Concurrent Systems and Applications

CST Part 1B, Michaelmas 2003

Programming with objects
Further Java topics
Concurrent systems
Distributed systems & transactions

Timothy L Harris

tim.harris@cl.cam.ac.uk

Lecture 1: Introduction

Course aims:

- to introduce the modular design of application software, using the facilities of the Java programming language as running examples
- to explore the need for and implementation of concurrency control and communication between processes and between threads
- to introduce the concept of transactions and their implementation and uses

Concurrent Systems and Applications was developed from the *Further Java* (and before that *Further Modula-3*) and *Concurrent Systems* courses

Where possible concrete examples and source code are used to illustrate topics in concurrent systems

More background information and principles will be given than in the old *Further Java* course

Feedback's useful at any point – either through the lab website, or e-mail *tlh20@cam.ac.uk* (or turn up at FN06).

Concurrency

‘Concurrent systems’ just means those consisting of multiple things that might be happening at the same time, e.g.

- Between the system as a whole and its user, external devices, etc.
- Between applications running at the same time on a computer – whether through context switching by the OS or by genuine parallelism on a multi-processor machine
- Explicitly between multiple threads within an application
- Implicitly within an application, e.g. when receiving call-backs through a user-interface tool-kit
- Other ‘housekeeping’ activities within an application, e.g. garbage collection

```
class Simple {
  public static void main(String args[]) {
    while (true) { }
  }
}
```

How many threads?

Concurrency (2)

- Run it in HotSpot Client VM 1.4 on Linux, interrupt it with Ctrl-backslash

```
"Signal Dispatcher" daemon prio=1 tid=0x0x807b

"Finalizer" daemon prio=1 tid=0x0x80744e0 nid=
        at java.lang.Object.wait(Native Method
        - waiting on <0x440e0490> (a java.lang
        at java.lang.ref.ReferenceQueue.remove
        - locked <0x440e0490> (a java.lang.ref
        at java.lang.ref.ReferenceQueue.remove
        at java.lang.ref.Finalizer$FinalizerTh

"Reference Handler" daemon prio=1 tid=0x0x8073
        at java.lang.Object.wait(Native Method
        - waiting on <0x440e0380> (a java.lang
        at java.lang.Object.wait(Object.java:4
        at java.lang.ref.Reference$ReferenceHa
        - locked <0x440e0380> (a java.lang.ref

"main" prio=1 tid=0x0x8051510 nid=0x519e runna
        at Simple.main(Simple.java:3)

"VM Thread" prio=1 tid=0x0x8070718 nid=0x51a1

"VM Periodic Task Thread" prio=1 tid=0x0x807a3

"Suspend Checker Thread" prio=1 tid=0x0x807ae1
```

- There are 7

Outline

Part 1 : Programming with objects

- Lecture 1: Introduction
- Lecture 2: Objects and classes
- Lecture 3: Packages, interfaces, nested classes
- Lecture 4: Design patterns

Part 2 : Further Java topics

- Lecture 5: Reflection & serialization
- Lecture 6: Graphical interfaces
- Lecture 7: Memory management
- Lecture 8: Miscellany

Outline (2)

Part 3 : Concurrent systems

- Lecture 9: Threads
- Lecture 10: Mutual exclusion
- Lecture 11: Deadlock
- Lecture 12: Condition synchronization
- Lecture 13: Worked examples
- Lecture 14: Low-level synchronization

Part 4 : Distributed systems & transactions

- Lecture 15: Distributed systems
- Lecture 16: Network sockets (TCP & UDP)
- Lecture 17: RPC & RMI
- Lecture 18: Transactions, isolation, serializability
- Lecture 19: Enforcing isolation
- Lecture 20: Crash recovery & logging

Resources

- http://www.cl.cam.ac.uk/Teaching/2003/ConcSys/
- Progamming documentation is available on the web. `http://java.sun.com/j2se/1.4.1/docs/api/`
- This includes the Java language specification + details about Java Bytecode and the Java Virtual Machine at `http://java.sun.com/j2se/1.4.1/docs/`
- The full-program examples I use are all in `$CLTEACH/tlh20/csaa-examples` on the PWF Linux system, along with the Eclipse development environment
- There's a local newsgroup `ucam.cl.java` (`nntp://ucam.cl.java` in most web browsers, or look at stand-alone news clients e.g. `trn`)

Books

These course notes are not intended as a complete reference text – either to the subject of concurrency or for practical programming in Java.

- Bacon, J. & Harris, T. (2003). *Operating Systems* Addison-Wesley.

  — Also see previous editions of Jean Bacon's book *Concurrent Systems* and the links from the teaching material page to on-line resources for it

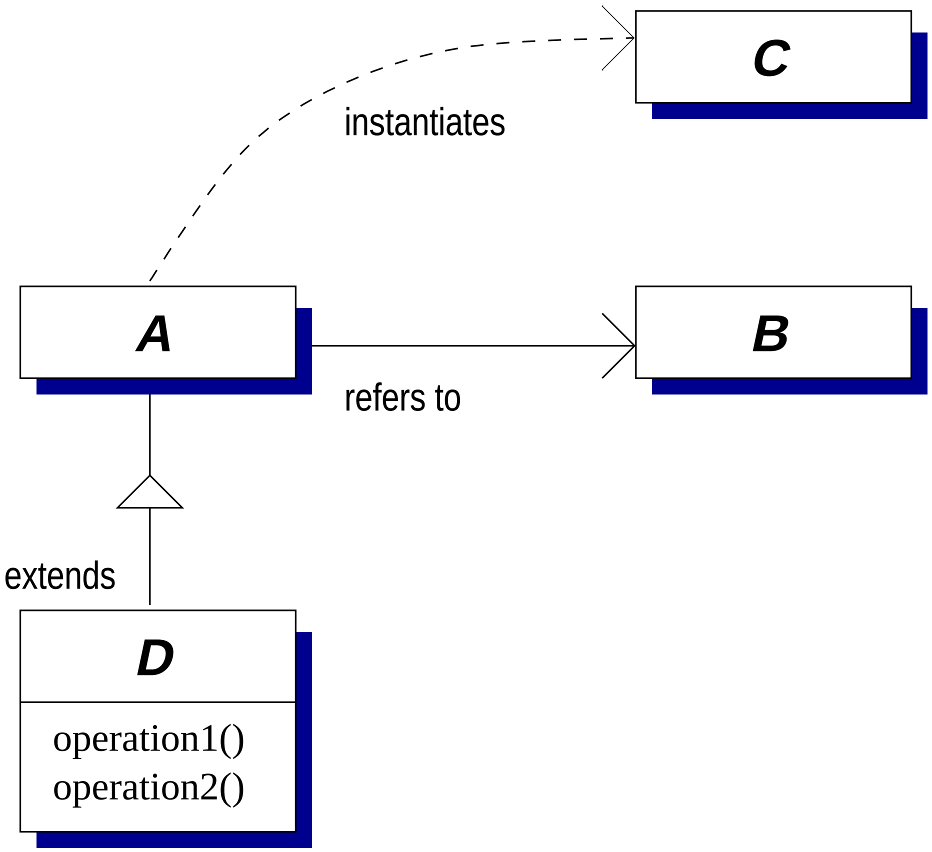
- Bracha, G., Gosling, J., Joy, B. & Steele, G. (2000). *The Java Language Specification*. Addison-Wesley (2nd ed.). `http://java.sun.com/docs/books/jls/`

- Lea, D. (1999). *Concurrent Programming in Java*. Addison-Wesley (2nd ed.)

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns*. Addison-Wesley

Notation

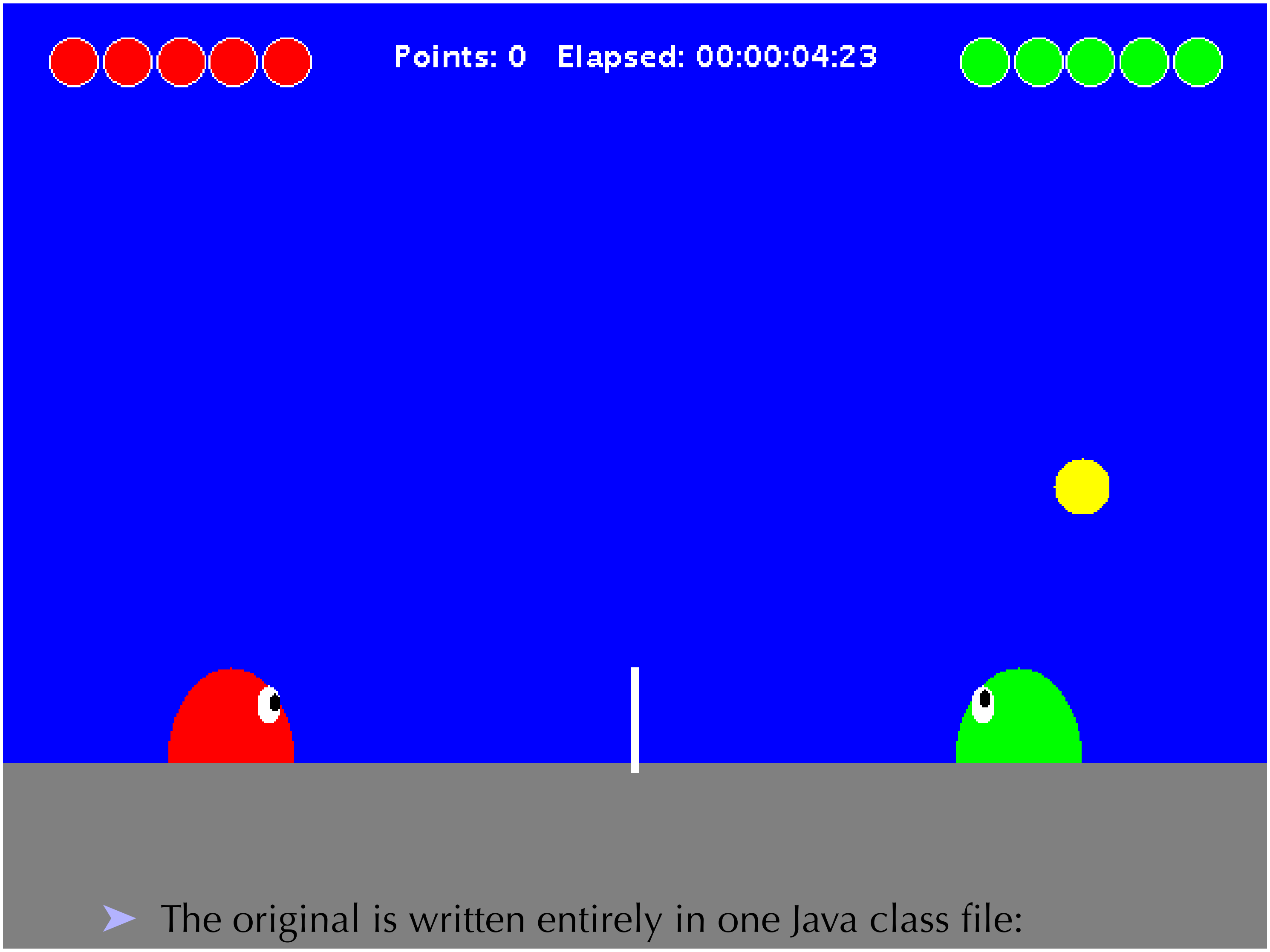
In many of the examples I've used UML-style class diagrams in which nodes represent classes and edges between them denote different kinds of relationship between those classes



The notation is consistent with Gamma *et al*'s text book; others may vary

Recap: basic Java

- The *Slime Volleyball* program will be used as a running example through the course



- The original is written entirely in one Java class file:
  - We'll restructure it into separate classes, improve the quality of the graphical display, add networked playing and an opponent-finding network service

Exercises

1-1 Compile the `Simple` class using the *javac* compiler.

1-2 Compile the source code of the initial version of *Slime Volleyball*.

The *javac* compiler will report that it uses a *deprecated* API, meaning one which still works but which should now be avoided. Practice using the on-line Java references by looking up the methods in question.

1-3 Now look at the source code of *Slime Volleyball*. Describe briefly the problems that may emerge (other than crowding around the keyboard) when changing it to a ‘doubles’ game with 2 players on each side.

Lecture 2: Objects and classes

Previous lecture

- Course structure etc.
- Recap of basic Java using the single-class *Slime Volleyball* example

Overview of this lecture

- Terminology: objects, classes, types, object references
- Composition
- Overloading methods
- Inheritance

Object-oriented programming

Programs in Java are made up of **objects**, packaging together data and the operations that may be performed on the data

For example, we could define:

```
class TelephoneEntry {
  String name;
  String number;

  TelephoneEntry(String name, String number) {
    this.name = name;
    this.number = number;
  }

  String getName () {
    return name;
  }

  TelephoneEntry duplicate() {
    return new TelephoneEntry(name, number);
  }
}
```

Object-oriented programming (2)

This example shows a number of concepts:

- Lines 1-17 comprise a complete **class** definition. A class defines how a particular kind of object works. Each object is said to be an *instance* of a particular class, e.g. line 15 creates a new instance of the `TelephoneEntry` class

- Lines 2-3 are **field** definitions. These are ordinary 'instance fields' and so a separate value is held for each object

- Lines 5-8 define a **constructor**. This provides initialization code for setting the field values for a new object

- Lines 10-12, 14-16 define two **methods**. These are 'instance methods' and so must be invoked on a specific object

This can support **encapsulation**: other parts of the program using a `TelephoneEntry` object can do so through its methods without knowing how its fields are defined

Object-oriented programming (3)

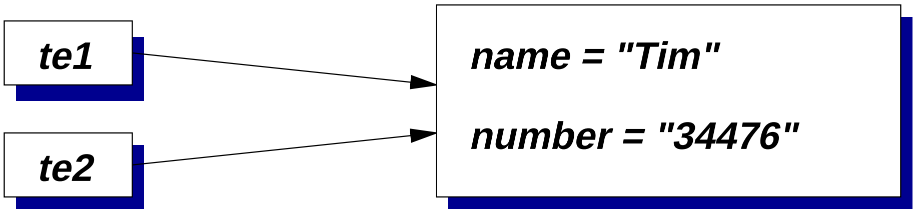
- A program manipulates objects through **object references**

- A value of an object reference type either (*i*) identifies a particular instance or (*ii*) is the special value `null`

- More than one reference can refer to the same object, for example:

  ```
  TelephoneEntry te1 =
      new TelephoneEntry ("Tim", "34476");

  TelephoneEntry te2 = te1;
  ```

  creates two references to the same object:



- If `te1.name` is updated then that new value can also be accessed by `te2.name`

Overloaded methods

- The same name can be used for more than one method in any class. They are said to be **overloaded**

- ...however, they must have distinct parameter types to disambiguate which one to call

- It is insufficient to merely have distinct return types, e.g. how would the following invocations behave?

```
void doSomething (String number) {
  this.number = number;
}

String doSomething (String c) throws IOException
{
  Runtime.getRuntime().exec(c);
  return "OK";
}

String s = o.doSomething ("rm -rf /");
o.doSomething ("12345");
```

- The choice would have to depend on the *context* in which an expression occurs

Overloaded methods (2)

- Calls to overloaded methods must also be unambiguous, e.g.

```
void f(int x, long y)
{
  ...
}

void f(long x, int y)
{
  ...
}
```

- Which should `f(10,10)` call? There is no best match

- However, unlike before, the caller can easily resolve the ambiguity by writing e.g. `f((long)10,10)` to convert the first parameter to a value of type `long`

Constructors

- Using constructors makes it easier to ensure that all fields are appropriately initialized

- If the constructor signature changes (e.g. an extra parameter is added) then other classes using the old signature will fail to compile: the error is detected earlier

- As with methods, constructors can be overloaded

```
class TelephoneEntry {
  TelephoneEntry (String name) {
    this(name, "");
  }

  ...
}
```

- Unlike methods, constructors do not have a declared return type or use the `return` statement

- A **default constructor** without any parameters is generated automatically if the programmer does not define any

Composition

- Placing a field of reference type in a class definition is a form of **composition**

- A new kind of data structure is defined in terms of existing ones, e.g.

  ```
  class TEList
  {
     TelephoneEntry te;
     TEList         next;
  }
  ```

- Used when modelling things related by a 'has a' relationship
  - e.g. a `Car` class might be expected to have a field of type `Engine` and a field of type `Wheels[]`
  - it would be less likely to have a field of type `Vehicle`

- By convention field names are spelled with an initial lower-case letter and have names that are nouns, e.g. `steeringWheel` or `leftChild`

Inheritance

- **Inheritance** is another way of combining classes – it typically models an *is a* relationship
  - e.g. between a `Car` class and a more general `Vehicle` class

- Inheritance defines a new **sub-class** in terms of an existing **super-class**. The sub-class is intended to be a *more specialized* version of the super-class. It can
  - add new fields
  - add new methods
  - provide new implementations of existing methods

```
class AddressEntry extends TelephoneEntry
{
  String addr;

  AddressEntry (String name,
          String number, String addr) {
    super(name, number);
    this.addr = addr;
  }

  ...
}
```

Types and inheritance

- Reference types in Java are associated with particular classes:

  ```
  class A {
    A anotherA; // Reference type A
  }
  ```

- Such fields can also refer to any object of a sub-class of the one named, e.g. `B` that `extends A`

- A particular object may be accessed through fields and variables of different *reference types* over the course of its lifetime; it's *class* is fixed at its time of creation

  ```
  someA.anotherA = new B();
  ```

- **Casting** operations convert references to an object between different reference types, e.g.

  ```
  1    A ref1 = new B();
  2    B ref2 = (B) ref1; // super -> sub
  3    ref1 = ref2; // no cast needed: sub -> super
  ```

- The cast in line 2 is needed because the variable `ref1` may refer to any instance of `A` or `B`. `ref2` may only refer to instances of `B`. Casts are checked for safety in Java

Arrays and inheritance

- If `B extends A` then how are `B[]` and `A[]` related?
- An array of type `A[]` can hold objects of class `B` or class `A`
- An array of type `B[]` can only hold objects of class `B`
- `B[]` is a sub-type of `A[]`

```
1   A[] array1 = new A[2];
2   B[] array2 = new B[2];
3   A[] temp;

4   temp = array1;
5   temp[0] = new B(); // A[] <- B, ok
6   temp[1] = new A(); // A[] <- A, ok

7   temp = array2;
8   temp[0] = new B(); // B[] <- B, ok
9   temp[1] = new A(); // B[] <- A, fails
```

- Line 9 fails at run-time: `array2` refers to an object that is an *array of references to things of type B* and so an object of class `A` is incompatible
- This scheme is said to be **covariant**

Fields and inheritance

- A field in the sub-class is said to **hide** a field in the super-class if it has the same name. The hidden field can be accessed by casting the object reference to the type on which the field is defined, or by writing `super.name` to get to the immediate superclass

- For example:

```
class A {
  int x;
  int y;
  int z;
}

class B extends A {
  String x;
  int y;

  void f () {
    x = "Field defined in B";
    y = 42;           // B
    super.x = 17;     // A
    ((A)this).y = 20; // A
    z = 23;           // A
  }
}
```

Methods and inheritance

A class inherits methods from its superclass

- It can overload them by making additional definitions with different signatures
- It can **override** them by supplying new definitions with the same signature

```
class A {
  void f () { }
}

class B extends A {
  void f () {
    System.out.println ("Override");
  }

  void f (int x) {
    System.out.println ("Overload");
  }
}
```

Methods and inheritance (2)

- When an overridden method is called, the code to execute is based on the *class* of the *target* object, not the *type* of the object reference

- Consequently, the type of an object reference does not affect the chosen method in these examples. A common mistake:

```
class A {
  void f () {
    System.out.println ("Super-class");
  }
}
class B extends A {
  void f () {
    System.out.println ("Sub-class");
    ((A)this).f(); // Try to call original
  }
}
```

- As with fields, the `super` keyword can be used:

```
    super.f();
```

Exercises

2-1 Write concise definitions of *object, class, object reference* and *type* with respect to a simple example in Java.

2-2 "If `s` is a sub-type of `t` then something of type `s` can be used anywhere something of type `t` can be used". Is this true for Java?

2-3 The `super` keyword can be used to access a field that has been hidden or a method that has been overridden. However, `super.super` is not valid in Java. What are the advantages and disadvantages of this restriction?

2-4* Suppose that instead of being *covariant*, array sub-typing in Java was *contravariant* – i.e. that if `B extends A` then `A[]` is considered a sub-type of `B[]`. Is this a reasonable proposition?

2-5* Is the lack of `super.super` something enforced just by the Java programming language, or also by the Java Virtual Machine?

'Starred' exercises are outside the syllabus of the course and are included as extensions or as topics for discussion

Lecture 3: Packages, interfaces, nested classes

Previous lecture

- Classes in Java
- Encapsulation
- Composition
- Inheritance

Overview of this lecture

- Packages for grouping related classes
- Modifiers and enforced encapsulation
- Interfaces & abstract classes
- Nested classes

Packages

- Java groups classes into **packages**. Classes within a package are typically written by co-operating programmers and expected to be used together

- Each class has a **fully qualified** name consisting of its package name, a full stop, and then the class name. e.g. `uk.ac.cam.cl.tlh20.TelephoneEntry`

- The `package` keyword is used to select which package a class definition is placed in, e.g.

```
package uk.ac.cam.cl.tlh20.examples;

class TelephoneEntry { ... }
```

- Definitions in the current package and `java.lang` can always be accessed. Otherwise, the `import` keyword can be used:

```
import java.util.*; // All from that package
import java.awt.Graphics; // Just named class
```

Modifiers

- This section looks at a number of **modifiers** that may be used when defining classes, fields and methods. Only access modifiers may be applied to constructors

```
<class-modifiers> class TelephoneEntry {

  <field-modifiers> String name;
  <field-modifiers> String number;

  TelephoneEntry () {
    /* Only access modifiers are allowed */
  }

  <method-modifiers> String getName () {
    return name;
  }

  <method-modifiers> String getNumber () {
    return number;
  }
}
```

Access modifiers

- Previous examples have relied on the programmer being careful when implementing encapsulation
  - e.g. to interact with classes through their methods rather than directly accessing their fields

- Access modifiers can be used to ensure that encapsulation is honoured and also, in some standard libraries, to ensure that untrusted downloaded code executes safely

	Same class	Same package	Sub-classes	Anywhere
`public`	✔	✔	✔	✔
`protected`	✔	✔	some	
`default`	✔	✔		
`private`	✔			

The `protected` modifier

- A `protected` entity is always accessible in the package within which it is defined

- Additionally, it is accessible within sub-classes (`B`) of the defining class (`A`), but only when actually accessed on instances of `B` or its sub-classes

```
 1  public class A {
 2    protected int field1;
 3  }
 4
 5  public class B extends A {
 6    public void method2 (B b_ref, A a_ref) {
 7      System.out.println (field1);
 8      System.out.println (b_ref.field1);
 9      System.out.println (a_ref.field1);
10    }
11  }
```

- Lines 7-8 are OK: `this` and `b_ref` must refer to instances of `B` or its sub-classes

- Line 9 is incorrect: `a_ref` may refer to any instance of `A` or its sub-classes

The `final` modifier

- A `final` method cannot be overridden in a sub-class – typically used because it allows faster calls to the method, but also used for security

- A `final` class cannot be sub-classed at all

- The value of a `final` field is fixed after initialization – either directly or in every constructor, e.g.

```
class FinalField {
  final String A = "Initial value";
  final String B;

  FinalField () {
    B = "Initial value";
  }
}
```

- `final` fields are also used to define constants, e.g.:

```
class ThreeColours {
  public static final int BLUE  = 1;
  public static final int WHITE = 2;
  public static final int RED   = 3;
}
```

The `abstract` modifier

- Used on class and method definitions. An `abstract` method is one for which the class does not supply an implementation

- A class is abstract if declared so or if it contains any abstract methods. Abstract classes cannot be instantiated

```
public class A {
  abstract int methodName ();
}

public class B extends A {
  int methodName () {
    return 42;
  }
}
```

- Abstract classes are used where functionality is moved into a super-class, e.g. an abstract super-class representing ‘sets of objects’ supporting iteration, counting, etc., but relying on sub-classes to provide the actual representation

- Note that fields cannot be `abstract`: they cannot be overridden in sub-classes

The `static` modifier

- The `static` modifier can be applied to any method or field definition. (It can also be applied to *nested* classes, discussed later)

- It means that the field/method is associated with the class as a whole rather than with any particular object

- For example, suppose the example `TelephoneEntry` class maintains a count of the number of times that it has ever been instantiated: there is only 1 value for the whole class, rather than a separate value for each object

- Similarly, `static` methods are not associated with a current object – unqualified instance field names and the `this` keyword cannot be used

- `static` methods can be called by explicitly naming the class within which the method is defined. The named class is searched, then its super-class, etc. Otherwise the search begins from the class in which the method call is made

The `static` modifier (2)

```
class Example {
  static int instantiationCount = 0;

  String name;

  Example (String name) {
    this.name = name;
    instantiationCount ++;
  }

  String getName () {
    return name;
  }

  static int getInstantiationCount () {
    return instantiationCount;
  }
}
```

Other modifiers

- A `strictfp` method is implemented at run-time using IEEE 754/854 arithmetic (see *Numerical Analysis 1*) – identical results are guaranteed on all computers. Can be applied to classes (⇒ all methods are then implicitly `strictfp`)

- There are four other modifiers to be covered later:
  - `synchronized` and `volatile` are used in multi-threaded applications
  - `transient` is used with the serialization API
  - A `native` method is implemented in native code – e.g. to interact with existing code or for (perceived) performance reasons. The mechanism for locating the native implementation is system-dependent

Interfaces

- There are often groups of classes that provide different implementations of the same kind of functionality
  - e.g. the *collection* classes – `HashSet` and `ArraySet` provide set operations, `ArrayList` and `LinkedList` provide list-based operations
- In that example there are some operations available on all *collections*, further operations on all *sets* and a third set of operations on the `HashSet` class itself
- Inheritance and abstract classes can be used to move common functionality into super-classes such as `Collection` and `Set`
  - Each class can only have a *single* super-class, so should `HashSet` extend a class representing the hashtable aspects of its behaviour, or a class representing the set-like operations available on it?
- More generally, it is often desirable to separate the definition of a standard programming *interface* (e.g. set-like operations) from their *implementation* using an actual data structure (e.g. a hash table)

Interfaces (2)

- Each Java class may only `extend` a single super-class, but it can `implement` a number of interfaces

```
interface Set {
  boolean isEmpty();
  void insert(Object o);
  boolean contains(Object o);
}

class HashSet implements Hashtable, Set {
  ...
}
```

- An `interface` definition just declares method signatures and `static final` fields

- An ordinary interface may have `public` or *default* access. All methods and fields are implicitly `public`

- An interface may `extend` one or more **super-interfaces**

- A class that `implements` an interface must supply definitions for each of the declared methods (or be declared an `abstract` class)

Nested classes

- A *nested* class/interface is one whose definition appears inside another class or interface

- There are four cases:

  - **inner classes** in which the enclosed class is an ordinary class (i.e. non-`static`)
  - **static nested classes** in which the enclosed definition is declared `static`
  - **nested interfaces** in which an interface is declared within an enclosing class or interface
  - **anonymous inner classes**

- Beware: the term *inner class* is sometimes used incorrectly to refer to all nested classes

  $$\text{inner classes} \subset \text{nested classes}$$

- In general nested classes are used (*i*) for programming convenience to associate related classes for readability (*ii*) as a shorthand for defining common kinds of relationship (*iii*) to provide one class with access to `private` members or local variables from its enclosing class

Nested classes (2)

- An *inner class* definition associates each instance of the *enclosed* class with an instance of the *enclosing* class, e.g.

```
1   class Bus {
2     Engine e;
3
4     class Wheel {
5       ...
6     }
7   }
```

- Each instance of `Wheel` is associated with an **enclosing instance** of `Bus`. For example methods defined at Line 5 can access the field `e` without qualification or access the enclosing `Bus` as `Bus.this`

- An instance of `Bus` must explicitly keep track of the associated `Wheel` instances, if it wishes to do so

- As with `static` fields and `static` methods, a `static` nested class is not associated with any instance of an enclosing class. They are often used to organise ‘helper’ classes that are only useful in combination with the enclosing class. Nested interfaces are implicitly static

Anonymous inner classes

- *Anonymous inner classes* provide a short-hand way of defining inner classes

```
class A {
  void method1 () {
    Object ref = new Object () {
      void method2 () { };
    };
  }
}
```

- An anonymous inner class may be defined using an interface name rather than a class name, e.g.

```
interface Ifc {
  public void interfaceMethod ();
}

class A {
  void method1 () {
    Ifc i = new Ifc () {
      public void interfaceMethod () {
      };
    };
  }
}
```

Exercises

3-1 Describe the facilities in Java for defining classes and for combining them through composition, inheritance and interfaces. Explain with a worked example how they support the principle of encapsulation in an object-oriented language.

3-2 Describe the differences and similarities between abstract classes and interfaces in Java. How would you select which kind of definition to use?

3-3 Why is it sensible that (*i*) interfaces cannot be `private`, (*ii*) methods signatures on interfaces are implicitly `public`, (*iii*) nested interfaces are implicitly static?

3-4 A common programming mistake in Java is to try to define a class to have more than one superclass. For example a naïve programmer may write

```
class FlyingHorse extends Horse, FlyingBeast
{
   ...
}
```

Describe *three* ways in which this problem can be resolved to produce (one or more) valid class definitions. What are the advantages and disadvantages of each?

Exercises (2)

3-5 An enthusiast for programming with *closures* proposes a new language, $\text{D}^\flat$, extending Java so that the following method definition would be valid:

```
Closure myCounter (int start) {
  int counter = start;
  return {
    System.out.println (counter ++);
  }
}
```

The programmer intends that no output would be made on `System.out` when this method is executed, but that it would return an object implementing a new built-in interface, `Closure`:

```
interface Closure {
  void apply();
}
```

Invoking `apply()` on the object returned by `myCounter` will cause successive values to be printed. By using a *nested* class definition, show how this example could be re-written as a valid Java program.

Lecture 4: Design patterns

Previous lecture

- Finished looking at the facilities for defining classing
- Access modifiers to enforce encapsulation
- Interfaces & abstract classes
- Nested classes

Overview of this lecture

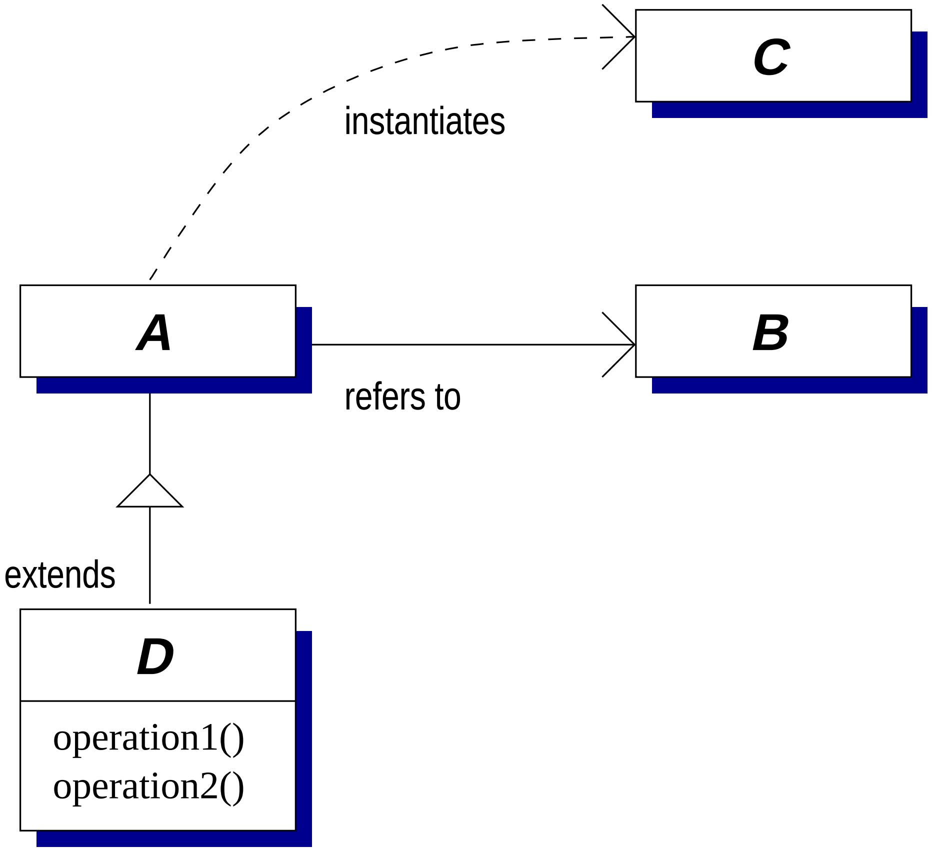
- Some ways of using these facilities effectively
- Think back to the first *Slime Volleyball* example and the difficulties there in supporting extra players, computer controlled players etc

Design patterns

A number of common idioms frequently emerge in object-oriented programming. Studying these **design patterns** provides

- common terminology for describing program organization in terms of inter-related classes

- examples of how to structure programs for flexibility and re-use



Abstract factory pattern

Suppose we've got a set of interfaces `Window`, `ScrollBar`, etc defining components used to build GUIs

There may be several sets of these components – e.g. with different visual appearance

How does an application get hold of appropriate instances of classes implementing those interfaces?

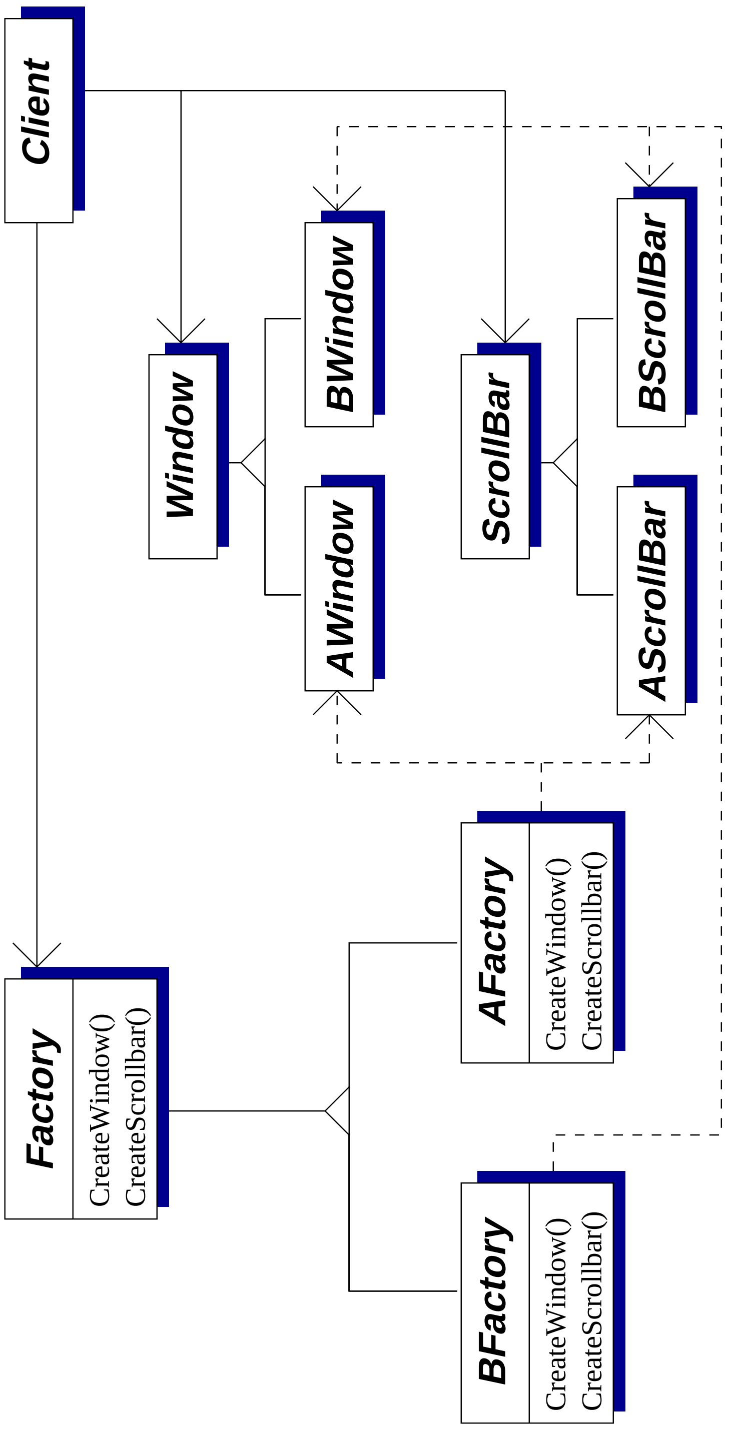
➤ Option 1: have it know about all of the options

```
switch (lf) {
  case MACINTOSH: w = new MacWindow (); break;
  case MOTIF: w = new MotifWindow (); break;
  ...
}
```

✘ All applications have to be changed to add a new family

✘ A buggy application might try to use a `MacWindow` with a `MotifScrollbar`...

Abstract factory pattern (2)



Abstract factory pattern (3)

➤ The client application invokes operations on an instance of abstract class `Factory`

➤ Code for these methods is provided by one of a number of sub-classes, e.g. `AFactory` or `BFactory`

➤ The factory class instantiates objects on behalf of the client from one of a family of related classes, e.g. `AFactory` instantiates `AWindow` and `AScrollBar`

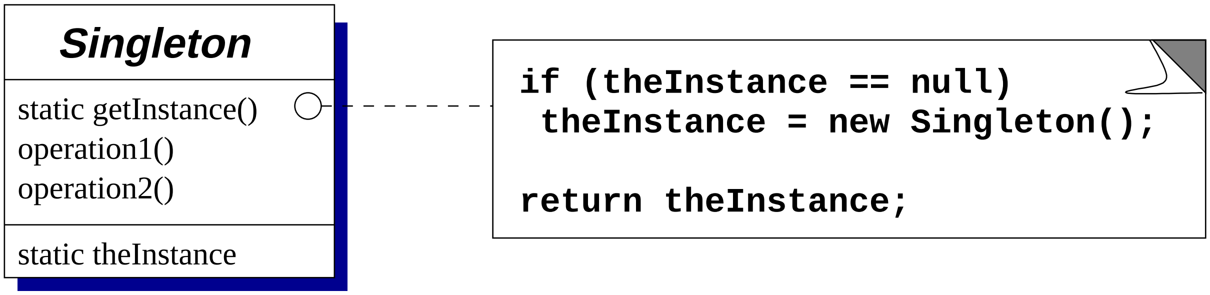
✔ New families can be introduced by providing the client with an instance of a new sub-class of `Factory`

✔ The factory can ensure classes are instantiated consistently; e.g. `AWindow` always with `AScrollBar`

✘ Adding a new operation involves co-ordinated change to the `Factory` class and all its sub-classes

...the problem hasn't entirely gone away: how does the application know which `Factory` to use?

Singleton pattern



Singleton

static getInstance()
operation1()
operation2()

static theInstance

```
if (theInstance == null)
 theInstance = new Singleton();

return theInstance;
```

- ➤ The Singleton pattern solves the *Highlander problem*: there can be only one instance of a particular class:
  - e.g. of a factory class in the previous Abstract Factory pattern

- ➤ Clients invoke `getInstance()` to retrieve the unique instance. The first invocation triggers instantiation of a (private) constructor

- ✔ More flexible than a suite of static methods: allows sub-classing, e.g. `getInstance` on `Toolkit` could return `MotifToolkit` / `MacToolkit` as appropriate

- ✔ The constraint is enforced (and could subsequently be relaxed) in a single place

- ✘ We'll return to the multi-threaded case later

Adapter pattern

➤ Suppose you've got an existing application that accesses a data structure through the `Dictionary` interface

```
public interface Dictionary {
  int size ();
  boolean isEmpty ();
  Object get(Object key);
  ...
}
```

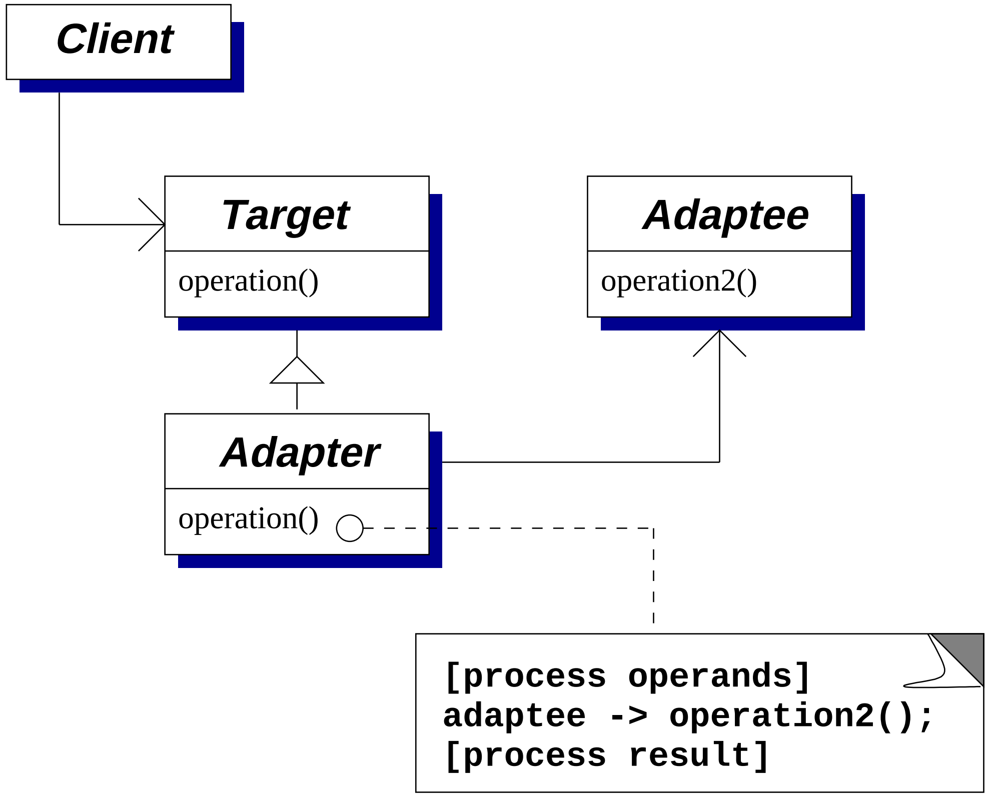
➤ ...and a good implementation `BinomialTree` that instead implements some other interface, say `LookupTable`:

```
public interface LookupTable {
  int numElements ();
  Object lookupKey (Object key);
  ...
}
```

✘ Rewrite `BinomialTree` so it implements `Dictionary`?

✘ Define a sub-class `DictionaryBinomialTree` that implements both?

Adapter pattern (2)



```
[process operands]
adaptee -> operation2();
[process result]
```

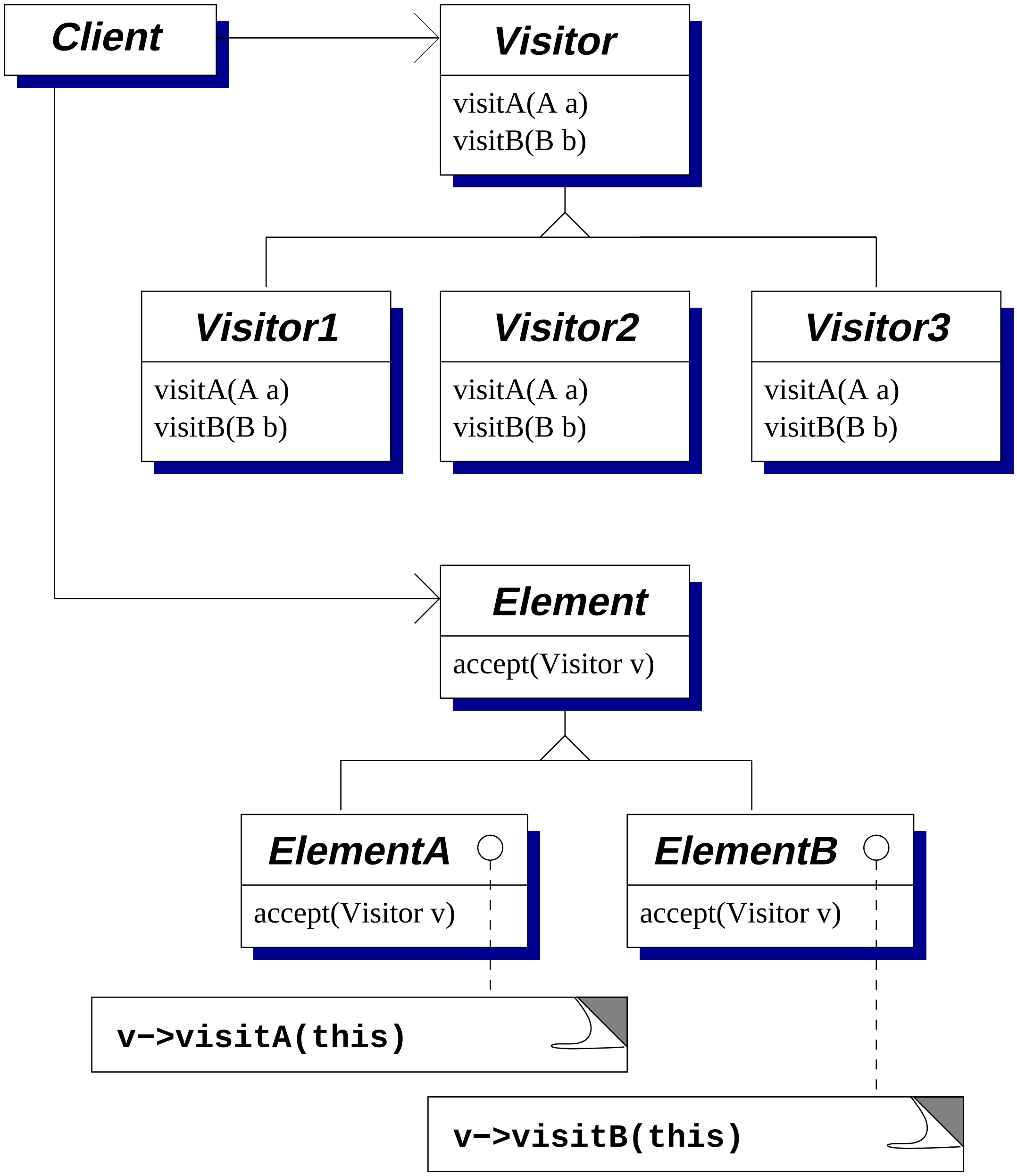
- The *Client* wishes to invoke operations on the *Target* interface (`Dictionary`) which the *Adaptee* (`BinomialTree`) does not implement

- The *Adapter* class implements the *Target* interface, in terms of operations the *Adaptee* supports

- ✔ The adapter can be used with any sub-class of the adaptee (unlike sub-classing adaptee directly)

Visitor pattern

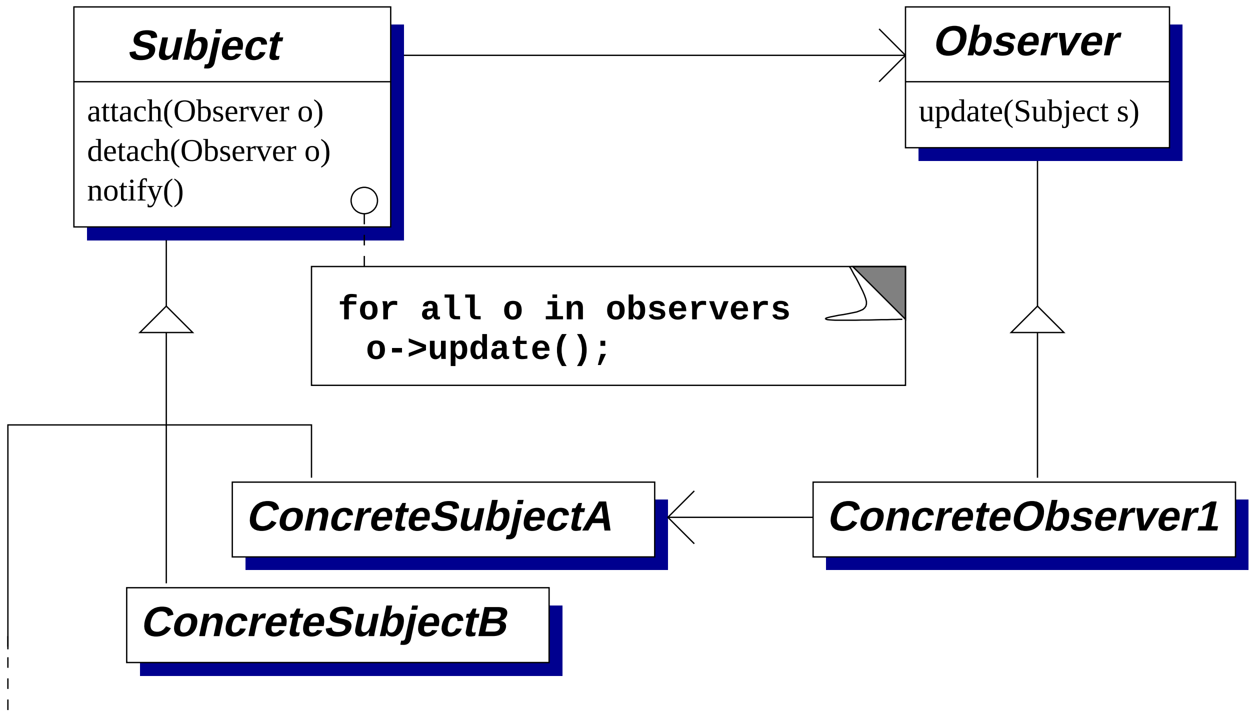
The Visitor pattern is one way of structuring operations that work on data structures comprising objects of different classes



Visitor pattern (2)

- ➤ The data structure is built from instances of *ElementA*, *ElementB*, etc., all sub-classes of *Element*
- ➤ These classes, or a separate object structure class, provide some mechanism for traversing the data structure
- ➤ The abstract *Visitor* class defines operations corresponding to each sub-class of *Element*
- ➤ A concrete sub-class of *Visitor* is constructed for each kind of operation on the data structure
- ✔ The methods implementing a particular operation are kept together in a single sub-class of *Visitor*
- ✔ Operations can be added and updated without changing the data structure's definition
- ✘ As with the Abstract Factory pattern, changing the data structure requires changes to many classes

Observer pattern



- ✔ In Java *Observer* can be an interface rather than a concrete class
- ✔ A many-to-many dynamically changing relationship can exist between subjects and observers
- ✘ The flexibility limits the extent of compile-time type-checking
- ✘ If observers can change the subject then cascading or cyclic updates may occur

Summary

Some common themes to the examples here:

- Explicitly creating objects by specifying their class (e.g. `new XYZ()`) commits to a particular implementation

  — it's often better to separate code responsible for instantiating objects (as in the Abstract Factory and Singleton patterns)

- Tight coupling between classes makes independent re-use difficult

  — e.g. the Visitor pattern separates the structure-traversal code from the visitor-specific operations to apply to each item found

  `java.util.Iterator` is a simpler example

- Extending functionality by sub-classing commits *at compile time* to a particular organisation of extensions

  — composition and delegation may be preferable (as in the Adapter pattern)

Exercises

4-1 The *Memento* design pattern can be used when designing an ‘undo’ facility for part of a program – e.g. in a graphical drawing environment to allow parts of the drawing to be restored to an earlier state.

In this pattern classes occupy one of three rôles: the *originator* is the class whose state must be saved and restored (e.g. a component in the graphical design), instances of the *memento* class represent snapshots of the saved state that are created and restored under the control of the *caretaker* class (typically the ‘main’ part of the application acting in response to user requests to undo operations). The originator is responsible for instantiating the memento class, for saving its state into such an instance and for restoring its state from an instance.

Suggest how the classes in the Memento pattern may be organized:

(a) As a UML class diagram.

(b) In terms of example Java code. Justify your answer’s use (or otherwise) of access modifiers, inheritance, interfaces and abstract or nested classes.

Lecture 5: Reflection & serialization

Previous section

- Recap of Java syntax
- Facilities for designing and using classes
- Design patterns

Overview of this section

- Important library facilities for:
- Saving & restoring object state
- Graphical interfaces
- Managing memory

Reflection

- Java provides facilities for **reflection** or *introspection* of type information about objects at run time

- Given the name of a class, a program can...
  - find the methods and fields defined on that class,
  - instantiate the class to create new objects

- Given an object reference, a program can...
  - determine the class of the object it refers to,
  - invoke methods or update the values in fields.

- It is *not* possible to obtain or change the source code of methods

- These facilities are often used ‘behind the scenes’ in the Java libraries, e.g. RMI, and in visual program developments environments – presenting a graphical representation of the facilities provided by each class, or showing the way in which classes are combined through composition or inheritance

Reflection (2)

- Reflection is provided by a number of classes in the `java.lang` and `java.lang.reflect` packages. Each class models one aspect of the Java programming language

- An instance of `Class` represents a Java class definition. The `Class` associated with an object is obtained by the `getClass()` method defined on it

- An instance of `Field` represents a field definition, obtained from the `Class` object by `getFields()`

- Instances of `Method` and `Constructor` represent method and constructor definitions, similarly obtained by `getMethods()` and `getConstructors()`

- Similarly for `getSuperclass()` and `getInterfaces()`

Reflection (3)

- For example:

```
1   public class ReflExample
2   {
3     public static void main (String args[])
4     {
5       ReflExample re = new ReflExample ();
6       Class reclass = re.getClass ();
7       String name = reclass.getName ();
8       System.out.println (name);
9     }
10  }
```

- Line 5 creates a new instance of `ReflExample`

- Line 6 obtains the `Class` object for that instance

- Line 7 obtains the name of that class

Reflection (4)

- We could do the same in reverse:

```
public class ReflExample2
{
  public static void main (String args[])
  {
    try
    {
      Class c = Class.forName (args[0]);
      Object o = c.newInstance ();
      System.out.println (o);
    }
    catch (Exception e)
    {
      System.out.println (e);
    }
  }
}
```

Reflection (5)

- Here we're taking a class name supplied as a parameter to the program and then instantiating it. For example

  ```
  $ java ReflExample2 java.lang.Object
  java.lang.Object@80cb54d
  ```

- We could have named any class on the command line

- By default a 0-argument constructor is called (and must exist)

- Specific constructors are also modelled by `Constructor` objects and define a `newInstance` method

- Why would anyone want to write

  ```
        Class c = Class.forName (args[0]);
        Object o = c.newInstance ();
  ```

  instead of an ordinary `new` expression?

Fields and reflection

- We can invoke `getFields()` on a `Class` object to obtain an array of the fields defined on that class

- As a shortcut we can also use `getField(...)`, passing the name required, to obtain information about an individual field

- If there is a security manager then its `checkMemberAccess` method must permit general access for `Member.PUBLIC` and `checkPackageAccess` must permit reflection within the package

- Only `public` fields are returned by `getFields()`

- A general `getDeclaredFields()` method provides full access (subject to a `checkMemberAccess` test for `member.DECLARED`)

- Given an instance of `Field` we can use...
  - `Class getDeclaringClass ()`
  - `String getName ()`
  - `int getModifiers ()`
  - `Class getType ()`

Fields and reflection (2)

```java
public class ReflExample3
{
  public static int field1 = 17;
  public static int field2 = 42;

  public static void main (String args[])
  {
    try
    {
      Class c = Class.forName (args[0]);
      java.lang.reflect.Field f;
      f = c.getField (args[1]);
      int value = f.getInt (null);
      System.out.println (value);
    }
    catch (Exception e)
    {
      System.out.println (e);
    }
  }
}
```

Fields and reflection (3)

- For example,

  ```
  $ java ReflExample3 ReflExample3 field1
  17

  $ java ReflExample3 ReflExample3 field2
  42

  $ java ReflExample3 ReflExample3 incorrect
  java.lang.NoSuchFieldException
  ```

- There are similar methods for setting the value of the field, e.g.

  ```
  f.setInt (null, 1234);
  ```

Methods and reflection

- The reflection API represents Java methods as instances of a `Method` class

- This has an `invoke` operation defined on it that calls the underlying method, for example, given a reference `m` to an instance of `Method`:

  ```
  Object parameters[] = new Object [2];
  parameters[0] = ref1;
  parameters[1] = ref2;
  m.invoke (target, parameters);
  ```

  is equivalent to making the call

  ```
  target.mth (ref1, ref2);
  ```

  where `mth` is the name of the method being called

Methods and reflection (2)

- The first value passed to `invoke` identifies the object on which to make the invocation. It must be a reference to an appropriate object (`target` in the example), or `null` for a static method

- Note how the parameters are passed as an array of type `Object[]`: this means that each element of the array can refer to any kind of Java object

- If a primitive value (such as an `int` or a `boolean`) is to be passed then this must be **wrapped** as an instance of `Integer`, `Boolean`, etc For example `new Integer(42)`

- The result is also returned as an object reference and may need unwrapping – e.g. invoking `intValue()` on an instance of `Integer`

Serialization

Reflection lets you inspect the definition of classes and manipulate objects without knowing their structure at compile-time

One use for this is automatically saving/loading data structures

- starting from a particular object you could use `getClass()` to find what it is, `getFields()` to find the fields defined on that class and then use the resulting `Field` objects to get the field values
- the data structure can be reconstructed by using `newInstance()` to instantiate classes and invocations on `Field` objects to restore their values

The `ObjectInputStream` and `ObjectOutputStream` classes automate this procedure

> Beware: the term's used with two distinct meanings. Here it means taking objects and making a 'serial' representation for storage. We'll use it in a different sense when talking about transactions.

Serialization (2)

In its simplest form the `writeObject()` method on `ObjectOutputStream` and `readObject()` method on `ObjectInputStream` transfer objects to/from an underlying stream, e.g.

```
FileOutputStream s = new FileOutputStream ("file");
ObjectOutputStream o = new ObjectOutputStream (s);
o.writeObject (drawing);
o.close ();
```

or

```
FileInputStream s = new FileInputStream ("file");
ObjectInputStream o = new ObjectInputStream (s);
Vector v = (Vector) o.readObject ();
o.close ();
```

- A real example must consider exceptions as well

- Fields with the `transient` modifier applied to them are not saved or restored

java.io.Serializable

These methods attempt to transfer the complete structure reachable from the initial object

However, classes must implement the `java.io.Serializable` interface to indicate that the programmer believes this is a suitable way of loading or saving instance state, e.g. considering

- whether field values make sense between invocations – e.g. time stamps or sequence numbrs
- whether the complete structure should be saved/restored – e.g. if it refers to a temporary data structure used as a cache
- any impact on application-level access control – e.g. if security checks were performed at instantiation time

The definition of `Serializable` is trivial:

```
public interface Serializable {
}
```

java.io.Serializable (2)

- A 0-argument constructor must be accessible to the subclass being serialized: it's used to initialize fields of non-serializable superclasses

- More control can be achieved by implementing `Serializable` *and* also two special methods to save and restore *that particular class's* aspect of the object's state:

```
private void writeObject(
       java.io.ObjectOutputStream out)
    throws IOException;

private void readObject(
       java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

- Further methods allow alternative objects to be introduced at each step, e.g. to canonicalize data structures:

```
ANY-ACCESS-MODIFIER Object writeReplace()
    throws ObjectStreamException;

ANY-ACCESS-MODIFIER Object readResolve()
    throws ObjectStreamException;
```

java.io.Externalizable

- `writeObject` and `readObject` are fiddly to use: they may require careful co-ordination within the class hierarchy. The documentation is unclear about the order in which they're called on different classes.

- The interface `java.io.Externalizable` is more useful in practice

```
public interface Externalizable
       extends java.io.Serializable
{
    void writeExternal(ObjectOutput out)
       throws IOException;
    void readExternal(ObjectInput in)
       throws IOException,
               ClassNotFoundException;
}
```

- It is invoked using the normal rules of method dispatch

- It is responsible for transferring the complete state of the object on which it is invoked

- But note: `readExternal` is called *after* instantiating the new object

Exercises

5-1 What is meant by *reflection* or *introspection* in Java? Give an example of how and why these facilities may be useful.

5-2 What are the advantages and disadvantages, with respect to encapsulation, of using `Externalizable` rather than `Serializable` for customised serialization?

5-3* Write a general-purpose implementation of methods

```
void writeObjectExternal (Object target,
                          ObjectOutputStream out);
void readObjectExternal (Object target,
                          ObjectInputStream in);
```

to save/restore the state of `target` to/from the the stream `out` or `in`. You will need to use the reflection API to interrogate the `target` object.

Hint: the slides headed 'Serialization' show how to instantiate an the input and output streams. Notice that `ObjectOutputStream` extends `ObjectOutput` (for saving fields of reference types) which extends `DataOutput` (for saving fields of int, char, boolean, etc).

5-4* Now implement it *without* using the `writeObject` or `readObject` methods on the stream classes.

Past exam questions: 1999 Paper 3 Q3

Lecture 6: Graphical interfaces

Previous lecture

- Reflection: APIs to interrogate objects and class definitions at run-time, access fields and call methods
- Serialization: API to save & restore object state e.g. to a file

Overview of this lecture

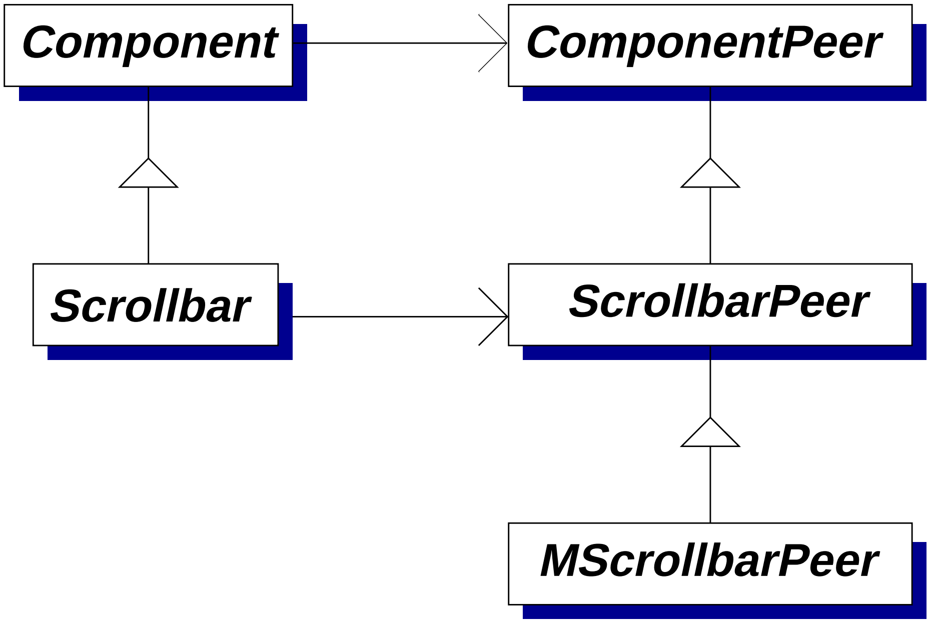
- Model-view-controller pattern
- Components & containers
- API specs are available on-line (`http://www.java.sun.com/products/jfc`)
- Examples of individual components in the “SwingExamples” demos

Java Foundation Classes (JFC)

- We'll cover the 'Swing' user interface components, part of JFC which also adds rendering APIs, drag 'n' drop & more

- AWT is the original set of GUI classes in Java: it's a lowest-common-denominator between the systems available at the time

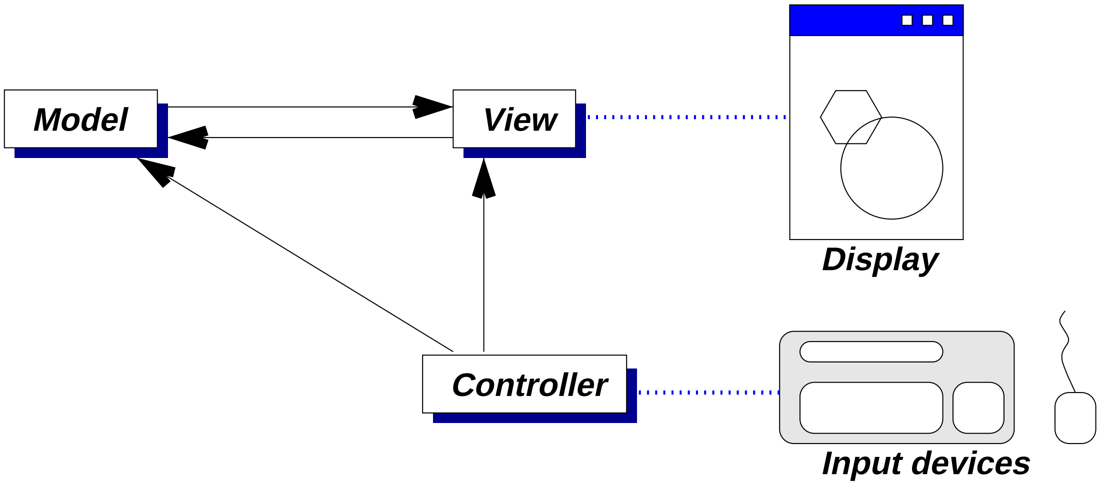
- AWT GUI components each had **peers** (using native methods) responsible for their rendering, e.g.:



- A `Toolkit` class puts all this together following the abstract factory pattern – e.g. `MToolkit` for Motif, instantiating `MScrollbarPeer`

Model-View-Controller

Swing components follow a **model-view-controller** pattern (derived from Smalltalk-80)



This separates three aspects of the component:

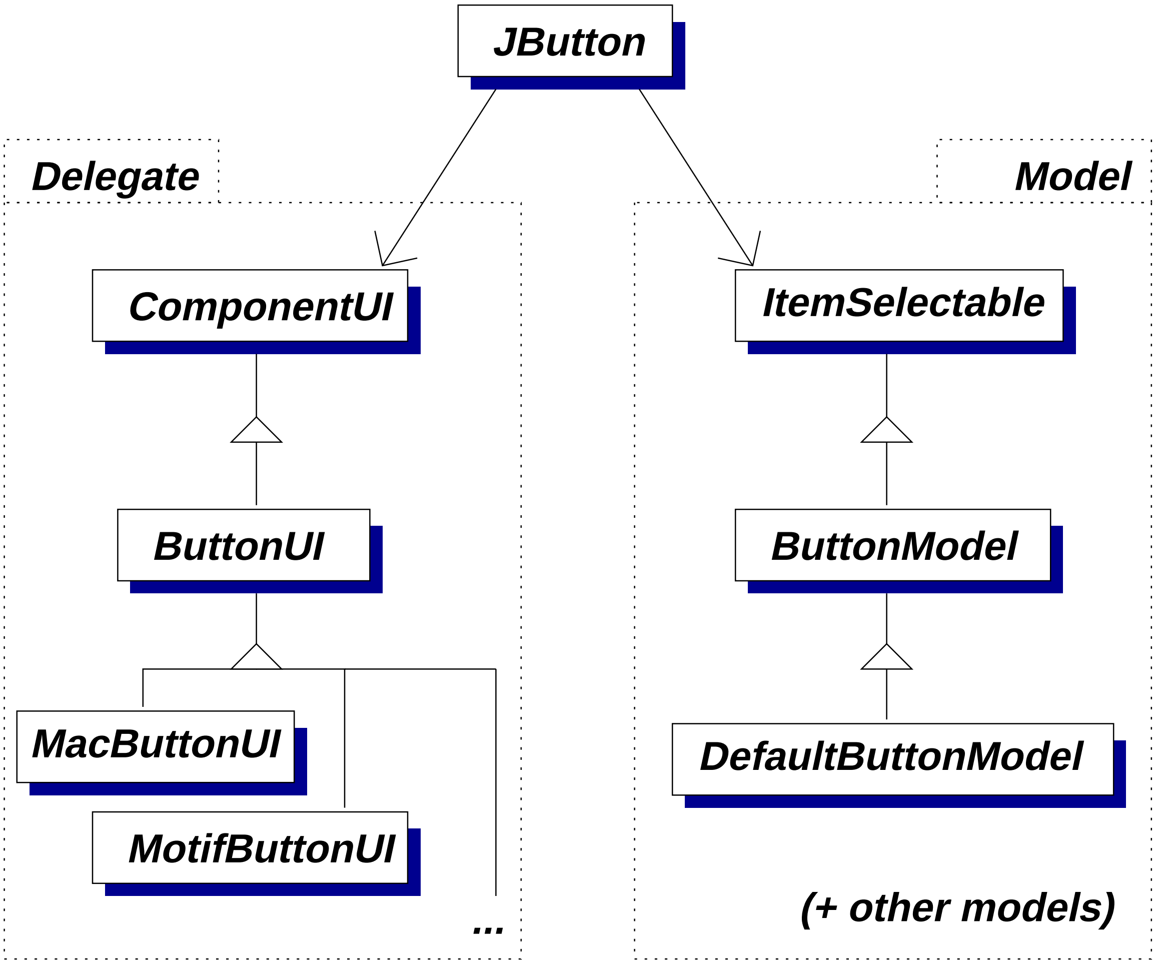
- The **view**, responsible for rendering it to the display
- The **controller**, responsible for receiving input
- The **model**, the underlying logical representation

Multiple views may be based on the same model (e.g. a table of numbers and a graphical chart). This separation allows views to be changed independently of application logic

Model-View-Controller (2)

For simplicity the controller and view are combined in Swing to form a **delegate**

The component itself (here `JButton`) contains references to the current delegate and current model



Graphics

- Basic rendering primitives are available on instances of `Graphics`, e.g. using Java applets as in Part-1A:

```
import java.awt.*;

public class E1 extends java.applet.Applet
{
  public void paint (Graphics g) {
    g.drawLine (0, 0, 100, 100);
  }
}
```

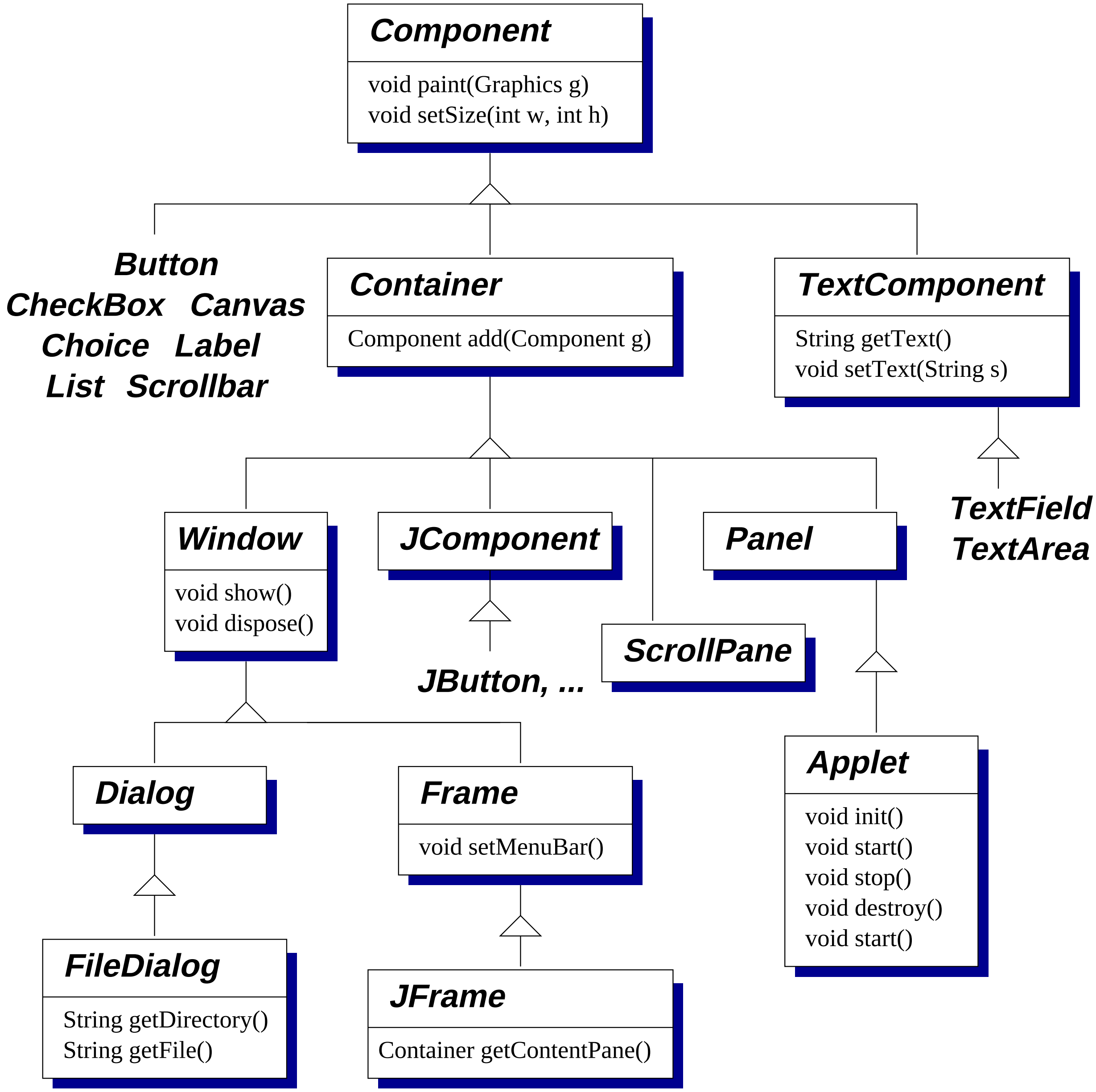
- Simple primitives are available – `setColor`, `copyArea`, `drawLine`, `drawArc`...

- More abstractly, an instance of `Graphics` represents the component on which to draw, a translation origin, the clipping mask, the font, etc

- Translation allows components to assume that they're placed at (0,0)

(Notice the running similarity between these basic functions as X11 / Motif...)

Components



- See the “SwingExamples” demos for illustrations of how to use many of these

Components (2)

- In general a graphical interface is built up from **components** and **containers**

- *Components* represent the building blocks of the interface, for example buttons, check-boxes or text boxes

- Each kind of component is modelled by a separate Java class (e.g. `javax.swing.JButton`). Instances of those classes provide particular things in particular windows – e.g. to create a button bar the programmer would instantiate the `JButton` class multiple times

- As you might expect, new kinds of component can be created by sub-classing existing ones – e.g. sub-classing `JPanel` (a blank rectangular area of the screen) to define how that component should be rendered by overriding its `paintComponent` method:

```
public void paintComponent (Graphics g) {
  super.paintComponent (g);
  ...
}
```

Containers

- *Containers* are a special kind of component that can contain other components – as expected, the abstract class `java.awt.Container` extends `java.awt.Component`

- Containers implement an `add` method to place components within them

- Containers also provide top-level windows – for example `javax.swing.JWindow` (a plain window, without title bar or borders) and `javax.swing.JFrame` (a 'decorated' window with a title bar etc)

- Other containers allow the programmer to control how components are organized – in the simplest case `javax.swing.JPanel`

- In fact, `java.applet.Applet` is actually a sub-class of `Panel`

Containers (2)

- Components are organized within a container under the control of a **layout manager**, e.g.

```
import java.awt.*;
import javax.swing.*;
public class ButtonsFrame extends JFrame {
  public ButtonsFrame() {
    super();
    Container cp;
    cp = getContentPane ();
    cp.setLayout(new BorderLayout());
    cp.add("North",  new JButton("North"));
    cp.add("South",  new JButton("South"));
    cp.add("East",   new JButton("East"));
    cp.add("West",   new JButton("West"));
    cp.add("Center", new JButton("Center"));
  }

  public static void main (String args[]) {
    ButtonsFrame b = new ButtonsFrame();
    b.pack();  b.setVisible(true);
  }
}
```

- A `JFrame` has a **root pane** which contains the main **content pane** and the menu bar

Containers (3)

- A common design technique is to develop a **spatial hierarchy** of nested containers

- `BoxLayout` is particularly useful: it places a series of components horizontally or vertically

- `Box` provides static methods to create special invisible components:
  - **Rigid-area** components which have a fixed size
  - **Struts** which have fixed height or wdith (used to space out other components)
  - **Glue** which expands/contracts if the window is resized and nothing else can change, e.g.:

```
cp.setLayout (
      new BoxLayout (cp, BoxLayout.X_AXIS));
cp.add (Box.createHorizontalStrut (10));
cp.add (left);
cp.add (Box.createHorizontalGlue ());
cp.add (right);
cp.add (Box.createHorizontalStrut (10));
```

- It is almost always easier to use nested `JPanel` components controlled with `BoxLayout` than to use any of the other older layout managers (`CardLayout`, `FlowLayout`, `GridLayout`, `GridBagLayout`)

Receiving input

- An **event-based** mechanism is used for delivering input to applications, broadly following the observer pattern

- Different kinds of event are represented by sub-classes of `java.awt.AWTEvent`. These are all in the `java.awt.event` package. e.g. `MouseEvent` is used for mouse clicks, `KeyEvent` for keyboard input, etc.

- The system delivers events by invoking methods on a **listener**. e.g. instances of `MouseListener` are used to receive `MouseEvent`:

```
public interface MouseListener
       extends EventListener
{
  public void mouseClicked(MouseEvent e);
  ...
}
```

  Components provide methods for registering listeners with them, e.g. `addMouseListener` on `Component`

- `AWTEvent` has a `getSource()` method, so a single listener can disambiguate events from different sources. Sub-classes add methods to obtain other details – e.g. `getX()` and `getY()` on a `MouseEvent`

Receiving input (2)

- All components can generate:

1. `ComponentEvent` when it is resized, moved, shown or hidden
2. `FocusEvent` when it gains or loses the focus
3. `KeyEvent` when a key is pressed or released
4. `MouseEvent` when mouse buttons are pressed or released
5. `MouseMotionEvent` when the mouse is dragged or moved

- Containers can generate `ContainerEvent` when components are added or removed

- Windows can generate `WindowEvent` when opened, closed, iconified etc

Input using inner classes

- Anonymous inner classes provide an effective way of handling some forms of input, e.g.

```
addActionListener (new ActionListener () {
  public void actionPerformed (ActionEvent e)
  {
    ...
  }
});
```

- A further idiom is to define inner classes that extend **adapter classes** from the `java.awt.event` package. These provide ‘no-op’ implementations of the associated interfaces

- The programmer just needs to override the methods for the kinds of event that they are interested in: there is no need to define empty methods for the entire interface

```
addMouseMotionListener
        (new MouseMotionAdapter () {
  public void mouseDragged (MouseEvent e)
  {
    ...
  } // no need to define mouseMoved
};
```

Accessibility (`javax.accessibility`)

Intended to allow interaction with Java applications through technologies such as screen readers and screen magnifiers

- Swing UI components implement `Accessible`, defining a single method `getAccessibleContext()` returning an `AccessibleContext`
- That instance describes and is used to interact with a particular UI component. It defines methods to retrieve associated instances of
  - `AccessibleAction` – representing operations that may be done on the component, named by strings
  - `AccessibleComponent` – represents the current visual appearance of the component. Allows colours, fonts, focus settings to be overridden
  - `AccessibleSelection` – e.g. items in a menu, table or tabbed pane
  - `AccessibleRole` – in terms of generic roles such as `SCROLL_PANE` or `SLIDER`
  - `AccessibleState` – e.g. `CHECKED`, `FOCUSED`, `VERTICAL`
  - `AccessibleText` – represents textual information
  - `AccessibleValue` – represents numerical values (e.g. scroll bar positions)

Exercises

6-1 Describe the way in which graphical output and interactive input are achieved when using Swing. Your answer should cover the use of

- hierarchies of classes,
- overriding methods,
- interfaces,
- inner classes,
- spatial hierarchy.

6-2 Define example classes showing how to use the `JComboBox`, `JProgressBar`, `JDesktopPane` and `JInternalFrame` components.

6-3 Describe the advantages and disadvantages of rendering components in Java (as with Swing) rather than using native components provided by system hosting the JVM (as with AWT).

Exercises (2)

6-4 Define a class that uses the `getAccessibleContext` method to extract a text-based description of a user interface, indicating the components that it comprises, their nesting within one another and their current state – for example whether or not a button is pressed.

How does your class perform when dealing with a kind of component for which you have not already tested it?

What are the advantages and disadvantages of having a separate `AccessibleContext` interface rather than using the reflection API?

Lecture 7: Memory management

Previous lecture

- Graphical user interfaces
- Model-view-controller pattern
- Components & containers

Overview of this lecture

- Garbage collection in Java
- Interaction between the application & the collector
- Reference objects

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

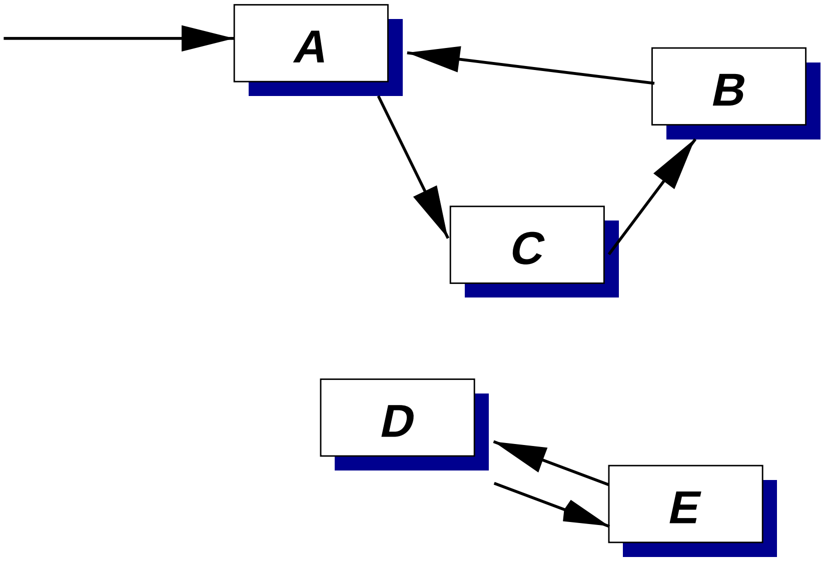
```
$ java -verbose:gc Loop
[GC 511K->95K(1984K), 0.0031071 secs]
[GC 607K->95K(1984K), 0.0006847 secs]
[GC 607K->95K(1984K), 0.0001221 secs]
[GC 607K->95K(1984K), 0.0002499 secs]
[GC 607K->95K(1984K), 0.0001106 secs]
```

Garbage collection (2)

There are lots of different techniques that might be used to implement the garbage collector (see Part 1B DS&A and Compiler Construction)

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,
- *still needs to be finalized,*
- is referred to by from another reachable object



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

Garbage collection (3)

Early languages with GC had a reputation for being slow and for adding annoying pauses to an application's execution. Modern collectors do better:

- **generational collection** ('most objects die young' → keep a small *young generation* which can be collected quickly and frequently)
- **parallel collection** (multiple processors work on GC at the same time → application pauses for less time)
- **concurrent collection** (GC occurs at the same time as application execution)
- **incremental collection** (GC occurs in small bursts, e.g. each time an object is allocated `-Xincgc`)

In complex multi-threaded systems GC may have indirect benefits over explicit deallocation:

- no need to agree which module is responsible for deallocation
- this may aid sharing of data structures rather than having each module take a private copy

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 network connection is set up in the constructor then perhaps the finalizer should close it so that the remote machine knows that the connection is no longer in use
- To aid debugging
  - e.g. to check that objects are being collected 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 them 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 (Lecture 12) or enter endless loops

Reference objects

Reference objects provide a more general mechanism for

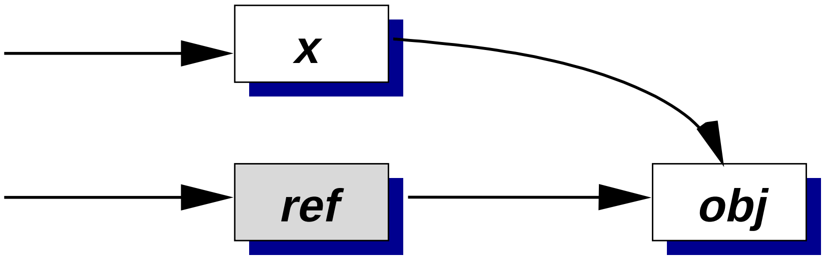
- scheduling cleanup actions when objects become unreachable via ordinary references,
- managing caches in which the presence of an object in the cache should not prevent its garbage collection, or
- accessing temporary objects which can be removed when memory is low

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:

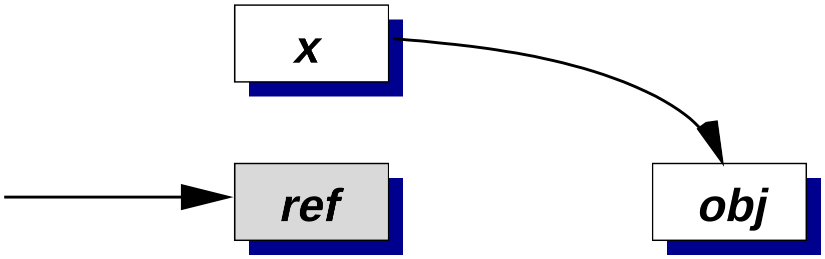
```
import java.lang.ref.*;

class RefExample {
  public static void main (String args[]) {
    int obj[] = new int[42];
    Reference ref = new WeakReference (obj);
    System.out.println ("ref: " + ref);
    System.out.println (ref.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 the object (`obj`) is accessible through a **weak reference object** (`ref`) and through an ordinary object (`x`):



If `x` becomes unreachable then `obj` is said to be **weakly reachable** and the GC is permitted to clear the reference in `ref`:



Further calls to `ref.get()` will return `null`. The reference object can be cleared explicitly by invoking `ref.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 ref = new WeakReference (obj, 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 (`ref`), not the referent (`obj`), that is appended to the queue

✔ This prevents the problem of ‘resurrected’ objects

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 (if necessary). `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

Exercises

7-1 The 'HotSpot' JVM has a command-line option to control the maximum size of the heap, e.g. `-Xmx25M` to set it to 5MB. Investigate the effect different settings have on performance using the `MMExample` program (or any other that allocates significant numbers of objects). Also investigate the `-Xincgc` option for incremental collection.

7-2 Describe how *soft references* can be used to implement a hashtable that discards objects when memory is tight. How can the implementation be extended to approximate a LRU (least-recently-used) policy for discard?

7-3 Compare and contrast *object finalizers* and *phantom references* as mechanisms to clean-up after objects that have become otherwise-unreachable. For each approach indicate, along with any specific problems or benefits:

(*i*) in which class the clean-up code is located,

(*ii*) in which thread or threads it may be executed,

(*iii*) what happens if the code blocks or takes a long time to execute.

Past exam questions: 2002 Paper 4 Q1

Lecture 8: Miscellany

Previous lecture

- Garbage collection
- Finalizers
- Reference objects

Overview of this lecture

- Native methods
- Class loaders

Native methods

The **Java Native Interface** (JNI) allows you to define method implementations in some other language (e.g. C or directly in assembly language) and to call them from Java

It might be useful

- to access facilities not provided by the standard APIs – e.g. some special hardware device,
- to re-use an existing well-engineered library,
- to allow careful optimization of part of an application

The latter reason has become less important

- Modern JVMs will compile Java bytecode to native code at run-time
- It can benefit from profile-directed optimization
- It is often hard to recoup the cost of making a JNI call (and accessing Java objects from within it)

The details of writing native methods in C are not examinable

Native methods (2)

```
 1 class HelloWorld {
 2   public native void displayHelloWorld();
 3
 4   static {
 5     System.loadLibrary("hello");
 6   }
 7
 8   public static void main(String[] args) {
 9     new HelloWorld().displayHelloWorld();
10   }
11 }
```

- Line 2 defines the signature of a native method
- Lines 4-6 are a **static initializer**, executed by the JVM when the class is loaded
- Line 9 instantiates the class and calls the native method
- Compile the Java and create the C function signatures:

```
$ javac HelloWorld.java
$ javah -jni HelloWorld
```

- Creates `HelloWorld.class` & `HelloWorld.h`

Native methods (3)

```
 1 /* DO NOT EDIT THIS FILE - it is machine generated */
 2 #include <jni.h>
 3 /* Header for class HelloWorld */
 4 
 5 #ifndef _Included_HelloWorld
 6 #define _Included_HelloWorld
 7 #ifdef __cplusplus
 8 extern "C" {
 9 #endif
10 /*
11  * Class:     HelloWorld
12  * Method:    displayHelloWorld
13  * Signature: ()V
14  */
15 JNIEXPORT void JNICALL
16   Java_HelloWorld_displayHelloWorld
17   (JNIEnv *, jobject);
18 
19 #ifdef __cplusplus
20 }
21 #endif
22 #endif
```

- Line 16 declares the function we must define for the `displayHelloWorld` method

Native methods (4)

- Now write `HelloWorldImp.c`:

```c
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env,
                                  jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

- On Linux or Solaris, we must build this to make the shared library `libhello.so` (the one named in the `System.loadLibrary` call)

```
$ JINCLUDE=/usr/java/j2sdk1.4.0_01/include
$ gcc HelloWorldImp.c -I$JINCLUDE \
     -I$JINCLUDE/linux -shared \
     -fpic -o libhello.so
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
$ java HelloWorld
```

- On Win32, we'd build `hello.dll`

Native methods (5)

- The header file `jni.h` defines the C or C++ functions that can be used within native methods and the correspondence between Java types and native types

- The `JNIEnv` parameter refers to an environment structure containing function pointers for these operations, e.g.
  - `FindClass` to get the `jclass` for a specified name
  - `GetSuperclass` to map one `jclass` to its parent
  - `NewObject` to allocate an object and execute a constructor on it
  - `CallObjectMethod`, `CallBooleanMethod`, `CallVoidMethod` etc for method calls
  - Similarly `GetObjectField`, `GetCharField` etc and corresponding `Set...Field` operations

- In C, references to Java objects (**local references**) are represented by structures of type `jobject`. The JVM tracks which objects have been passed to active native methods. These cannot be collected by the GC

- If a Java object is to be kept alive through references from native data structures then a **global reference** must be created to inform the GC about it (`NewGlobalRef`) and removed when it may be collected (`DeleteGlobalRef`)

Class loaders

- The examples we've seen so far are based on compiling a number of `.java` files to get `.class` files placed in per-package directories

- **Class loaders** can be used to supply the JVM with class definitions from other sources – e.g. across the network, or dynamically generated

- Class loaders are Java objects extending `java.lang.ClassLoader`

- `c.loadClass(name)` requests that `c` loads the named class, returning a `java.lang.Class` object. The default implementation:
  - tests whether the class is already loaded,
  - delegates to a **parent class loader** to load it,
  - otherwise calls `c.findClass(name)`

- A new kind of class loader should override the protected `findClass` method so that the delegation model remains consistent

- `findClass` can then call `c.defineClass(name, b, off, len)` to create a new `Class` object from the bytes at `off` → `off+len`

Class loaders (2)

An aside:

- `findClass` is a good example of the use of a `protected method` – note how the modifier prevents one kind of class loader calling `findClass` on a different kind

Within the JVM classes are identified by the pair of their fully-qualified name and the class loader that created them

- i.e. there can be several different classes of the same name
- If a class `A` refers to a class `B` (e.g. its superclass, a field of that type, etc) then the class loader that defined `A` is requested for `B`
- The delegation model ensures that all classes agree on (e.g.) `java.lang.System`

A good library for creating class definitions at run time: `http://jakarta.apache.org/bcel/`

Hosting separate applications

Class loaders provide part of a solution for hosting separate applications within one JVM

✔ Each application can have a separate class loader so name-space clashes are avoided...

✘ ...but they will still share static fields in the standard libraries (e.g. `System.out`)

✘ ...and there's no resource management at all

```
while (true) { /* Nothing */ }
```

```
while (true) {
  Thread t = new Thread ();
  t.start ();
}
```

```
while (true) {
  int a[] = new int[1000000];
}
```

Exercises

8-1 Describe an example situation in which it might be appropriate to use JNI. Suggest how the Java programming language, or the standard libraries that it supports, could be extended so that Java could be used instead of native code.

8-2 Implement a simple class loader that prints a list of the class names for which it is requested.

8-3* If you are familiar with C or C++ then experiment with calls to a simple JNI method to determine the overhead introduced by invoking a native method when compared with an ordinary Java method. Similarly, compare the time taken to access a Java field from Java code and from native code.

Exercises (2)

8-4* A class `C1`, loaded and defined by classloader `L1`, contains the code

```
S s_object = C2.getS ();
```

calling a static method on `C2` to receive an object of type `S`, again loaded by `L1`. `C2` has been defined by a different class loader, `L2` and contains the definition:

```
static S getS ( ) { return new S(); }
```

(*i*) Show how the type safety of the JVM could be compromised if the class named `S` in `C2` is loaded by `L2`.

(*ii*) Implement classes `C1`, `C2`, `L1`, `L2` and the two versions of `S` (to be loaded by `L1` and `L2` respectively).

(*iii*) Is `L2` actually requested to load `S`? If so then what happens if it supplies a different `Class` object from the one already loaded by `L1`?

The paper *Dynamic Class Loading in the JVM* (on the teaching material web site) discusses this problem more formally and various solutions that were proposed – it was a long-standing type safety problem in early versions of the JVM

Past exam question: 2003 Paper 4 Q8

Lecture 9: Threads

Previous section

- Reflection & serialization
- Memory management
- Swing and AWT

Overview of this section

- Multi-threaded programming
- Concurrent data structures

Concurrency

The next section of the course concerns different ways of structuring systems in which concurrency is present and, in particular, co-ordinating multiple threads, processes and machines accessing shared resources and data

Two main scenarios:

- Tasks operating with a shared address space – e.g. multiple threads created within a Java application
- Tasks communicating between address spaces – e.g. different processes, whether on the same or separate machines

In each case we must consider

- How shared resources and data are named and referred to by the participants
- Conventions for representing shared data
- How access to resources and data is controlled
- What kinds of system failure are possible

Concurrency (2)

- Previous examples have been implemented using a single thread that runs the `main` method of a program

- Java supports *lightweight* concurrency within an application – multiple threads can be running at the same time

- Can simplify code structuring and aid interactive response – e.g. one thread deals with user interaction, another thread deals with computation
  - Easier to add additional tasks as new threads?

- Can benefit from multi-processor hardware
  - e.g. the new HPCF machines have 106 processors...

- Implementation schemes vary substantially. We'll look at how multiple threads are available to the Java programmer, and what you can assume when writing portable code

Concurrency (3)

Most OS introduce a distinction between *processes* (as discussed in Part 1A) and *threads*

Processes are the unit of *protection* and *resource allocation*. Each processes has a **process control block** (PCB) holding:

- Identification (e.g. PID, UID, GID)
- Memory management information
- Accounting information
- (References to) one or more TCBs. . .

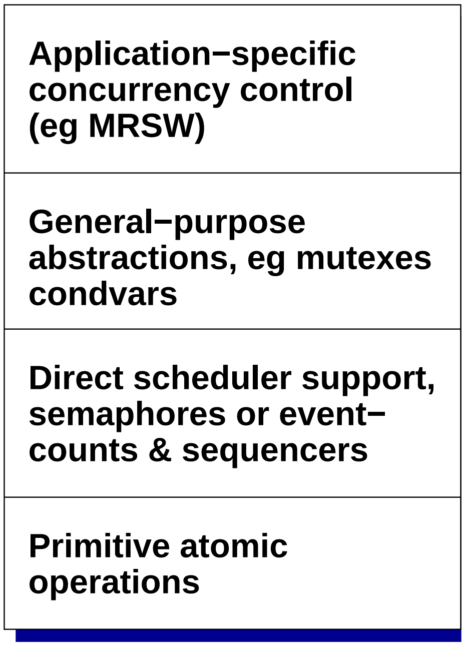
Threads are the entities considered by the scheduler. Each thread has a **thread control block** (TCB) holding:

- Thread state
- Saved context information
- References to user (and kernel?) stack
- Scheduling parameters (e.g. priority)

Concurrency (4)

Structure of this section:

- Managing threads in Java
- Simple shared objects – shared counters, shared hashtables etc
  - Mutual exclusion locks (mutexes)
- Shared objects in Java with internal blocking – queues, multi-reader single-writer (MRSW) locks
  - Condition variables (condvars)
- Implementation of mutexes and condvars
  - Direct scheduler support
  - Semaphores
  - Event counts / sequencers
- Alternative abstractions
  - Monitors
  - Active objects



Application−specific concurrency control (eg MRSW)

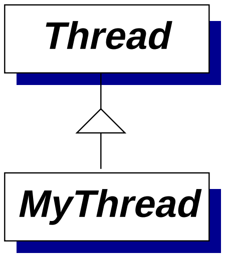
General−purpose abstractions, eg mutexes condvars

Direct scheduler support, semaphores or event−counts & sequencers

Primitive atomic operations

Creating threads in Java

- There are two ways of creating a new thread. The simplest is to define a sub-class of `java.lang.Thread` and to override the `run()` method, e.g.



Thread

MyThread

```
class MyThread extends Thread {
  public void run() {
    while (true) {
      System.out.println ("Hello from " +
                                   this);
      Thread.yield ();
    }
  }

  public static void main (String args[]) {
    Thread t1 = new MyThread ();
    Thread t2 = new MyThread ();
    t1.start ();  t2.start ();
  }
}
```

Creating threads in Java (2)

- The `run` method of the class `MyThread` defines the code that the new thread(s) will execute. Just defining such a class does not create any threads

- Lines 11–12 instantiate the class to create two objects representing the two threads that will be executed

- Line 13 actually starts the two threads executing

- The program continues to execute until all ordinary threads have finished, even after the `main` method has completed

  ```
  Hello from Thread[Thread-5,5,main]
  Hello from Thread[Thread-4,5,main]
  Hello from Thread[Thread-5,5,main]
  ```

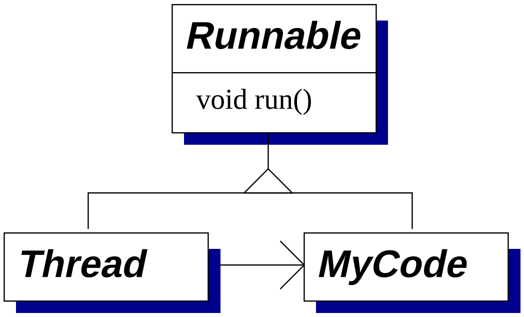
  etc...

- A **daemon** thread will not prevent the application from exiting:

  ```
  t1.setDaemon(true);
  ```

Creating threads in Java (3)

- The second way of creating a new thread is to define a class that `implements` the `java.lang.Runnable` interface, e.g.



```
class MyCode implements Runnable {
  public void run() {
    while (true) {
      System.out.println ("Hello from " +
                          Thread.currentThread());
      Thread.yield ();
    }
  }
  public static void main (String args[]) {
    MyCode mt = new MyCode ();
    Thread t_a = new Thread (mt);
    Thread t_b = new Thread (mt);
    t_a.start (); t_b.start ();
  }
}
```

Creating threads in Java (4)

- As before, Lines 2–8 define the code that the new threads will execute
- Lines 11–12 instantiate two `Thread` objects, passing a reference to an instance of `MyCode` to them as their *target*
- Line 13 starts these two threads executing
- Note that here the `run` methods of the two threads are being executed on the *same* `MyCode` object, whereas two separate `MyThread` objects were required
- The second way of creating threads is more complex, but also more flexible
- Generally, fields in the class containing the `run` method will hold per-thread state – e.g. which part of a problem a particular thread is tackling

Creating threads in Java (5)

- In some cases anonymous inner classes can be used to simplify thread creation, e.g.

```
 1   class Example {
 2     public static void main (String args[]) {
 3       Thread t = new Thread () {
 4         public void run () {
 5           System.out.println ("Hello world!");
 6         }
 7       };
 8
 9       t.start ();
10     }
11   }
```

- Recall that Lines 3–7 define and instantiate a new anonymous class that extends `Thread`

- As before, line 9 actually starts the thread executing

Terminating a thread

- A thread can be forced to exit by invoking the `stop` method on it. This method throws an exception *into* the thread – the thread behaves as if the exception had suddenly been thrown at the point at which it was executing
  - Usually, an instance of `java.lang.ThreadDeath` is thrown. The programmer may pass some other object as a parameter to `stop`

```
Thread t = new MyThread ();
t.start ();
t.stop ();
```

- `stop` is now deprecated: it should not be used. The exception may be delivered to the target thread when it is executing a `finally` block

- The correct approach is to use the `interrupt()` method on `java.lang.Thread`

- A thread is responsible for periodically calling `isInterrupted()`

Terminating a thread (2)

- In some situations a thread is interrupted immediately if it is blocked – e.g. `sleep` may throw `InterruptedException`. For example:

```
 1  class Example {
 2    public static void main (String args[])
 3    {
 4      Thread t = new Thread () {
 5        public void run () {
 6          try {
 7            do {
 8              Thread.sleep (1000); // sleep 1s
 9            } while (true);
10          } catch (InterruptedException ie) {
11            // Interrupted: exit
12          }
13        }
14      };
15      t.start (); // Start...
16      t.interrupt (); // ...interrupt
17    }
18  }
```

- If the thread didn't block then line 9 could perhaps be

```
 9            } while (!isInterrupted());
```

Join

- The `join` method on `java.lang.Thread` causes the currently running thread to wait until the target thread dies

```
1   class Example {
2     public void startThread (void)
3                   throws InterruptedException
4     {
5       Thread t = new Thread () {
6         public void run () {
7           System.out.println ("Hello world!");
8         }
9       };
10
11      t.start (); // Start thread...
12      t.join (0); // ...wait for it to exit
13    }
14  }
```

- Line 12 waits for the thread started at Line 11 to finish. The parameter specifies a time in milliseconds (0 ⇒ wait forever)

- The `throws` clause in line 3 is required: the call to `join` may be interrupted

Priority controls

- Methods `setPriority` and `getPriority` on `java.lang.Thread` allow the priority to be controlled

- A number of standard priority levels are defined: `MIN_PRIORITY`, `NORM_PRIORITY`, `MAX_PRIORITY`

- The programmer can also try to influence thread scheduling using the `yield` method on `java.lang.Thread`. This is a hint to the system that it should try switching to a different thread – note how it was used in the previous examples
  - In a non-preemptive system even low priority threads may continue to run unless they periodically `yield`

- Selecting priorities becomes complex when there are many threads or when multiple programmers are working together

> Although it may work on some systems, the variation in behaviour between different JVMs means that it is *never* correct to use thread priorities to control access to shared data in portable code

Thread scheduling

- The choice of exactly which thread(s) execute at any given time can depend both on the operating system and on the JVM

- Some systems are **preemptive** – i.e. they switch between the threads that are eligible to run. Typically these are systems in which the OS supports threads directly, i.e. maintaining separate PCBs and TCBs

- Other systems are **non-preemptive** – i.e. they only switch when the running thread yields, becomes blocked or exits. Typically these systems implement threads within the JVM

- The Java language specification says that, in general, threads with high priorities will run in preference to those with lower priorities

> To write correct portable code it's therefore important to think about what the JVM is *guaranteed* to do – not just what it does on one system. Different behaviour may occur at different nodes within a distributed system

The `volatile` modifier

```
static boolean 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 boolean 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`.

As we'll see, 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`)

Exercises

9-1 Describe the facilities in Java for creating multiple threads of execution.

9-2 What is the difference between a *preemptive* and a *non-preemptive* scheduler? Write a Java class containing a method

```
        boolean probablyPreemptive();
```

which returns `true` if the JVM running it appears to be preemptive. (Hint: your solution will probably need to start multiple threads which then perform some kind of experiment)

9-3 A Java-based file server is to use a separate thread for each user granted access. Discuss the merits of this approach from the point of view of security, possible performance and likely ease of implementation.

9-4 Examine the behaviour that one or more JVMs provides for the following aspects of thread management:

(*i*) whether scheduling is preemptive,

(*ii*) whether the highest-priority runnable thread is guaranteed to run,

(*iii*) the impact on performance of making a frequently-accessed field `volatile`.

Lecture 10: Mutual exclusion

Previous lecture

- Creating & terminating threads
- `volatile`

Overview of this lecture

- Shared data structures
- Mutual exclusion locks

Safety

In concurrent environments we must ensure that the system remains **safe** no matter what the thread scheduler does – i.e. that ‘nothing bad happens’

- Unlike type-soundness, this cannot usually be checked automatically by compilers or tools (although some exist to help)
- It’s often useful to think of safety in terms of **invariants** – things that must remain true, no matter how different parts of the system evolve during execution
  - e.g. a ‘transfer’ operation between bank accounts preserves the total amount in them
- We can then identify **consistent** object states in which all invariants are satisfied
- ...and aim that all of the operations available on the system keep it consistent
- Therefore many of the problems we’ll see come down to deciding when different threads can be allowed access to objects in various ways

Liveness

As well as safety, we'd also like liveness – i.e. 'something good eventually happens'. We often distinguish per-thread and system-wide liveness

Standard problems include:

- **Deadlock** – a circular dependency between processes holding resources and processes requiring them. Typically the 'resources' are exclusive access to locks
- **Livelock** – a thread keeps executing instructions, but makes no useful progress, e.g. busy-waiting on a condition that will never become true
- **Missed wake-up** (**wake-up waiting**) – a thread misses a notification that it should continue with some operation and instead remains blocked
- **Starvation** – a thread is waiting for some resource but never receives it – e.g. a thread with a very low scheduling priority may never receive the CPU
- **Distribution failures** – of nodes or network connections in a distributed system

Shared data

- Most useful multi-threaded applications will share data between threads

- Sometimes this is straightforward e.g. data passed to a thread through fields in the object containing the `run` method

- More generally, threads may share state through...
  - `static` fields in mutually-accessible classes, e.g. `System.out`
  - objects to which multiple threads have references

- What happens to field `o.x`:

Thread A *Thread B*

```
o.x = 17;        o.x = 42;
```

- Most fields accesses are **atomic** – the value read from `o.x` after those updates will be either 17 or 42

- The only exceptions are numeric fields of type `double` or type `long` – some third value may be read in that case

Shared data (2)

- This is an example of a **race condition**: the result depends on the uncontrolled interleaving of the threads' execution

- We need some way of controlling how threads are executed when accessing shared data

- The basic notion is of **critical regions**: parts of a program during which a thread should have exclusive access to some data structures while making a number of operations on them

- Careful programming is rarely sufficient, e.g.

```
boolean busy;
int x;

...

while (busy) { /* nothing */ }
busy = true;
x = x + 1;
busy = false;
```

- Using `x++` would be no better

Locks in Java

- Simple shared data structures can be managed using **mutual exclusion locks** (‘mutexes’) and the `synchronized` keyword to delimit critical regions

- The JVM associates a separate mutex with each object. It acts like the ′busy′ flag except
  - There’s no need to spin while waiting for it – the thread is blocked
  - The race condition in lines 6–7 is avoided

- The `synchronized` keyword can be used in two ways – either applied to a method or applied to a block of code

- For example, suppose we want to maintain an invariant between multiple fields:

```
class BankAccounts {
  private int balanceA;
  private int balanceB;

  synchronized void transferToB (int v) {
    balanceA = balanceA - v;
    balanceB = balanceB + v;
  }
}
```

Locks in Java (2)

- When a synchronized method is called, the thread must lock the mutex associated with the object

- If the lock is already held by another thread then the caller is blocked until the lock becomes available

- Locks therefore operate on a *per-object* basis – that is, only one synchronized method can be called on a particular object at any time
  - ...similarly, it is OK for multiple threads to be calling the same method, so long as they do so on different objects

- Locks are **re-entrant**, meaning that a thread may call one synchronized method from another

- If a `static synchronized` method is called then the thread must acquire a lock associated with the *class* rather than on with an individual *object*

- The `synchronized` modifier cannot be used directly on classes or on fields

Locks in Java (3)

- The second form of the `synchronized` keyword allows it to be used within methods, e.g.

```
 1  void methodA (Object x) {
 2    synchronized (x) {
 3      System.out.println ("1");
 4    }
 5
 6    ...
 7
 8    synchronized (x) {
 9      System.out.println ("2");
10    }
11  }
```

- The `synchronized` region at line 2 locks the mutex associated with the object `x` refers to, performs the `println` operation at line 3 and then releases the lock at line 4

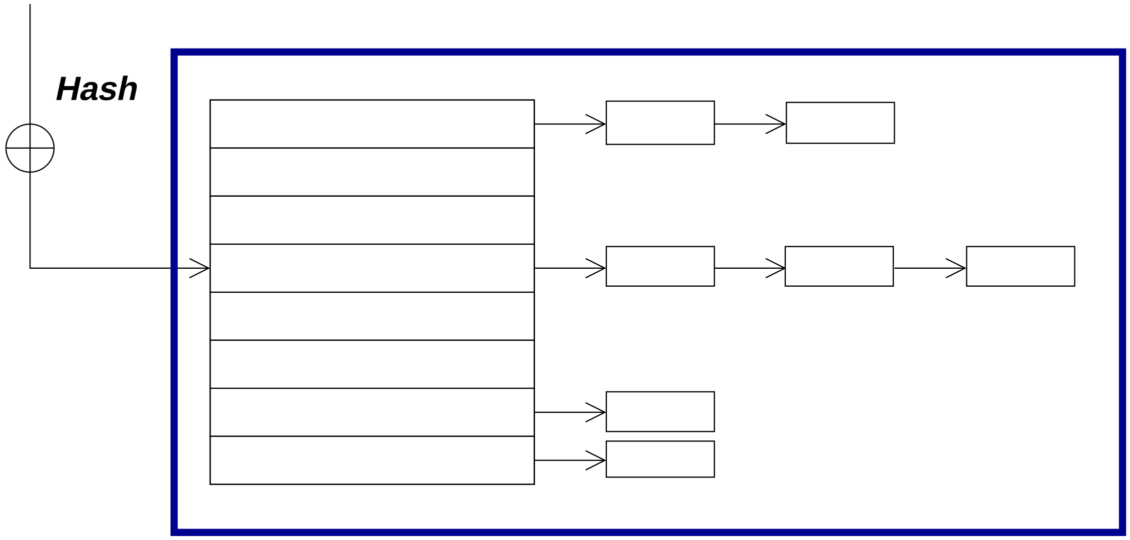
- The lock must be re-acquired at line 8

This kind of usage is good if an intervening operation, not requiring mutual exclusion, may take a long time to execute: other threads may acquire the lock

Compound data structures

➤ How can we use locks on a data structure built from multiple objects, e.g. a hashtable?

➤ One “big lock” associated with the hashtable object itself:



✔ Easy to implement

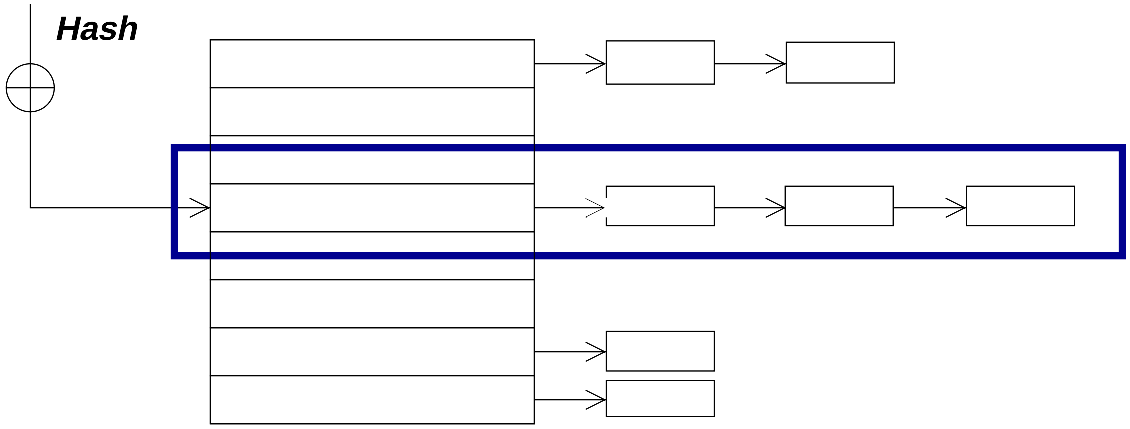
✔ “Obviously correct”

✔ Good performance under light load: only one lock to acquire/release per operation

✘ Poor performance in most other cases – only one operation can proceed at a time

Compound data structures (2)

- ➤ Separate "small locks", e.g. associated with each bucket of the hashtable:



- ✔ Operations using different buckets can proceed concurrently

- ✘ Harder to implement – e.g. consider resizing...

In general designing an effective fine-grained locking scheme is hard:

- ➤ A poor scheme may leave the program spending its time juggling locks rather than doing useful work

- ➤ Many locks does not automatically imply better concurrency

- ➤ Deadlock problems...

Exercises

10-1 Describe how mutual-exclusion locks provided by the `synchronized` keyword can be used to control access to shared data structures.

10-2 Describe what a *race condition* is, with the aid of example code.

"A Java class is safe for use by multiple threads if all of its methods are synchronized."

To what extent do you agree with this statement?

10-3 Suppose that, instead of using mutual exclusion locks, a programmer attempts to support critical regions by manipulating the running thread's scheduling priority in a class extending `java.lang.Thread`:

```
void enterCriticalRegion {
  oldPriority = getPriority ();
  setPriority (Thread.MAX_PRIORITY);
}

void exitCriticalRegion {
  setPriority (oldPriority);
}
```

What assumptions are needed to guarantee this works?

Lecture 11: Deadlock

Previous lecture

- Safety & liveness requirements
- Mutual exclusion locks

Overview of this lecture

- Deadlock
- Automatic detection
- Avoidance

Deadlock

Suppose that `a` and `b` refer to two different shared objects,

Thread P *Thread Q*

```
synchronized (a)       synchronized (b)
  synchronized (b)       synchronized (a)
  {                      {
    ...                    ...
  }                      }
```

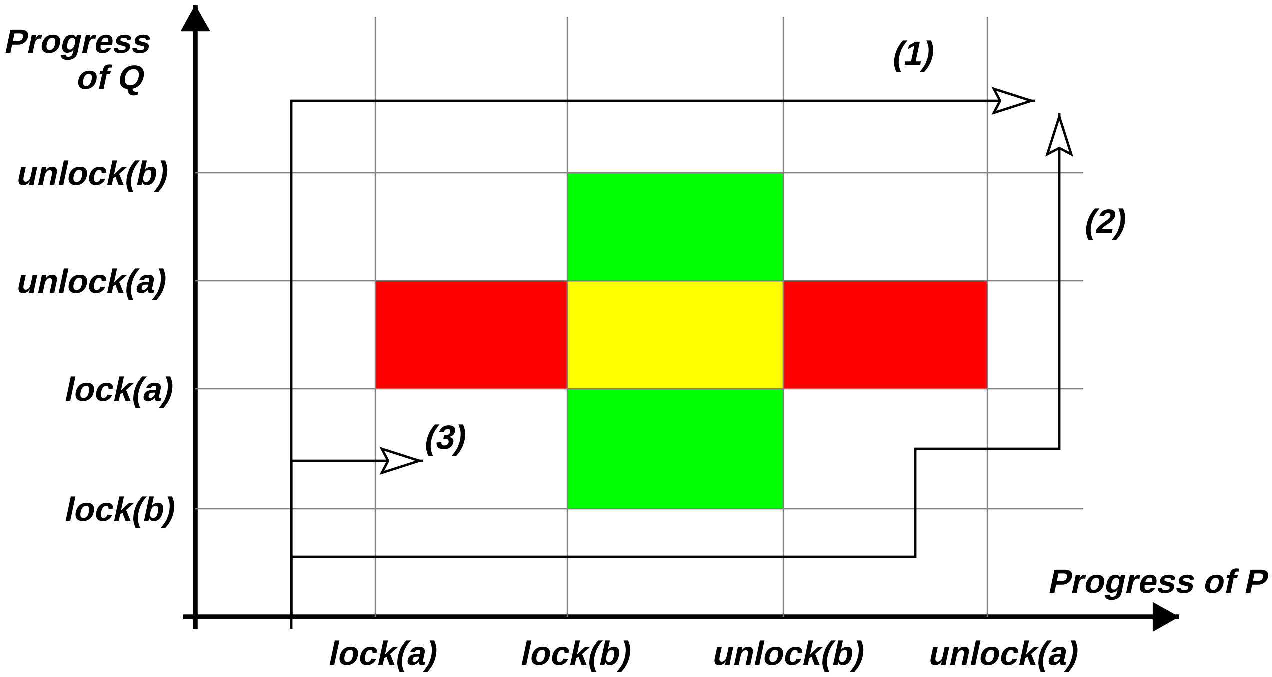
➤ If P locks both `a` and `b` then it can complete its operation and release both locks, thereby allowing Q to acquire them

➤ Similarly, Q may acquire both locks, then release them and then allow P to continue

✘ If P locks `a` and Q locks `b` then neither thread can continue: they are *deadlocked* waiting for the resources that the other has

Deadlock (2)

Whether this deadlock actually occurs depends on the dynamic behaviour of the applications. We can show this graphically in terms of the threads' progress:



- In the horizontal area one thread is blocked by the other waiting to lock `a`. In the vertical area it is lock `b`

- Paths `(1)` and `(2)` show how these threads may be scheduled without reaching deadlock

- Deadlock is *inevitable* on path `(3)` (but hasn't yet occurred in the position indicated)

Requirements for deadlock

If all of the following conditions are true then deadlock exists:

1. A resource request can be refused – e.g. a thread cannot acquire a mutual-exclusion lock because it is already held by another thread

2. Resources are held while waiting – e.g. when a thread blocks waiting for a lock it does not have to release any others it holds

3. No preemption of resources – e.g. once a thread acquires a lock then it's up to that thread to choose when to release it

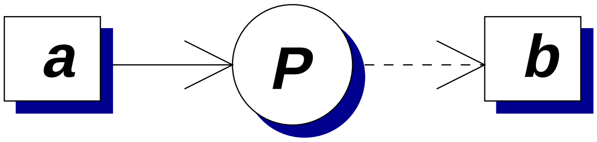
4. Circular wait – a cycle of threads exist such that each holds a lock requested by the next process in the cycle, and that request has been refused

In the case of mutual exclusion locks in Java, 1–3 are always true, and so the existence of a circular wait leads to deadlock

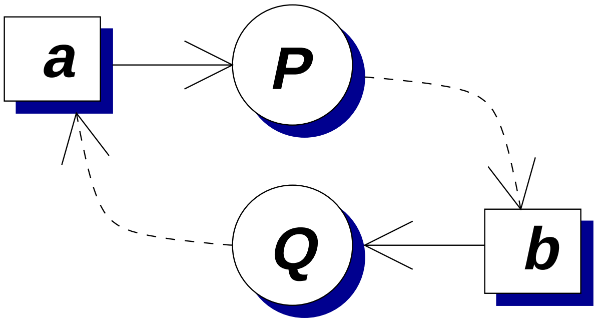
Object allocation graphs

An **object allocation graph** shows the various tasks in a system and the resources that they have acquired and are requesting. We'll use a simplified form in which resources are considered to be individual objects

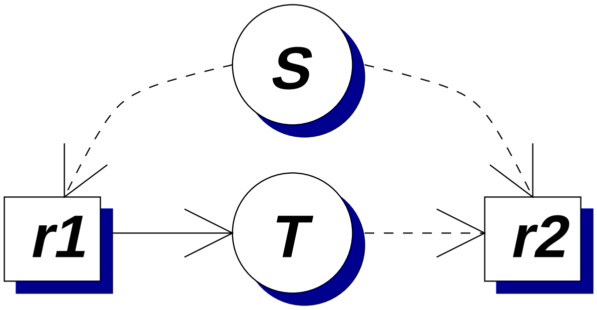
a is held by thread P and P is requesting object b:



a is held by P, b is held by Q:



Should r2 be allocated to S or T?



Deadlock detection

Deadlock can be detected by looking for cycles (as in the second example on the previous slide)

Let A be the object allocation matrix, with one thread per row and one column per object. A_{ij} indicates whether thread i holds a lock on object j

Let R be the object request matrix. R_{ij} indicates whether thread i is waiting to lock object j

We proceed by *marking* rows of A indicating threads that are *not* part of a deadlocked set. Initially no rows are marked. A working vector W indicates which objects are available

1. Select an unmarked row i such that $R_i \leq W$ – i.e. a thread whose requests can be met. Terminate if none
2. Set $W = W + A_i$, mark row i, and repeat

This identifies when deadlock *has occurred* – we may be interested in other properties such as whether deadlock is

- inevitable (must happen in some possible execution path)
- possible (may happen in some path)

Deadlock detection (2)

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

1. $W = (0, 0, 0, 1, 1)$
2. Thread 3's requests can be met ⇒ it's not deadlocked, so can continue and may release object 1
3. $W = (1, 0, 0, 1, 1)$
4. Thread 4's requests can now be met ⇒ it's not deadlocked
5. $W = (1, 0, 0, 1, 1)$

✘ Nothing more can be done: threads 1 and 2 are both deadlocked

Deadlock avoidance

A conservative approach:

- ➤ Require that each process identifies the maximum set of resources that it may ever lock, C_{ij}
- ➤ When thread i requests a resource then construct a hypothetical allocation matrix A' in which it has been made and a hypothetical request matrix B' in which every other process makes its maximum request
- ➤ If A' and B' do not indicate deadlock then the allocation is safe
- ✔ It does avoid deadlock – may be preferable to deadlock recovery
- ✘ Need to know maximum requests
- ✘ Run-time overhead
- ✘ What if there are no safe states?
- ✘ Objects are usually instantiated dynamically...

Deadlock avoidance (2)

It's often more practical to prevent deadlock by careful design. How else can we tackle the four requirements for deadlock?

- Use locking schemes that allow greater concurrency – e.g. multiple-readers, single-writer in preference to mutual exclusion

- Do not hold resources while waiting – e.g. acquire all necessary locks at the same time

- Allow preemption of locks and roll-back (not a primitive in Java if using built-in locks)

Two practical schemes that are widely applicable:

- Coalesce locks so that only one ever needs to be held – e.g. have one lock protecting all bank accounts

- Enforce a lock acquisition order, making it impossible for circular waits to arise, e.g. lock 2 'bank account' objects in account number order

...trade-off between simplicity of implementation and possible concurrency

Priority inversion

Another liveness problem in priority-based systems:

- Consider low, medium and high priority threads called $\mathtt{P}_{\mathtt{low}}$, $\mathtt{P}_{\mathtt{med}}$ and $\mathtt{P}_{\mathtt{high}}$ respectively.
  1. First $\mathtt{P}_{\mathtt{low}}$ starts, and acquires a lock on object `a`.
  2. Then the other two processes start.
  3. $\mathtt{P}_{\mathtt{high}}$ runs since highest priority, tries to lock `a` and blocks.
  4. Then $\mathtt{P}_{\mathtt{med}}$ gets to run, thus preventing $\mathtt{P}_{\mathtt{low}}$ from releasing `a`, and hence $\mathtt{P}_{\mathtt{high}}$ from running.

- Usual solution is **priority inheritence**:
  - associate with every lock the priority p of the highest priority process waiting for it.
  - then temporarily boost priority of *holder* of the lock up to p.
  - can use handoff scheduling to implement.

- Windows 2000 “solution”: priority boosts
  - checks if ∃ ready thread not run $\geq$ 300 ticks.
  - if so, doubles quantum & boosts priority to 15

- What happens in Java?

Priority inversion (2)

- With basic priority inheritance we can distinguish (assuming a uni-processor with strict-priority scheduling)
  - *Direct blocking* of a thread waiting for a lock
  - *Push-through blocking* of a thread at one priority by an originally-lower-priority thread that has inherited a higher priority
- A thread $\mathtt{P}_{\mathtt{high}}$ can be blocked by each lower priority thread $\mathtt{P}_{\mathtt{low}}$ for at most one of $\mathtt{P}_{\mathtt{low}}$'s critical sections
- A thread $\mathtt{P}_{\mathtt{high}}$ can experience push-through blocking for any lock accessed by a lower-priority thread and by a job which has (or can inherit) a priority $\geq \mathtt{P}_{\mathtt{high}}$

This can give an upper bound on the total blocking delay a thread encounters, but

- chains of blocking may limit bounded & practical performance: the former a particular problem for real-time systems
- remember: does not prevent deadlock

Exercises

11-1 In the *dining philosophers* problem, five philosophers spend their time alternately *thinking* and *eating*. The each have a chair around a common, circular table. In the centre of the table is a bowl of spaghetti and the table is set with five forks, one between each pair of adjacent chairs. From time to time philosophers may get hungry and try to pick up the two closest forks. A philosopher may only pick up one fork at a time. It is an axiom of philosophic thought that one is only allowed to eat with the aid of two forks and that, of course, both forks are put down while thinking.

Model this problem in Java using a separate thread for each philosopher.

Does your simulation illustrate either deadlock or livelock? If so then what changes could you make to avoid it?

11-2 Write a Java class that attempts to cause priority inversion with a medium-priority thread preventing a high-priority thread from making progress. Do you observe priority inversion in practise?

Exercises (2)

11-3 Show how the deadlock detection algorithm can be extended to manage locks that support a separate *write* mode (in which it can be held by at most one thread at a time) and a *read* mode (in which it can be held by multiple threads at once). The lock cannot be held in both modes at the same time.

11-4* One way to avoid deadlock is for a thread to simultaneously acquire all of the locks it needs for an operation. However, Java's `synchronized` keyword can only acquire or release a single lock at a time.

Sketch the design of a class `LockManager` that implements the `LockManagerIfc` interface (below) so that the `doWithLocks` operation:

1. appears to atomically acquire locks on all of the objects in the array `o`,
2. invokes `op.doOp(arg)` keeping the result of that method as the eventual result of `doWithLocks`,
3. releases all of the locks initially acquired.

Exercises (3)

```
interface Operation {
  Object doOp(Object arg);
}

interface LockManagerIfc {
  Object doOpWithLocks (Object o[],
                        Operation op,
                        Object arg);
}
```

[Hint: one approach is to assume initially some mechanism for mapping each object to a unique integer value and to then examine how to provide that mechanism.]

Past exam questions: 1998 Paper 4 Q2

Lecture 12: Condition synchronization

Previous lecture

- Deadlock
- Ordered acquisition
- Priority inversion & inheritance

Overview of this lecture

- Condition synchronization
- `wait`, `notify`, `notifyAll`

Limitations of mutexes

- Suppose we want a one-cell buffer with a `put` operation (store something if cell empty) and a `remove` operation (read something if anything there):

```
1   class Cell {
2     private int value;
3     private boolean full;
4
5     public synchronized int removeValue () {
6       if (full) {
7         full = false;
8         return value;
9       } else {
10        /* ??? */
11      }
12
13      ...
14
15  }
```

- What can we put in line 10?

Limitations of mutexes (2)

- We could keep testing `full` (a ‘spin lock’)...

```
1   class Cell { /* Incorrect */
2     private int value;
3     private boolean full;
4
5     public int removeValue () {
6       while (full) {
7         /* Nothing */
8       }
9       synchronized (this) {
10        full = false;
11        return value;
12      }
13    }
14  }
```

But this is:

- ✘ Incorrect: if multiple threads try to remove values then they may both see `full` false at line 6 and independently execute 9–12

- ✘ Inefficient: threads consume CPU time while waiting ⇒ this may impede a thread about to put a value into the cell

Limitations of mutexes (3)

- Another problem: what if we want to enforce some other kind of concurrency control?
- e.g. if we identify *read-only* operations which can be executed safely by multiple threads at once
- e.g. if we want to control which thread gets next access to the shared data structure
  - perhaps to give preference to threads performing update operations
  - or to enforce a first-come first-served regime
  - or to choose on the basis of the threads′ scheduling priority?
- All that mutexes are able to do is to prevent more than one thread from running the code on a particular object at the same time

Condition synchronization

- What we might like to write:

```
1  class Cell { /* Not valid Java */
2    private int value;
3    private boolean full;
4
5    public synchronized int removeValue () {
6      wait_until (full);
7      full = false;
8      return value;
9    }
```

- Line 6 would have the effect of
  - If `full` is false blocking the caller atomically with doing the test and releasing the lock on the cell (to allow another thread to put items into it)
  - Unblocking the thread when `full` becomes true and the lock can be re-acquired (so the lock prevents multiple ‘removes’ of the same value)

- We can’t directly implement `wait_until` in Java
  - Call-by-value ⇒ `full` would only be evaluated once
  - We’d need some way of releasing the lock on the `Cell`

Condition variables

- **Condition variables** provide one solution

- In general, condition variables support two kinds of operation:
  - a *cv.CVWait(m)* operation causes the current thread to atomically release a lock on mutex `m` and to block itself on condition variable `cv`, re-acquiring the lock on `m` before it completes
  - a *cv.CVNotify()* operation that wakes up threads blocked on `cv`

- Such operations would be more cumbersome in this simple example than a general `wait_until`

```
class Cell { /* Not valid Java */
  private int value;
  private boolean full;
  private ConditionVariable cv =
            new ConditionVariable();

  public synchronized int removeValue () {
    while (!full) cv.CVWait (this);
    full = false;
    cv.CVNotify ();
    return value;
  }
```

Condition variables in Java

- Java doesn't (currently) provide individual condition variables in this way

- Instead, each object `o` has an associate condition variable which is accessed by
  - `o.wait()`
  - `o.notify()`
  - `o.notifyAll()`

- `o.wait()` acts as the equivalent of `cv.CVWait(o)` on the condition variable associated with `o`

- This means that `o.wait()` *always* releases the mutual exclusion lock held on `o`

- ...and therefore that the caller may *only* use `o.wait()` when holding that lock (otherwise `IllegalMonitorStateException` is thrown)

- `notify()` unblocks exactly one thread (if any are waiting), otherwise it does nothing

- `notifyAll()` unblocks all waiting threads

Condition variables in Java (2)

```
class Cell {
  private int value;
  private boolean full = false;

  public synchronized int removeValue ()
            throws InterruptedException
  {
    while (!full) wait ();

    full = false;
    notifyAll ();
    return value;
  }

  public synchronized void putValue (int v)
            throws InterruptedException
  {
    while (full) wait ();

    full = true;
    value = v;
    notifyAll ();
  }
}
```

Condition variables in Java (3)

- Line 8 causes a thread executing `removeValue` to block on the condition variable until the cell is full

- Line 10 updates the object to mark it empty

- Line 11 notifies all threads currently blocked on the condition variable

- Similarly, line 18 causes a thread executing `putValue` to block on the condition variable until the cell is empty

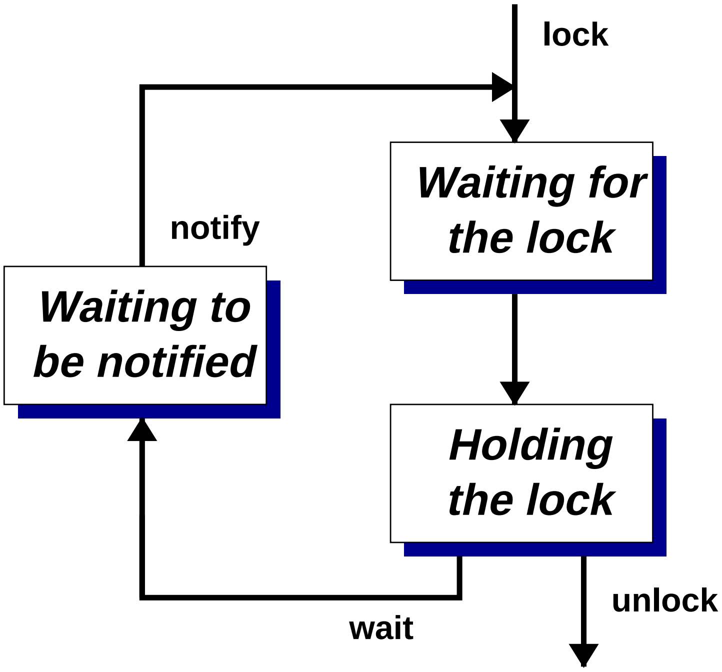
- Lines 20–21 update the fields to mark the cell full and store the value in it, 22 notifies waiting threads

`InterruptedException` will be thrown in the thread is interrupted while waiting. In general it should be propagated until it can be handled. Be wary of writing:

```
try {
  while (full) wait ();
} catch (InterruptedException ie) {
  /* Nothing */
}
```

Condition variables in Java (4)

- Note how there are now two different ways that a thread may be blocked:



- It may have entered a `synchronized` region for an object and found that the associated mutual exclusion lock is already held

- It may have called `wait` on an object and blocked until the associated condition variable is notified

- When notified, the thread must compete for the lock once more

Condition variables in Java (5)

- When should `notify()` be used, when should `notifyAll()` be used?
- With `notifyAll()` the programmer must ensure that every thread blocked on the condition variable can continue safely
  - e.g. line 8 in the example surrounds the `wait()` operation with a `while` loop
  - if a 'removing' thread is notified when there is no work for it, then it just waits again
- `notify()` selects arbitrarily between the waiting threads: the programmer must therefore be sure that the exact choice does not matter
- In the `Cell` example we can't use `notify()` because although only one thread is to be woken
  - a successful `removeValue` must allow a call blocked in `putValue` to proceed

`notify()` does not guarantee to wake the longest waiting thread

Suspending threads

- The `suspend` and `resume` methods on `java.lang.Thread` allow one thread to temporarily stop and start the execution of another

```
Thread t = new MyThread ();
t.suspend ();
t.resume ();
```

- As with `stop`, the `suspend` and `resume` methods are deprecated

- This is because the use of `suspend` can lead to deadlocks if the target thread is holding locks. It also risks missed wake-up problems:

```
1   public int removeValue () {
2     if (!full) {
3       Thread.suspend (Thread.currentThread());
4     }
```

The status may change between executing 2 and 3 ⇒ a lost wakeup problem

`suspend` should never be used: even if the program does not explicitly take out locks the JVM may use locks in its implementation

Exercises

12-1 Describe the facilities in Java for restricting concurrent access to critical regions. Explain how shared data can be protected through the use of objects.

12-2 Consider the following class definition:

```
class Example implements Runnable {
  public static Object o = new Object();
  int count = 0;

  public void run() {
    while (true) {
      synchronized (o) {
        count ++;
      }
    }
  }
}
```

(*i*) Show how to start two threads, each executing this `run` method on separate instances of `Example`.

(*ii*) When this program runs, only one of the `count` fields is found to increment, even though threads are scheduled preemptively. Why might this be?

Past exam questions: 2002 Paper 5 Q4, 1999 Paper 4 Q3

Lecture 13: Worked examples

Previous lecture

- Condition synchronization
- `wait`, `notify`, `notifyAll` in Java

Overview of this lecture

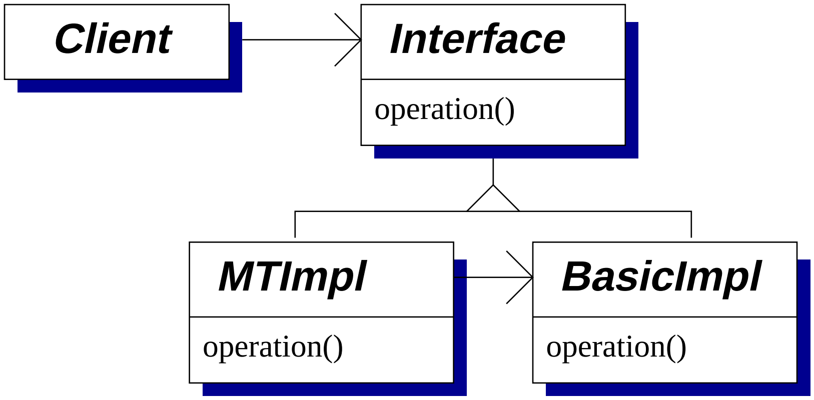
- Further examples of how to use these facilities
- Common design features

Design

Suppose that we wish to have a shared data structure on which multiple threads may make read-only access, or a single thread may make updates

- How can this be implemented using the facilities of Java,
  - In terms of a well-designed OO structure?
  - In terms of the concurrency-control features?

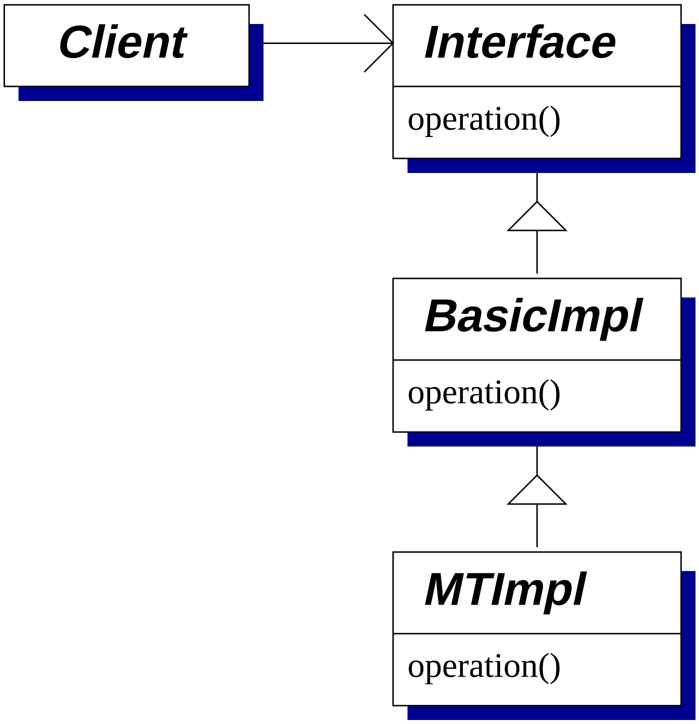
  One option is based on delegation and the Adapter pattern,



- `BasicImpl` provides the actual data structure implementation, conforming to `Interface`. The class `MTImpl` wraps each operation with appropriate code for its use in a multi-threaded application, delegating calls to an instance of `BasicImpl`

Design (2)

➤ How does that compare with:



✔ Sub-classes enforce encapsulation and mean that only one instance is needed

✔ Delegation may be easier, just use `super.operation()`

✘ Separate sub-classes are needed for each implementation of `Interface`

✘ Composition of wrappers is fixed at compile time

Design (3)

In each of these cases the class `MTImpl` will define methods that can be split into three sections:

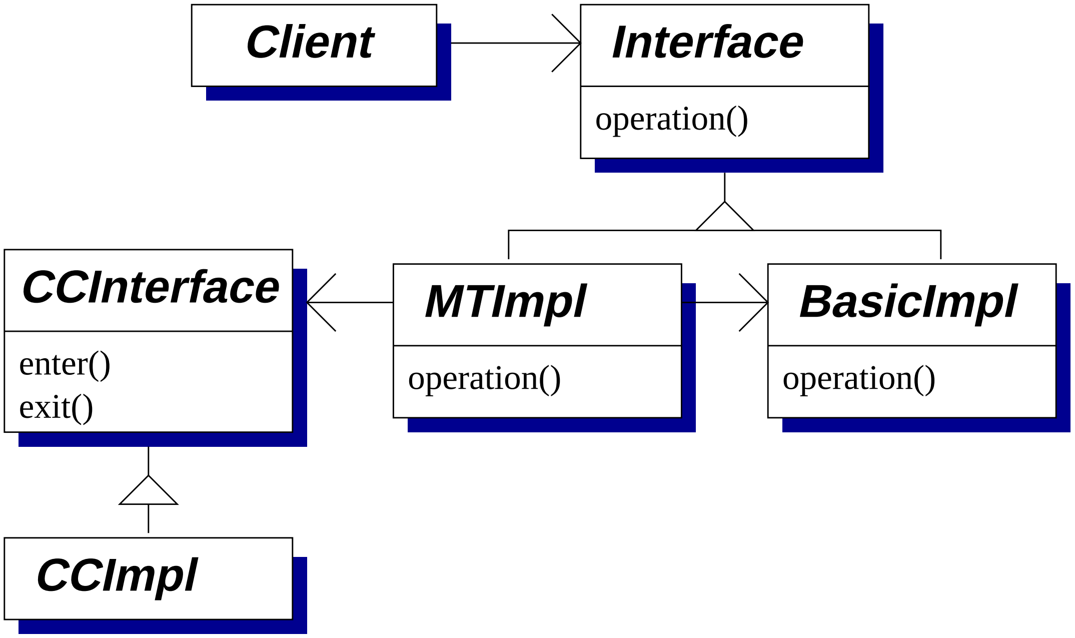
1. An **entry protocol** responsible for concurrency control – usually waiting until it is safe for the operation to continue

2. *Delegation* to the underlying data structure implementation (either by an ordinary method invocation on an instance of `BasicImpl` or a call using the `super` keyword)

3. An **exit protocol** – generally selecting the next thread(s) to perform operations on the structure

This common structure often motivates further separation of concurrency control protocols from the data structure

Design (4)



`MTImpl` now just deals with delegation, wrapping each invocation on `Interface` with appropriate calls to `enter()` and `exit()` on a general concurrency-control interface (`CCInterface`).

Sub-classes, e.g. `CCImpl`, provide specific entry/exit protocols. A factory class may be used to instantiate and assemble these objects

- ✔ Concurrency-control protocols can be shared
- ✔ Only a single `MTImpl` class is needed per data structure interface

Multiple readers, single writer

- As a more concrete example:

```
interface MRSW {
  public void enterReader ()
    throws InterruptedException;
  public void enterWriter ();
    throws InterruptedException;
  public void exitReader ();
  public void exitWriter ();
}
```

- This could be used as:

```
class MTHashtable implements Dictionary {
  ...
  Object get (Object key) {
    object result;
    cc.enterReader ();
    try {
      result = ht.get (key);
    } finally {
      cc.exitReader ();
    }
  }
```

- Why is `try`... `finally` used like this? How should `InterruptedException` be managed?

Multiple readers, single writer (2)

- We'll now look at implementing an example protocol, MRSW

```
class MRSWImpl1 implements MRSW {
  private int numReaders = 0;
  private int numWriters = 0;
```

- A reader must wait until `numWriters` is zero. A writer must wait until both fields are zero:

```
  synchronized void enterReader ()
        throws InterruptedException
  {
    while (numWriters > 0)
      wait ();
    numReaders ++;
  }

  synchronized void enterWriter ()
        throws InterruptedException
  {
    while ((numWriters > 0) ||
           (numReaders > 0))
      wait ();
    numWriters ++;
  }
```

Multiple readers, single writer (3)

The exit protocols are more straightforward:

```
  synchronized void exitRead () {
    numReaders --;
    notifyAll ();
  }

  synchronized void exitWrite () {
    numWriters --;
    notifyAll ();
  }
}
```

✔ Simple design: (1) create a class containing the necessary fields (2) write entry protocols that keep checking these fields and waiting (3) write exit protocols that cause any waiting threads to assess whether they can continue.

✘ `notifyAll()` may cause too many threads to be woken – the code is safe but may be inefficient

Is that efficiency likely to be a problem?

Could `notify()` be used instead?

➤ 1999 Paper 4 Q3

Giving writers priority

- …how else could `MRSW` be implemented?

```
 1 class PrioritizedWriters implements MRSW {
 2   private int numReaders = 0;
 3   private int numWriters = 0;
 4   private int waitingWriters = 0;
 5
 6   synchronized void enterReader ()
 7             throws InterruptedException
 8   {
 9     while ((numWriters > 0) || (waitingWriters > 0))
10       wait ();
11     numReaders ++;
12   }
13
14   synchronized void enterWriter ()
15            throws InterruptedException
16   {
17     waitingWriters ++;
18     while ((numWriters > 0) || (numReaders > 0))
19       wait ();
20     waitingWriters --;
21     numWriters ++;
22   }
```

- What about interruptions at 19?

First-come first-served ordering

- Suppose now we want an ordinary lock that provides FCFS semantics – the longest waiting thread is given access next

```
class FCFSImpl implements CCInterface {
  private int currentTurn = 0;
  private int nextTicket = 0;
```

Threads take a ticket and wait until it becomes their turn:

```
  synchronized void enter ()
    throws InterruptedException
  {
    int myTicket = nextTicket ++;
    while (currentTurn < myTicket)
      wait ();
  }

  synchronized void exit ()
  {
    currentTurn ++;
    notifyAll ();
  }
}
```

First-come first-served ordering (2)

- ✔ The implementation is simple
- ✘ If a thread is interrupted during `wait` then its ticket is lost
- ✘ `notifyAll()` will wake all threads waiting in `enter()` on this object – in this case we know that only one can continue
- ✘ What happens if the program runs for a long time and `nextTicket` overflows?

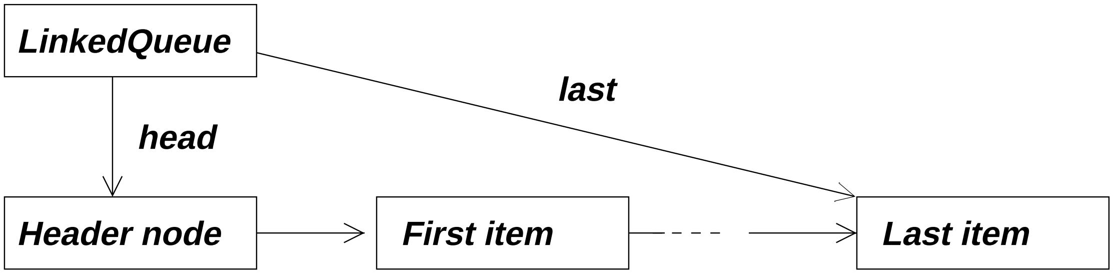
Resolving these issues in an effective way depends on the context in which the class is being used, e.g.

- ➤ *Lots of waiting threads and frequent contention:* have an explicit queue of per-thread objects and use `notify()` on the object at the head of the queue
- ➤ *Safe with arbitrary interruption:* allow the `enter()` method to manage aborted waiters, e.g. using a queue as above with an `abandoned` field in each entry
- ➤ *No undetected failures:* would `long`s ever overflow here?

Splitting locks

Our last example shows how instances of `java.lang.Object` can be used to create separate locks to protect different parts of a data structure

- This technique is generally useful to obtain more concurrency
  - e.g. different locks to protect different operations that are safe concurrently,
  - in a list, 'push on tail' is usually safe to run in parallel with 'pop from head'



We'll use one lock to protect the `head` field, one for the `last` field and then separate locks for each entry in the list protecting its `next` field

- Do this after seeing that a lack of concurrency is a problem; the code is rarely as clear and often is wrong :-)
- More of this in Part-2 Advanced Systems Topics

Splitting locks (2)

```
// Based on 2.4.2.4 in Doug Lea’s book
class LinkedQueue {
  Node headNode = new Node (null);
  Node lastNode = headNode;
  Object headField = new Object();
  Object lastField = new Object();

  public void pushTail (Object x) {
    Node n = new Node (x);
    synchronized (lastField) {
      synchronized (lastNode) {
        // insert after last &
        // update last
      }
    }
  }

  public Object popHead () {
    synchronized (headField) {
      synchronized (headNode) {
        // read value from the
        // node after head & make
        // that node the new head
      }
    }
  }
}
```

Exercises

13-1 In the `PrioritizedWriters` example the `numWriters` field is supposed to be a count of the number of threads executing in the body of the `enterWriter` method. How can this invariant be broken? Correct the code.

13-2 Update the `FCFSImpl` class so that it

(*i*) allows threads to safely be interrupted during `wait`

(*ii*) uses `notify()` instead of `notifyAll()`

(*iii*) will not suffer from the ticket counter overflowing.

How does the performance of your new implementation compare with that of the basic one?

13-3 Update the `FCFSImpl` class so that the lock can be held *recursively*: i.e. a thread already holding the lock can make subsequent calls to `enter()` without blocking. The lock is released only when a matched number of calls to `exit()` have been made.

13-4* Complete the implementation of the `LinkedQueue` class by giving a suitable definition for `Node` and filling in the missing code in `pushTail` and `popHead`. Can you write `pushHead`? What about `popTail`?

Past exam question: 2003 Paper 5 Q4

Lecture 14: Low-level synchronization

Previous lecture

- Integrating concurrency control
- Several examples: MRSW, FCFS
- General design methods for other cases

Overview of this lecture

- Implementing mutexes and condvars
- Direct scheduler support
- Semaphores
- Event counts / sequencers
- Alternative language features

Implementing mutexes and condvars

- Nowadays mutexes and condvars are usually implemented using a combination of:

  - operations provided by the scheduler to suspend and resume threads

  - atomic assembly language instructions, e.g. **compare-and-swap**

    ```
    seen = CAS(addr, old, new)
    ```

    Read from address `addr`, if it matches `old` then store `new` at that address. Return the value `seen`.

- Care is needed to avoid the problems seen with Java's `Thread.suspend()` and `Thread.resume()` methods

- Some implementations provide "lower level" primitives and build mutexes and condvars over these:

  - semaphores

  - event counts and sequencers

  This layering is no longer typical, although we'll still briefly look at these other primitives

Implementing mutexes

- Using CAS we can build a simple **spin-lock**:

```
class Mutex {
  int lockField = 0;

  void lock() {
    while (CAS(&lockField, 0, 1) != 0) {
      /* Someone else has the lock */
    }
  }

  void unlock() {
    lockField = 0;
  }
}
```

- Many performance problems: most importantly the `lock` operation consumes CPU time while waiting

- Also, if multiple threads are waiting, then the data-cache line holding `lockField` will bounce between different CPUs on a multi-processor machine
  - More about cache implications in Part-2 Advanced Systems Topics

Implementing mutexes (2)

- To avoid spinning, each mutex usually has a queue of blocked threads associated with it

- A thread attempts to acquire the lock directly (e.g. using `CAS`), if it succeeds then it's done

- If it doesn't succeed then it adds itself to the queue and invokes a `suspend` operation on the scheduler

- After releasing the lock, a thread checks whether the queue is empty

- If the queue is non-empty the thread selects an entry and `resumes` it. To avoid lost-wake up problems, either outstanding `resume` operations must be remembered:

```
Thread A, LOCK:      Thread B, UNLOCK:
see lock held
add A to queue
                     release lock
                     take A from queue
                     resume A
suspend A
```

or the scheduler should support a "disable thread switches" operation

Implementing condvars

- Condition variables are more intricate, but can build on very similar techniques

- Recall that a condition variable in general supports two operations:

  - a *cv.CVWait(m)* operation causes the current thread to atomically release a lock on mutex `m` and to block itself on condition variable `cv`, re-acquiring the lock on `m` when it is woken

  - a *cv.CVNotify()* operation that causes threads blocked on `cv` to continue

- Internally, in a typical implementation, each condvar has private fields that hold:

  - a queue of threads that are waiting on the condition variable

  - an additional mutex `cvLock` that is used to give the atomicity required by `CVWait`

Implementing condvars (2)

- `cv.CVWait(m)` then proceeds by:

```
Acquire mutex cv.cvLock
  Add the current thread to cv.queue
  Release mutex m
Release mutex cv.cvLock
Suspend current thread
Re-acquire mutex m
```

- and `cv.CVNotify()` proceeds by:

```
Acquire mutex cv.cvLock
  Remove one thread from cv.queue
  Resume that thread
Release mutex cv.cvLock
```

- Again, it's important to avoid lost wake-up problems – typically by remembering resumptions

- A real implementation is more complex – e.g. in Java it's necessary to deal with threads being interrupted
  - (See `linuxthreads/condvar.c` for a Linux implementation)

Semaphores

- These examples have used the language-level *mutexes* and *condition variables*

- **Semaphores** provide basic operations on which the language-level features could be built. In Java-style pseudo-code:

```
class CountingSemaphore {

  CountingSemaphore (int x) {
    ...
  }

  native void P();

  native void V();
}
```

- P (sometimes called *wait*) decrements the value and then blocks if the result is less than zero

- V (sometimes called *signal*) increments the value and then, if the result is zero or less, selects a blocked thread and unblocks it

- Using semaphores directly is intricate – the programmer must ensure `P()`/`V()` are paired correctly

Programming with semaphores

- Typically the integer value of a counting semaphore is used to represent the number of instances of some resource that are available, e.g.:

```
class Mutex {
  CountingSemaphore sem;

  Mutex () {
    sem = new CountingSemaphore (1);
  }

  acquire () {
    sem.P();
  }

  release () {
    sem.V();
  }
}
```

- The mutex is considered unlocked when the value is 1 (it is initialized unlocked)

- ...and locked when the value is 0 or less

- How does this mutex differ from a Java-style one?

Programming with semaphores (2)

```
class CondVar { // Not examinable!
  int numWaiters = 0;
  Mutex cv_lock = new Mutex();
  CountingSemaphore cv_sleep =
          new CountingSemaphore (0);

  void CVWait (Mutex m) {
    cv_lock.acquire ();
    numWaiters ++;
    m.release ();
    cv_lock.release ();
    cv_sleep.P ();
    m.acquire ();
  }

  void CVNotify () {
    cv_lock.acquire ();
    if (numWaiters > 0) {
      cv_sleep.V();
      numWaiters --;
    }
    cv_lock.release ();
  }
}
```

Event counts and sequencers

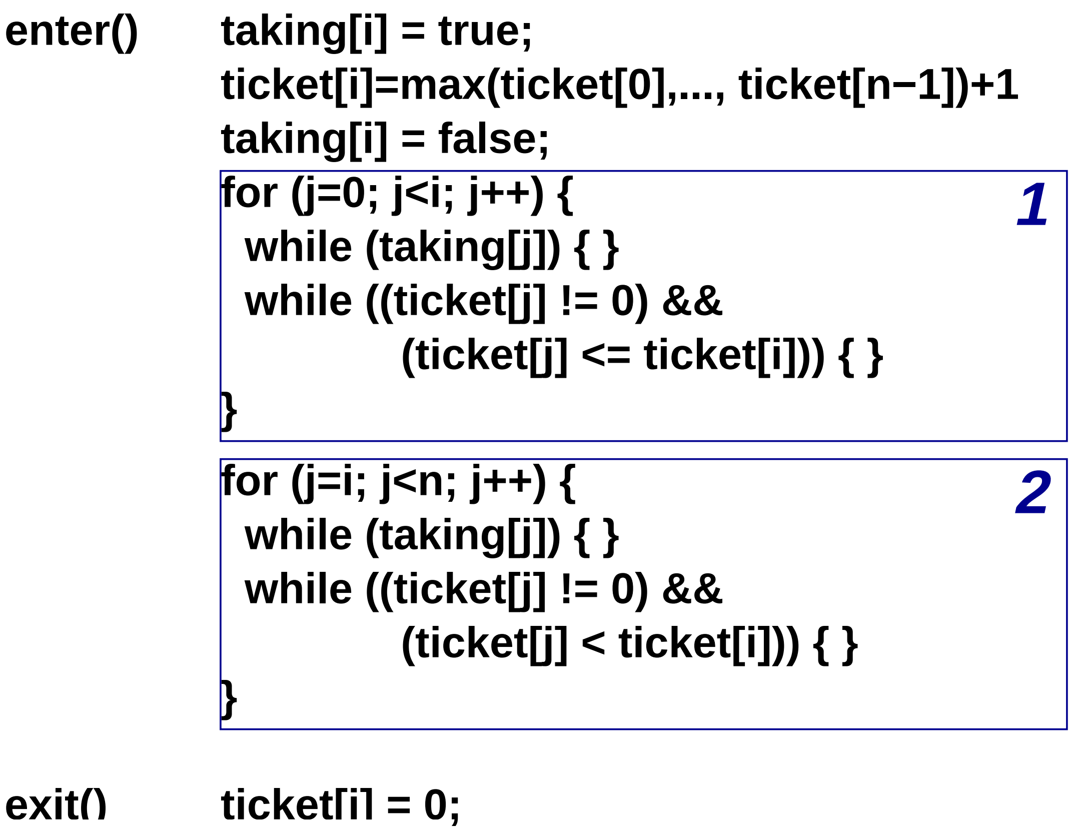
A further style of concurrency control is presented by **event count** and **sequencer** primitives

- An event count is represented by a positive integer, initialized to zero, supporting the following atomic operations:
  - `advance()` – increment the value by one, returning the new value
  - `read()` – return the current value
  - `await(i)` – wait until the value is greater than or equal to `i`
- A sequencer is again represented by a positive number, initialized to zero, supporting a single atomic operation:
  - `ticket()` – increment the value by one, returning the old value
- Mutual exclusion is easy: a thread takes a ticket entering a critical region and then invokes `await` to receive its turn (c.f. `FCFSImpl`)
- The values returned by `await` can be used directly in implementing a single-producer single-consumer N-slot buffer: they give the modulo-N indices to read/write

Mutexes without hardware support

- What can we do if there isn't a `cas` or `tas` instruction, just atomic read and write? (e.g. the ARM7 only has a `swap` operation)

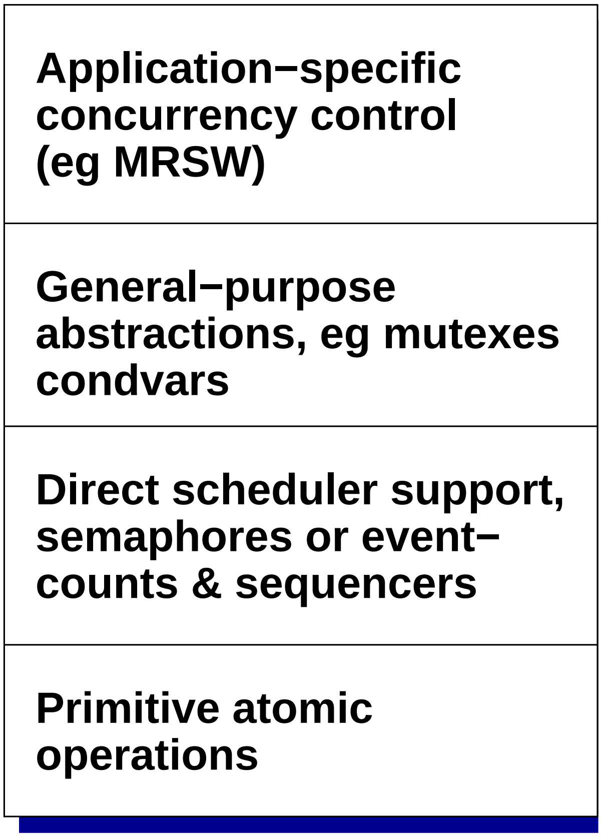
- 'Bakery' algorithm due to Lamport (1974) – this algorithm is now an example: not for practical use!



```
enter()     taking[i] = true;
            ticket[i]=max(ticket[0],..., ticket[n−1])+1
            taking[i] = false;
            for (j=0; j<i; j++) {
              while (taking[j]) { }
              while ((ticket[j] != 0) &&
                        (ticket[j] <= ticket[i])) { }
            }
            for (j=i; j<n; j++) {
              while (taking[j]) { }
              while ((ticket[j] != 0) &&
                        (ticket[j] < ticket[i])) { }
            }

exit()      ticket[i] = 0;
```

- Threads enter the critical region in ticket order, using their IDs (`i`) as a tie-break

Recap



Application−specific concurrency control (eg MRSW)

General−purpose abstractions, eg mutexes condvars

Direct scheduler support, semaphores or event−counts & sequencers

Primitive atomic operations

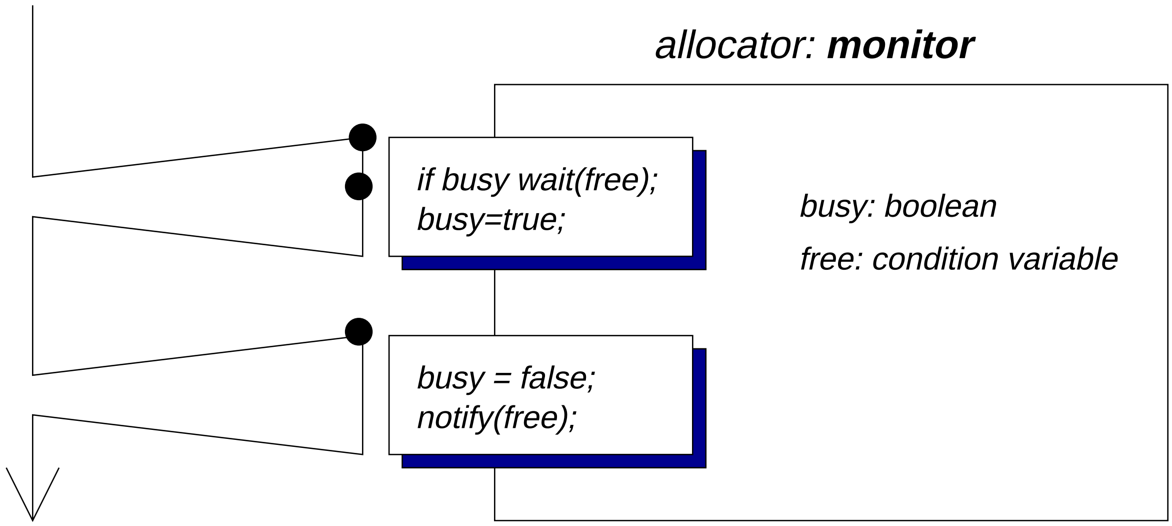
The details of exactly what is implemented where vary greatly between systems, e.g.

- Whether the thread scheduler is implemented in user-space or in the kernel
- Which synchronization primitives can be used between address spaces

Similarly, unless the application builds it, FCFS semantics and fairness are rarely guaranteed

Alternative language features

A **monitor** is an abstract data type in which mutual exclusion is enforced between invocations of its operations. Often depicted graphically showing the internal state and external interfaces, e.g. in pseudo-code



When looking at a definition such as this, independent of a specific language, it's important to be clear on what semantics are required of `wait` and `notify`

- Does `notify` wake at most one, exactly one or more than one waiting thread?
- Does `notify` cause the resumed thread to continue immediately (if so, must the notifier exit the monitor)?

Alternative language features (2)

An **active object** achieves mutual exclusion between operations by (at least conceptually) having a dedicated thread that performs them on behalf of external callers, e.g.

```
loop
  SELECT
    when count < buffer-size
      ACCEPT insert(param) do
        [insert item into buffer]
      end;
    increment count;
    [manage ref to next slot for insertion]
    or when count > 0
      ACCEPT remove(param) do
        [remove item from buffer]
      end;
    decrement count;
    [manage ref to next slot for removal]
  end SELECT
end loop
```

- Guarded ACCEPT statements provide operations and pre-conditions that must hold for their execution

- Management code occurs outside the ACCEPT statements

Exercises

14-1 Using the `CountingSemaphore` class (and not the `synchronized` keyword) implement a *sequencer*. The sequencer should hold a single positive number, initialized to zero, and support an atomic operation `ticket()` which increments the value by one and returns the old value.

14-2 Using the example `EventCount` and `Sequencer` classes, implement a single-cell buffer supporting an arbitrary number of producers and consumers, but holding only a single value at once.

14-3* A *binary semaphore* in a simplified version of the *counting semaphore* from the slides. Rather than an integer count value it has a binary flag. $\mathtt{P}_b$ blocks (if necessary) until the flag is set and then atomically clears it. $\mathtt{V}_b$ sets the flag (atomically unblocking one thread, if any, blocked in $\mathtt{P}_b$ on that semaphore).

(*i*) In pseudo-code, show how a binary semaphore can be built using atomic compare-and-swap (`cas`) or test-and-test (`tas`).

(*ii*) In pseudo-code, show how a counting semaphore can be built using binary semaphores. Your solution may need more than one binary semaphore and another field to hold the count value.

Exercises (2)

14-4* Some data structures can be implemented directly using the `cas` primitive without needing mutual exclusion locks. Suppose that a Java-like language supports a `cas` operation on fields. Show how a single-ended queue could be defined (implemented using a singly-linked list) supporting `push` and `pop` operations at the head of the queue.

Past exam questions: 1998 Paper 3 Q2

Lecture 15: Distributed systems

Previous section

- Multi-threaded programs
- Communication between threads

Overview of this section

- Distributed systems
- Naming
- Network communication
- Compound operations
- Crash-tolerance

Communication between processes

What problems emerge when communicating

- between separate address spaces
- between separate machines?

How do those environments differ from previous examples?

Recall that

- within a process, or with a shared virtual address space, threads can communicate naturally through ordinary data structures – object references created by one thread can be used by another
- failures are rare and at the granularity of whole processes (e.g. `SIGKILL` by the user)
- OS-level protection is also performed at the granularity of processes – as far as the OS is concerned a process is running on behalf of one user

Communication between processes (2)

Introducing separate address spaces means that data is not directly shared between the threads involved

- Access mechanisms and appropriate protection must be constructed
- At a low-level the representation of different kinds of data may vary between machines – e.g. big endian v little endian
- Names used may require translation – e.g. object locations in memory (at a low-level) or file names on a local disk (at a somewhat higher level)

Any communicating components need

- to agree on how to exchange data – usually by the sender **marshalling** from a local format into an agreed common format and the receiver **unmarshalling**

  — similar to using the serialization API to read/write an object to a file on disk

- to agree on how to name shared (or shareable) entities

Distributed systems

More generally, four recurring problems emerge when designing distributed systems:

- Components execute in parallel

  — maybe on machines with very different performance

- Communication is not instantaneous

  — and the sender does not know when/if a message is received

- Components (and/or their communication links) may fail independently

  — usually need explicit failure detection and robustness against failed components/links restarting

- Access to a global clock cannot be assumed

  — different components may observe events in a different order

To varying degrees we can provide services to address these problems. Is complete transparency possible?

Distributed systems (2)

Focus here is on basic naming and communication. Other courses cover access control (Part 1B: OS, Intro. Security) and algorithms (Part 2: Distributed Systems, Advanced Systems Topics)

We'll look at two different communication mechanisms:

➤ Remote method invocation

- ✔ Remote invocations look substantially like local calls: many low-level details are abstracted
- ✘ Remote invocations look substantially like local calls: the programmer must remember the limits of this transparency and still consider problems such as independent failures
- ✘ Not well suited to streaming or multi-casting data

➤ Low-level communication using network sockets

- ✔ A 'lowest-common-denominator': the TCP & UDP protocols are available on almost all platforms
- ✘ Much more for the application programmer to think about; many wheels to re-invent

Interface definition

The provider and user of a network service need to agree on how to access it and what parameters / results it provides. In Java RMI this is done using Java interfaces

- ✔ Easy to use in Java-based systems
- ✘ What about interoperability with other languages?

Java RMI is rather unusual in using ordinary language facilities to define remote interfaces. Usually a separate **Interface Definition Language** (IDL) is used

- ➤ This provides features common to many languages
- ➤ The IDL has **language bindings** that define how its features are realized in a particular language
- ➤ An IDL compiler generates per-language stubs (contrast with the `rmic` tool that only generates stubs for the JVM)

(An aside: they must also agree on *what* the service does, but that needs human intervention!)

Interface definition: OMG IDL

We'll take OMG IDL (used in CORBA) as a typical example

```
//POS Object IDL example
module POS {
  typedef string Barcode;

  interface InputMedia {
    typedef string OperatorCmd;
    void barcode_input(in Barcode item);
    void keypad_input(in OperatorCmd cmd);
  };
};
```

- A `module` defines a namespace within which a group of related type definitions and interface definitions occur

- Interfaces can be derived using multiple inheritance

- Built-in types include basic integers (e.g. `long` holding $-2^{31} \ldots 2^{31} - 1$ and `unsigned long` holding $0 \ldots 2^{32} - 1$), floating point types, 8-bit characters, `boolean`s and `octet`s

- Parameter modifiers `in`, `out` and `inout` define the direction in which parameters are copied

Interface definition: OMG IDL (2)

Type constructors allow structures, discriminated unions, enumerations and sequences to be defined:

```
struct Person {
  string name;
  short age;
};

union Result switch(long) {
  case 1 : ResultDataType r;
  default : ErrorDataType e;
};

enum Color { red, green, blue };

typedef sequence<Person> People;
```

Interfaces can define *attributes* (unlike Java interfaces), but these are just shorthand for pairs of method definitions:

```
attribute long value;

                 →

                   long _get_value();
                   void _set_value(in long v);
```

Interface definition: OMG IDL (3)

IDL construct	*Java construct*
`module`	`package`
`interface`	`interface` + classes
`constant`	`public static final`
`boolean`	`boolean`
`char`, `wchar`	`char`
`octet`	`byte`
`string`, `wstring`	`java.lang.String`
`short`	`short`
`unsigned short`	`short`
`long`	`long`
`unsigned long`	`long`
`float`	`float`
`double`	`double`
`enum`, `struct`, `union`	`class`
`sequence`, `array`	`array`
`exception`	`class`
`readonly attribute`	Read-accessor method
`attribute`	Read,write-accessor methods
`operation`	Method

- ‘Holder classes’ are used for `out` and `inout` parameters – these contain a field appropriate to the type of the parameter

Interface defintion: .NET

Instead of defining a separate IDL and per-language bindings, the Microsoft .NET platform defines a **common language subset** and programming conventions for making definitions that conform to it

Many familiar features: static typing, objects (classes, fields, methods, properties), overloading, single inheritance of implementations, multiple implementation of interfaces, . . .

Metadata describing these definitions is available at run-time, e.g. to control marshalling

➤ Interfaces can be defined in an ordinary programming language and do not need an explicit IDL compiler

➤ Languages vary according to whether they can be used to write clients or servers in this system – e.g. JScript and COBOL vs VB, C#, SML

Naming

How should processes identify which resources they wish to access?

Within a single address space in a Java program we could use object references to identify shared data structures and either

- pass them as parameters to a thread's constructor
- access them from static fields

When communicating between address spaces we need other mechanisms to establish

- unambiguously which item is going to be accessed
- where that item is located and how communication with it can be achieved

Late binding of names (e.g. `elite.cl.cam.ac.uk`) to addresses (`128.232.8.50`) is considered good practice – i.e. using a **name service** at run-time to resolve names, rather than embedding addresses directly in a program

Names

Names are used to identify things and so they should be *unique* within the context that they are used. (A **directory service** may be used to select an appropriate name to look up – e.g. “find the nearest system providing service xyz”)

In simple cases unique IDs (UIDs) may be used – e.g. process IDs in UNIX

- ➤ UIDs are simply numbers in the range $0 \ldots 2^N - 1$ for an N-bit namespace. (Beware: UID $\neq$ user ID in this context!)
- ✔ Allocation is easy if N is large – just allocate successive integers
- ✘ Allocation is centralized (designs for allocating process IDs on highly parallel UNIX systems are still the subject of research)
- ✘ What can be done if N is small? When can/should UIDs be re-used?

Names (2)

More usually a **hierarchical** namespace is formed – e.g. filenames or DNS names

- ✔ The hierarchy allows **local allocation** by separate allocators if they agree to use non-overlapping prefixes
- ✔ The hierarchy can often follow administrative delegation of control
- ✔ Locality of access within the structure may help implementation efficiency (if I lookup one name in `/home/tlh20/` then perhaps I'm likely to lookup other names in that same directory)
- ✘ Lookups may be more complex. Can names be arbitrarily long?

Names (3)

We can also distinguish between **pure** and **impure** names

A pure name yields no information about the identified object – where it may be located or where its details may be held in a distributed name service

– e.g. a UNIX process ID on a multi-processor system does not say on which CPU the process should run, or which user created it

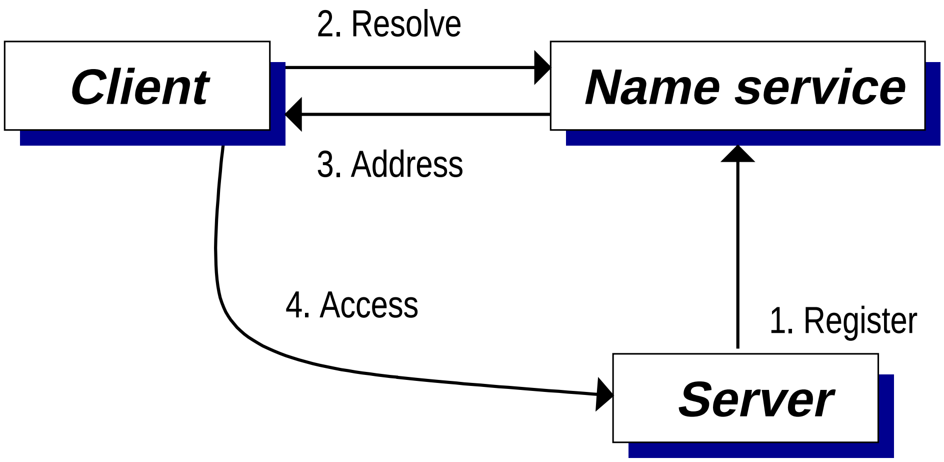
An impure name contains information about the object – e.g. e-mail to `tlh20@cam.ac.uk` will always be sent to a mail server in the University

- ➤ Are DNS names, e.g. `elite.cl.cam.ac.uk` pure or impure?
- ➤ Are IPv4 addresses, e.g. `128.232.8.50` pure or impure?

Names may have structure while still being pure – e.g. Ethernet MAC addresses are structured 48-bit UIDs and include manufacturer codes, and broadcast/multicast flags. This structure avoids centralized allocation

In other schemes, pure names may contain location *hints*. Crucially, impure names prevent the identified object from changing in some way (usually moving) without renaming

Name services



- A **namespace** is a collection of names recognised by a name service – e.g. process IDs on one UNIX system, the filenames that are valid on a particular system or the Internet DNS names that are defined

- A **naming domain** is a section of a namespace operated under a single administrative authority – e.g. management of the `cl.cam.ac.uk` portion of the DNS namespace is delegated to the Computer Lab

- **Binding** or **name resolution** is the process of making a lookup on the name service

How does the client know how to contact the name service?

Name services (2)

Although we've shown the name service here as a single entity, in reality it may

- be *replicated* for availability (lookups can be made if any of the replicas are accessible) and read performance (lookups can be made to the nearest replica)
- be *distributed*, e.g. separate systems may manage different naming domains within the same namespace (updates to different naming domains require less co-ordination)
- allow *caching* of addresses by clients, or caching of partially resolved names in a hierarchical namespace

Security

In a distributed system, access control is needed to:

- control communication to/from the various components involved,
  - e.g. consider an industrial system with a component on one computer recording the temperature and responding to queries from another computer that controls settings on a machine its attached to
  - how does the controller know that the temperature readings come from the intended probe?
  - how does the probe know that it's being queried by the intended controller?
- control operations that one component does on behalf of users,
  - e.g. a file server may run as the privileged `root` on a UNIX machine
  - when accessing a file on behalf of a remote client it needs to know who that client is and either cause the OS to check access would be OK, or to do those checks itself
- Again, covered more fully in the security and distributed systems courses

Security (2)

We'll look at basic sensible things to do when writing distributed systems in Java

- use a **security manager** class to limit what the JVM is able to do

  — e.g. limiting the IP addresses to which it can connect or whether it is permitted to write to your files

- if using network sockets directly then make the program robust to unexpected input

  — less of a concern in Java than in C...

A security manager provides a mechanism for enforcing simple controls

- A security manager is implemented by `java.lang.SecurityManager` (or a sub-class)

- An instance of this is installed using `System.setSecurityManager(...)` (itself an operation under the control of the current security manager)

Security (3)

- Most checks are made by delegating to a `checkPermission` method, e.g. for dynamically loading a native library

  ```
  checkPermission(
     new RuntimePermission(
        "loadLibrary."+lib));
  ```

- Decisions made by `checkPermission` are relative to a particular **security context**. The current context can be obtained by invoking `getSecurityContext` and checks then made on behalf of another context

- Permissions can be granted in a policy definition file, passed to the JVM on the command line with `-Djava.security.policy=`*filename*

  ```
  grant {
    permission java.net.SocketPermission
            "*:1024-65535", "connect,accept";
  };
  ```

`http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html`

Exercises

15-1 If you have access both to a big-endian (e.g. SPARC) and a little-endian machine (e.g. Intel) then test whether an object serialized to disk on one is able to be recreated successfully on the other. Examine what happens if the object refers to facilities intrinsic to the originating machine – e.g. if it contains an open `FileOutputStream` or a reference to `System.out`.

15-2 Suppose that two people are communicating by sending and receiving mobile-phone text messages. Messages are delayed by varying amounts. Some messages are lost entirely. Design a way to get reliable communication (so far as is possible). You may need to add information to each message sent, and possibly create further messages in addition to those sent ordinarily.

15-3 Convert the `POS` module definition from OMG IDL into a Java interface that provides similar RMI functionality.

15-4* Suppose that frequent updates are made to part of a hierarchical namespace, while other parts are rarely updated. Lookups are made across the entire namespace. Discuss the use of *replication*, *distribution*, *caching* or other techniques as ways of providing an effective name service.

Lecture 16: Network sockets (TCP & UDP)

Previous lecture

- Distributed systems
- Interface definitions
- Naming

Overview of this lecture

- Communication using network sockets
- UDP
- TCP

Low-level communication

Two basic network protocols are available in Java: datagram-based **UDP** and stream-based **TCP**

UDP sockets provide *unreliable datagram-based* communication that is subject to:

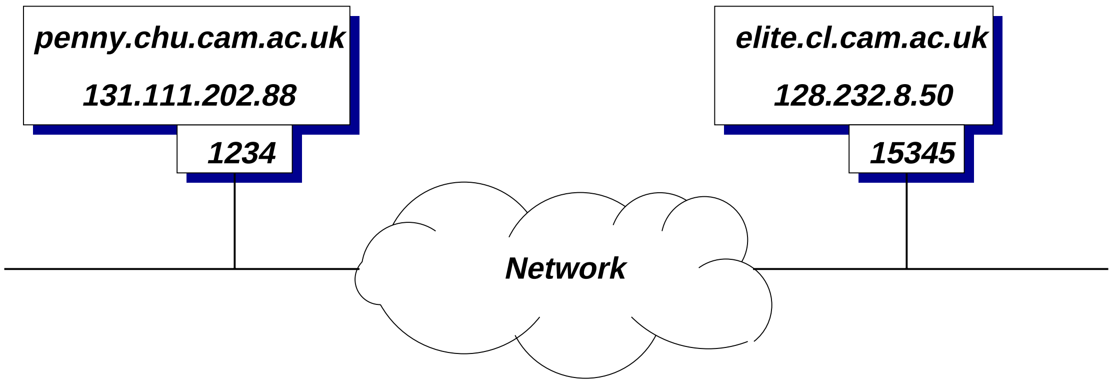
- **Loss**: datagrams that are sent may never be received,
- **Duplication**: the same datagram is received several times,
- **re-ordering**: datagrams are forwarded separately within the network and may arrive out of order

What is provided:

- A checksum is used to guard against corruption (corrupt data is discarded by the protocol implementation and the application perceives it as loss)
- The **framing** within datagrams is preserved – a UDP datagram may be **fragmented** into separate packets within the network, but these are reassembled by the receiver

Low-level communication (2)

Communication occurs between UDP **sockets** which are addressed by giving an appropriate IP address and a UDP port number (0..65535, although 0 not accessible through common APIs, 1..1023 reserved for privileged use)



Naming is handled by

- Using the DNS to map textual names into IP addresses, `InetAddress.getByName("elite.cl.cam.ac.uk")`
- Using 'well-known' port numbers for particular UDP services which wish to be accessible to clients (See the `/etc/services` file on a UNIX system)

As far as we're concerned here, the network acts as a 'magic cloud' that conveys datagrams – see Digital Communication I for *layering* in general and examples of how UDP is implemented over IP and IP over (e.g.) ethernet

UDP in Java

- UDP sockets are represented by instances of `java.net.DatagramSocket`. The 0-argument constructor creates a new socket that is bound to an available port on the local host machine. This identifies the *local* endpoint for the communication

- Datagrams are represented in Java as instances of `java.net.DatagramPacket`. The most elaborate constructor:

  ```
  DatagramPacket(byte buf[], int length,
    InetAddress address, int port)
  ```

  specifies the data to send (`length` bytes from within `buf`) and the destination `address` and `port`

- `MulticastSocket` defines a UDP socket capable of receiving multicast packets. The constructor specifies the port number and then methods

  ```
  joinGroup (InetAddress g);
  leaveGroup (InetAddress g);
  ```

  join and leave a specified group operating on that port

- Multicast group addresses are a designated subset of the IPv4 address space. Allocation policies are still in flux ⇒ check the local policy before using

UDP example

```
import java.net.*;

public class Send {
  public static void main (String args[]) {
    try {
      DatagramSocket s = new DatagramSocket ();
      byte[]         b = new byte[1024];
      int            i;

      for (i = 0; i < args.length - 2; i ++)
        b[i] = Byte.parseByte (args[2 + i]);

      DatagramPacket p = new DatagramPacket (
            b, i,
            InetAddress.getByName (args[0]),
            Integer.parseInt (args[1]));

      s.send(p);

    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

UDP example (2)

```
import java.net.*;

public class Recv {
  public static void main (String args[]) {
    try {
      DatagramSocket s = new DatagramSocket ();
      byte[]         b = new byte[1024];
      DatagramPacket p =
                  new DatagramPacket (b, 1024);

      System.out.println("Port: " +
                  s.getLocalPort());

      s.receive(p);

      for (int i = 0; i < p.getLength (); i ++)
        System.out.print ("" + b[i] + " ");

      System.out.println ("\nFrom: " +
        p.getAddress () + ":" + p.getPort ());
    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Problems using UDP

Many facilities must be implemented manually by the application programmer:

- ✗ Detection and recovery from loss
- ✗ Flow control (preventing the receiver from being swamped with too much data)
- ✗ Congestion control (preventing the network from being overwhelmed)
- ✗ Conversion between application data structures and arrays of bytes (*marshalling*)

Of course, there are situations where UDP is directly useful

- ✓ Communication with existing UDP services (e.g. some DNS name servers)
- ✓ Broadcast and multicast are possible (e.g. address 255.255.255.255 ⇒ all machines on the local network – but note problems of port assignment and more generally of multicast group naming)

TCP sockets

The second basic form of inter-process communication is provided by TCP sockets

- Naming is again handled using the DNS and well-known port numbers as before. There is no relationship between UDP and TCP ports having the same number
- TCP provides a reliable bi-directional connection-based byte-stream with flow control and congestion control

What doesn't it do?

- Unlike UDP the interface exposed to the programmer is not datagram based: framing must be provided explicitly
- Marshalling must still be done explicitly – but serialization may help here
- Communication is always one-to-one

In practice TCP forms the basis for many internet protocols – e.g. FTP and HTTP are both currently deployed over it

TCP sockets (2)

Two principal classes are involved in exposing TCP sockets in Java:

- `java.net.Socket` represents a connection over which data can be sent and received. Instantiating it directly initiates a connection from the current process to a specified address and port. The constructor blocks until the connection is established (or fails with an exception)

- `java.net.ServerSocket` represents a socket awaiting incoming connections. Instantiating it starts the local machine listening for connections on a particular port. `ServerSocket` provides an `accept` operation that blocks the caller until an incoming connection is received. It then returns an instance of `Socket` representing that connection

The system will usually buffer only a small (5) number of incoming connections if `accept` is not called

Typically programs that expect multiple clients will have one thread making calls to `accept` and starting further threads for each connection

TCP example

```
import java.net.*;
import java.io.*;

public class TCPSend {
  public static void main (String args[]) {
    try {
      Socket s = new Socket (
          InetAddress.getByName (args[0]),
          Integer.parseInt (args[1]));

      OutputStream os = s.getOutputStream ();

      while (true) {
        int i = System.in.read();
        os.write(i);
      }

    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

TCP example (2)

```
import java.net.*;
import java.io.*;

public class TCPRecv {
  public static void main (String args[]) {
    try {
      ServerSocket serv = new ServerSocket (0);
      System.out.println ("Port: " +
              serv.getLocalPort ());
      Socket          s = serv.accept ();
      System.out.println ("Remote addr: " +
              s.getInetAddress());
      System.out.println ("Remote port: " +
              s.getPort());
      InputStream    is = s.getInputStream ();
      while (true) {
        int i = is.read ();
        if (i == -1) break;
        System.out.write (i);
      }
    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Server design

The examples have only illustrated the basic use of the operations on `DatagramSocket`, `ServerSocket` and `Socket`:

- ➤ Typically a server would be expected to manage multiple clients

Doing so efficiently can be a problem if there are lots of clients:

- ➤ Could have one thread per client:
  - ✔ Can exploit multi-processor hardware
  - ✘ Many active clients ⇒ frequent context switches
  - ✘ The JVM (+ usually the OS) must maintain state for all clients, whether active or not
- ➤ Could have a single thread which services each client in turn:
  - ✔ Simple, avoids context switching
  - ✘ No 'wait for any input stream' operation in java (cf `select` in UNIX): must poll each client whether needed or not
- ➤ The `javo.nio` package now supports asynchronous I/O

Exercises

16-1 Write a class `UDPSender` which sends a series of UDP packets to a specified address and port at regular 15 second intervals. Write a corresponding `UDPReceiver` which receives such packets and records the inter-arrival time. How does the performance differ if (*i*) both programs run on the same computer, (*ii*) both run on computers on the University network or (*iii*) one runs on the University network and another on a dial-up internet connection. Do you see packets that are lost, duplicated or re-ordered? Do the packets arrive regularly spaced?

16-2 Write similar classes `TCPSender` and `TCPReceiver` which establish a TCP connection over which single bytes are sent at 15 second intervals. How does the performance compare with the UDP implementation. Is it necessary to call `flush` on the `OutputStream` after sending each byte?

16-3 Consider a server for a noughts-and-crosses game. The two players communicate with it over UDP. Describe a possible structure for the server – in terms of the major data structures, the threads used, the format of the datagrams sent and the concurrency-control techniques.

Lecture 17: RPC & RMI

Previous lecture

- UDP: connectionless, unreliable
- TCP: connection-oriented, reliable

Overview of this lecture

- Java RMI
- Implementing RPC

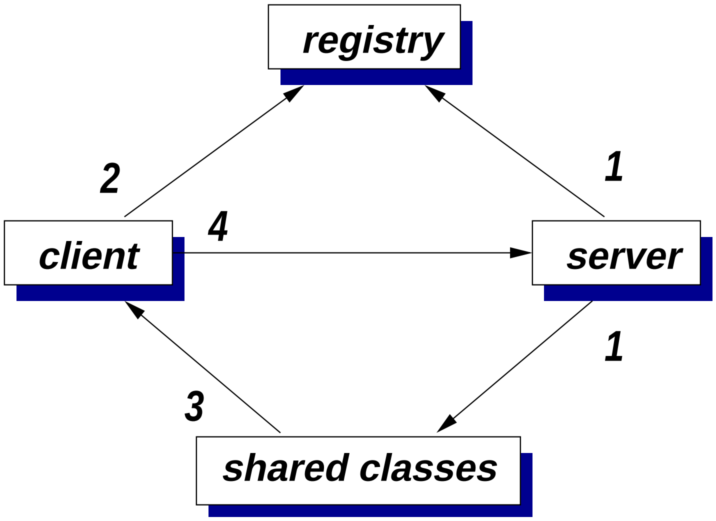
Remote method invocation

Using UDP or TCP it was necessary to

- Decide how to represent data being sent over the network – either packing it into arrays of bytes (in a `DatagramPacket`) or writing it into an OutputStream (using a `Socket`)
- Use a rather inflexible naming system to identify servers – updates to the DNS may be difficult, access to a specific port number may not always be possible
- Distribute the code to all of the systems involved and ensure that it remains consistent
- Deal with failures (e.g. the remote machine crashing – something a 'reliable' protocol like TCP cannot mask)

Java RMI presents a higher level interface that addresses some of these concerns. Although it is **remote method invocation**, the principles are the same as for **remote procedure call** (RPC) systems

Remote method invocation (2)



1. A server registers a reference to a remote object with the **registry** (a basic name service) and deposits associated `.class` files in a shared location, the **RMI codebase**

2. A client queries the registry to obtain a reference to a remote object

3. If they are not directly available, the client obtains the `.class` files needed to access the remote object

4. The client makes an RMI call to the remote object

The registries act as a name service, with names of the form `rmi://linux2.pwf.cl.cam.ac.uk/tlh20/example-1.2`

Remote method invocation (3)

Parameters and results are generally passed by making **deep copies** when passed or returned over RMI

- i.e. copying proceeds recursively on the object passed, objects reachable from that etc (⇒ take care to reduce parameter sizes)
- The structure of object graphs is preserved – e.g. data structures may be cyclic
- Remote objects are passed *by reference* and so both caller and callee will interact with *the same* remote object if a reference to it is passed or returned

Note that Java only supports remote *method* invocation – changes to fields must be made using `get`/`set` methods

Other RPC systems make different choices:

- Perform a shallow copy and treat other objects reachable from that as remote data (as above, would be hard to implement in Java) or copy them incrementally
- Emulate 'pass by reference' by passing back any changes with the method results (what about concurrent updates?)

RMI - Interfaces

Suppose that we wish to define a simple remote object on which a single method `getOpponent` is defined:

```
1  package remifc;
2
3  import java.rmi.*;
4
5  public interface PlayerFinder extends Remote
6  {
7    public static final String NAME
8      = "rmi://elite.cl.cam.ac.uk/tlh20/playerfinder";
9
10   public PlayerLocation
11       getOpponent(PlayerLocation me)
12       throws RemoteException;
13 }
```

- All RMI invocations are made across **remote interfaces** extending `java.rmi.Remote`

- The field NAME in Lines 7–8 says which RMI registry will be used (the one on `elite.cl.cam.ac.uk`) and the name to registry the service as (`tlh20/playerfinder`)

- All remote methods must throw `RemoteException`

RMI - Client

- In class `client.SlimeV5`:

```java
// Bind to RMI service
System.out.println ("Looking for " +
          PlayerFinder.NAME);
PlayerFinder pf = (PlayerFinder)
          Naming.lookup(PlayerFinder.NAME);

// Start listening for connections from opponents
ServerSocket ss =
          new ServerSocket (0, 1,
                   InetAddress.getLocalHost ());

// Determine our network address and port number
PlayerLocation my_location =
          new PlayerLocation(ss.getInetAddress(),
                             ss.getLocalPort());

System.out.println ("My location is " +
          my_location);

// Wait to be paired up with an opponent
PlayerLocation op_location =
          pf.getOpponent (my_location);

System.out.println ("Opponent location is " +
          op_location);
```

RMI - Client (2)

Note how few differences there are in the client compared with local invocations on an instance of a class implementing `PlayerFinder`:

- Lines 2–5 obtain an instance of a **stub class** implementing the `PlayerFinder` interface. Invocations on this instance will be forwarded to a remote object registered under the name `PlayerFinder.NAME`
- Lines 7–15 are particular to the slime volley ball example: as in the basic networked game, they set up a server socket to await connections from opponents
- Lines 21–21 perform the actual invocation
- The opponents receive one another's network addresses: the one with the lower address will connect to the one with the higher address
- The code will require an exception handler to deal with:
  - `NotBoundException` – no remote object has been associated with the name `PlayerFinder.NAME`
  - `RemoteException` – if the RMI registry could not be contacted (4–5) or if there was a problem with the call (21–22)
  - `AccessException` – if the operation has not been permitted by the installed security manager

RMI - Server

- In class `server.PlayerFinderImpl`:

```
public static void main (String args[])
{
  try
    {
      // Instantiate server
      PlayerFinderImpl s =
            new PlayerFinderImpl ();

      // Allow connections to the registry
      System.setSecurityManager (
            new RMISecurityManager ());

      // Bind name in the registry
      Naming.rebind (PlayerFinder.NAME, s);

      System.out.println (NAME +
            " server running");
    }
  catch (Exception e)
    {
      System.out.println ("Failed: " + e);
      e.printStackTrace ();
    }
}
```

Putting it all together

- Select which machine to run the registry on and update `remifc.PlayerFinder.NAME` (usually use the same machine for the registry & the server)

- Compile the interface, client and server:

  ```
  $ javac remifc/*.java server/*.java client/*.java
  ```

- Generate stub classes from the server:

  ```
  $ rmic -v1.2 server.PlayerFinderImpl
  ```

  creating `server.PlayerFinderImpl_Stub.class`

- Generate a security policy file for the server, e.g. `security.policy`:

  ```
  grant {
    permission java.net.SocketPermission
        "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission
        "*:80", "connect";
    permission java.util.PropertyPermission
        "java.rmi.server.codebase", "read";
    permission java.util.PropertyPermission
        "user.name", "read,write";
  };
  ```

Putting it all together (2)

- Make sure that the RMI registry is running. If not then:

```
$ rmiregistry
```

- Select an RMI codebase for the server – this should be available to the client and the registry (e.g. a directory on a shared file system, or on a web site)

```
$ CODEBASE=file:/homes/tlh20/19.RMISVB/
```

Do not forget the trailing `/`

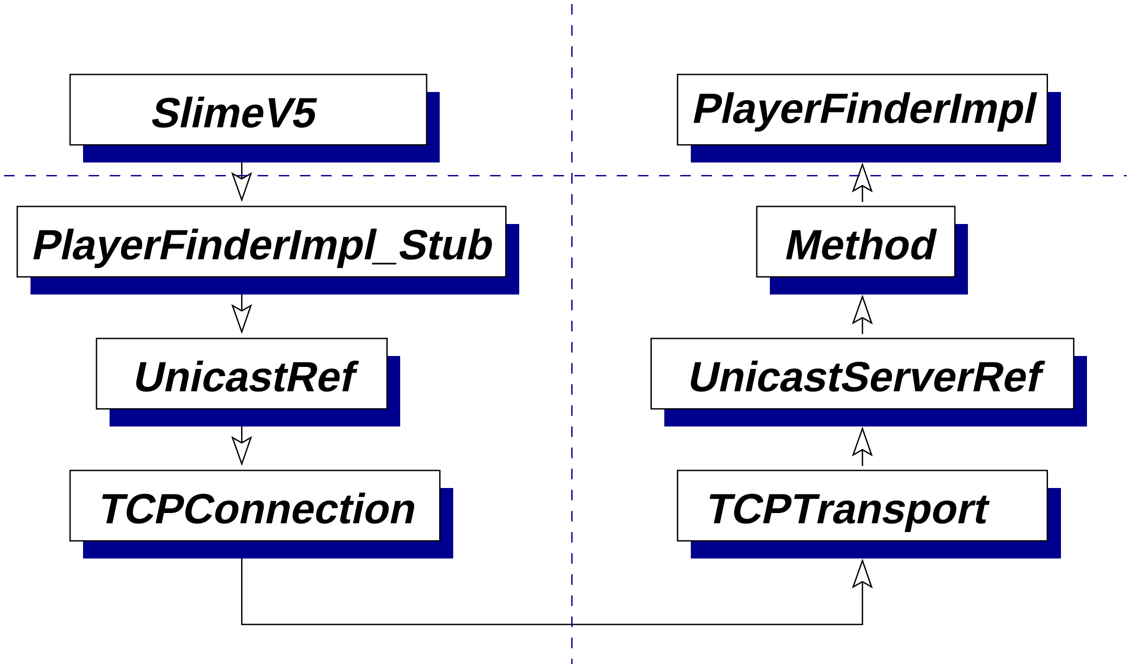
- Start the server running:

```
$ java -Djava.rmi.server.codebase=$CODEBASE \
       -Djava.security.policy=security.policy \
       server.PlayerFinderImpl
rmi://elite.cl.cam.ac.uk/tlh20/playerfinder server run
```

- Start clients...

```
$ java client.SlimeV5
Looking for rmi://elite.cl.cam.ac.uk/tlh20/playerfinde
My location is (elite.cl.cam.ac.uk/128.232.8.50:56283)
Opponent location is (elite.cl.cam.ac.uk/128.232.8.50:
Waiting for connections on elite.cl.cam.ac.uk/128.232.
Connection from /128.232.8.50:56283
```

RMI implementation



- The _Stub class is the one created by the rmic tool – it transforms invocations on the PlayerFinder interface into generic invocations of an invoke method on UnicastRef

- UnicastRef is responsible for selecting a suitable network transport for accessing the remote object – in this case TCP

- UnicastServerRef uses the ordinary reflection interface to dispatch calls to remote objects

RMI implementation (2)

With the TCP transport RMI creates a new thread on the server for each incoming connection that is received

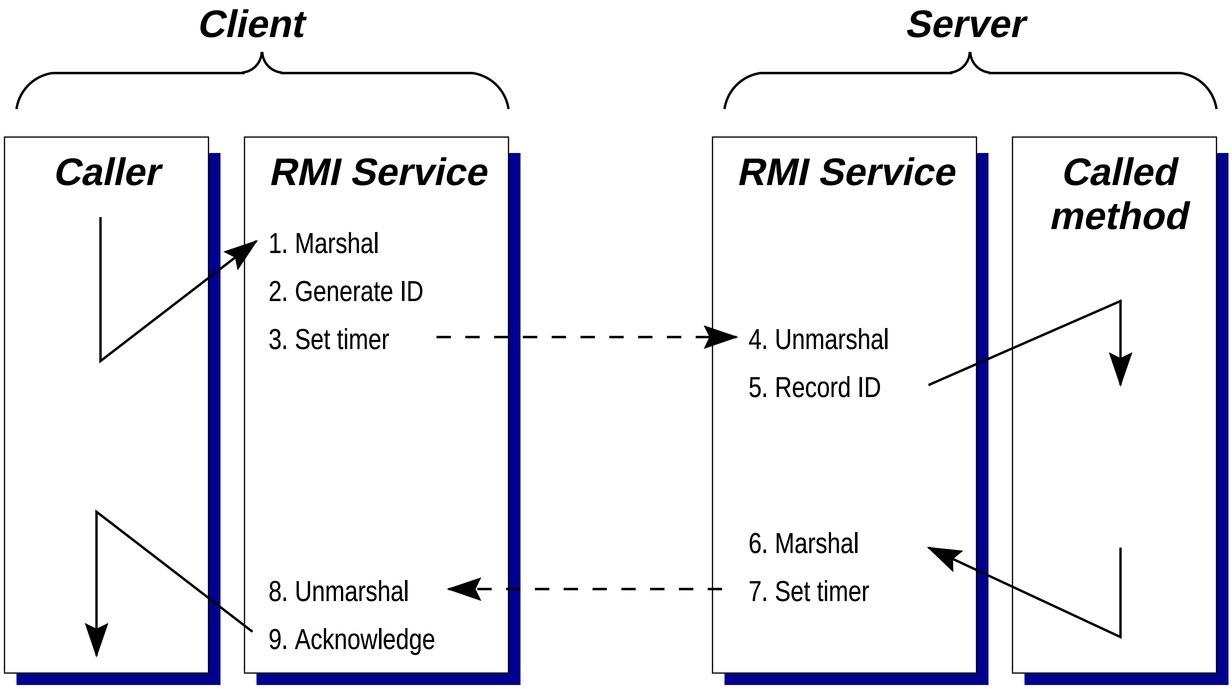
➤ A remote object should be prepared to accept concurrent invocations of its methods

✔ This concurrency avoids deadlock if remote method `A.m1` invokes an operation on remote method `B.m2` which in turn invokes an operation `A.m3`

✘ The application programmer must be aware of how many threads might be created and the impact that they may have on the system

➤ Remember: the `synchronized` modifier applies to a method's *implementation*. It must be applied to the definition in the server class, not the interface

RMI implementation (3)



What could be done without TCP?

We need to manually implement:

- Reliable delivery of messages subject to loss in the network
- Association between invocations and responses – shown here using a per-call RPC identifier with which all messages are tagged

RMI implementation (4)

Even this simple protocol requires multiple threads: e.g. to re-send lost acknowledgements after the client-side RMI service has returned to the caller

What happens if a timeout occurs at 3? Either the message sent to the server was lost, or the server failed before replying

- **At-most-once** semantics ⇒ return failure indication to the application
- **'Exactly'-once** semantics ⇒ retry a few times with the same RPC id (so server can detect retries)

What happens if a timeout occurs at 7? Either the message sent to the client was lost, or the client failed

No matter what is done, the client cannot distinguish, on the basis of these messages, server failures before / after making some change to persistent storage

Exercises

17-1 Compile and execute the RMI example yourself. Use it with the stub class held on a web server as well as with the stub class available directly through the file system.

17-2 Modify the `UDPSender` and `UDPReceiver` example so the sender initiates an RMI call to the receiver at regular 15 second intervals. How does the performance compare now to the UDP and TCP examples?

17-3 To what extent can the fact that a method invocation is remote be made transparent to the programmer? In what ways is complete transparency not possible?

17-4 A client and a server are in frequent communication using the RPC protocol described in the slides and implemented over UDP. Design and outline an alternative protocol that sends fewer datagrams when loss is rare.

17-5* All remote method invocations in Java may throw `RemoteException` because of the failure modes introduced by distribution. Do you agree that `RemoteException` should be a checked exception rather than an unchecked exception (such as `NullPointerException`) which is usually fatal?

Lecture 18: Transactions

Previous section

- Communication using UDP or TCP
- Remote method invocation
- ...and before that concurrency control between threads

Overview of this section

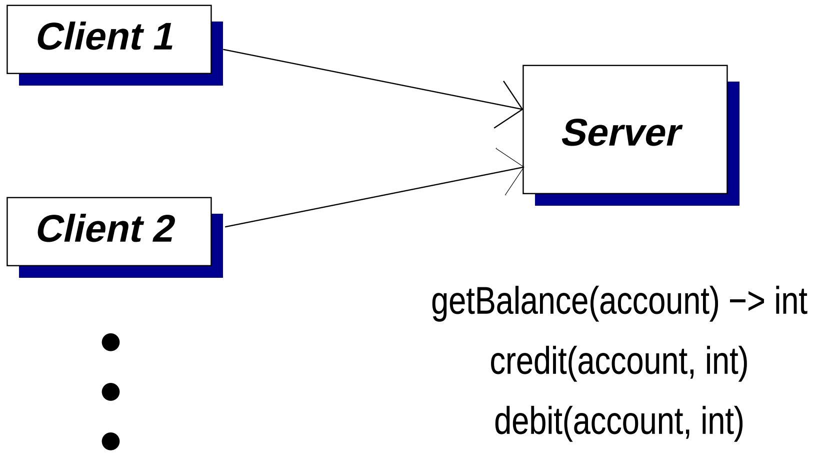
- Compound operations
- Correctness requirements
- Implementation

Compound operations

We’ve now seen mechanisms for

- Controlling concurrent access to objects
- Providing access to remote objects

Using these facilities correctly, and particularly in combination, is extremely difficult.



- Client 1 tries to get the total amount in two accounts
- Client 2 tries to transfer some money between the accounts using `credit` then `debit`
- It can all go horribly wrong, even if `getBalance`, `credit` and `debit` are safe for multi-threaded use

Compound operations (2)

What can go wrong?

- Client 1 may look at the two accounts after one has been credited but before the other is debited
  ⇒ the total will be wrong

- Client 2 may crash after doing its `credit` but before the matching `debit`
  ⇒ the recipient could be lucky...

- The network may fail, even if the clients are well behaved

- The server may crash

What can be done about these problems?

- Have the server provide `lockAccount` and `unlockAccount` operations

- Have the server provide `transfer` as an atomic operation

- Use some kind of ‘downloadable code’ system

Transactions

Transactions provide a more general abstraction

Ideally the programmer may wish to write something like

```
transaction {
  if (server.getBalance(src) >= amount) {
    server.credit (dest, amount);
    server.debit (src, amount);
    return true;
  } else {
    return false;
  }
}
```

The intent is that code within a `transaction` block will execute without interference from other activities, in particular

- other operations on the same objects
- system crashes (within reason...)

We say that a transaction either **commits** atomically (if it completes successfully) or it **aborts** (if it fails for some reason). Aborted transactions leave the state unchanged.

Transactions (2)

In more detail we'd like committed transactions to satisfy four **ACID properties**:

Atomicity – either all or none of the transaction's operations are performed

— programmers do not have to worry about 'cleaning up' after a transaction aborts; the system ensures that it has no visible effects

Consistency – a transaction transforms the system from one consistent state to another

— the programmer must design transactions that preserve desired invariants, e.g. totals across accounts

Isolation – each committed transaction executes isolated from the concurrent effects of others

— e.g. another transaction shouldn't read the `source` and `destination` amounts mid-transfer and then commit

Durability – the effects of committed transactions endure subsequent system failures

— when the system confirms the transaction has committed it must ensure any changes will survive faults

Transactions (3)

These requirements can be grouped into two categories:

- Atomicity and durability refer to the persistence of transactions across system failures.

  We want to ensure that no 'partial' transactions are performed (atomicity) and we want to ensure that system state does not regress by apparently-committed transactions being lost (durability)

- Consistency and isolation concern ensuring correct behaviour in the presence of concurrent transactions

  As we'll see there are trade-offs between the ease of programming within a particular transactional framework, the extent that concurrent execution of transactions is possible and the isolation that is enforced

In some cases – where data is held entirely in main memory – we may just be concerned with controlling concurrency

- Note the distinction with the concurrency control schemes based (e.g.) on programmers using mutexes and condition variables: here the system enforces isolation

Isolation

Recall our original example:

```
transaction {
  if (server.getBalance(src) >= amount) {
    server.credit (dest, amount);
    server.debit (src, amount);
    return true;
  } else {
    return false;
  }
}
```

What can the system do in order to enforce isolation between transactions specified in this manner and initiated concurrently?

A simple approach: have a single lock that's held while executing a transaction, allowing only one to operate at once

✔ Simple, 'clearly correct', independent of the operations performed within the transaction

✘ Does not enable concurrent execution, e.g. two of these operations on separate sets of accounts

✘ What happens if operations can fail?

Isolation – serializability

This idea of executing transactions serially provides a useful correctness criterion for executing transactions in parallel:

- A concurrent execution is **serializable** if there is some serial execution of the same transactions that gives the same result — the programmer *cannot distinguish* between parallel execution and the simple one-at-a-time scheme

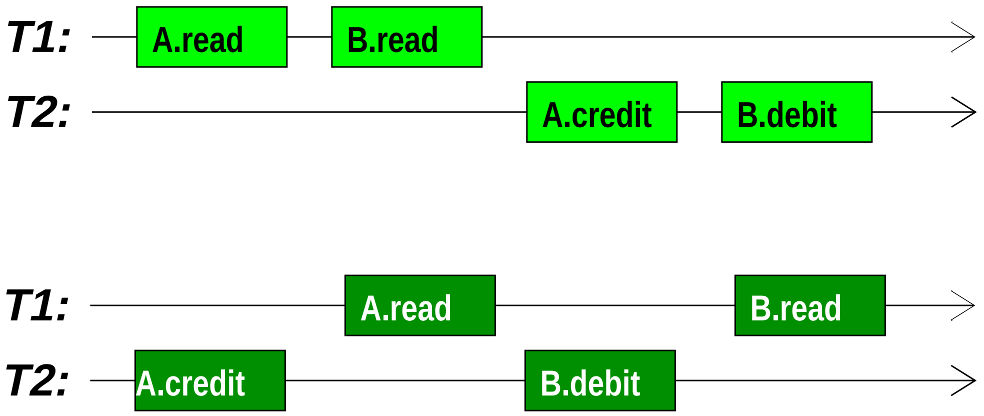
Suppose we have two transactions:

```
T1: transaction {
  int s = server.getBalance (A);
  int t = server.getBalance (B);
  return s + t;
}

T2: transaction {
  server.credit (A, 100);
  server.debit (B, 100);
}
```

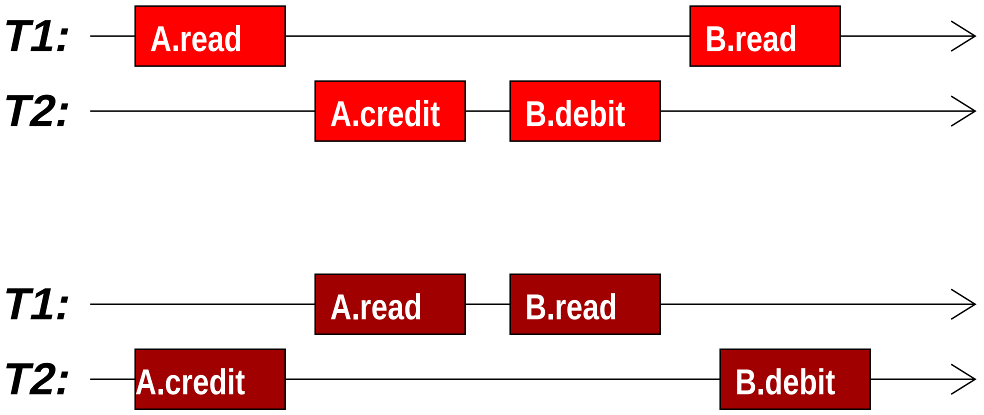
If we assume that the individual `getBalance`, `credit` and `debit` operations are atomic (e.g. `synchronized` methods on the server) then an execution without further concurrency control can proceed in 6 ways

Isolation – serializability (2)

Both of these concurrent executions are OK:



Neither of these concurrent executions is valid:



In each case some – but not all – of the effects of T2 have been seen by T1, meaning that we have not achieved isolation between the transactions

Isolation – serializability (3)

We can depict *a particular execution* of a set of concurrent transactions by a **history graph**

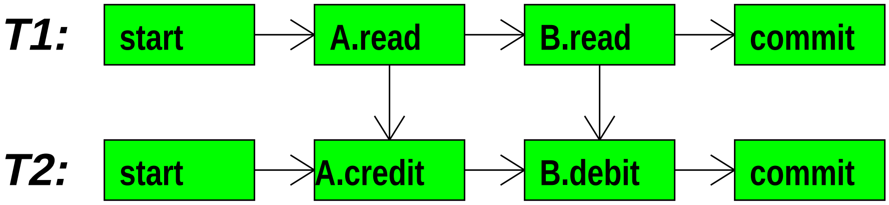
- Nodes in the graph represent the operations comprising each transaction, e.g. `T1: A.read`
- A directed edge from node `a` to node `b` means that `a` **happens before** `b`
  - Operations within a transaction are totally ordered by the **program order** in which they occur
  - **Conflicting** (i.e. non-commutative) operations on the same object are ordered by the object's implementation

For clarity we usually omit edges that can be inferred by the transitivity of *happens before*

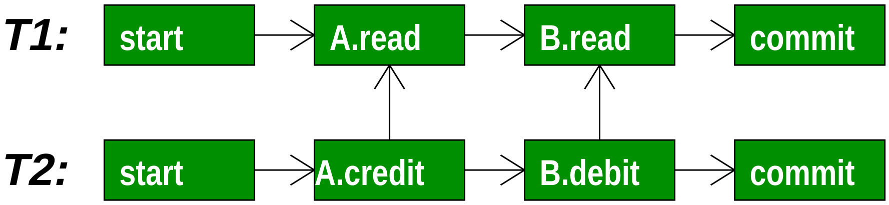
Suppose again that we have two objects `A` and `B` associated with integer values and run transaction `T1` that reads values from both and transaction `T2` that adds to `A` and subtracts from `B`

Isolation – serializability (4)

These histories are OK. Either both the `read` operations see the old values of `A` and `B`:

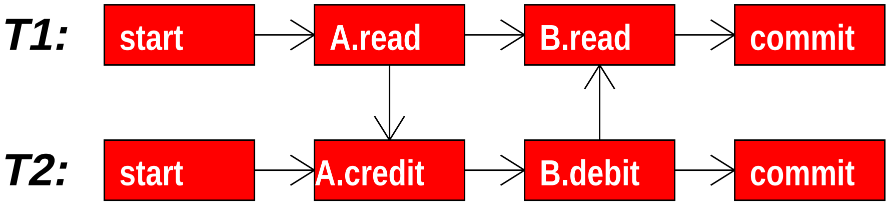


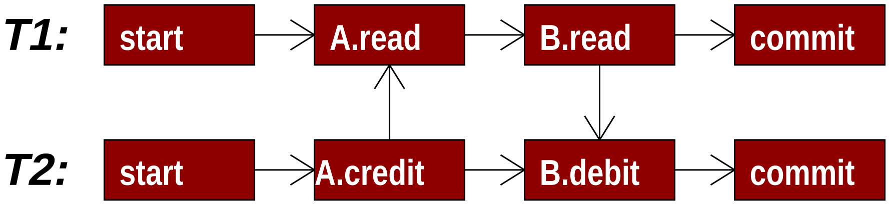
or both `read` operations see the new values:



Isolation – serializability (5)

These histories show non-serializable executions in which one `read` sees an old value and the other sees a new value:





In general, cycles are caused by three kinds of problem:

- **Lost updates** (e.g. by another transaction overwriting them before they are commmitted)
- **Dirty reads** (e.g. of updates before they are committed)
- **Unrepeatable reads** (e.g. before an update by another transaction overwrites it)

Isolation & strict isolation

Here we're interested in avoiding all three kinds of problem so that committed transactions built from simple *read* and *update* operations satisfy serializable execution

We can distinguish between enforcing:

- **Strict isolation:** actually ensure that transactions are isolated during their execution – prohibit all three problems

- **Non-strict isolation:** ensure that a transaction was isolated before it's allowed to commit

Non-strict isolation may permit more concurrency but can lead to **delays on commit** (e.g. a transaction that performed a dirty read cannot commit until the writer has) and **cascading aborts** (if the writer actually aborts)

- NB: in some situations weaker guarantees are accepted for higher concurrency

  — In systems using locks to enforce isolation: so long as all transactions avoid lost updates, the decision to avoid dirty & unrepeatable reads can be made on a per-transaction bases

Exercises

18-1 Define the ACID properties for transactions using a simple example (such as transfers between a number of bank accounts) as illustration. For each property, give a possible (incorrect) execution which violates it.

18-2 Slide 18-8 says that there are 6 ways in which execution can proceed, but Slide 18-9 depicts only 4. Illustrate the remaining 2 possible executions and construct history graphs for them.

18-3* If Java were to support a `transaction` keyword then its semantics would need to be defined carefully. Describe how it could behave when the transactional code:

(*i*) accesses local variables

(*ii*) accesses fields

(*iii*) throws exceptions

(*iv*) makes method calls

(*v*) uses mutexes and condition variables

(*vi*) creates threads

You should assume that it is for multi-threaded use on a single computer, rather than needing to support RMI or other kinds of external communication.

Lecture 19: Enforcing isolation

Previous lecture

- Problems of compound operations
- ACID properties for transactions
- Serializability

Overview of this lecture

- Implementing isolation
- Two-phase locking
- Timestamp ordering
- Optimistic concurrency control

Isolation – two-phase locking

We'll now look at some mechanisms for ensuring that transactions are executed in a serializable manner while allowing more concurrency than an actual serial execution would achieve

In **two-phase locking** (2PL) each transaction is divided into

- a phase of acquiring locks
- a phase of releasing locks

Locks must exclude other operations that may conflict with those to be performed by the lock holder.

Operations can be performed *during both phases* so long as the appropriate locks are held.

Simple mutual exclusion locks may suffice, but could limit concurrency. In the example we could use a MRSW lock, held in read mode for `getBalance` and write mode for `credit` and `debit`

Isolation – two-phase locking (2)

How does the system know when (and how) to acquire and release locks if transactions are defined in the form:

```
1 transaction {
2   if (server.getBalance(src) >= amount) {
3     server.credit (dest, amount);
4     server.debit (src, amount);
5     return true;
6   } else {
7     return false;
8   }
9 }
```

- Could require explicit invocations by the programmer, e.g. expose `lock` and `unlock` operations on the server
  - acquire a read lock on `src` before 2, release if the `else` clause is taken,
  - upgrade to a write lock on `src` before 3,
  - acquire a write lock on `dest` before 4,
  - release the lock on `src` any time after acquiring both locks,
  - release the lock on `dest` after 4

Isolation – two-phase locking (3)

How well would this form of two-phase locking work?

- ✔ Ensures serializable execution if implemented correctly
- ✔ Allows arbitrary application-specific knowledge to be exploited, e.g. using MRSW for increased concurrency over mutual exclusion locks
- ✔ Allowing other transactions to access objects as soon as they have been unlocked increases concurrency
- ✘ Complexity of programming (e.g. 2PL ⇒ MRSW needs an *upgrade* operation here)
- ✘ Would be nice to provide `startTransaction` & `endTransaction` rather than individual lock operations
- ✘ Risk of deadlock
- ✘ If T_b locks an object just released by T_a then isolation requires that
  - T_b cannot commit until T_a has
  - T_b must abort if T_a does ('cascading aborts')

Some of these problems can be addressed by **Strict 2PL** in which all locks are held until commit/abort: transactions *never* see partial updates made by others

Isolation – timestamp ordering

Timestamp ordering (TSO) is another mechanism to enforce isolation:

- Each transaction has a timestamp – e.g. of its start time. These must be totally ordered
- The ordering between these timestamps will give a serializable order for the transactions
- If T_a and T_b both access some object then they must do so according to the ordering of their timestamps

Basic implementation:

- Augment each object with a field holding the timestamp of the transaction that most recently invoked an operation on it
- Check the object's timestamp against the transaction's each time an operation is invoked:
  - ✓ The operation is allowed if the transaction's timestamp is latest
  - ✗ The operation is rejected as *too late* if the transaction's timestamp is earlier

Isolation – timestamp ordering (2)

One serializable order is achieved: that of the timestamps of the transactions, e.g.

```
T1,1:startTransaction     T2,1:startTransaction
T1,2:server.getBalance(A) T2,2:server.credit(A, 100)
T1,3:server.getBalance(B) T2,3:server.debit(B, 100)
```

- ✔ `T1,1` executes, → timestamp 17
- ✔ `T1,2` executes, A: 17,`read`
- ✔ `T2,1` executes, → timestamp 42
- ✔ `T2,2` executes, OK (later) A: 42,`credit`
- ✔ `T2,3` executes, B: 42,`debit`
- ✘ `T1,3` attempted: too late 17 earlier than 42 and `read` conflicts with `credit`

In this case both transactions could have committed if `T1,3` had been executed before `T2,3`

Isolation – timestamp ordering (3)

- ✔ The decision of whether to admit a particular operation is based on information local to the object

- ✔ Simple to implement – e.g. by interposing the checks on each invocation at the server (contrast with non-strict 2PL)

- ✔ Avoiding locking may increase concurrency

- ✔ Deadlock is not possible

- ✘ Needs a roll-back mechanism

- ✘ Cascading aborts are possible – e.g. if `T1,2` had updated `A` then it would need to be undone and `T2` would have to abort because it may have been influenced by `T1`

  — could delay `T2,2` until `T1` either commits or aborts (still avoiding deadlock)

- ✘ Serializable executions can be rejected if they do not agree with the transactions' timestamps (e.g. executing `T2` in its entirety, then `T1`)

Generally: the low overheads and simplicity make TSO good when conflicts are rare

Isolation – OCC

Optimistic Concurrency Control (OCC) is the third kind of mechanism we will look at for enforcing isolation

- Optimistic schemes assume that concurrent transactions rarely conflict
- Rather than ensuring isolation during execution a transaction proceeds directly and serializability is checked at commit time
- Assuming this check usually succeeds (and is itself fast) then OCC will perform well
- ...if the check often fails then performance may be poor because the work done executing the transaction is wasted

For instance consider implementing a shared counter using atomic compare and swap:

```
do {
  old_val = counter;
  new_val = old_val + 1;
} while (CAS (&counter, old_val -> new_val));
```

Isolation – OCC (2)

More generally, a transaction proceeds by taking **shadow copies** of each object it uses (when it accesses it for the first time). It works on these shadows so changes remain local.

Upon commit it must:

- ➤ **Validate** that the the shadows were consistent...
- ➤ ...and no other transaction has committed an operation on an object which conflicts with one intended by this transaction
- ✔ If OK then commit the updates to the persistent objects, in the same transaction-order at every object
- ✘ If not OK then abort: discard shadows and retry

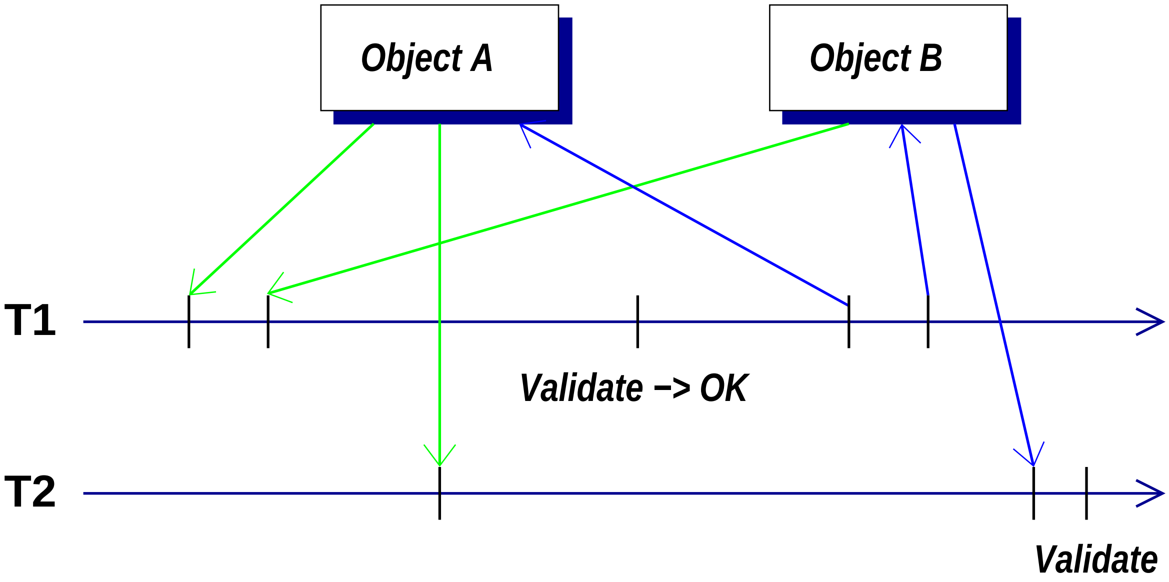
Until commit, updates are made locally. Abort is easy. No need for a roll-back mechanism

No cascading aborts or deadlock

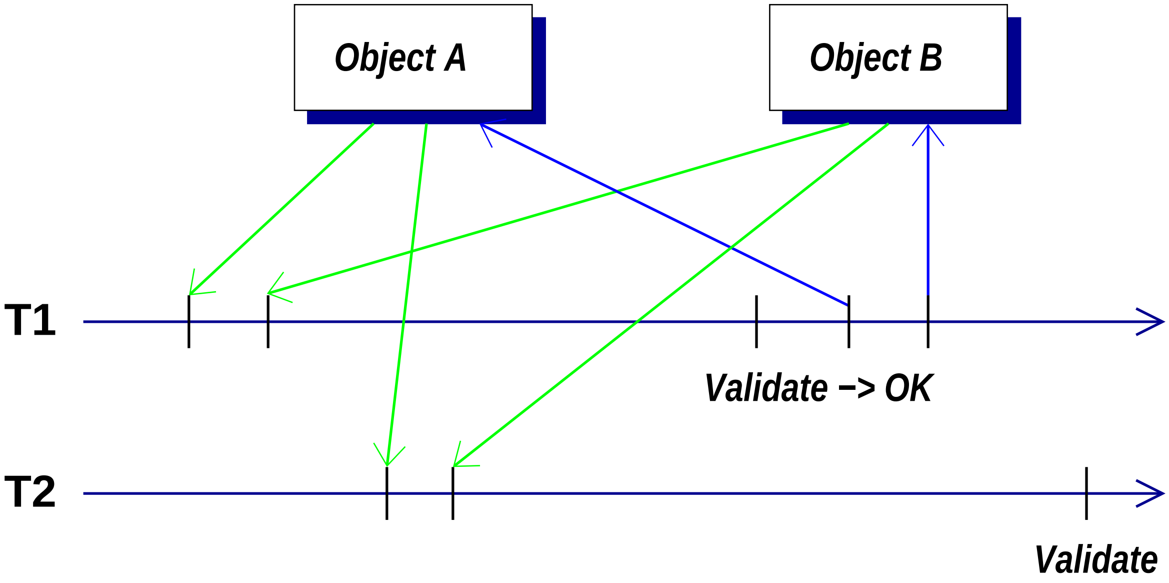
But conflicts force transactions to retry

Isolation – OCC (3)

- The first step avoids unrepeatable reads, e.g. T2 has seen T1's update to B but not seen T1's update to A:



- The second step avoids lost updates, e.g. T1 updates A and B, which T2 would overwrite if it was accepted:



Implementing validation

Validation is the complex part of OCC. As usual there are trade-offs between the implementation complexity, generality and likelihood that a transaction must abort

We'll consider a validation scheme using

- a single-threaded validator
- the usual distinction between *conflicting* and *commutative* operations

Transactions are assigned timestamps when they pass validation, defining the order in which the transactions have been serialized. We'll assign timestamps when validation starts and then either

- confirm during validation that this gives a serializable order, or
- discover that it does not and abort the transaction

Elaborate schemes are probably unnecessary: OCC assumes transactions do not usually conflict

Implementing validation (2)

The validator maintains a list of transactions it has accepted:

Validated transaction	Validation timestamp	Objects updated	Updates written back
$T1$	10	A, B, C	Yes
$T2$	11	D	Yes
$T3$	12	A, E	No

- Once a transaction passes validation, it can proceed to write its updates back to the shared objects
- Then the 'written back' flag can be set for it

Each object records the timestamp of the most recent transaction to update it:

Object	Timestamp
A	12
B	10
C	10
D	11
E	9

- In this case $T3$ is still writing back its updates: it's done A but not yet reached E

Implementing validation (3)

Consider $T4$ which updates B and E. Before it starts:

- ➤ Record the timestamp of the most recently validated fully-written-back transaction – in this case 11. This will be $T4$'s *start time*

When $T4$ accesses any object for the first time:

- ➤ Take a shadow copy of the object's current state
- ➤ Record the timestamp seen (e.g. 10 for B and 9 for E)

Validation phase 1:

- ➤ Compare each shadow's timestamp against the start time
  - ✓ Shadow earlier/equal: part of a consistent snapshot at the start time (B, E both OK here)
  - ✗ Shadow later: it may have seen a subsequent update not seen by other shadows

Validation phase 2:

- ➤ Compare the transaction $T4$ against each entry in the list after its start time:
  - ✓ No problem if they do not conflict
  - ✗ Abort $T4$ if a conflict is found (with $T3$ on E in this case)

Isolation – recap

We've seen three schemes:

1. 2PL uses explicit locking to prevent concurrent transactions performing conflicting operations. Strict 2PL enforces strict isolation and avoids cascading aborts. Both may allow deadlock

   ✓ Use when contention is likely and deadlock avoidable. Use strict 2PL if transactions are short or cascading aborts problematic

2. TSO assigns transactions to a serial order at the time they start. Can be modified to enforce strict isolation. Does not deadlock but serializable executions may be rejected

   ✓ Simple and effective when conflicts are rare. Decisions are made local to each object: well suited for distributed systems

3. OCC allows transactions to proceed in parallel on shadow objects, deferring checks until they try to commit

   ✓ Good when contention is rare. Validator may allow more flexibility than TSO

Exercises

19-1 A system is to support abortable transactions that operate on a data structure held only in main memory.

(a) Define and distinguish the properties of *isolation* and *strict isolation*.

(b) Describe *strict two-phase locking* (S-2PL) and how it enforces strict isolation.

(c) What impact would be made by changing from S-2PL to ordinary 2PL?

You should say what the consequences are (*i*) during a transaction's execution, (*ii*) when a transaction attempts to commit and (*iii*) when a transaction aborts?

Exercises (2)

19-2 You discover that a system does not perform as well as intended using S-2PL (measured in terms of the mean number of transactions that commit each second). Suggest why this may be in the following situations and describe an enhancement or alternative mechanism for concurrency control for each:

(a) The workload generates frequent contention for locks. The commit rate sometimes drops to (and then remains at) zero.

(b) Some transactions update several objects, then perform private computation for a long period of time before making one final update.

(c) Contention is extremely rare.

Exercises (3)

19-3* A system is using S-2PL to ensure the serializable execution of a group of transactions. Suppose that a new kind of transaction is to be supported which is tolerant to *dirty reads* and to *unrepeatable reads*.

(a) Describe how the new transaction could proceed, in terms of when it must acquire and release locks on the objects from which it (*i*) reads and (*ii*) updates.

(b) Does supporting this new kind of transaction have any impact on the S-2PL algorithm used by the existing ones?

Past exam questions: 1994 Paper 6 Q7, 2001 Paper 3 Q1

Lecture 20: Crash recovery & logging

Previous lecture

- Enforcing isolation
- Two-phase locking
- Timestamp ordering
- Optimistic concurrency control

Overview of this lecture

- Logging
- Crash recovery
- Checkpoints

Persistent storage

Assume a **fail-stop** model of crashes in which

- the contents of main memory (and above in the memory hierarchy) is lost
- non-volatile storage is preserved (e.g. data written to disk)

If we want the state of an object to be preserved across a machine crash then we must either

- ensure that sufficient replicas exist on different machines that the risk of losing all is tolerable (*Part-II Distributed Systems*)
- ensure that the enough information is written to non-volatile storage in order to recover the state after a restart

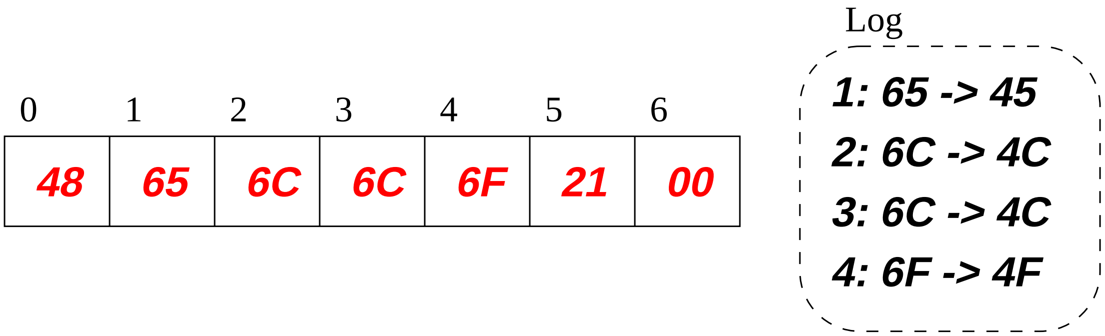
Can we just write object state to disk before every commit? (e.g. invoking `flush()` on any kind of Java `OutputStream`)

✘ Not directly: the failure may occur part-way through the disk write (particularly for large amounts of data)

Persistent storage – logging

We could split the update into stages:

1. Write details of the proposed update to an **write-ahead log** – e.g. in a simple case giving the old and new values of the data, or giving a list of smaller updates as a set of $(address, old, new)$ tuples

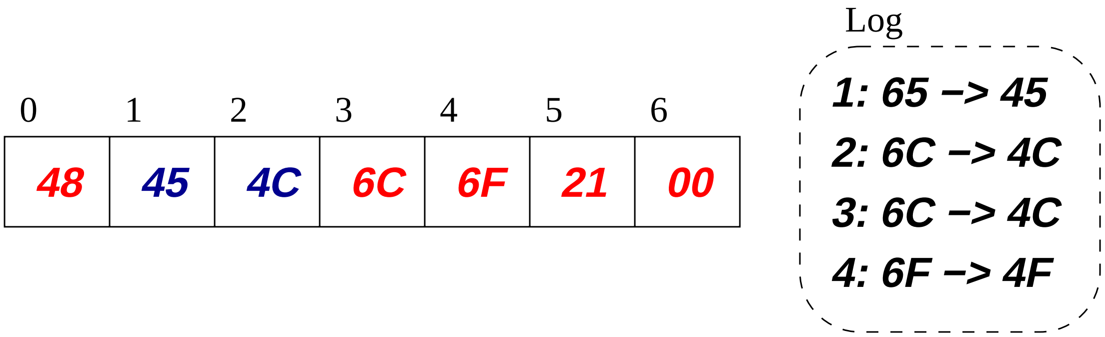


0	1	2	3	4	5	6
48	65	6C	6C	6F	21	00

Log

1: 65 -> 45
2: 6C -> 4C
3: 6C -> 4C
4: 6F -> 4F

2. Proceed through the log making the updates



0	1	2	3	4	5	6
48	45	4C	6C	6F	21	00

Log

1: 65 −> 45
2: 6C −> 4C
3: 6C −> 4C
4: 6F −> 4F

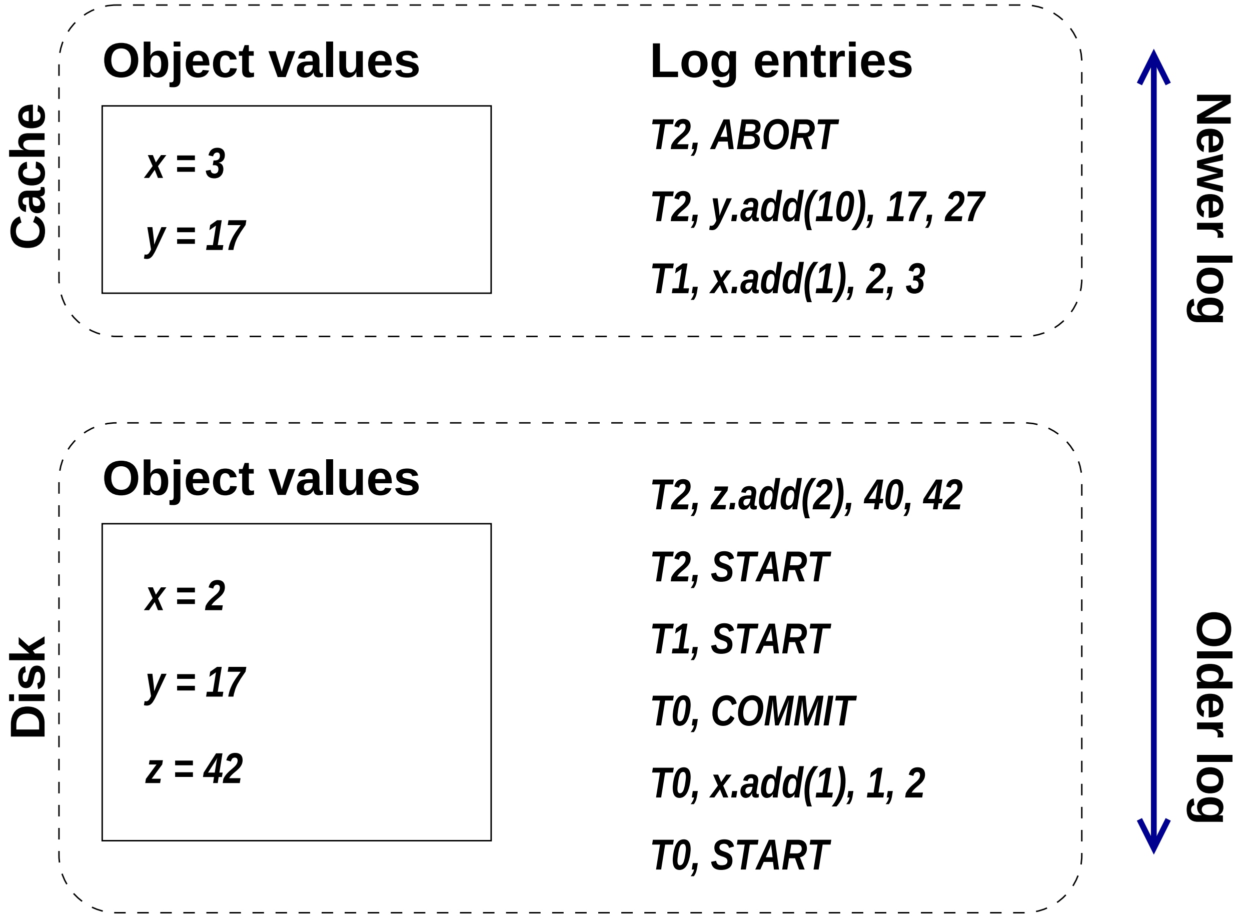
Crash during 1 ⇒ no updates performed

Crash during 2 ⇒ re-check log, either undo (so no changes) or redo (so all changes made)

Persistent storage – logging (2)

More generally we can record details of multiple transactions in the log by associating each with a *transaction id*. Complete records, held in an append-only log, may be of the form:

- $(transaction, operation, old, new)$
- or $(transaction,$ start/abort/commit$)$



Persistent storage – logging (3)

We can cache values in memory and use the log for recovery

- A portion of the log may also be held in volatile storage, but records for a transaction must be written to non-volatile storage before that transaction commits
- Values can be written out lazily

This allows a basic recovery scheme by processing log entries in turn (oldest → youngest)

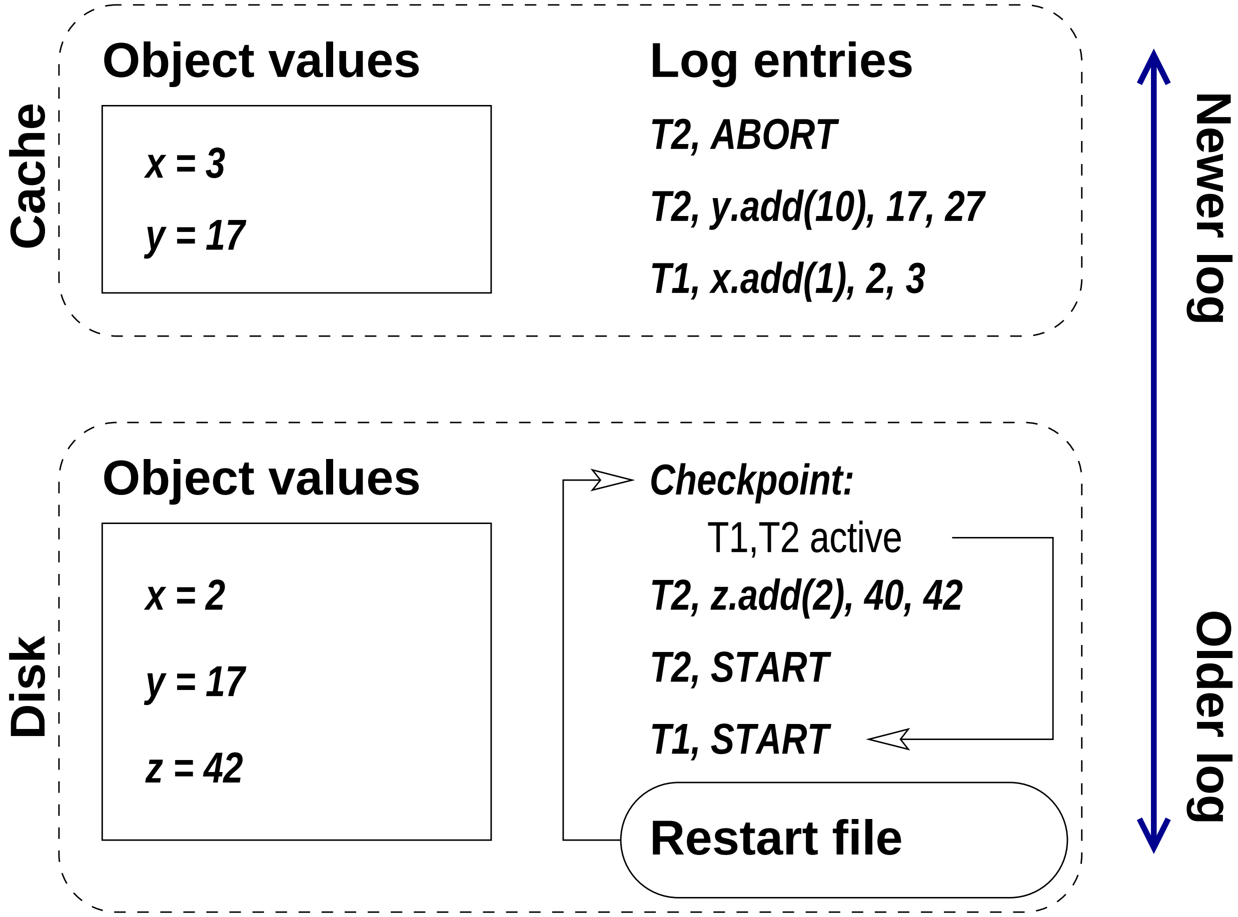
- Note the need for an **idempotent** record of an update – e.g. for `add` we keep the new & old values as well as the difference
- The old value lets us undo a transaction that's either logged as aborted...
- ...or for which the log stops before we know its outcome

The naïve recovery algorithm can be inefficient

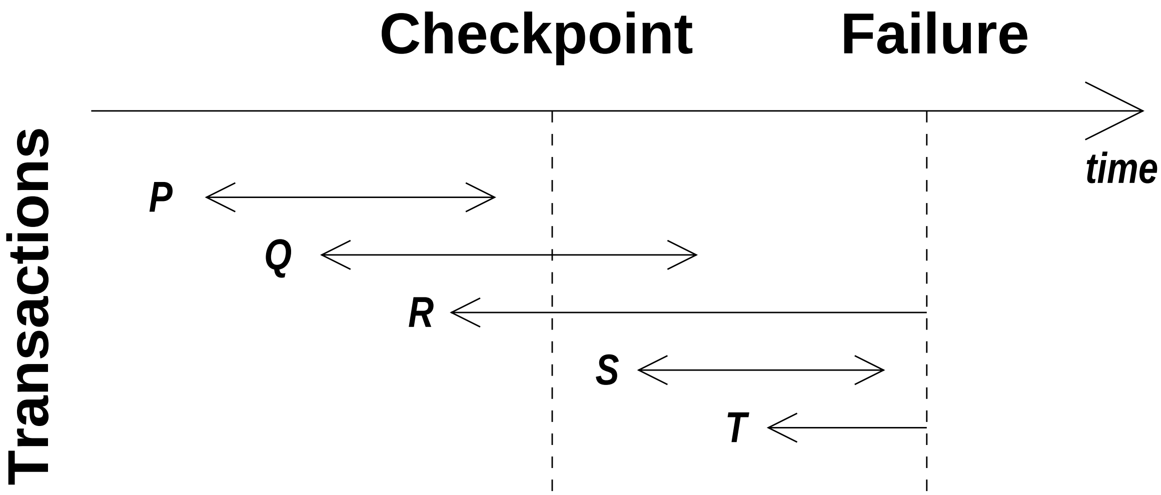
Persistent storage – logging (4)

A checkpoint mechanism can be used, e.g. every x seconds or every y log records. For each checkpoint:

- Force log records out to non-volatile storage
- Write a special **checkpoint** record that identifies the then-active transactions
- Force cached updates out to non-volatile storage



Persistent storage – logging (5)



P already committed before the checkpoint – any items cached in volatile storage must have been flushed

Q active at the checkpoint but subsequently committed – log entries must have been flushed at commit, REDO

R active but not yet committed – UNDO

S not active but has committed – REDO

T not active, not yet committed – UNDO

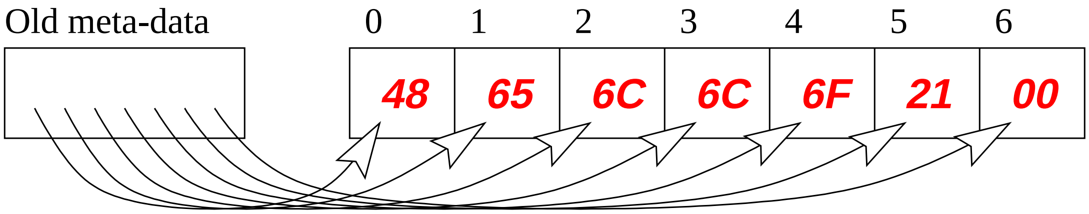
Persistent storage – logging (6)

A general algorithm for recovery:

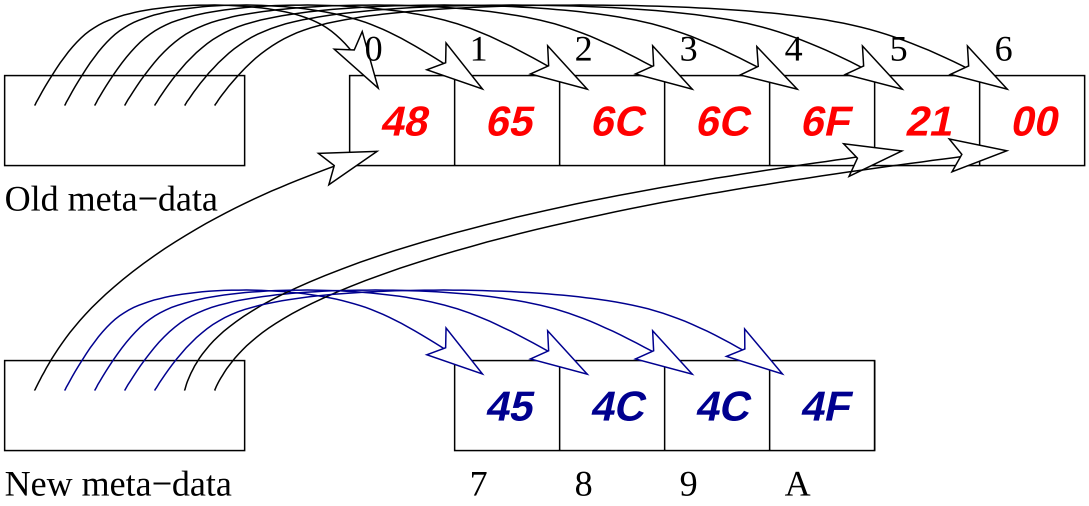
- The **recovery manager** keeps UNDO and REDO lists
- Initialize UNDO with the set of transactions active at the last checkpoint
- REDO is initially empty
- Search forward from the checkpoint record:
  - Add transactions that `start` to the UNDO list
  - Move transactions that `commit` from the UNDO list to the REDO list
- Then work backwards through the log from the end to the checkpoint record:
  - UNDOing the effect of transactions on the UNDO list
- Then work forwards from the log from the checkpoint record:
  - REDOing the effect of transactions in the REDO list

Persistent storage – shadowing

An alternative to logging: create separate old and new versions of the data structures being changed



An update starts by constructing a new ‘shadow’ version of the data, possibly sharing unchanged components:



The change is committed by a single in-place update to a location containing a pointer to the current version. This last change must be guaranteed atomic by the system

How can this be extended for persistent updates to multiple objects?

Exercises

20-1 Consider the basic logging algorithm (without checkpointing). Show how it enforces atomicity and durability of committed transactions.

While it is not necessary to construct a formal proof, you should be methodical and consider the different operations that the system may perform (e.g. updating objects in memory, starting and concluding transactions, transfers between disk and the in-memory object cache and writing of log entries). Consider the effect of failure and recovery after each one.

20-2 Suppose that you wish to augment the *Slime Volleyball* game with a high-score table held on disk. Is it necessary to use any of the schemes presented here for persistent storage? If so then suggest which would be most appropriate. If not then say why none is needed.

Past exam questions: 1999 Paper 4 Q2, 1995 Paper 3 Q1, 1997 Paper 4 Q2