SE 552: Lecture 5

Overview

Confinement

Confinement across methods

Confinement within threads

Confinement within objects

Confinement within groups

Confinement

We are trying to arrange for threads to have *exclusive access* to resources.

Consider the code:

```
public class Outer {
  protected final Inner contents = new Inner ();
  public synchronized void inc () { contents.inc (); }
  public int get () { return contents.get (); }
}
class Inner {
  protected int value = 0;
  synchronized void inc () { value++; }
  int get () { return value; }
}
```

Does `Outer.inc` need to be synchronized?

Does `Inner.inc` need to be synchronized?

Confinement

This is an example of *confinement*: the `contents` object never *leaks* from the `Outer` container.

Ways an outer method `m` can leak an inner `foo` object:

- Return `foo` as a result.
- Pass `foo` as an argument to a callback.
- Store `foo` in a public field.
- Leak an object `bar` which in turn leaks `foo`.

If one or more of these hold, we say `foo` *escapes*.

Confinement

Does `bar` escape from this code?

```
public class Foo {
  public final Bar bar = new Bar ();
}
```

What about in this code?

```
public class Foo {
  protected final Bar bar = new Bar ();
  public Bar get () { return bar; }
}
```

What about in this code?

```
public class Foo {
  protected final Bar bar = new Bar ();
  public int get () { return bar.get (); }
}
```

Confinement

What about in this code?

```
public class Foo {
  protected final Bar bar = new Bar ();
  public void baz (final Callback c) { c.do (bar); }
}
```

What about in this code?

```
public class Foo {
  protected final Bar bar = new Bar ();
  public void baz (final Callback c) { c.do (bar.get ()); }
}
```

Confinement

What about in this code?

```
public class Foo {
  protected final Bar bar = new Bar ();
  protected final Baz baz = new Baz (bar);
  public Baz get () { return baz; }
}
public class Baz {
  protected final Bar bar;
  public Baz (final Bar bar) { this.bar = bar; }
  public Bar get () { return bar; }
}
```

Confinement

What about in this code?

```
public class Foo {
  protected final Bar bar = new Bar ();
  protected final Baz baz = new Baz (bar);
  public Baz get () { return baz; }
}
public class Baz {
  protected final Bar bar;
  public Baz (final Bar bar) { this.bar = bar; }
  public int get () { return bar.get (); }
}
```

Confinement across methods

One form of confinement is *method* confinement.

A method m confines an object foo if:

- m creates foo.
- m does not let foo escape, except
- the very last instruction m executes may let foo escape.

This way, the thread calling m has unique access to foo during m's execution.

Confinement across methods

```
public class A {
  public void m () {
    B b = new B ();
    b.stuff ();
  }
}
class B {
  synchronized stuff () { critical section }
  ...
}
```

Does stuff need to be synchronized?

Confinement across methods

```
public class A {
  public void m (CallBack c) {
    B b = new B ();
    c.do (b);
    b.stuff ();
  }
}
class B {
  synchronized stuff () { critical section }
  ...
}
```

Now does stuff need to be synchronized?

Confinement across methods

```
public class A {
  public void m (CallBack c) {
    B b = new B ();
    c.do (b.clone ());
    b.stuff ();
  }
}
class B {
  **synchronized** stuff () { *critical section* }
  ...
}
```

Now does stuff need to be synchronized?

Confinement within threads

A brute force approach to confinement: make sure each thread gets access to a different object.

```
public interface Foo {
  public void bar ();
  ...
  public static FooFactory factory = new FooFactoryImpl ();
}
public interface FooFactory {
  public Foo build ();
}
class FooFactoryImpl implements FooFactory {
  protected final Foo singleton = new FooSingleton ();
  public Foo build () { return singleton; }
}
```

Confinement within threads

```
class FooSingleton implements Foo {
  protected final WeakHashMap cache = new WeakHashMap ();
  protected Foo get () {
    final Thread current = Thread.currentThread ();
    Foo result = (Foo)(cache.get (current));
    if (result == null) {
      result = new FooImpl ();
      synchronized (cache) { cache.put (current, result); }
    }
    return result;
  }
  public void bar () { get ().bar (); }
  ...
}
class FooImpl implements Foo {
  public synchronized void bar () { critical section which doesn’t leak this }
  ...
}
```

Does FooImpl need synchronized methods?

Confinement within threads

Why a WeakHashMap?

Java has a ThreadLocal class which does much of this code.

What are the pros and cons of this approach compared to using locks?

Confinement within objects

Another form of confinement is *object* confinement:

```
public class Outer {
  protected final Inner contents = new Inner ();
  ...
}
class Inner {
  ...
}
```

As long as the outer class doesn't leak the inner `contents` field, then the only way to access the inner object is through the outer object.

So if we synchronize the outer object, we *don't* need to synchronize the inner object!

This is *aggregation* in Patterns-speak.

Confinement within objects

Example: an adapter class:

```
public interface Counter {
  public int get ();
  public void inc ();
}
public class SafeCounter implements Counter {
  protected final Counter contents = new UnsafeCounter ();
  public synchronized int get () { return contents.get (); }
  public synchronized void inc () { contents.inc (); }
}
class UnsafeCounter implements Counter {
  protected int value = 0;
  int get () { return value; }
  void inc () { value++; }
}
```

SafeCounter is safe, even though it uses an unsafe class, because it never leaks the contents object.

Confinement within objects

What about this:

```
public interface Balance {
  public void deposit (int amount);
  public void withdraw (int amount);
  ...
}
public class SafeBalance implements Balance {
  public final Balance contents = new UnsafeBalance ();
  public synchronized void deposit (int amount) { contents.deposit (amount); }
  public synchronized void withdraw (int amount) { contents.withdraw (amount); }
  ...
}
class UnsafeBalance implements Balance {
  int balance = 0; int withdrawls = 0; int deposits = 0;
  public void deposit (int amount) { balance += amount; deposits += amount; }
  public void withdraw (int amount) { balance -= amount; withdrawls += amount; }
}
```

Confinement within groups

Aggregation works, but is often very restrictive.

Often, resources need to be passed around amongst many objects.

Example: the token solution to Dining Philosophers.

Group confinement is based on *tokens*, *batons*, *linear objects*, *capabilities*... we will call these objects *resources*.

Properites of resources:

- If a thread owns a resource, it has access to extra functionality.
- If a thread owns a resource, then nobody else does.
- If a thread gives a resource away, they no longer own it.
- If a thread destroys a resource, then no other thread ever owns it.

Example: a token ring for unique access to a printer.

Confinement within groups

```
public class Owner {
  protected Resource ref = null;
  protected final Object lock = new Object ();
  public void acquire (final ResourceFactory factory) { synchronized (lock) {
    this.ref = factory.build ();
  } }
  public void forget () { synchronized (lock) {
    this.ref = null;
  } }
  public void give (final Owner other) { synchronize (lock) { synchronize (other.lock) {
    other.ref = this.ref; this.ref = null;
  } } }
  public void take (final Owner other) { synchronize (lock) { synchronize (other.lock) {
    this.ref = other.ref; other.ref = null;
  } } }
  public void exchange (Owner other) { synchronize (lock) { synchronize (other.lock) {
      final Resource tmp = this.ref; this.ref = other.ref; other.ref = tmp;
  } } }
}
```

What about deadlock? And aren't we trying to avoid using locks?

Summary

A common problem in concurrent programming is unique access to objects.

Locks achieve this, but at the cost of possible deadlock.

May be better to achieve unique access to objects by other means, and ensuring the object does not *escape*.

‘Other means’ include: confinement within methods, threads, objects, or groups of threads.

Next week: midterm exam. Homework due in two weeks!

Week after that: Lea, Chapter 3.