name of the object.

Stack-based Allocation

- Each thread of execution has a stack of frames
- A frame is allocated when a method is called
  - Contains return address, parameters, local variables, and return value
- The frame is popped off the stack when the method ends




```
def fibonacci(index: Int): Int = {
  if (index == 0) return 0
  var n = 1; var last = 0
  for (i <- 1 until index) {
    val temp = n
    n = n + last
    last = temp
  }
  return n
}
```

Stack-based Allocation

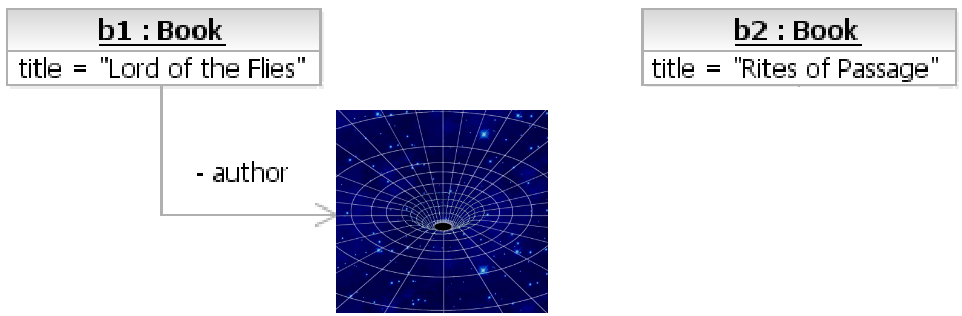
- Some OO languages (e.g., C++) can store objects on the stack
  - objects are constructed when a method starts
  - objects are destroyed when a method ends
  - using references to objects on stack causes problems
    - references become invalid when a method terminates
- Java/Scala cannot store objects on the stack
  - avoids the problem of references to objects on stack
  - stack frames in Java/Scala…
    - … store only local variables
    - … cannot be manipulated directly by the programmer

Heap-based Allocation

- Main way of allocating memory in Java/Scala
- Heap: global memory pool
  - maximum size determined by -Xmx JVM option
- Memory is allocated as needed using `new`
- But when to release memory?
  - upon termination
  - must release memory as it becomes unused
    - programs such as servers will otherwise run out of memory
- Two approaches to memory reclamation:
  - explicit (i.e., manual) reclamation
  - automatic garbage collection

Manual Memory Reclamation

- In some languages, unused objects must be removed from memory explicitly
  - e.g., in C++, one can write `delete b2.author`



- If we `delete b2.author`, we must update `b1.author` as well
- If we forget, then `b1.author` points to an invalid object
  - AKA dangling reference
  - reading from `b1.author` returns an arbitrary value
  - writing into `b1.author` results in a run-time error

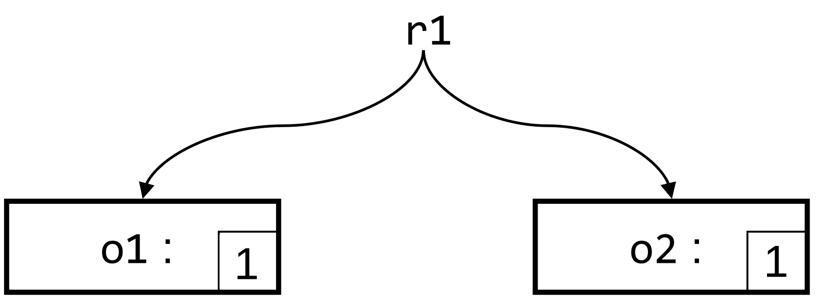
the most common source of bugs in practice!

Automatic Garbage Collection

- Alternative: release memory automatically when it becomes unused
  - typically done by the virtual machine or the runtime library
  - AKA (automatic) garbage collection
- Benefits:
  - no problems with dangling references
  - no need to worry about when objects become unused
  - less error-prone
- Main drawback: efficiency
  - overhead in determining which objects are unused
  - unsuitable for systems with critical performance
- Fundamental question: what is unused memory?

Reference Counting

- Main idea:
  - associate a counter `o.cnt` with each object `o`
  - increase `o.cnt` when attaching `o` to a reference
  - decrease `o.cnt` when detaching `o` from a reference
  - if `o.cnt` is zero, then `o` is unused



Now `o1.cnt = 0`, so `o1` is unused and its memory can be reclaimed.

`r1 = new Object()`

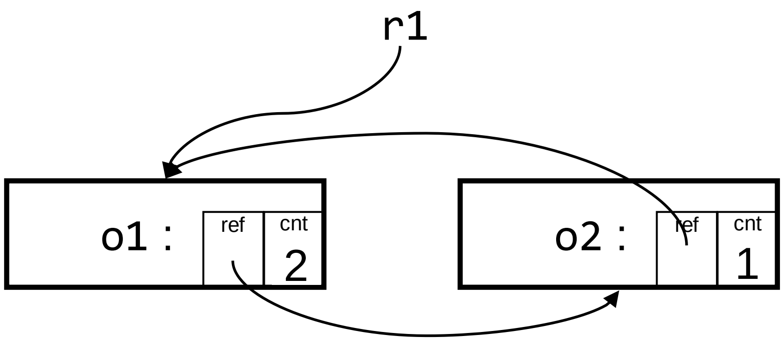
- create `o1`
- assign `o1` to `r1`
- increase `o1.cnt`

`r1 = new Object()`

- create `o2`
- decrease `o1.cnt`
- assign `o2` to `r1`
- increase `o2.cnt`

Limits of Reference Counting

```
class A {
  var ref: A = null
}
```



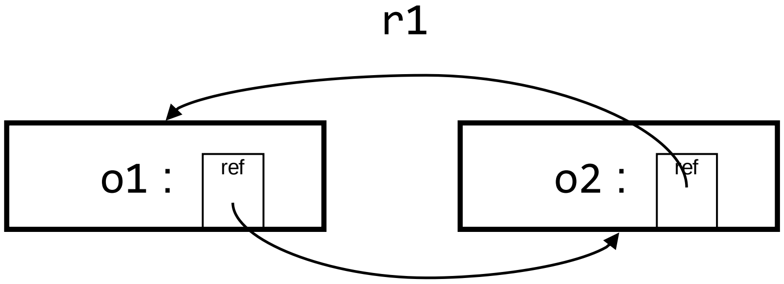

```
r1 = new A()
r1.ref = new A()
r1.ref.ref = r1
r1 = null
```

- Both o1 and o2 are unused,…
- …but their counters are 1
  - the memory of o1 and o2 never get released

Reference counting cannot deal with cyclic references

Unreachable Objects

- An object is unused if it cannot be reached through references starting from the active code
  - i.e., starting from the references on the stack
  - to ensure an object is not reachable, set the reference to `null`
- Mark-and-sweep garbage collection:
  - stop the execution of all threads
  - phase 1: mark all objects that are reachable from the active code
  - phase 2: release all unmarked objects



Neither object is reachable from `r1`
⇒ Neither object is marked
⇒ Both objects can be reclaimed

When is Garbage Collection Run?

- JVM offers no guarantees!
  - can be source of performance problems:
    - the program needs to do something promptly, but the JVM is running garbage collection
- Garbage collection is typically run when…
  - …the available heap memory is exhausted
  - …the program is idle
- Can be requested manually
  - `System.gc()`

Memory Leaks

- Memory leak: unused memory is not reclaimed due to programmer's oversight
  - common problem in C++ and other languages with explicit memory mgmt
  - But can this happen in languages with automatic memory management (e.g., in Java or Scala)?
- Example:
  - creating two `Region` objects with the same (x, y, width, height) wastes memory
  - solution: internalize `Region`
    - create a global cache of `Region` objects
    - reuse the object with the same (x, y, width, height)
  - internalization is a common memory usage optimization technique
    - internalized objects are still value objects
    - Java strings provide a similar mechanism

Internalizing `Region`

```
object Region {
  protected val regions = new mutable.HashMap[Region, Region]()
}
```

`regions` is common to all instances of `Region`

```
class Region(...) {

   // ... must implement equals() and hashCode()

  def internalize(): Region =
    Region.regions.get(this) match {
      case Some(r) => r
      case None => {
        regions(this) = this
        this
    }
}
```

- Looks for an equal object in `regions`
- If not found, then registers itself there
- Returns the object from `regions`

Internalization and Memory Leaks

```
val r1 = new Region(10, 10, 40, 40).internalize()
val r2 = new Region(10, 10, 40, 40).internalize()
assert(r1 eq r2) // AnyRef.eq - reference equality
```

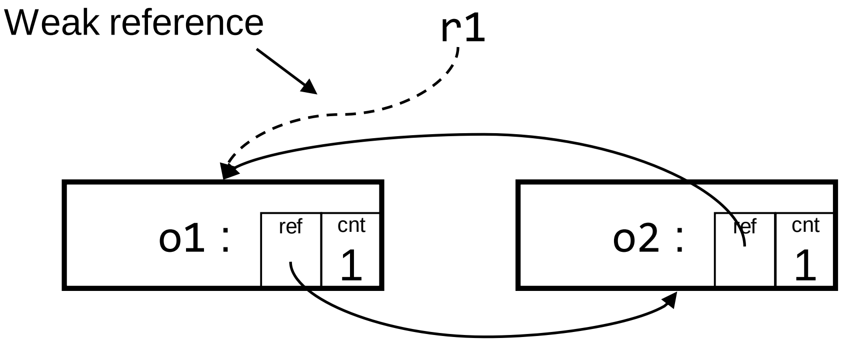
- Assertion holds!
  - internalization guarantees that each object exists only once
- But internalized objects are never freed!

```
r1 = null
r2 = null
```

  - releasing explicit references does not help:
    - `regions` holds an active reference to the region (10, 10, 40, 40)
    - static objects are reachable from active code
- $\Rightarrow$ With time, internalization uses ever more memory
  - memory leak!
- Solutions:
  - do nothing – acceptable if memory exhaustion is unlikely
  - use weak references

Weak References

- Weak reference: does not prevent an object from being garbage collected
- Example:



▪ o1 and o2 are not reachable through strong references

$\Rightarrow$ they are unused and can be freed

Internalization w/o Memory Leaks

```
object Region {
  protected val regions =
    new mutable.WeakHashMap[Region, WeakReference[Region]]()
}

class Region(...) {

   // ... must implement equals() and hashCode()

  def internalize(): Region =
    Region.regions.get(this) match {
      case Some(r) if r.get != null => r.get
      case _ => {
        regions(this) = new WeakReference[Region](this)
        this
    }
}
```

java.lang.ref.WeakReference

Checks if the object has not been garbage collected

Source of Memory Leaks in Java/Scala

- Memory leaks occur when objects are added to long-lived collections
  - static collections
  - collections that exist as long as the application
    - associated with windows and GUI elements
  - improper usage or bugs in third-party libraries
  - improper usage of the Observer pattern
    - common in GUI applications
- Memory leaks can occur in native code

Finalizers

- `finalize()`:
  - called before an object is garbage collected
  - provides an opportunity for cleanup
- Example: files must be closed after use

```
class FileInputStream extends InputStream {
  ...
  override def finalize(): Unit = {
    if (...) {    // checks whether the file is open
      close()
    }
  }
}
```

  - without the finalizer, a `FileInputStream` object could be garbage-collected without releasing the resources
    - results in resource leak!

Nondeterministic Nature of Finalizers

```
object A {
  private var numFinalized = 0

  def main(args: Array[String]) = {
    for (i <- 0 until 1000000) {
        val a = new A
    }
    System.out.println(A.numFinalized)
  }
}
class A {
  override def finalize(): Unit = {
    A.numFinalized += 1
  }
}
```

Five runs on my machine:

571222
562831
522754
270059
554158

Releasing Resources

- Do not rely on finalizers to release resources!
  - finalizers are not destructors from C++!
  - no guarantee when or if finalizers are invoked
- Release resources explicitly as soon as possible
  - e.g., close a file as soon as you are done using it
  - add an explicit method for releasing resources
    - e.g., `close()`, `dispose()`
- Add finalizers whenever possible as an "insurance" against careless programmers

Releasing Resources and Exceptions

- Resources are often not released due to exceptions
  - solution: add a `finally` block

If an exception is thrown here, `os` is not closed!

```
val os: FileOutputStream = ...
os.write(...)
os.close()
```

close() is called even if an exception is thrown.

```
val os: FileOutputStream = ...
try {
  os.write(...)
}
finally {
  os.close()
}
```

Summary

- Memory is divided into static, stack and heap
- Java/Scala objects are stored on heap
- Manual memory reclamation is error-prone
- Garbage Collection automatizes memory reclamation
  - detects and disposes unreachable objects
  - but cannot prevent memory leaks
- Weak references suitable for caches
- Releasing resources as soon as not needed