Further Java topics

The `volatile` modifier

```
static bool signal = false;

public void run() {
  while (!signal) {
    doSomething();
  }
}
```

If some other thread sets the `signal` field to `true` then what will happen?

- The thread running the code above may keep executing the `while` loop

- This might happen if the JVM produces machine code that loads the value of `signal` into a processor register and just tests that register value each time around the loop

→ Such behaviour is valid and may help performance

The `volatile` modifier (2)

`volatile` is a modifier that can be applied to fields, e.g.

```
volatile static bool signal = false;
```

When a thread reads or writes a `volatile` field it must actually access the memory location in which that field's value is held

The precise rules about when a value held in a register may be re-used are still being formulated. However, in general, if a shared field is being accessed then either:

- the thread updating the field must release a mutual exclusion lock that the thread reading from the field acquires,
- or the field should be `volatile`.

Note that the first condition is satisfied by the usual use of `synchronized` methods → `volatile` is therefore rarely seen in practice

For more details: section 2.2. of Doug Lea's book (online at `http://gee.cs.oswego.edu/dl/cpj/jmm.html`)

Garbage collection

As with Standard ML, a Java program does not need to explicitly reclaim storage space from objects and arrays that are no longer needed

```
class Loop {
  public static final void main (String args[]) {
    while (true) {
      int x[] = new int[42];
    }
  }
}
```

- This code will run forever without any problems and without requiring additional storage space for each iteration of the loop
- The *garbage collector* is responsible for identifying when storage space can be reclaimed

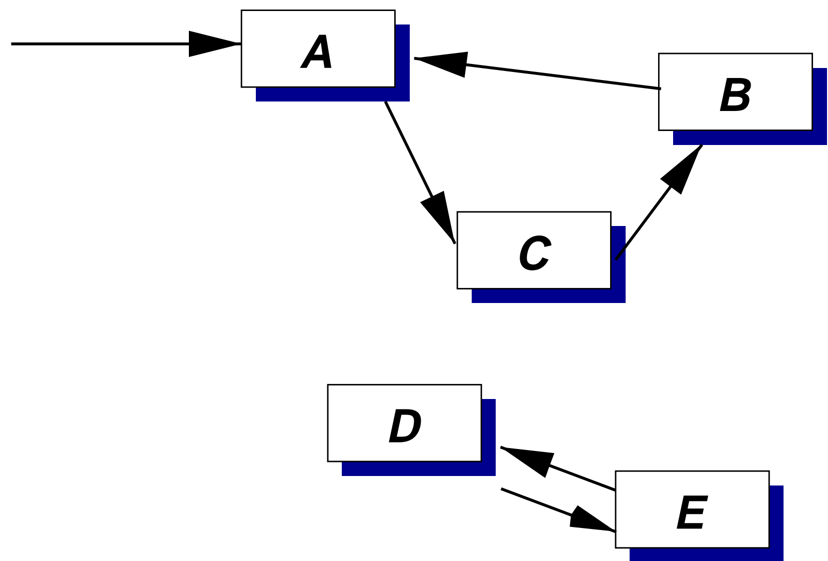
```
$ java -verbosegc Loop
[GC 511K->95K(1984K), 0.0032864 secs]
[GC 607K->95K(1984K), 0.0008373 secs]
[GC 607K->95K(1984K), 0.0002556 secs]
[GC 607K->95K(1984K), 0.0002518 secs]
[GC 607K->95K(1984K), 0.0002451 secs]
```

Garbage collection (2)

There are lots of different techniques that might be used to implement the garbage collector (see Part 1B DS&A, Part 2 Advanced Algorithms)

The JVM guarantees that storage space will not be reclaimed while an object remains *reachable*, i.e. if it

- is referred to by a `static` field in a class,
- is referred to by a local variable in a running thread,
- is referred to by from another reachable object,
- *still needs to be finalized*



Objects `A`, `B`, `C` are all reachable. Objects `D` and `E` are not

Finalizers

When the GC detects that an object is otherwise unreachable (e.g. `D` and `E` on the previous slide) then it can run a *finalizer* method on it. These are ordinary methods that override a default version defined on `java.lang.Object`

```
protected void finalize() throws Throwable { }
```

Why might this be useful?

- To perform some clean-up operation
  - although the GC can reclaim the storage space allocated to the object, it will not be able to reclaim other resources associated with it
  - e.g. if a TCP connection is set up in the constructor then perhaps the finalizer should invoke `close()` on the associated `TCPConnection` object so that the remote machine knows that the connection is no longer in use
- To aid debugging
  - e.g. to check that objects are becoming unreachable at the times at which the programmer intended

Finalizers (2)

What about examples like this? The `Restore` class implements a simple singly-linked-list:

```
class Restore {
  int     value;
  Restore next;

  static Restore found;

  Restore (int value) {
    this.value=value; this.next=null;
  }

  public void finalize () {
    synchronized (Restore.class) {
      this.next = found;
      found = this;
    }
  }
}
```

The `finalize` method will be invoked on objects once they cease to be accessible to the application...

...but it then restores access to through the static `found` field. This is perfectly safe, but very unclear

Finalizers (3)

Beware! The JVM gives few guarantees about exactly when a finalizer will be executed

- A finalizer will not be run on an object before it becomes unreachable. It is invoked *at most once* on an object
- The method `System.runFinalization()` will cause the JVM to 'make a best effort' to complete any outstanding finalizations
- There is no built-in control over the order in which finalizers are executed on different objects
- There is no control over the thread that executes finalizer methods – there may be a dedicated thread for executing them, there may be one thread per class, they may be executed by one of the threads performing garbage collection

Finalizers (and everything they access!) should be written defensively: assume that they may run concurrently with anything else and make sure that they do not deadlock or enter endless loops

Reference objects

Reference objects provide a more general mechanism for

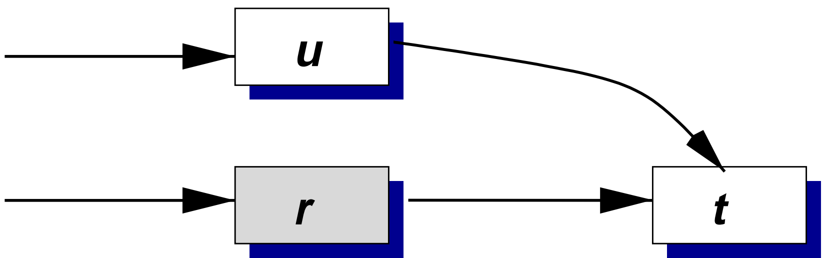
- scheduling cleanup actions when objects become unreachable via ordinary references,
- managing caches in which the presence of an object in the cache should not prevent its garbage collection, or
- accessing temporary objects which can be removed when memory is low

A reference object holds a reference to some other object introducing an extra level of indirection. The *referent* is selected at the time that the reference object is instantiated and can subsequently be obtained using the `get` method:

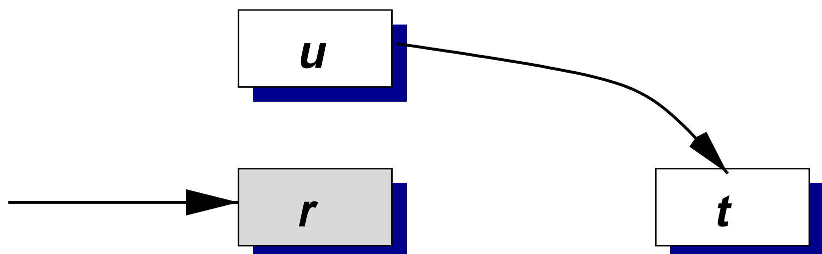
```
import java.lang.ref.*;

class RefExample {
  public static void main (String args[]) {
    int a[] = new int[42];
    Reference r = new WeakReference (a);
    System.out.println ("r: " + r);
    System.out.println (r.get());
  }
}
```

Reference objects (2)

The garbage collector is aware of reference objects and will clear the reference that they contain in certain situations. Suppose that an object (`t`) is accessible through a weak reference object (`r`) and through an ordinary object (`u`):



If `u` becomes unreachable then `t` is said to be *weakly reachable* and the GC is permitted to clear the reference in `r`:



Further calls to `r.get()` will return `null`. The reference object can be cleared explicitly by invoking `r.clear()`

✘ Traversal requires extra calls to `get()`

✔ ...but reference objects are simpler conceptually than separate 'weak reference types' to the language

Reference objects (3)

A reference object can be associated with a *reference queue* (instantiated from `java.lang.ref.ReferenceQueue`):

```
Reference r = new WeakReference (a, rq);
```

After clearing reference objects the garbage collector will (possibly some time later) append those associated with reference queues to the appropriate queue

→ it is the reference object (`r`), not the referent (`t`), that is appended to the queue

A reference queue supports three operations:

- `poll()` attempts to remove a reference object from the queue, returning `null` if none is available
- `remove(x)` attempts to remove a reference object, blocking up to `x` milliseconds
- `remove()` attempts to remove a reference object, blocking indefinitely

Reference objects (4)

There are actually three different classes defining successively weaker kinds of reference object:

- `SoftReference` – a soft reference may be cleared by the GC if memory is tight, so long as the referent is not reachable by ordinary references. Useful for memory-sensitive caches

- `WeakReference` – may be cleared by the GC once the referent is not reachable by ordinary references or soft references. Useful for hashtables from which data can be discarded when no longer in use elsewhere in the application

- `PhantomReference` – useful in combination with reference queues as a more flexible alternative to finalizers. Enqueued once the referent is not reachable through ordinary, soft or weak references and once it has been finalized. `get` always returns `null`

  In practice `PhantomReference` would be sub-classed and instances of those sub-classes would maintain any information needed for clean-up