Concurrent Systems and Applications

CST Part 1B, Michaelmas 2004

John K Fawcett

`john.fawcett@cl.cam.ac.uk`

Lecture 1: Introduction

Concurrent Systems and Applications is closely based on last year's course of the same name by Tim Harris, and is a development of *Further Java* and *Further Modula-3*, which came before. This course also includes topics from *Concurrent Systems* and examples relating to Andy Hopper's *Additional Topics* series in Part II.

Although the content is familiar, many of the slides are new this year and I would be grateful to receive by email any feedback and, of course, notice of any errors. Send email to *john.fawcett@cl.cam.ac.uk* or use the lab's feedback webpage. Thanks are due to Jon Davies and all the others who proof read early versions of these notes.

This course consists of 20 lectures and 2 examples classes. The examples classes will demonstrate the topics covered in Concurrent Systems and Applications by giving substantial source code that would be impractical to discuss inline during lectures.

Like last year's course, this course provides more background information of several of the topics discussed than was possible in the earlier *Further Java* course.

Programs will be demonstrated during the lectures. These notes contain appropriate spaces for you to make notes.

Resources

- `http://www.cl.cam.ac.uk/Teaching/2004/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/`
- Past Tripos questions relevant to this course can be found under
  - Concurrent Systems and Applications
    `http://www.cl.cam.ac.uk/tripos/t-ConcurrentSystemsandApplications.html`
  - Concurrent Systems
    `http://www.cl.cam.ac.uk/tripos/t-ConcurrentSystems.html`
  - Further Java
    `http://www.cl.cam.ac.uk/tripos/t-FurtherJava.html`
- 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`)
- Recommended reading list...

Recommended reading

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

- Write your own programs
- Take notes in the Lectures
- Read the recommended texts

- * Bacon, J., Harris, T. (2003) *Operating systems*
- Bacon, J. (1997) *Concurrent systems (2nd ed.)* Addison-Wesley.
- Lea, D. (1999). *Concurrent programming in Java (2nd ed.)* Addison-Wesley.
- Bracha, G., Gosling, J., Joy, B., Steele, G. (2000). *The Java language specification. Addison-Wesley (2nd ed.)* http://java.sun.com/docs/books/jls/
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design patterns* Addison-Wesley.
- Bruce Eckel ((December 2002) *Thinking in Java (3rd ed.)* Prentice-Hall and downloadable http://www.mindview.net/Books/TIJ/

Where do we find concurrency?

- 'Concurrent systems' just means those where several independent or related activities are proceeding at the same time. For example...
- between the system as a whole and its user, external devices, etc.
- between applications running at the same time on a computer:
  - context switching by the OS
  - 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
- 'housekeeping' activities within an application, e.g. garbage collection
- SIMD instructions in the CPUs and GPUs.
- symmetric multi-threading (SMT, '*hyperthreading*')

  How many threads when you run this?

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

HotSpot Client VM

- Interrupt the program (ctrl+backslash)...
- 7 threads

Outline of CS&A

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

  Lectures notes in this handout

- Part 2: Further Java Topics
  - 4 Lectures
- Part 3: Concurrent Systems
  - 6 Lectures
- Part 4: Distributed Systems and Transactions
  - 6 Lectures

  Lecture notes will be provided later

- Examples classes: threaded, distributed, bluetooth-enabled, database-backed, tank of fish with attitude.
  - Java-enabled mobile 'phones, time permitting

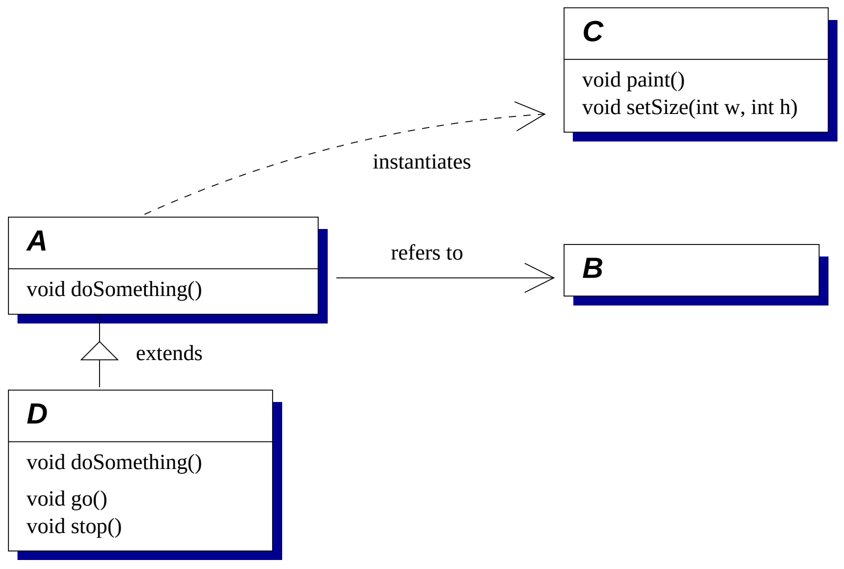
Timetable

- 24 timetabled slots
- First 12 lectures in the Heycock Lecture Theatre

  Then a change of venue...
- Lecture 13 (Fri 2004-11-05) in the William Gates Building, LT1
- *No Lecture on Monday 8 or Wednesday 10 November*
- Remaining Lectures in WGB::LT1 from Fri 2004-11-12
- Examples classes at the end
  - Mon 2004-11-29
  - Wed 2004-12-01

Notation

In many of the examples depicting hierarchies UML-style class diagrams are used. The nodes of these graphs represent classes and the edges between them denote relationships of different kinds. “Refers to” is shown with a solid arrow from the class doing the referring to that being referred to. “A instantiates C” is indicated by an broken arrow from A to C. An arrow with the head half way along indicates the class being pointed from extending the other. Interesting methods are listed.



This notation is consistent with Gamma *et al*’s text book; others may vary.

Recap of basic Java

- Look through notes from last year
- Have another look at the Ticked Exercises, especially the starred exercises
- Have a look at the code snippets from last year's course

Exercises

1. Compile the `Simple` class using the javac compiler.
2. Read the online Java API documentation and answer these questions:
   - What is a *deprecated* method? Why should we avoid using them?
   - Find `java.lang.Object`; describe the intended purpose of as many of the methods it provides as you can.
3. Find `java.lang.Throwable` in the documentation.
   - A programmer believes that although the Java compiler says a number of exceptions might be thrown by one of his methods, no exception will ever occur in practice. He is a busy chap and gets rid of the compiler's complaints by wrapping the code in a `try...catch` block as below. Describe the problems that he or others might encounter with this approach.

   ```
   try {
     ...
   } catch (java.lang.Throwable t) {}
   ```
   - Why might Throwable objects implement Serializable? [Answer in a later lecture.]

4. (Harder) Read the documentation for `java.lang.Object.finalize()`. Write the contents of the empty method in the program below (without printing

either “finalize” or “constructed” yourself!) for four versions of the program so that it prints:

- “finalize” as many times as “constructed”
- “finalize” more often than “constructed”
- “finalize” fewer times than “constructed”
- “constructed” precisely three times (harder still)

...or explain why such is impossible. It is cheating to change anything other than the contents of `yourMethod()`!

```
class Ex1Point4 {
  public static void main(String [] args)
   throws Throwable {
    A a = new A();
    for (int x=0;x<5;x++) {
      try {
        Ex1Point4 e1p4 = new Ex1Point4(a);
      } catch (Throwable t) {}
    }
    System.gc();
  }

  Ex1Point4(A a) throws Throwable {
    a.yourMethod();
    System.out.println("constructed");
  }

  protected void finalize() {
    System.out.println("finalize");
  }

  static class A {
    A() {}
    void yourMethod() throws Throwable {
       // write your code here
    }
  }
}
```

Lecture 2: Objects and classes

Previous lecture

- Course structure, administration details.
- Recap of basic Java using a number of Excercises

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)

- 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
  - identifies a particular instance
  - is the special value `null`
- More than one reference can refer to the same object, for example:

  ```
  TelephoneEntry tel =
      new TelephoneEntry("John", "76019");

  TelephoneEntry tel2 = tel;
  ```

  ...creates two references to the same object.

  If `tel.name` is updated, the new value can also be accessed by `tel2.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;   // nice and innocent!
}

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 an expression like `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 initialised.
- 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 – before you deploy the program!
  - however, be wary of forgetting to `make clean`!
- As with methods, constructors can be overloaded:

```
class TelephoneEntry {
  TelephoneEntry(String name) {
    this(name,"0");
  }

  ...
}
```

- 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 constructors.

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 nxt;
}
```

- We construct new data types by combining old ones together to model 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 `Wheel[]`.
  - ... but 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 `favouriteTellytubby`

Inheritance

- **Inheritance** is another way to combine classes–it typically models an ‘is a’ relationship, e.g.
  - between `Bicycle` and a more general `PersonnelTransport`
  - between `SpaceElevator` and a more general `SatelliteLauncher`
- Inheritance defines a new **sub-class** in terms of an existing **super-class**. The sub-class is intended to be a *more specialised* version of the super-class. It might
  - add new fields
  - add new methods
  - override existing methods to change their behaviour

```
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. if we have a class `B` that `extends A` then `anotherA` could refer to an instance of class `B`.
- A particular object may be accessed through fields and variables of different *reference types* over the course of its lifetime; its *class* is fixed at the 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; // cast 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`, which may or may not turn out to be an instance of `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` (or a mixture).
- An array of type `B[]` can hold only 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 runtime because `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 super-class.
- For example:

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

class B extends A {
  String x;   // hides A’s int called ‘x’
  int y;      // hides A’s int called ‘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 used in the code to specify the method's name.
- Consequently, the type of an object reference does not affect the chosen method in these examples. A common mistake:

```
 1   class A {
 2     void f() {
 3       System.out.println("Super-class");
 4     }
 5   }
 6   class B extends A {
 7     void f() {
 8       System.out.println("Sub-class");
 9       ((A)this).f();  // try to call original
10     }
11   }
```

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

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

Exercises

1. Write concise definitions of *object*, *class*, *object reference*, and *type* with respect to a simple example in Java.
2. Is this statement true for Java? "If `S` is a sub-type of `T` then an object of type `S` can be used anywhere that an object of type `T` can be used." Explain your answer.
3. Read about the `instanceof` keyword in the online Java documentation and write a program using it to distinguish between these two classes:

   ```
   class A {}
   class B extends A {}
   ```

4. `super.super` is not valid syntax in Java even though it might appear to provide a means to access the super-class of a class' super-class. Why might the designers have disallowed this?
5. In some languages (like *SmallTalk*), a class can change which other class it extends at runtime! Suppose this were permitted in Java; outline how this could lead to confusion by means of a small example program. Are any safeguards needed to prevent the programmer doing something insane? Would this feature ever be useful?

The Programming Challenge

- Can you write an additional Java class which creates an object that, when passed to the test method causes it to print “Here!”? As I say in the code, editing the class A itself, or using library features like reflection, serialization, or native methods are considered cheating! I’ll provide some hints in lectures if nobody can spot it in a week or so. None of the PhD students has got it yet.

```
public class A  {
  // Private constructor tries to prevent A
  // from being instantiated outside this
  // class definition
  //
  // Using reflection is cheating :-)

  private A() { }

  // ’test’ method checks whether the caller has
  // been able to create an instance of the ’A’
  // class.  Can this be done even though the
  // constructor is private?

  public static void test(Object o) {
    if (o instanceof A) {
      System.out.println ("Here!");
    }
  }
}
```

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 and abstract classes
- Nested classes

Packages (1)

- Java groups classes into **packages**. Classes within a package are typically written by co-operating programmers and are 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.jkf21.TelephoneEntry
  ```

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

  ```
  package uk.ac.cam.cl.jkf21.examples;

  class TelephoneEntry { ... }
  ```

- Don't have to *create* the package in any way; just quote the name in any `package` statement.
  - Watch out for typos in your package names!
  - Errors are often manifested as compiler errors saying that it can't find referenced or extended classes.
- Some compilers create subdirectories in the file system, nesting one directory level for each full stop in the package's fully-qualified name.
- By convention, package names are spelled with an initial capital letter, capital letters to begin each word, and are nouns. e.g. `uk.ac.cam.cl.jkf21.AwesomeStuff`

Packages (2)

- Definitions in the current package and in `java.lang` can always be accessed without specifying the package name. Otherwise:
  - the `import` keyword can be used to tell the compiler that we mean to use items from other packages:

```
// import all classes+interfaces from here...
import java.util.*;

// import just this single class...
import java.awt.Graphics;
```

  - the fully-qualified class name can be used in the code:

```
java.util.Map = new java.util.HashMap();
```

- Defensive programming – always using fully-qualified names prevents your code misbehaving or failing to compile if you or someone else later adds to a class called "Map" or "HashMap" to the local package.
  - I prefer using fully-qualified names to `import`ing too much stuff.
  - But all my friends hate my code because the lines are too wide.
  - But I use long names anyway.

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 legal */
  }

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

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

Access modifiers (1)

- 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.

```
class UniversityCard {

  MifareApplicationID mad[];
  Byte checksum;

  UniversityCard(MifareApplicationID mad[],
                                        Byte checksum) {
    this.mad = mad;
    this.checksum = checksum;
  }

  void changeMADEntry(int index,
      MifareApplicationID newval) {
    mad[index & 0x0F] = newval;
    recomputeChecksum();
  }

  void recomputeChecksum() {
    // Do clever stuff
    ...
  }

  void programCard() {
    // talk to the printer
    ...
  }
}
```

Access modifiers (2)

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

	same class	same package	sub-class	anywhere
`public`	Y	Y	Y	Y
`protected`	Y	Y	some	
`default`	Y	Y		
`private`	Y			

- `protected` access permits access by sub-classes in the same package, or in a different package provided the type of the object reference is that of the sub-class...

The `protected` modifier

- A `protected` entity is always accessible in the package within which it is defined.
- Additionally, it is accessible within the 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` might 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.
  - Also for security.
- A `final` class cannot be sub-classed at all.
- The value of a `final` field is fixed after initialisation, either directly or in every constructor, e.g.

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

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

- Every code path that creates instances of class `FinalField` **must** initialise the `final` fields.
- `final` fields are also used to define constants, e.g.

```
public static final int BLUE  = 1;
public static final int WHITE = 2;
public static final int RED   = 3;
```

The `abstract` modifier (1)

- 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.
  - Look for examples in the java class libraries – `java.util.AbstractMap`, `java.util.AbstractList`, etc.
- Note that fields cannot be `abstract`: they cannot be overridden in sub-classes.

The `abstract` modifier (2)

- It is not permitted to instantiate an abstract class, but we can use object references of the type of the abstract class, e.g.

```
public abstract class A {
  int x=20;
  abstract int methodname(A a);
}

public class B extends A {
  int methodname(A a){
    System.out.println("The A's x is "+a.x);
    return 42;
  }

  public static void main(String [] args) {
    new B().methodname(new A());  // invalid
    new B().methodname(new B());  // fine
  }
}
```

- Some compilers require explicit declaration as `abstract` of classes containing abstract methods.

The `abstract` modifier (3)

- Ask the audience: is this valid?

```
public abstract class A {
  int x=20;
  abstract int methodname(A a);
}

public class B extends A {
  int methodname(A a){
    System.out.println("The A's x is "+a.x);

    // Am I allowed to access the
    // abstract method in A?
    a.methodname(a);

    // Does casting to a B make any
    // difference?
    ((B) a).methodname(a);

    return 42;
  }

  public static void main(String [] args) {
    new B().methodname(new B());  // fine
  }
}
```

The `abstract` modifier (4)

- Moving functionality into the super-class...

```
public abstract class Collection {
  protected java.lang.Object [] objs;

  abstract void sortTheStuff();

  void printInOrder() {
    // Sub-classes will implement
    // collections in different ways.
    // Lets have our sub-class sort
    // the things in this Collection
    // in whatever way it needs to...
    sortTheStuff();

    // Right, now we can print them...
    for (int x=0;x<objs.length;++x)
      System.out.println(objs[x]);
  }
}
```

The `static` modifier (1)

- The `static` modifier can be applied to any method or field definition (and also 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 instantitated: there is only one 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 floating point arithmetic (see *Numerical Analysis I*). This ensures that identical results are produced on all computers, regardless of the processors' floating point hardware.
  - `strictfp` can be applied to classes too—all methods are implemented using strict floating point.
- There are four other modifiers to be covered in later lectures:
  - `synchronized` ensures that several threads in a multi-threaded applications do not access the same class/method at the same time.
  - `volatile` causes the JVM to re-read a variable from memory each time the value is required—caching is not permitted in many circumstances.
  - `transient` fields are used with the `Serialization` API. `transient` fields are not sent over the network when classes are copied from machine to machine—e.g. temporary values, or large structures that can be recomputed more quickly at the far end than the network can transfer them.
  - 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 (1)

- There are often groups of classes that provide different implementations of the same kind of functionality.
  - e.g. the *collection* classes in `java.util`—`HashSet` and `ArraySet` provide set operations; `ArrayList` and `LinkedList` provide list-based operations.
- In that example there are some operations available on all *collection*s, 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 has a *single* super-class (in Java), 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 desireable 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 `extend` only 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 (constants).
- 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 either:
  - supply definitions for each of the declared methods; or
  - be declared an abstract class.

Nested classes (1)

- 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; and
  - **anonymous inner classes**.
- Beware: the term *inner class* is sometimes used incorrectly to refer to all nested classes. In fact, *inner classes* only form a subset of *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; and
  - (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 qualificiation 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. There is no mechanism for an instance of `Bus` to acquire a list of the instances of `Wheel` that are associated to it.
- 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 inerface name rather than a class name—providing inline implementations of all the methods in the interface.

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

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

Exercises (1)

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.
2. Describe the differences and similarities between abstract classes and interfaces in Java. How would you select which kind of definition to use?
3. Why is it sensible that (i) interfaces cannot be private; (ii) method signatures on interfaces are implicitly public; (iii) nested interfaces are implicitly static?
4. A common grumble about Java is the lack of *multiple inheritance*—being permitted to `extend` more than one class. Describe *three* ways in which this problem can be resolved to produce (one or more) class definitions. What are the advantages and disadvantages of each approach?

Exercises (2)

5. An enthusiast for programming with *closures* proposes a new language 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 valies 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 OO design
- Access modifiers to enforce encapsulation
- Interfaces and abstract classes
- Nested classes

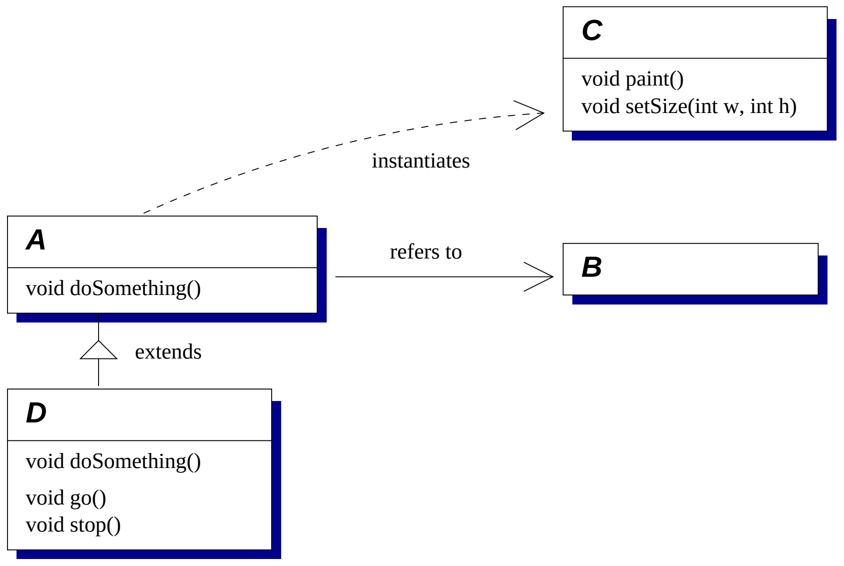
Overview of this lecture

- Some ways of using these facilities effectively
- Think about the ways we might implement parts of the distributed fish tank.

Design patterns

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

- common terminology for describing program organisation and conveying the purposes of inter-related classes; and
- examples of how to structure programs for flexibility and re-use.



Observer pattern

- Suppose we have a user interface in which we need to permit the user to specify the colour of an on-screen item (like a fish in the tank...).
- The client's specification demands there be three ways to specify colours:
  - Text boxes for red, green, and blue elements;
  - Sliders for red, green, and blue elements; and
  - Selecting from a palette of commonly-used colours.
- BUT more than one of these input mechanisms might be displayed simultaneously.
- AND when the user operates one of them, the others must update appropriately.

- Question: how might we represent this in an object-oriented program?

Bad hack (1)

- Write three classes: one for each input mechanism.
- Write code in each one to update the other two each time it is used:

```
class ColourTextBoxes {
  ColourSlider cs;
  ColourPalette cp;
  ...
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
    cs.update(r,g,b);
    cp.update(r,g,b);
  }
}

class ColourSlider {
  ColourTextBoxes ct;
  ColourPalette cp;
  ...
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
    ct.update(r,g,b);
    cp.update(r,g,b);
  }
}

class ColourPalette {
  ColourTextBoxes ct;
  ColourSlider cs;
  ...
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
    ct.update(r,g,b);
    cs.update(r,g,b);
  }
}
```

Bad hack (2)

Advantages

- Simple and efficient.

Disadvantages

- If we want to add a fourth way to select colours then we have to change the code in several places.
- In general, each of the colour selection classes needs to know about all the others.

Second attempt (1)

- Move the code that updates each object in to a common super-class:
  - it would be centralised; and
  - there would be only a single method to change if we add a fourth type of colour selector.

```
class Colour {
  int r,g,b;
  ColourTextBoxes ct;
  ColourSlider cs;
  ColourPalette cp;
  void setColour(int r,int g,int b) {
    this.r = r; this.g = g; this.b = b;
    ct.update();
    cs.update();
    cp.update();
  }
}
class ColourTextBoxes extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}
class ColourSlider extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}
class ColourPalette extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}
```

Second attempt (2)

Advantages

- Remains simple.
- Efficient.
- Easier to maintain.

Disadvantages

- Messy—why should `Colour` have to know about all the sub-classes?
- Adds clutter to `Colour`.

Observer pattern (1)

- Lets keep `Colour` clean:

```
class Colour {
  int r,g,b;
  void setColour(int r,int g,int b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }
}
```

- Use the following as the super-class for `ColourTextBoxes`, `ColourSlider`, and `ColourPalette` instead:

```
abstract class ColourObserver {
  ColourSubject colsubj;
  abstract void update();
}
```

- ...where `ColourSubject` is a class designed to handle all the calls to update the other colour selectors:

```
class ColourSubject {
  Colour c;
  ColourObserver observers [];
  void setColour(int r,int g,int b) {
    c.setColour(r,g,b);
    for (int x=0;x<observers.length;++x)
      observers[x].update();
  }
  void addObserver(ColourObserver co) {
    /* insert into observers[] */
    ...
  }
}
```

Observer pattern (3)

- So the sub-classes now look like this:

```
class ColourSlider extends ColourObserver {
  void update() {
    /* read colour, redraw our GUI */
    ...
  }

  void changeTo(int r,int g,int b) {
    colsubj.setColour(r,g,b);
  }
}
```

Observer pattern (4)

- Also known as the **Model-View-Controller** pattern.

Advantages

- In Java, *observers* can be implemented as interfaces rather than as concrete classes.
  - Doesn't use up the single opportunity to sub-class another class.
- A many-to-many, dynamically changing relationship can exist between subjects and observers.

Disadvantages

- The flexibility limits the extent of compile-time type-checking.
- If observers can change the subject than cascading or cyclic updates could occur.
- Potential for a large amount of computational overhead.
  - Consider slowly dragging the slider from left to right.

Singleton pattern

- Ensures that a class can be instantiated at most once.
- Private constructor ensures external classes cannot instantiate the class by calling `new`.
- A static method creates an instance when first called and subsequently returns an object reference to the same instance.

```
class Singleton {
  static Singleton theInstance = null;

  private Singleton() {...}

  static Singleton getInstance() {
    if (theInstance == null)
      return (theInstance = new Singleton());
    return theInstance;
  }

  void method1() {...}
  void method2() {...}
  void method3() {...}
}
```

- More flexible than a suite of static methods: allows sub-classing, e.g. `getInstance` on `ToolKit` might return `MotifToolKit` or `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.

Abstract factory pattern (1)

Suppose we have 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 appearances.

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

- We could have the application know about all the options and, whenever it needs to construct a new instance, switch on the mode in which it is running:

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

Disadvantages

- It would be lots of work to add support for a new GUI system.
- And we would have to change every application!
- A buggy application might try to use a `MacWindow` with a `MotifScrollBar`.

Abstract factory pattern (2)

- Define an abstract super-class `Factory`.
- Code for the methods is provided by one of a number of sub-classes, each implementing a different family, e.g. `MotifFactory`, `MacFactory`, …
- The factory class instantiates objects on behalf of the client from one of a family of related classes, e.g. `MotifFactory` instantiates `MotifWindow` and `MotifScrollBar`.
- 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. `MotifWindow` always with `MotifScrollBar`.
- Adding a new operation involves co-ordinated change to the `Factory` class and all its sub-classes.

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

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 you have a good implementation `BinomialTree` that instead implements some other interface, say `LookupTable`

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

- **Terminology**: the *Client* wishes to invoke operations on the *Target* interface which the *Adaptee* 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 (1)

- Use a hierarchy of classes to represent the nodes in a data structure, e.g.

```
class TreeNode {
  Object myData;
  TreeNode [] children;

  void accept(Visitor v) {
    // apply the visitor to my data...
    v.apply(myData);

    // ...and to each child’s data...
    for (int x=0;x<children.length;++x)
      children[x].accept(v);
  };
}

class TreeNode2 extends TreeNode {
  TreeNode2 () {children = new TreeNode[2];}
}
class TreeNode3 extends TreeNode {
  TreeNode2 () {children = new TreeNode[3];}
}
class TreeNode4 extends TreeNode {
  TreeNode2 () {children = new TreeNode[4];}
}
```

Visitor pattern (2)

- Implement different visitors for different tasks, e.g.

```
abstract class Visitor {
  abstract void apply(Object o);
}

class VisitorCountNulls extends Visitor {
  int count=0;
  void apply(Object o) {if (o == null) ++count;}
}

class VisitorSumIntegers extends Visitor {
  int sum=0;
  void apply(Object o) {
    if (o instanceof Integer) {
      sum += ((Integer) o).intValue();
    }
  }
}
```

Visitor pattern (3)

- The data structure is built from instances of `TreeNode2`, `TreeNode3`, etc.—all sub-classes of `TreeNode`.
- These classes, or a separate object structure class, provide some mechanism for traversing the data structure.
- The abstract *Visitor* class defines operations to perform on each node.
  - It might perform different tasks on each different sub-class of `TreeNode`.
- 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*.
- But changing the data structure requires changes to many classes.

Summary

Some common themes...

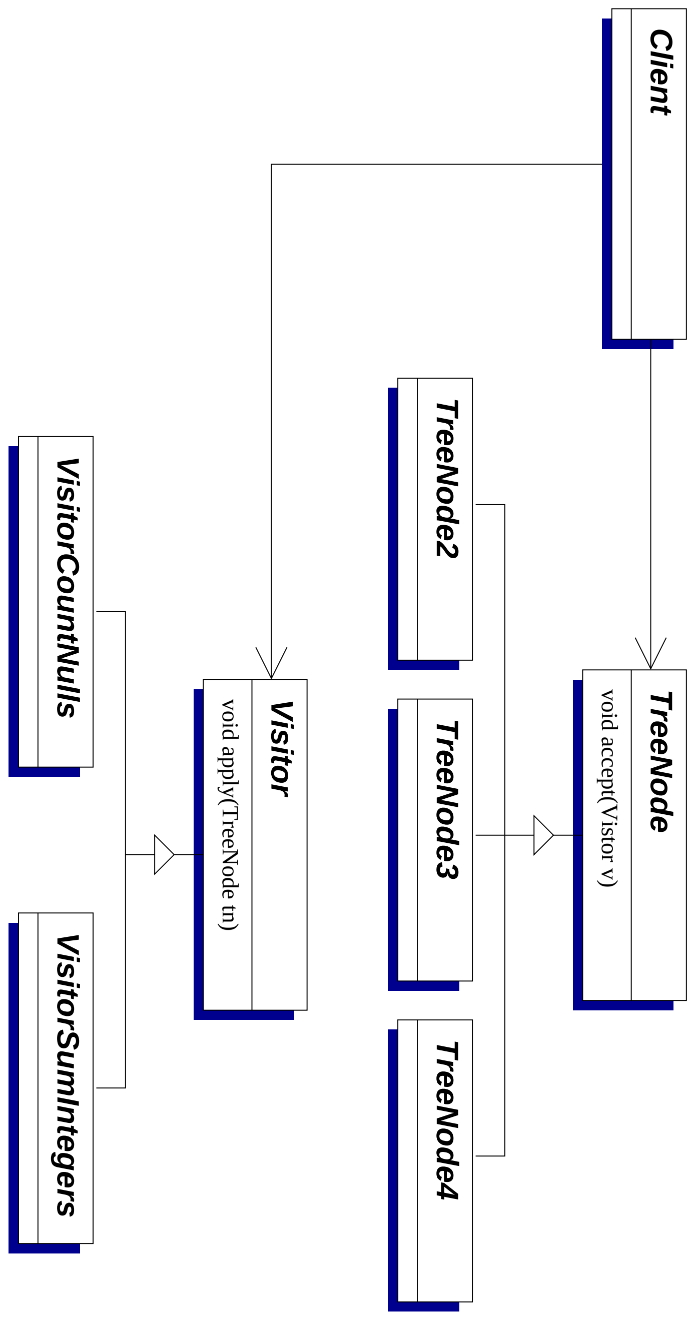
- Explicitly creating objects by specifying their class commits to a particular implementation.
  - It is often better to separate code responsible for instantiating objects—Abstract Factory and Singleton patterns.
- Tight coupling between classes makes independent re-use difficult.
  - e.g. the Visitor pattern separates the structure-traversal code from the visitor-specific operation to apply to each item found.
- Extending functionality by sub-classing commits *at compile time* to a particular organisation of extensions.
  - Composition and delegation may be prefereable (as in the Adapter pattern).

Exercises

1. Represent each of the design patterns as UML diagrams.
2. Have a look at `java.util.Iterator`. Describe how the methods implement the *Visitor* pattern.
3. What is the scope to use design patterns together?—e.g. in the Visitor pattern, why might the data structure nodes be adapters?

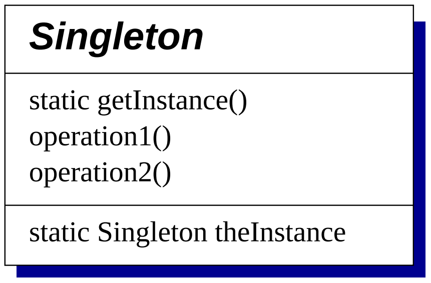
Answers to UML questions (1)

- Visitor pattern



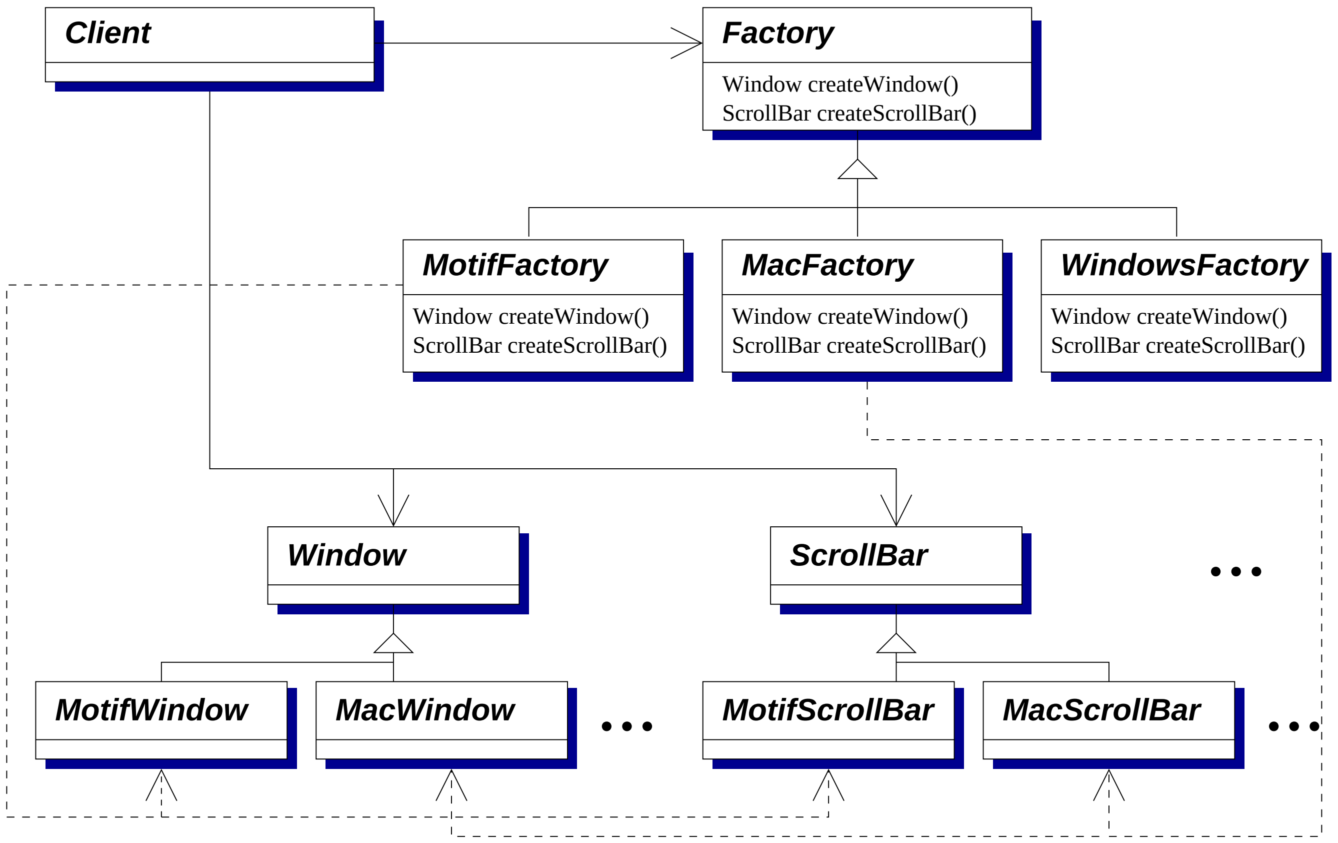
Answers to UML questions (2)

- Singleton pattern



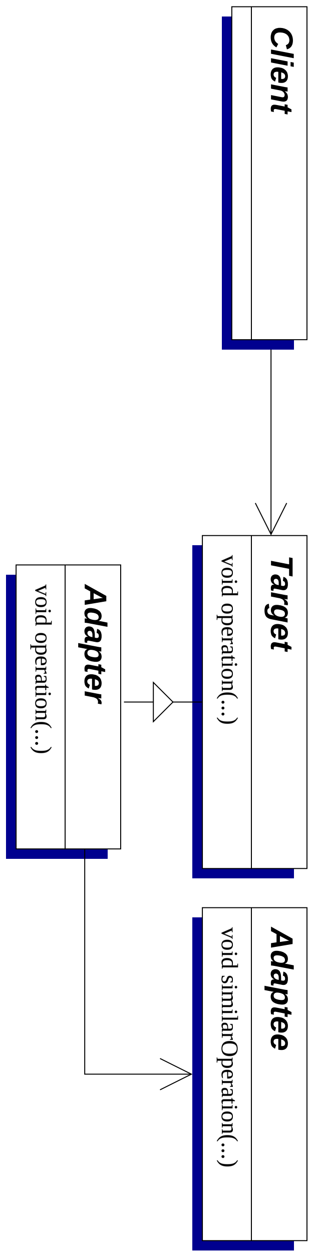
Answers to UML questions (3)

- Abstract factory pattern



Answers to UML questions (3)

- Adapter pattern



Lecture 5: Reflection and serialization

Now in Part 2 of this course.

Previous section
Programming with Objects

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

Overview of this section
Further Java

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

Reflection (1)

- 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; and
  - instantiate the class to create new objects.
- Given an object reference, a program can...
  - determine the class of the object it refers to; and
  - 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 development 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 a method inherited from `java.lang.Object`:

  ```
  java.lang.Class getClass()
  ```

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

- Line 3 creates a new instance of `ReflExample`.
- Line 4 obtains the `Class` object for that instance.
- Line 5 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);
    }
  }
}
```

- But what about `Exceptions` that might be thrown in the constructor?
- In fact, there are lots of `Exceptions` and `Errors` to think about...

Reflection (5)

- Errors and Exceptions for `Class.forName(String)`...
  - `java.lang.ClassNotFoundException`: the `ClassLoader` was unable to find any definition of the class whose name was supplied as argument.
  - `java.lang.LinkageError`: Subclasses of LinkageError indicate that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.
  - `java.lang.ExceptionInInitializerError`: the initialization provoked by this method failed—see `Throwable.getCause()` to find out what went wrong.

- Errors and Exceptions for `Class.newInstance()`...
  - `java.lang.InstantiationException`: the Class represents an abstract class, an interface, an array class, a primitive type, or void; or the class has no nullary constructor; or the instantiation failed for some other reason.
  - `java.lang.IllegalAccessException`: the class or its nullary constructor is not accessible.
  - `java.lang.ExceptionInInitializerError`: the initialization provoked by this method failed.
  - `java.lang.SecurityException`: there is no permission to create a new instance.

Reflection (6)

- We're taking a class name supplied as a parameter to the program and instantiating it. We can name any class we like...!
- By default a 0-argument constructor is called (and must exist).
- Specific constructors are also modelled by `Constructor` objects and each will define a `newInstance(Object [])` method; the `Object []` is a list of the arguments to the constructor.
- Think about why anyone would want to write

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

  instead of an ordinary `new` expression...

Fields and reflection (1)

- We can invoke `getFields()` on a `Class` object to obtain an array of the public fields defined on that class (or interface) and all super-classes (or super-interfaces).
- As a shortcut we can also use `getField(...)`, passing the name of an individual field to obtain information about it.
- 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 `checkMemberAcess` test for `member.DECLARED`) to a list of the `public`, `protected`, *default*, and `private fields`, excluding inherited fields.
- Given an instance of `Field` we can use...
  - `Class getDeclaringClass()`
  - `String getName()`
  - `int getModifiers()`
  - `Class getType()`

Fields and reflection (2)

```
public class ReflExample3 {
  public static int field1 = 10;
  public static int field2 = 11;

  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);
    }
  }
}

class A {
  public static int ax;
  public int ay;

  static{
    System.out.println(
       "This is A’s static initializer.");
    ax=20;
  }

  A() {
    System.out.println(
       "This is A’s constructor.");
    ay=42;
  }
}
```

Fields and reflection (3)

- For example:

```
$ java ReflExample3 ReflExample3 field1
10


$ java ReflExample3 ReflExample3 field2
11


$ java ReflExample3 ReflExample3 field3
java.lang.NoSuchFieldException: field3


$ java ReflExample3 A ax
This is A’s static initializer.
20


$ java ReflExample3 A ay
This is A’s static initializer.
java.lang.NullPointerException
```

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

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

Methods and reflection (1)

- The reflection API represents Java methods as instances of a `java.lang.reflect.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 represented by `m`.

  Well, almost equivalent...

Methods and reflection (2)

```
public class ReflExample4 {
  public static int ctr=0;
  public void mth(Object a, Object b) {++ctr;}

  public static void main(String [] args) {
    try {
      ReflExample4 re4 = new ReflExample4();
      Class c = Class.forName("ReflExample4");
      java.lang.reflect.Method m;
      m = c.getMethod("mth",new Class[] {
        Class.forName("java.lang.Object"),
        Class.forName("java.lang.Object")
      });
      Object params[] = new Object[2];
      params[0] = params[1] = null;

      long st1=System.currentTimeMillis();
      for (int x=0;x<10000000;++x)
        m.invoke(re4,params);
      long speed1=System.currentTimeMillis()-st1;
      System.out.println("Reflection took "+speed1
        +" ms to do "+ctr+" iterations.");

      ctr=0;
      long st2=System.currentTimeMillis();
      for (int x=0;x<10000000;++x)
        re4.mth(null,null);
      long speed2=System.currentTimeMillis()-st2;
      System.out.println("Direct invocation took "
       +speed2+" ms to do "+ctr+" iterations.");

      System.out.println("Reflection took "+
        ((double)speed1/speed2)+" times longer!");
    } catch (Exception e) {System.out.println(e);}
  }
}
```

Methods and reflection (3)

With Sun's JDK1.4.2_05...

```
$ java ReflExample4

Reflection took 1842 ms to
do 10000000 iterations.

Direct invocation took 41 ms to
do 10000000 iterations.

Reflection took 44.926829268292686
times longer!
```

With Sun's JDK1.5.0...

```
$ java ReflExample4
Reflection took 3503 ms to
do 10000000 iterations.

Direct invocation took 144 ms to
do 10000000 iterations.

Reflection took 24.32638888888889
times longer!
```

Methods and reflection (4)

- 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 first 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 `boolean`) is to be passed then the value 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 (1)

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 structures 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 ‘serialization’ is 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 their simplest forms 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.
  - So what value do transient fields have in restored classes?

`java.io.Serializable` (1)

The methods defined in this interface 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 numbers;
- whether the complete structure should be saved/restored—e.g. if it refers to a temporary data structure used as a cache; or
- 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 sub-classes being serialized: it is used to initialize fields of the non-serializable super-classes.
- 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 java.io.IOException;

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

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

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

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

`java.io.Externalizable`

- `writeObject` and `readObject` are fiddley 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 java.io.IOException;

  void readExternal(ObjectInput in)
    throws java.io.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

1. What is meant by *reflection* or *introspection* in Java? Give an example of how and why these facilities might be useful.
2. What are the advantages and disadvantages, with respect to encapsulation, of using `Externalizable` rather than `Serializable` for customised serialization?

Lecture 6: Graphical interfaces

Previous lecture

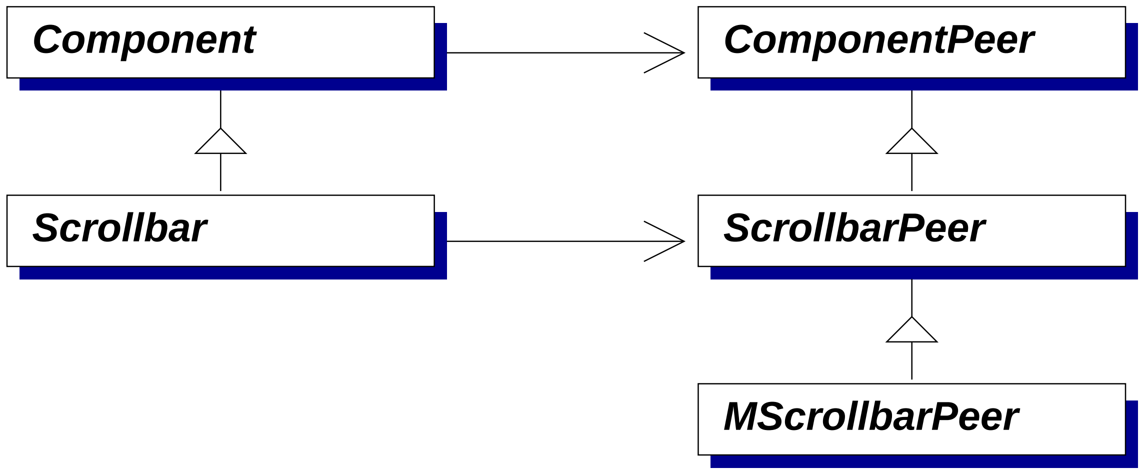
- Reflection: APIs to interrogate objects and class definitions at run-time, access fields, and call methods.
- Serialization: API to save and restore object state, e.g. to a file.

Overview of this lecture

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

Java Foundation Classes (JFC)

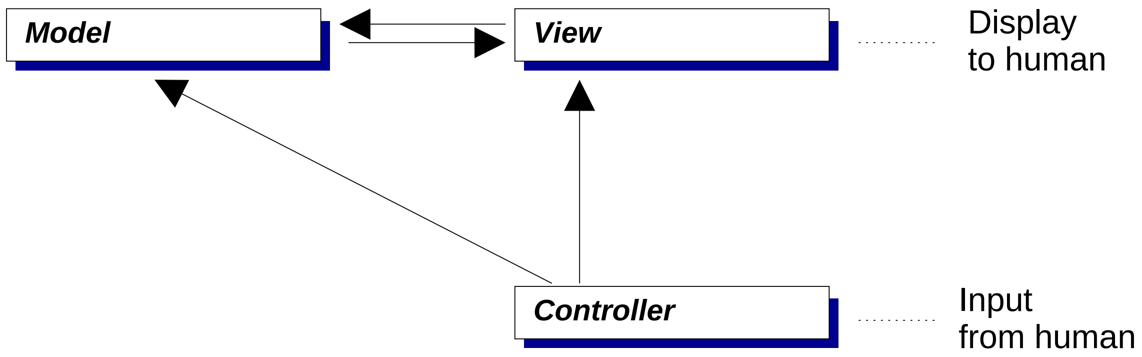
- We will cover the 'Swing' user interface components, part of JFC which also adds rendering APIs, drag 'n' drop, and more!
- The Abstract Windowing Toolkit (AWT) is the original set of GUI classes in Java: it is a lowest-common-denominator between the systems available at the time.
- AWT GUI components 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 (1)

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



This separates three aspects of the component:

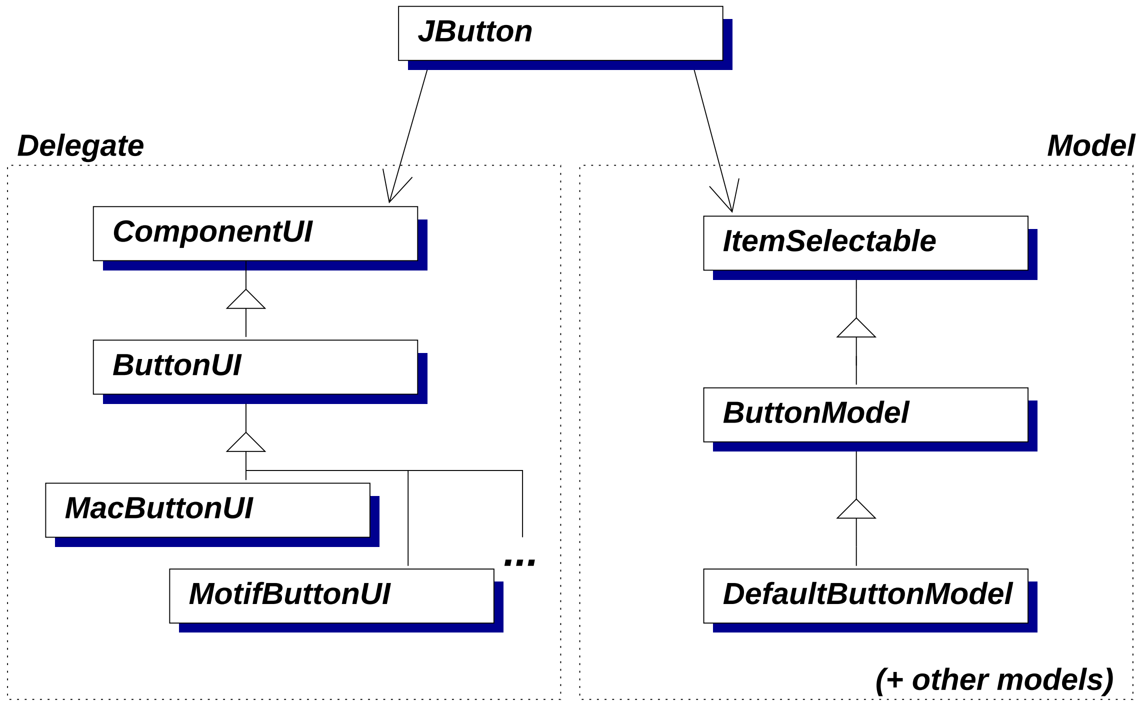
- the **view**, responsible for rendering it to the display;
- the **controller**, responsible for receiving input; and
- the **model**, being the underlying logical representation.

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 the current model.



Graphics

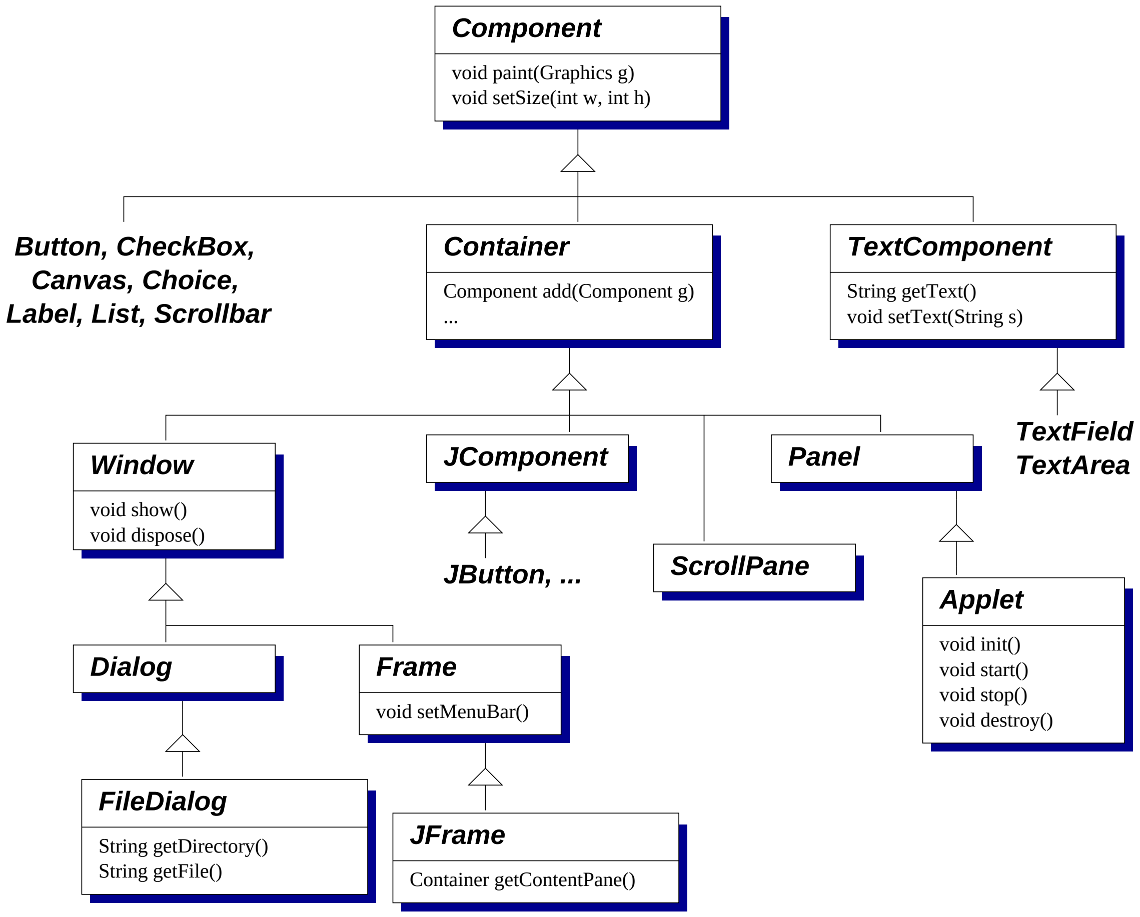
- Basic rendering primitives are available on instances of `Graphics`, e.g. using Java applets as in Part 1A:

  ```
  import java.awt.*;

  public class E1 extends java.applet.Applet {
    public void paint(java.awt.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 (1)



➤ 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, text boxes, etc.
- 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. by 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 (1)

- *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 a title bar or borders) and `javax.swing.JFrame` (a 'decorared' 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("Centre", new JButton("Centre"));
  }

  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 a fixed height or width (used to space out other components); and
  - **Glue** which expands/contracts if the window is resized and nothing else can change.

```
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 a BoxLayout than to use any of the other, older, layout managers (CardLayout, FlowLayout, GridLayout, GridBagLayout).

Receiving input (1)

- 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. For example, `MouseEvent` is used for mouse clicks; `KeyEvent` for keyboard input, etc.
- The system delivers events by invoking methods on a **listener**. For example, instances of `MouseListener` are used to receive `MouseEvents`:

  ```
  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; and
  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. They provide ‘no-op’ implementations of the associated interfaces.
  - Saves a lot of typing when we want to use only one of several actions provided by listener interface.
- The programmer just needs to override the method(s) 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 performed 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. the 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; and
  - `AccessibleValue`—represents numerical values (e.g. scroll bar positions).

Exercises (1)

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; and
   - spatial hierarchy.
2. Define example classes showing how to use the `JComboBox`, `JProgressBar`, `JDesktopPane`, and `JInternalFrame` components.
3. Describe the advantages and disadvantages of rendering components in Java (as with Swing) rather than using native components provided by the system hosting the JVM (as with AWT).

Exercises (2)

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?

5. Define a class that draws a fish at `(0,0)`. Implement this interface:

```
public interface AquaticSentience
  extends java.lang.Runnable {

  public void render(java.awt.Graphics g);

  public void setSize(int s);
  public int getSize();
  public void setSpeed(int s);
  public int getSpeed();
  public void setDirection(int d);
  public int getDirection();
  public void setDepthInScene(int d);
  public int getDepthInScene();
}
```

   Some of the best fish will be included in the tank in later lectures...

Lecture 7: Memory management

Previous lecture

- Graphical user interfaces
- Reminder of model-view-controller pattern
- Components and containers

Overview of this lecture

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

Garbage collection (1)

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

Advantages

- Frees the programmer from handling memory deallocation.
- More rapid code development.
- Fewer bugs than in similar programs written in languages not supporting run-time garbage collection.
- Increased run-time performance "for free" because garbage collection can perform tidy-up during system idle time.
- Garbage collection is often unnecessary for short-lived programs.
- Increased stability for very-long-running programs, e.g. server daemons.
- In complex, multi-threaded systems, GC may have indirect benefits over explicit deallocation.
  - No need to agree which module is responsible for deallocation.
  - Aids sharing of data structures rather than having each module take a private copy.
- Makes possible some data structures and algorithms that otherwise are not possible, or are very difficult to implement.
  - Lock-free data structures (see later!)

Garbage collection (2)

Disadvantages

- Increased overhead at execution time.
  - e.g. in some implementations, assignment requires counter manipulation, and perhaps even mutex operations!
- Programmer never thinks about memory allocation—not always a good thing!
- Less control over the memory footprint of the process at run-time.
  - Harder or impossible to optimise locations and sizes of data structures in memory to take advantage of CPU data/instruction/page table caches.
- Might be more threads at run-time.
- Antithesis of real-time execution.

Garbage collection (3)

- The code below will run forever without any problems and without requiring additional storage space for each iteration of the loop.

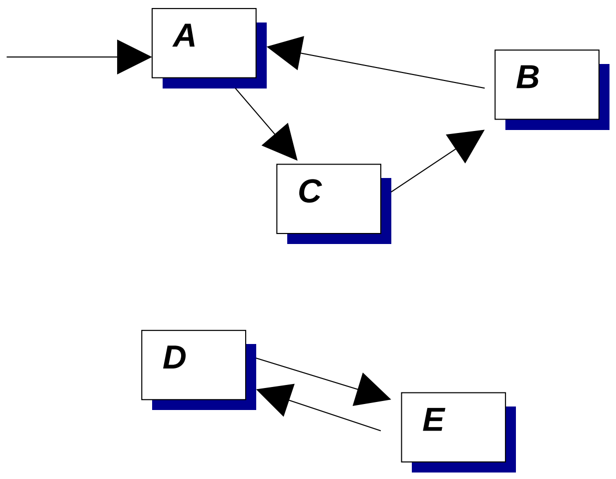
```
class Loop {
  public static void main(String args[]) {
    while (true) {
      int x[] = new int[42];
    }
  }
}
```

- The **garbage collector** is responsible for identifying when storage space can be reclaimed.

```
$ java -verbose:gc Loop
[GC 512K->94K(1984K), 0.0051620 secs]
[GC 606K->94K(1984K), 0.0020410 secs]
[GC 606K->94K(1984K), 0.0013290 secs]
[GC 606K->94K(1984K), 0.0006350 secs]
[GC 606K->94K(1984K), 0.0006550 secs]
[GC 606K->94K(1984K), 0.0006440 secs]
[GC 606K->94K(1984K), 0.0006870 secs]
[GC 606K->94K(1984K), 0.0006490 secs]
[GC 606K->94K(1984K), 0.0006480 secs]
[GC 606K->94K(1984K), 0.0006840 secs]
```

Garbage collection (4)

- There are lots of different techniques that might be used to implement the garbage collector. See Part 1B *DS&A* and *Compiler Construction* for details of algorithms like:
  - buddy systems;
  - mark and sweep;
  - reapers;
  - incremental garbage collectors;
  - ...
- The JVM guarantees that storage space will not be reclaimed while an object remains **reachable**, defined as being 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*; or
  - is referred to by another reachable object.



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

Garbage collection (5)

- Early languages with GC had reputations for being slow and for adding annoying pauses to an application's execution. Modern collectors do better thanks to features of newer GC techniques:

  Generational collection: "most objects die young" $\longrightarrow$ keep a small *young generation* which can be collected quickly and frequently.

  Parallel collection: multiple processors work on GC at the same time $\longrightarrow$ 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`.

Finalizers (1)

When the GC detects that an object is otherwise unreachable (e.g. `D` and `E` two slides previous) 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 down 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 (beautifully) 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 might run concurrently with anything else and make sure that they do not deadlock (Lecture 12) or enter endless loops.

Reference objects (1)

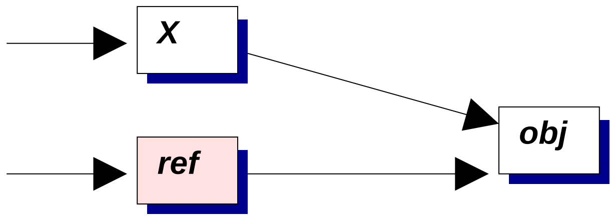
- **Reference objects** provide a more general mechanism for:
  - scheduling clean-up actions when objects become unreachable via ordinary references;
  - managing caches in which the presence of an object in the cache should not prevent its garbage collection; or
  - accessing temporary objects which 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());
  }
}
```

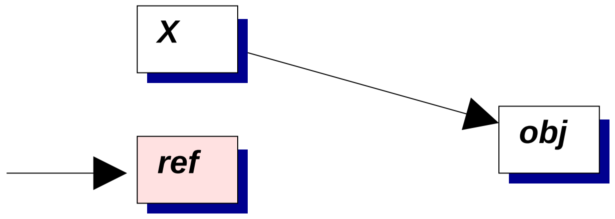
Reference objects (2)

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()`.

Disadvantage

- Traversal requires extra calls to `get()`.

Advantage

- Reference objects are simpler conceptually than separate 'weak reference types' to the language.

Reference objects (3)

A reference object can be assoicated 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 assoicated 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; and
- `remove()` attempts to remove a reference object, blocking indefinitely.

Reference objects (4)

Compiling RefExample.java...

```
$ javac RefExample.java

Note: RefExample.java uses unchecked or unsafe
operations.

Note: Recompile with -Xlint:unchecked for
details.
```

```
$ javac -Xlint RefExample.java

RefExample.java:6: warning: [unchecked] unchecked
call to WeakReference(T) as a member of the raw
type java.lang.ref.WeakReference

    Reference ref = new WeakReference(obj);
                    ^
1 warning
```

```
$ java RefExample
ref: java.lang.ref.WeakReference@10b62c9
[I@82ba41
```

Reference objects (5)

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

1. The ‘HotSpot’ JVM has a command-line option to control the maximum size of the heap: `-Xmx`. 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 garbage collection.
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?
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 might be executed; and

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

Lecture 8: Miscellany

Previous lecture

- Garbage collection
- Finalizers
- Reference objects

Overview of this lecture

- Native methods
- Class loaders

Native methods (1)

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; or
- 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; and
- 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 (but JNI is).

Native methods (2)

Here's how it's done...

```
 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` and `HelloWorld.h`

Native methods (3)

So what did we get?

```
 1  /* DO NOT EDIT THIS FILE - it is machine gener
 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
```

➤ Lines 15–17 declare the function we must define for the `displayHelloWorld` method.

Native methods (4)

- Now we write `HelloWorldImp.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=/home/jkf/java/jdk1.5.0/include

  $ gcc HelloWorldImp.c -I$JINCLUDE \
        -I$JINCLUDE/linux -shared \
        -fpic -o libhello.so

  $ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH

  $ java HelloWorld
  Hello World!
  ```

- 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 the function pointers for these operations, eg.
  - `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 (1)

- 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 call `c.defineClass(name, b, off, len)` to create a new `Class` object from the bytes at `off` $\rightarrow$ `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 super-class, 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 is no resource management at all.

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

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

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

Exercises (1)

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.
2. Implement a simple class loader that prints a list of the class names for which it is requested.
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. Compare versions 1.4.2_05 and 1.5.0 of Sun's JDK.

Exercises (2)

4. A class `C1` loaded and defined by class loader `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.

Lecture 9: Threads

Now in Part 3 of this course.

Previous section
Further Java

- Reflection and serialization
- Memory management
- Swing and AWT

Overview of this section
Concurrency Issues

- Multi-threaded programming
- Concurrent data structures

Concurrency (1)

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.
- Can benefit from multi-processor hardware
  - e.g. the new High-Performance Computing Facility (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 process has a **process control block** (PCB) holding:

- identification (e.g. PID, UID, GID);
- memory management information;
- accounting information; and
- (references to) one or more TCB...

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; and
- 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 counters / sequences
- Alternative abstractions
  - Monitors
  - Active objects

Creating threads in Java (1)

- 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.
  - `run()` provides the code that the thread will execute while it is on the CPU.
  - The thread terminates when `run()` returns.
  - It is common to have daemonic threads in a loop: `while (!done) <code>`

```
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.
- The two calls to `new` instantiate the class to create two objects representing the two threads that will be executed.
- The calls to `start()` actually start the two threads executing.
- The program continues to execute until all ordinary threads have finished, even after the `main` method has completed.

```
$ java MyThread
Hello from Thread[Thread-0,5,main]
Hello from Thread[Thread-1,5,main]
Hello from Thread[Thread-0,5,main]
Hello from Thread[Thread-1,5,main]
...
```

- 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.

```
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, the `MyCode` class defines the code that the new threads will execute.
- The two calls to `new` instantiate two `Thread` objects, passing a reference to an instance of `MyCode` to them as their *target*s.
- The two calls to `start()` set the 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.
  - It doesn't consume the single opportunity to sub-class a parent class.
- Generally, the 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 situations a thread is interrupted immediately if it is blocked—e.g. `sleep` may throw `InterruptedException`. For example:

```
class Example {
  public static void main(String [] args) {
    Thread t = new Thread() {
      public void run() {
        try {
          do {
            Thread.sleep(1000); // 1s sleep
          } while (true);
        } catch (InterruptedException ie) {
          // Interrupted: better exit
        }
      }
    };
    t.start();
    t.interrupt();
  }
}
```

- If the thread didn't block then the `while (true)` could perhaps be

```
while (!isInterrupted());
```

Join

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

```
class Example {
  public void startThread(void)
    throws InterruptedException
  {
    Thread t = new Thread() {
      public void run() {
        System.out.println("Hello world!");
      }
    };

    t.start();
    t.join(0);
  }
}
```

- The call to `join` waits for the thread started on the previous line to finish. The parameter specifies a time in milliseconds (0 $\longrightarrow$ wait forever).
- The `throws` clause on `startThread` 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—not 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 higher priorities will run in preference to those with lower priorities.
- **To write correct, portable code it is therefore important to think about what the JVM is *guaranteed* to do—not just what it does on one system. Different behaviour might occur at different nodes within a distributed system!**

The `volatile` modifier (1)

```
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 might 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.
  - This is common when the body of the loop is short—no need for compiler to re-use the register containing `signal`.
  - Commonly seen in embedded C/Java systems.
- Such behaviour is valid and might help performance.

The `volatile` modifier (2)

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

```
static volatile 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 it is permitted for the JVM to re-use a value that is held in a register 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 acqiures; or
- the field should be `volatile`.

As we will see, the first condition is satisfied by the usual use of `synchronized` methods (or classes) $\longrightarrow$ `volatile` is rarely seen in practice.

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

Exercises

1. Describe the facilities in Java for creating multiple threads of execution.
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 that perform some kind of experiment.)
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.
4. Examine the behaviour that one or more JVMs provide for the following aspects of thread management:

   (i) whether scheduling is preemptive;

   (ii) whether the highest-priority runnable-thread is guaranteed to run; and

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

Lecture 10: Mutual exclusion

Previous lecture

- Creating and 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, it is usually the case that this cannot be checked automatically by compilers or tools (although some exist to help).
- It is 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.
- We can identify **consistent** object states in which all invariants are satisfied.
- ...and aim that each of the operations available on the system keeps it in a consistent state.
- Therefore many of the problems that we will see come down to deciding when different threads can be allowed access to objects in various ways.

Liveness

As well as safety, we would 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 (1)

- 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 field accesses are **atomic** in Java (and many other languages)—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 those cases.

Shared data (2)

- This is an example of a **race condition**: the result depends on the uncontrolled interleaving of the threads' executions.
- We need some way of controlling how threads are executed when accessing shared data.
- The basic notion is of **critical regions**: parts of the 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 (1)

- 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. Each acts like the 'busy' flag on the previous slide except:
  - There is no need to spin while waiting for it—the thread is blocked.
  - The race condition between the `while` loop and `busy=true;` 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 called thread 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.
  - ...and 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 the 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 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.

```
void methodA(Object x) {
  synchronized (x) {
    System.out.println("1");
  }

  ...

  synchronized (x) {
    System.out.println("2");
  }
}
```

- The first `synchronized` region locks the mutex associated with the object to which `x` refers, performs the `println` operation, and then releases the lock.
- Before entering the second region, the mutex must be re-acquired.

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

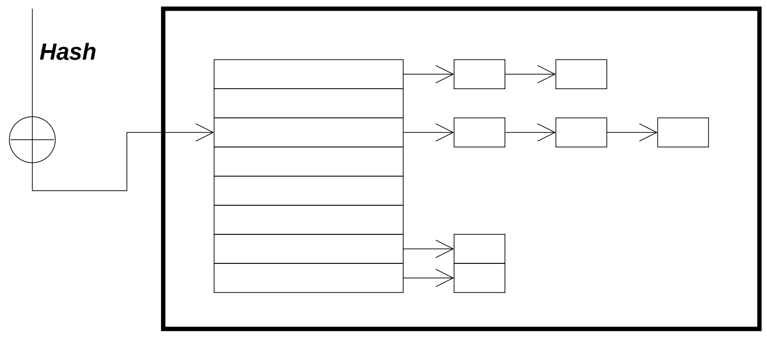
What about exceptions and errors?

- What if an exception is thrown inside a `synchronized` region or a `synchronized` method?
- If the exception is not caught inside the region/method, then the flow of execution leaves the synchronized region.
- The JVM will release the mutex automatically *before* executing the `catch` block.
  - This helps prevent deadlock caused by accidentally not releasing a mutex.
  - But sometimes we need to exercise a little caution...

```
void screwItUp(Bank college) {
  try {
    synchronized (college) {
      college.credit(fees);
    }
  } catch (OutOfMoneyException oome) {
    System.out.println("Credit to "+
      college+" failed!");
    // oops... that access on 'college'
    //         wasn't thread-safe!
  }
}
```

Compound data structures (1)

- 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:



Advantages

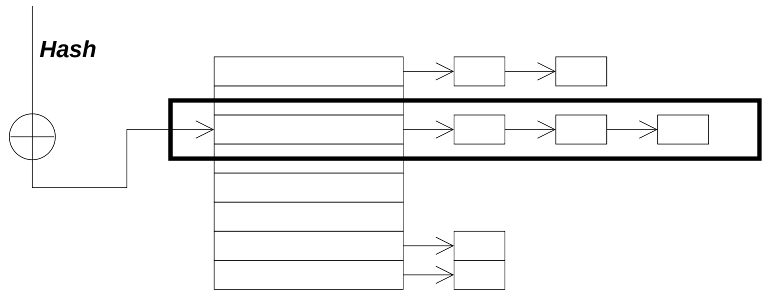
- Easy to implement
- "Obviously correct"
- Good performance under light load: only one lock to acquire/release per operation.

Disadvantage

- 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:



Advantage

- Operations using different buckets can proceed concurrently.

Disadvantage

- Harder to implement—consider resizing the hashtable...

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.
- Having many locks does not automatically imply better concurrency.
- Deadlock problems...

Exercises

1. Describe how the mutual-exclusion locks provided by the `synchronized` keyword can be used to control access to shared data structures.
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?
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? Does your JVM guarantee them?

Lecture 11: Deadlock

Previous lecture

- Safety and liveness requirements
- Mutual exclusion locks

Overview of this lecture

- Deadlock
- Automatic detection
- Avoidance

Deadlock (1)

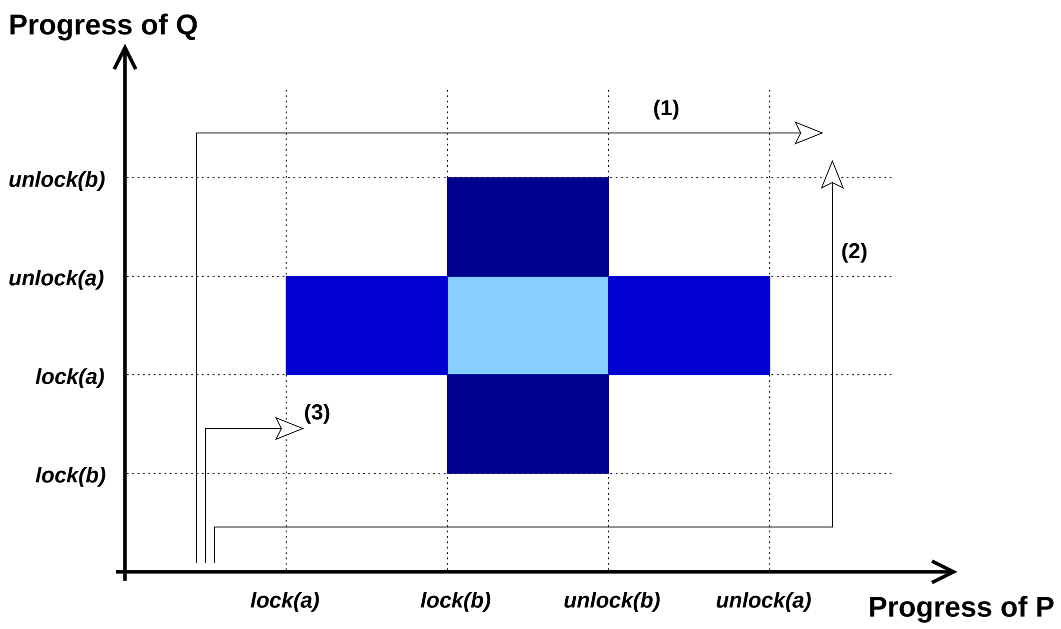
Suppose that `a` and `b` refer to two different shared objects,

```
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 might acquire both locks, then release them and thus allow P to continue.
- But, if P locks `a` and Q locks `b` then neither thread can continue: they are each *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:

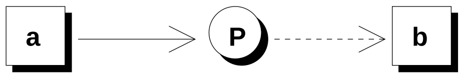
1. A resource request can be refused—e.g. a thread cannot acquire a mutual-exclusion lock because it is already held by another thread.
2. Resources are held while waiting—e.g. while a thread blocks waiting for a lock it does not have to release any others that it holds.
3. No preemption of resources is permitted—e.g. once a thread acquires a lock then it is up to that thread to choose when to release it, it cannot be taken away from the thread.
4. Circular wait—a cycle of threads exists 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 (they are static properties of the language), and so the existence of a circular wait leads to deadlock.

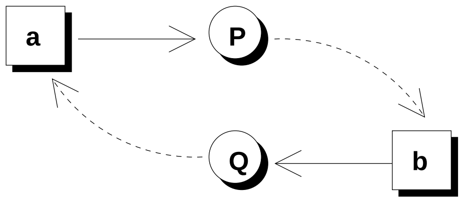
Object allocation graphs

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

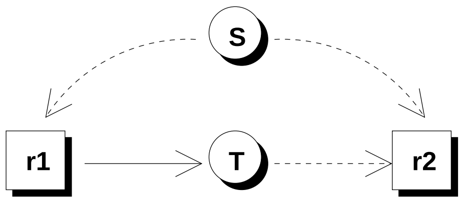
`a` is held by thread `P` and `P` is requesting object `b`:



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



Should `r2` be allocated to `S` or `T`?



Deadlock detection (1)

Deadlock can be detected by looking for cycles in object allocation graphs (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 no such row exists.
2. Set $W = W + A_i$, mark row i, and repeat.

This identifies when deadlock *has occurred*—we might be interested in other properties such as whether deadlock is:

- inevitable (must happen in all possible execution paths); or
- possible (might happen in some paths).

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

➤ Nothing more can be done: threads 1 and 2 are deadlocked.

Deadlock avoidance (1)

A conservative approach:

- Require that each process identifies the maximum set of resources that it might ever lock, C_{ij}.
- When a 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.

Advantage

- This does avoid deadlock—might be preferable to deadlock recovery.

Disadvantages

- Need to know maximum requests.
- Run-time overhead.
- What if there are no safe states?
- Objects are usually instantiated dynamically...

Deadlock avoidance (2)

It is 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).
  - Hardware/software **transactional memories**...

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 two 'bank account' objects in order of increasing account number.

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

Priority inversion (1)

Another liveness problem in priority-based systems:

- Consider low, medium, and high priority threads called P_{low}, P_{med}, and P_{high} respectively.
  1. First P_{low} starts, and acquires a lock on object a.
  2. Then the other two processes start.
  3. P_{high} runs since it has the highest scheduling priority, tries to lock a, and blocks.
  4. Then P_{med} gets to run, thus preventing P_{low} from releasing a, and hence P_{high} from running.
- Usual solution is **priority inheritance**:
  - associate with every lock the priority p of the highest priority process waiting for it; then
  - temporarily boost the priority of the *holder* of the lock up to p.
  - We can use handoff scheduling to implement this.
- Windows 2000 "solution": priority boosts
  - checks if $\exists$ a thread in the ready-to-run state but not run for $\geq$ 300 ticks.
  - if so, double the on-CPU time quantum and boost 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; and
  - *push-through blocking* of a thread at one priority by an originally-lower-priority thread that has inherited a higher priority.
- A thread P_{high} can be blocked by each lower priority thread P_{low} for at most one of P_{low}'s critical sections.
- A thread P_{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 $\geq P_{high}$.

  This can give an upper bound on the total blocking delay that a thread encounters, but
- chains of blocking may limit the bounded and practical performance: the former is a particular problem for real-time systems; and
- remember: does not prevent deadlock.

Exercises (1)

1. In the *dining philosophers* problem, five philosophers spend their time alternately *thinking* and *eating*. They 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 might get hungry and try to pick up the two closest forks. A philosopher may only pick up one fork at a time. It is a common 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?

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)

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.
4. One way to avoid deadlock is for a thread to simultaneously acquire all of the locks that 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()`; and
   3. releases all of the locks initially acquired.

Exercises (3)

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

interface LockManagerIfc {
  Object doWithLocks(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 later to examine how to provide that mechanism.

Lecture 12: Condition synchronization

Previous lecture

- Deadlock
- Ordered acquisition
- Priority inversion and inheritance

Overview of this lecture

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

Limitations of mutexes (1)

- Suppose we want a one-cell buffer with a `putValue` operation (store something if the cell is empty) and a `removeValue` operation (read something if there is anything in the cell).

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

  public synchronized int removeValue() {
    if (full) {
      full = false;
      return value;
    } else {
      /* ??? */
    }
  }

  ...
}
```

- What can we write in place of "`/* ??? */`" to finish the code?

Limitations of mutexes (2)

➤ We could keep testing `full`—i.e. implement a 'spin lock'...

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

But this is...

1. incorrect: if multiple threads try to remove values then they might each see `full` false at Line 6 and independently execute 7–10;
2. inefficient: threads consume CPU time while waiting $\longrightarrow$ this might impede a thread about to put a value into the cell; and
3. incorrect: `full` needs to be `volatile` anyway!

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 priorities?
- 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      }
10    }
```

- 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—the method is `synchronized`—to allow another thread to put items into it; and
  - 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 semantics mean that `full` would be evaluated only once!
  - we would 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 causing 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; and
  - a *cv.CVNotify(m)* operation that wakes up (one? all?) threads blocked on `cv`.
- Such operations would be more cumbersome in this simple example than a general `wait_until` primitive:

```
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 (1)

- Java doesn't (currently) provide individual condition variables in this way.
- Instead, each object `o` has an associated condition variable which is accessed by:
  - `o.wait()`
  - `o.notify()`
  - `o.notifyAll()`
- Calling `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 the caller may *only* use `o.wait()` when holding that lock (otherwise an `IllegalMonitorStateException` is thrown).
- `o.notify()` unblocks exactly one thread (if any are waiting), otherwise it does nothing—no wake-up waiting is left.
- `o.notifyAll()` unblocks all waiting threads.

Condition variables in Java (2)

```java
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.
  - Think about whether I really need the `while` loop around `wait()` (answer in 4 slides' time...).
- 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, Line 22 notifies waiting threads.

An `InterruptedException` will be thrown if 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 */
}
/* did we get here because wait() succeeded
   or were we interrupted?    */
```

Condition variables in Java (4.1)

Is this code cunning or broken?

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

  public synchronized int removeValue()
    throws InterruptedException
  {
    while (!full) {
      wait();
      /* Should a put’er or a remove’er have
         been woken up?  */
      if (!full) {
        // pass on the nofitication,
        // hopefully to a put’er.
        notify();
      }
    }

    full = false;
    notify();  // surely waking one thread
               // is better than waking all?
    return value;
  }
```

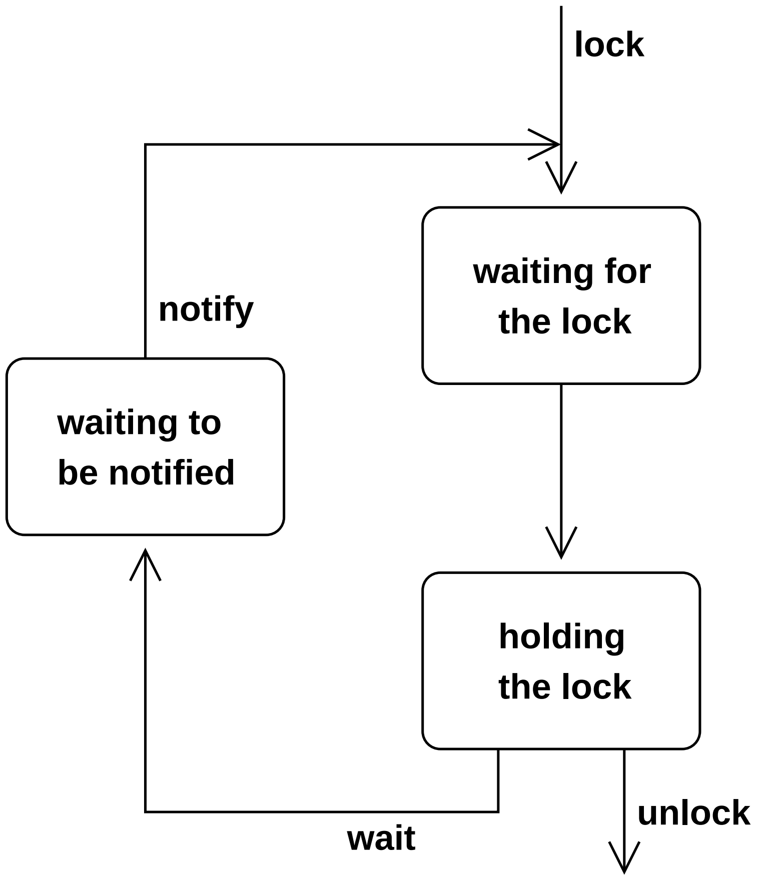
Condition variables in Java (4.2)

Is this code cunning or broken (continued)?

```
  public synchronized void putValue(int v)
    throws InterruptedException
  {
    while (full) {
      wait();
      /* Should a put’er or a remove’er have
         been woken up?  */
      if (full) {
        // pass on the nofitication,
        // hopefully to a remove’er.
        notify();
      }
    }

    full = true;
    value = v;
    notify();  // surely waking one thread
               // is better than waking all?
  }
}
```

Condition variables in Java (5)

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



- It might have entered a `synchronized` region for an object and found that the associated mutual exclusion lock is already held.
- It might 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 and 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 invocation of `wait()` with a `while` loop;
  - if a 'removing' thread is notified when there is no work for it, 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 rather than another thread that is blocked in `removeValue`—we cannot control which thread will be notified by the `notify()` call.

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

Suspending threads

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

```
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 might change between executing Lines 2 and 3 $\longrightarrow$ a lost wake-up problem!

`suspend()` **should never be used: even if the program does not explicitly take out locks the JVM might use locks in its implementation.**

Exercises

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

(iii) If two threads are run on the **same** instance of class `Example`, would you expect the value of `count` to increase more rapidly or less rapidly than a single thread running `while (true) ++count;`? Why? Does it make any difference if the machine being used has several processors instead of just one?

(iv) Compared to a uni-processor, approximately how rapidly would you expect `count` to increase on a dual-processor machine running the code if the synchronization on `o` was removed from the `while` loop entirely?

Lecture 13: Worked examples

Previous lecture

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

Overview of this lecture

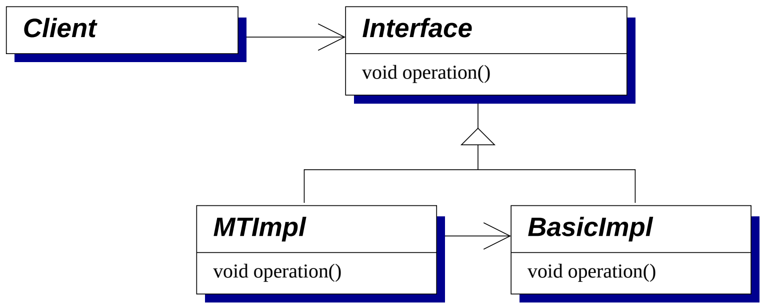
- Further examples of how to use these facilities
- Common design features

Design (1)

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 $\Rightarrow$ the multiple-reader, single-writer problem.

- 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?

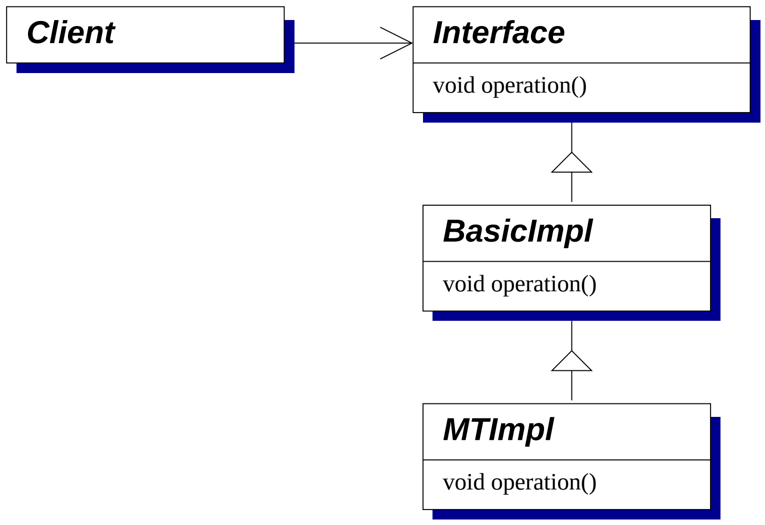
One option is based on delegation and the Adapter pattern:



- `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:



Advantages

- Sub-classes enforce encapsulation and mean that only one instance is needed.
- Delegation may be easier; just use `super.operation()`.

Disadvantages

- Separate sub-classes are needed for each implementation of `Interface`.
- Composition of wrappers is fixed at compile time.

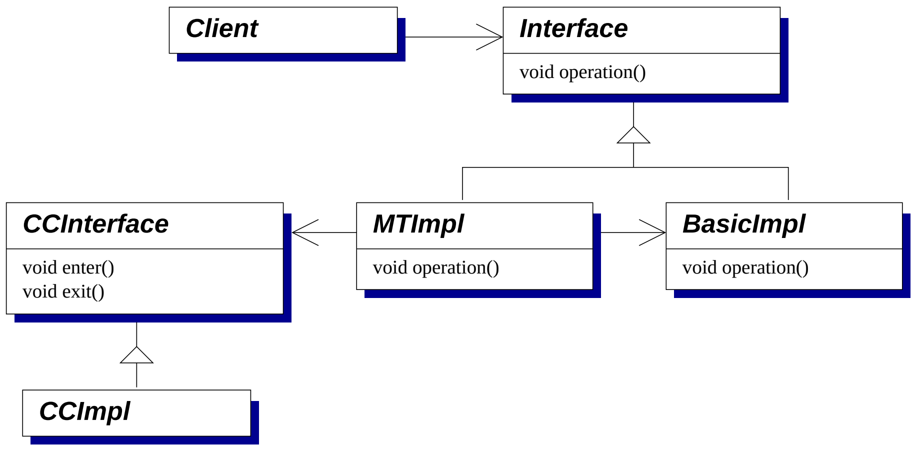
Design (3)

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 ordinay 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 might be used to instantiate and assemble these objects.

Advantages

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

Multiple readers, single writer (1)

- 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();
  }
}
```

Advantage

- 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.

Disadvantage

- `notifyAll()` may cause too many threads to be woken—the code is safe but might be inefficient.

Is that inefficiency likely to be a problem?

Could `notify()` be used instead?

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      while ((numWriters>0)||(waitingWriters>0))
 9        wait();
10      numReaders++;
11    }
12
13    synchronized void enterWriter()
14    throws InterruptedException {
15      waitingWriters++;
16      while ((numWriters>0)||(numReaders>0))
17        wait();
18      waitingWriters--;
19      numWriters++;
20    }
21  }
```

- What happens to instances of `PrioritizedWriter` if the code is interrupted at line 17?

First-come first-served ordering (1)

- 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)

Advantages

- The implementation is simple!

Disadvantages

- 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?

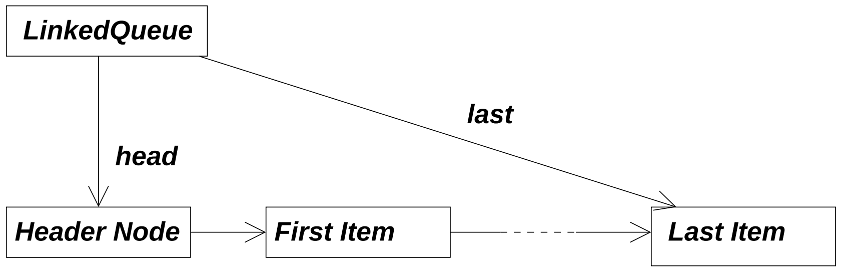
Resolving these issues in an effective way depends on the context in which the class is being used, e.g.

- *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 `longs` ever overflow here?

Splitting locks (1)

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,
  - e.g. in a list, 'push on tail' is usually safe to run in parallel with 'pop from head'.



We will 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 the Part II course "*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
        // and update last
      }
    }
  }

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

Exercises

1. In the `PrioritizedWriters` example the `waitingWriters` 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.
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 version?
3. Update the `FCFSImpl` class so that the lock can be held *recursively*—a thread already holding the lock can make subsequent calls to `enter()` without blocking. The lock is released only when a matched number of (properly-nested) calls to `exit()` have been made.
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`?

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 condition variables
- 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 will still briefly look at these other primitives.

Implementing mutexes (1)

- 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 the Part II course "*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 is 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 `resume`s it. To avoid lost wake-up problems, either outstanding `resume` operations must be remembered:

```
1     Thread A, LOCK():         Thread B, UNLOCK():
2     see that lock is held
3     add A to queue
4                               release lock
5                               take A from queue
6                               resume A
7     suspend A
```

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

Implementing condvars (1)

- 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; and
  - 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; and
  - an additional mutex `cvLock` that is used to give the atomicity required by `CVWait()`.

Implementing condvars (2)

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

```
1    Acquire mutex cv.cvLock
2      Add the current thread to cv.queue
3      Release mutex m
4    Release mutex cv.cvLock
5    Suspend current thread
6    Re-acquire mutex m
```

- `cv.CVNotify()` proceeds by:

```
1    Acquire mutex cv.cvLock
2      Remove one thread from cv.queue
3      Resume that thread
4    Release mutex cv.cvLock
```

- Again, it is important to avoid lost wake-up problems—typically by remembering resumptions.
- A real implementation is more complex—e.g. in Java it is 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 (1)

- 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);
  }

  void acquire() {
    sem.P();
  }

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

- The mutex is considered unlocked when the value is 1 (it is initialized unlocked)...
- ...and is locked when the value is 0 or less.
- How does this mutex differ from a Java-style one?

Programming with semaphores (2)

```
class CondVar {
  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; and
  - `await(i)`—wait until the value is greater than or equal to `i`.
- A sequencer is again represented by a positive integer, 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 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.

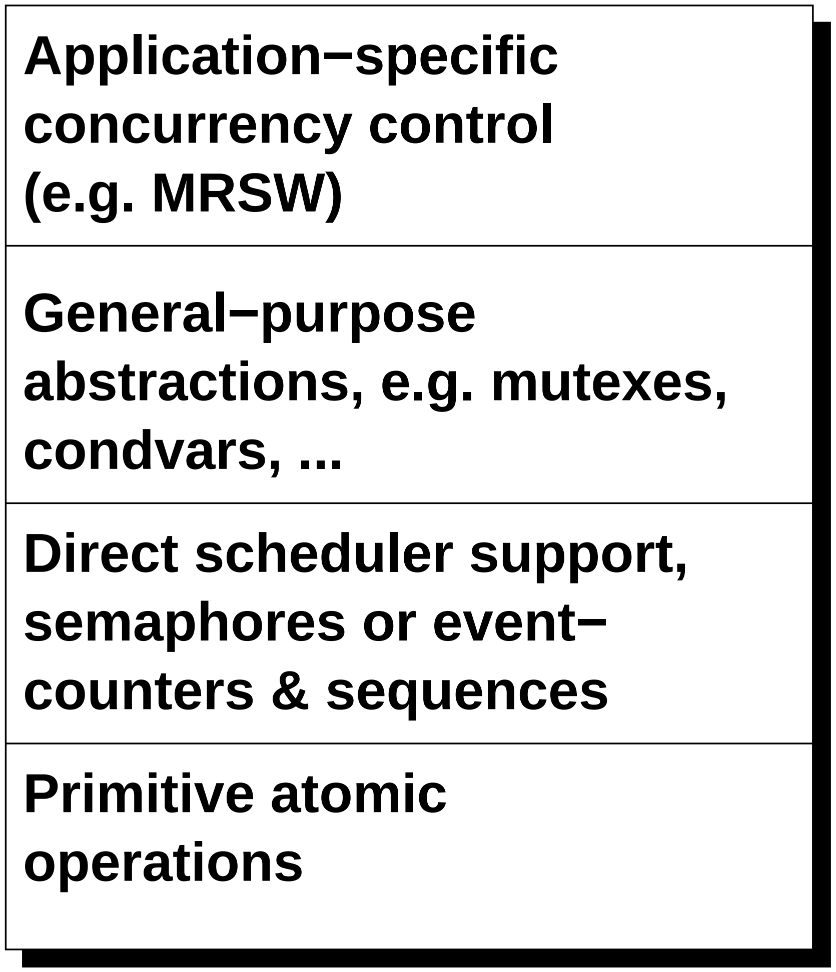
Mutexes without hardware support

- 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)
- The '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++) {                       1
                while (taking[j]) {}
                while ((ticket[j] != 0) &&
                          (ticket[j] <= ticket[i])) {}
              }

              for (j=i; j<n; j++) {                       2
                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 (e.g. MRSW)
General−purpose abstractions, e.g. mutexes, condvars, ...
Direct scheduler support, semaphores or event−counters & sequences
Primitive atomic operations

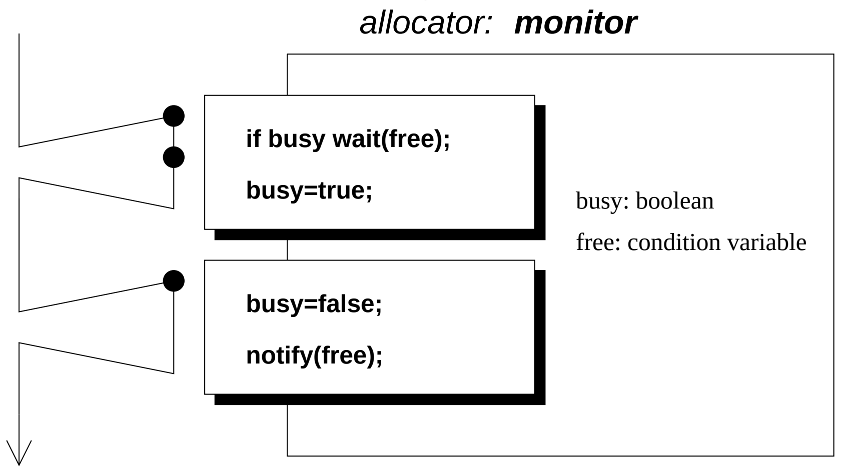
The details of exactly what is implemented where vary greatly between systems, e.g.

- 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 (1)

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 programming language, it is 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 (1)

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.
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.
3. A *binary semaphore* is a simplified version of the *counting semaphore* from the slides. Rather than an integer count value it has a binary flag. P_b blocks (if necessary) until the flag is set and then atomically clears it. V_b sets the flag (atomically unblocking one thread, if any blocked in 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-set (`TAS`) machine instructions.
   - (ii) In pseudo-code, show how a counting semaphore can be built using binary semaphores. Your solution might need more than one binary semaphore and another field to hold the count value.

Exercises (2)

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.

Lecture 15: Distributed Systems

Now in Part 4 of this course.

Previous section
Concurrency Issues

- Multi-threaded programming
- Concurrent data structures
- Communication between threads

Overview of this section
Distributed Systems

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

Communication between processes (1)

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—because all the parties share a single copy of the data.
- 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 (precisely) one user.

Communication between processes (2)

Introducing separate address spaces means that the 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 vs little-endian architectures.
- 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 (1)

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

- Components execute in parallel
  - maybe on machines with very different performances
- 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 different orders from one another

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, Introduction to Security) and algorithms (Part II: Distributed Systems, Advanced Systems Topics).

We will look at two different communication mechanisms:

- Remote method invocation

  Advantage

  - Remote invocations look substantially like local calls: many low-level details are abstracted.

  Disadvantages

  - 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

  Advantage

  - A 'lowest-common-denominator': the TCP and UDP protocols are available on almost all platforms.

  Disadvantage

  - 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.

Advantage

- Easy to use in Java-based systems

Disadvantage

- 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 lanaguage** (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 programming language.
- An IDL compiler generates per-language *stubs* (contrast with the `rmic` tool that only generates stubs for the JVM).

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

Interface definition: OMG IDL (1)

We will 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(s) 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;
  string 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, e.g.:

```
attribute long value;
```

can be transformed to:

```
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`
`exception`	`class`
`readonly attribute`	read-accessor method
`attribute`	read-/write-accessor methods
`operation`	method

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

Interface definition: .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 versus VB, C-sharp, 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; or
- access them from static fields.

When communicating between address spaces we need other mechanisms to establish:

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

Late binding of names (e.g. `magic.voidstar.org.uk`) to addresses (`193.117.99.117`) 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 (1)

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 ID (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!

Advantage

- Allocation is easy if N is large—just allocate successive integers

Disadvantages

- 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.

Advantages

- 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/jkf21/` then perhaps I'm likely to lookup other names in that same directory).

Disadvantage

- 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 stay on which CPU the process should run, or which user created it.

An impure name contains information about the object—e.g. email to `jkf21@cam.ac.uk` will always be sent to a mail server in the University.

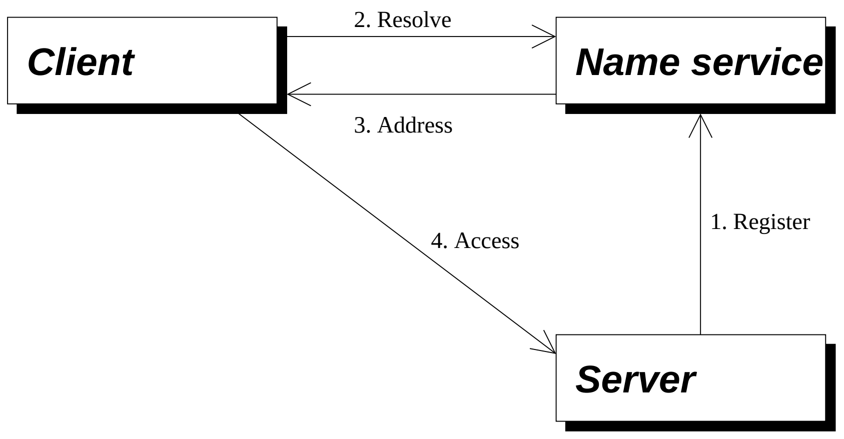
➤ Are DNS names, e.g. `www.cl.cam.ac.uk`, pure or impure?

➤ Are IPv4 addresses, e.g. `192.168.1.1`, 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 centralised allocation.

In other schemes, pure names may contain location *hints*. Crucially, impure names prevent the identified objects from changing in some way (usually moving) without renaming.

Name services (1)



- 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 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 Laboratory.
- **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 is accessible) and read performance (lookups can be made to the nearest replica).
- be *distributed*, e.g. separate systems might 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 (1)

In a distributed system, access control is needed to:

- control communications 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 it is attached to.
  - how does the controller know that the temperature readings came from the intended probe?
  - how does the probe know that it is being queried by the intended controller?
- control operations that one component does on behalf of users,
  - e.g. a file server might 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 in Part II.

Security (2)

We will 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

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`.
2. Suppose that two people are communicating by sending and receiving mobile 'phone text messages (SMS). 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.
3. Convert the `POS` module definition from OMG IDL into a Java interface that provides similar RMI functionality.
4. Suppose that frequent updates are made to part of a hierachical 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 (1)

Two basic network protocols are available in Java:

- datagram-based **UDP**, the user datagram protocol; and
- stream-based **TCP**, the transport control protocol.

UDP sockets provide *unreliable datagram-based* communication that is subject to:

➤ **Loss**: datagrams that are sent might never be received.

➤ **Duplication**: the same datagram might be received several times.

➤ **Re-ordering**: datagrams are forwarded separately within the network and might arrive out of order.

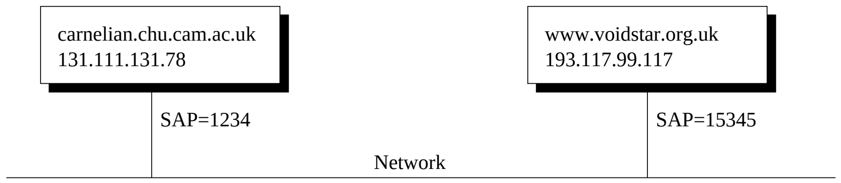
What is provided:

➤ A checksum is used to guard against corruption (corrupt data is discarded by the protocol implementation and the application perceives it as packet loss).

➤ The **framing** within datagrams is preserved—a UDP datagram might 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 is not accessible through most common APIs, 1..1023 reserved for privileged use and well-known services).



Naming is handled by:

- using the DNS to map textual names to IP addresses, `InetAddress.getByName("www.voidstar.org.uk")`
- using 'well-known' port numbers for particular UDP services which wish to be accessible to clients (see `/etc/services` on a UNIX system).

As far as we're concerned here, the network acts as a 'magic cloud' that conveys datagrams—see Digital Communications I for *layering* in general and examples of how UDP is implemented over IP, and how IP is implemented over (e.g.) ethernet by encapsulation and/or fragmentation and reassembly.

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 machine. This identifies the *local* endpoint for the communication.
- Datagrams are represented in Java as instances of `java.net.DatagramPacket`. The most elaborate constructor is:

  ```
  DatagramPacket(byte buf[], int length,
    InetAddress address, int port)
  ```

  ... and specifies the data to be sent (`length` bytes from within `buf`) and the destination address and port for this packet.
- `MulticastSocket` defines a UDP socket capable of receiving multicast packets. The constructor specifies the port number and the methods

  ```
  joinGroup (InetAddress g);
  leaveGroup(InetAddress g);
  ```

  ... join and leave a specified group operating on that port.
- Multicast group addresses are a designated subnet of the IPv4 address space. Allocation policies are still in flux ⇒ check the local policy before using multicast!

UDP example (1)

```
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 in the network;
- Flow control (preventing the receiver from being swamped with too much data);
- Congestion control (preventing the network from being overwhelmed); and
- 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); and
- 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 (1)

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 being thrown).
- `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 (or the wait is interrupted). When it returns, an instance of `Socket` representing the connection is passed to the caller. Methods on `Socket` provide a mechanism to discover the address and port number of the remote process.

  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 (1)

```
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 $\Rightarrow$ 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 `:-)`
  - Implementation not easy due to absence of a 'wait for any input stream' system operation in Java (c.f. `select` in UNIX): must poll each client whether needed or not `:-(`
- ➤ The `java.nio` package now supports asynchronous I/O.

Exercises

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 Data Network; or (iii) one runs on the University network and another on a dial-up internet connection? Do you see that packets are lost, duplicated, or re-ordered? Do the packets arrive regularly spaced?
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?
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 and RMI

Previous lecture

- UDP: connectionless, unreliable
- TCP: connection-oriented, reliable

Overview of this lecture

- Java Remote Method Invocation (RMI)
- Implementing RPC

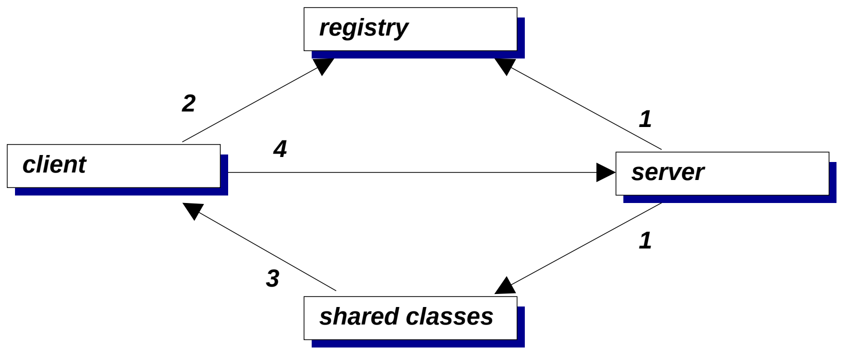
Remote method invocation (1)

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 might 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 registry acts as a name service, with names being of the form
`rmi://linux2.pwf.cl.cam.ac.uk/jkf21/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 use `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 `getFish` is defined:

```
1  package remifc;
2
3  import java.rmi.*;
4
5  public interface FishFinder extends Remote
6  {
7    public static final String NAME
8  = "rmi://magic.voidstar.org.uk/jkf/fishfinder";
9
10   public FishLocation
11     getFish(FishLocation 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 `magic.voidstar.org.uk`) and the name under which to register the service (`jkf/fishfinder`).
- All remote methods must throw `RemoteException`.

RMI—Client (1)

- Example code to construct a shoal of fish...

```
// Bind to RMI service
System.out.println("Looking for " +
  FishFinder.NAME);
FishFinder ff = (FishFinder)
  Naming.lookup(FishFinder.NAME);

// Start listening for connections from other fish
ServerSocket ss = new ServerSocket(0,1,
  InetAddress.getLocalHost());

// Determine our network address and port number
FishLocation my_location =
  new FishLocation(ss.getInetAddress(),
                   ss.getLocalPort());

System.out.println("My location is " +
  my_location);

// Wait to be paired up with another fish
FishLocation friendlyfish_location =
  ff.getFish(my_location);

System.out.println("Other fish’s location is " +
  friendlyfish_location);
```

RMI—Client (2)

Note how few differences there are in the client compared with local invocations on an instance of a class implementing `FishFinder`:

- The call to `Naming.lookup` obtains an instance of a **stub class** implementing the `FishFinder` interface. Invocations on this instance will be forwarded to a remote object registered under the name `FishFinder.NAME`.
- The next three commands start a server socket to await connections from external processes.
- The actual invocation is performed by `ff.getFish(...)`.
- The code requires an exception handler to deal with:
  - `NotBoundException`—no remote object has been associated with the name `FishFinder.NAME`
  - `RemoteException`—if the RMI registry could not be contacted (`Naming.lookup`) or if there was a problem with the call (`ff.getFish`).
  - `AccessException`—if the operation has not been permitted by the installed security manager.

RMI—Server

```
public static void main(String args[]) {
  try {
    // Instantiate the server
    FishFinderImpl s =
      new FishFinderImpl();

    // Allow connections to the registry
    System.setSecurityManager(
      new RMISecurityManager());

    // Bind name in the registry
    Naming.rebind(FishFinder.NAME,s);

    System.out.println(NAME +
      " server running");
  } catch (Exception e) {
    System.out.println("Failed: " + e);
    e.printStackTrace();
  }
}
```

Putting it all together (1)

- Select which machine is to run the registry and update `remifc.FishFinder.NAME` (usually use the same machine for the registry and the server).
- Compile the interface, the client, and the server:

  ```
  $ javac remifc/*.java server/*.java client/*.java
  ```

- Generate stub classes from the server:

  ```
  $ rmic -v1.2 server.FishFinderImpl
  ```

  creating `server.FishFinderImpl_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/jkf21/19.RMISVB/
  ```

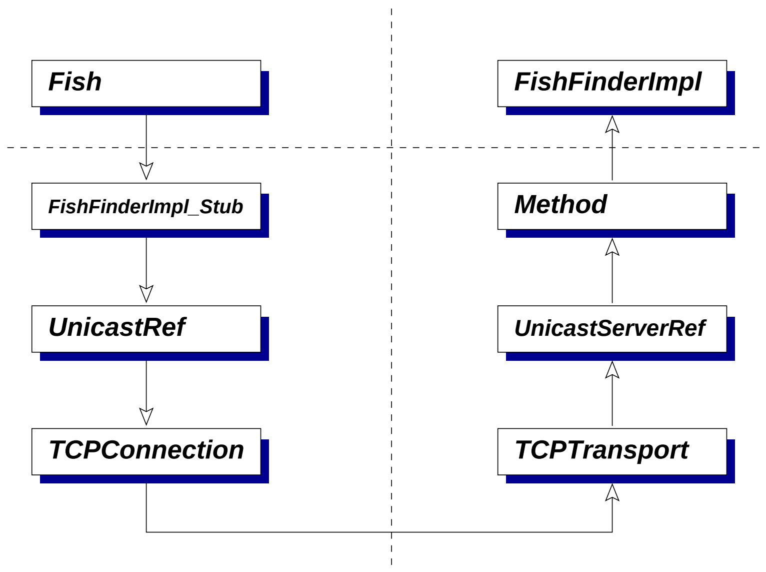
  Do not forget the trailing /
- Start the server running:

  ```
  $ java -Djava.rmi.server.codebase=$CODEBASE \
         -Djava.security.policy=security.policy \
         server.FishFinderImpl
  ```

- Start the clients...

  ```
  $ java client.Fish
  ```

RMI implementation (1)



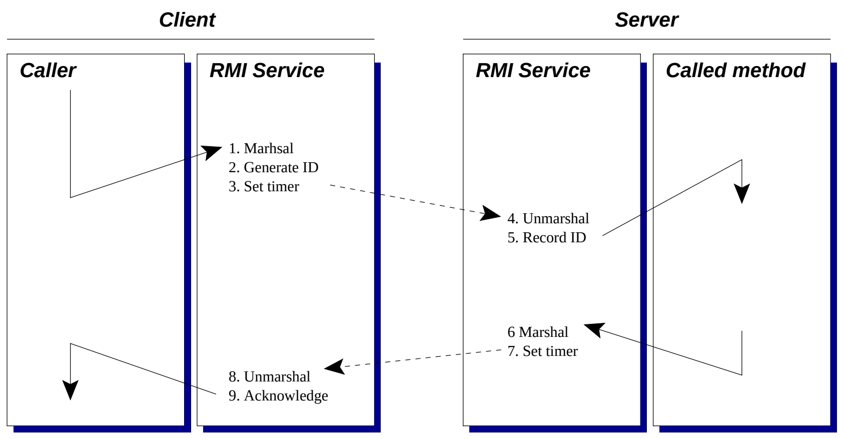
- The _Stub class is the one created by the rmic tool—it transforms invocations on the FishFinder interface into generic invocations of an invoke method on UnicastRef.
- UnicastRef is responsible for selecting a suitable network transport for accessing the remote object—in this case, TCP.
- UnicastServerRef 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: e.g. if a remote method `A.m1` invokes an operation on remote method `B.m2` which in turn invokes an operation `A.m3`.
- But the application programmer must be aware of how many threads might be created and the impact that they might 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 server has returned to the caller.

What happens if a timeout occurs at step 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 the 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 alone, server failures before/after making some change to persistent storage.

Exercises

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.
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?
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?
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.
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 lecture

- Communication using UDP or TCP
- Remote method invocation
- RPC semantics

Overview of this lecture

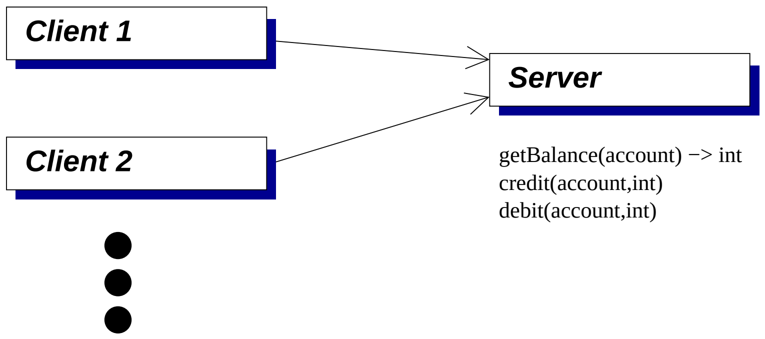
- Compound operations
- Correctness requirements
- Implementation

Compound operations (1)

We've now seen mechanisms for

- controlling concurrent access to objects; and
- 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 might look at the two accounts after one has been credited but before the other has been debited.
  - $\Rightarrow$ the total amount will be incorrect.
- Client 2 might crash after doing its `credit` but before the matching `debit`.
  - $\Rightarrow$ the recipient could be lucky...
- The network might fail, even if the clients are well behaved.
- The server might 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 (1)

Transactions provide a more general abstraction.

Ideally the programmer might 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 the code within a `transaction` block will execute without interference from other activities, in particular

- other operations on the same objects as those mentioned within the block (`src` and `dest` in this case); and
- system crashes (within reason...).

We say that a transaction **commits** atomically (if it completes successfully) or it **aborts** (if it fails for some reason). Aborted transactions leave the state **wholly** unchanged.

Transactions (2)

In more detail we would like committed transactions to satisfy four **ACID properties**:

A tomicity—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.

C onsistency—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.

I solation—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.

D urability—the effects of committed transactions endure subsequent system failures.

- When the system confirms the transaction has committed it must ensure that changes will survive faults—e.g. don't report "commit" until the disk caches have been drained to the disk surfaces.

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 might 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 is held while executing a transaction, allowing only one to operate at once.

Advantage

- Simple.

Disadvantages

- Does not enable concurrent execution, e.g. of two of these operations on separate sets of accounts.
- What happens if operations can fail?

Isolation—Serializability (1)

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.

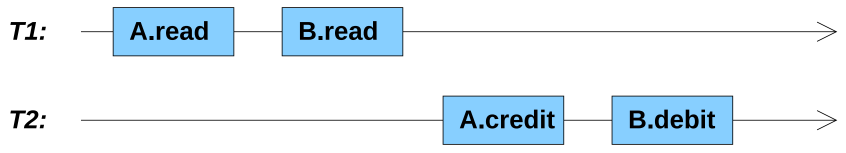
Suppose we have two transactions:

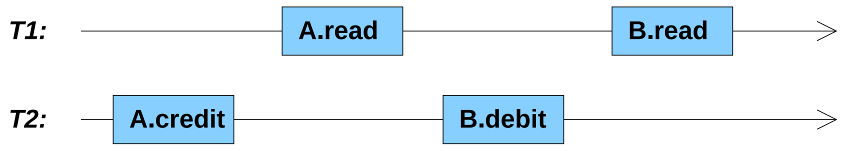
```
T1: transaction {
  int s = server.getBalance(A);
  int t = server.getBalance(B);
  return s + t;
}

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 any of 6 ways.

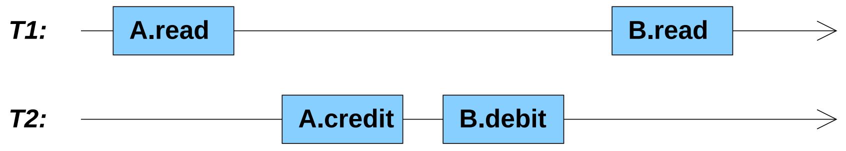
Isolation—Serializability (2)

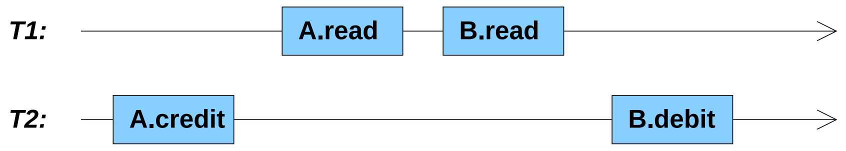
Each of these concurrent executions is OK:





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**.

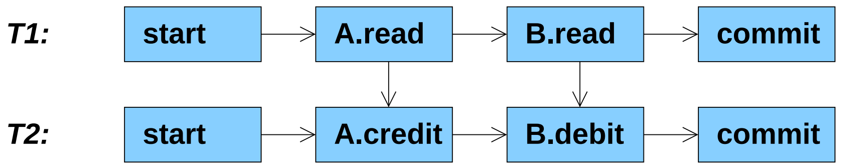
- Nodes in the graph represent the operations comprising each transaction, e.g. `T1: A.read`.
- A directed edge from node `a` to node `b` means that `a` **happens before** `b`.
  - Operations within a transaction are totally ordered by the **program order** in which they occur.
  - **Conflicting** (i.e. 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*.

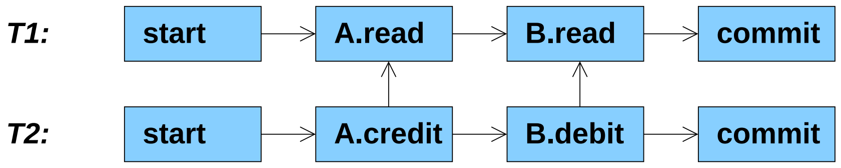
Suppose again that we have two objects `A` and `B` associated with integer values and run transaction `T1` that reads values from both and transaction `T2` that adds to `A` and subtracts from `B`.

Isolation—Serializability (4)

These histories are OK. Either (i) both the `read` operations see the old values of `A` and `B`:

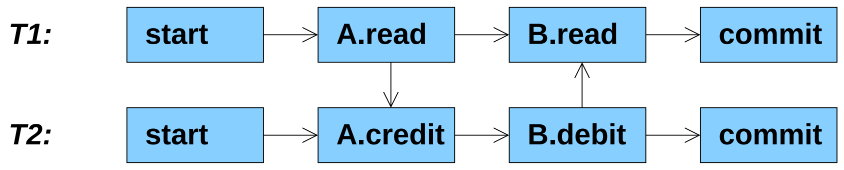


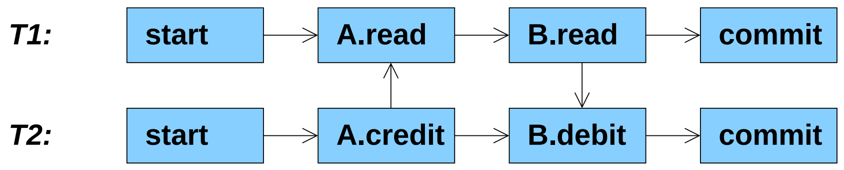
or (ii) both `read` operations see the new values:



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 committed)
- **Dirty reads** (e.g. of updates before they are committed)
- **Unrepeatable reads** (e.g. before an update by another transaction overwrites it)

Isolation and 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 and unrepeatable reads can be made on a per-transaction basis.

Exercises

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.
2. Earlier it was claimed that there are 6 ways in which execution can proceed but only 4 were presented. Illustrate the remaining 2 possible executions and construct history graphs for them.
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; and

   (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 (1)

We will 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; and
- a phase of releasing locks.

Locks must exclude other operations that might 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 might suffice but could limit the degree of concurrency. In the example we could use a MRSW lock, held in read mode for `getBalance()` and in 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  }
```

- 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; and
  - release the lock on `dest` after 4.

Isolation—two-phase locking (3)

How well would this form of two-phase locking work?

Advantages

- 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.

Disadvantages

- Complexity of programming (e.g. 2PL $\Rightarrow$ MRSW needs an *upgrade* operation here).
- Would be nice to provide `startTransaction` and `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 done so; and
  - T_b must abort if T_a does (a cascading abort).

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 (1)

Timestamp ordering (TSO) is another mechanism to enforce isolation.

- Each transaction has a timestamp—e.g. of its start time. These must be subject to a total ordering.
- The ordering between these timestamps will give a serializable order for the transactions.
- If T_a and T_b each 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 the later of the two.
  - 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 $\longrightarrow$ timestamp 17
- `T1,2` executes: set A:17,`read`
- `T2,1` executes $\longrightarrow$ timestamp 42
- `T2,2` executes—this is OK (later timestamp): set A:42,`credit`
- `T2,3` executes: set 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)

Advantages

- The decision of whether to admit a particular operation is based on information local to the object.
- Simple to implement—e.g. by interposing checks on each invocation at the server (contrast with non-strict 2PL).
- Avoiding locking might increase concurrency.
- Deadlock is not possible.

Disadvantages

- 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 might have been influenced by `T1`.
  - could delay `T2,2` until `T1` either commits or aborts (still avoiding deadlock).
- Serializable executions might 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 (1)

Optimistic Concurrency Control (OCC) is the third kind of mechanism we will look at in our survey of means to enforce 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.
- But, if the check often fails then performance might 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 = counter + 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—isolated from other transactions.

Upon commit it must:

- **Validate** that the shadows were consistent...
- ...and that no other transaction has committed an operation on an object which conflicts with the 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 the shadows and retry.

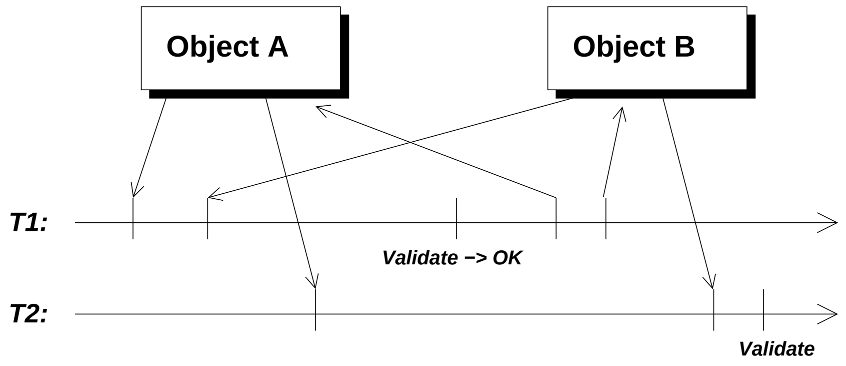
Until commit, updates are made locally. Abort is easy. There is no need for a roll-back mechanism.

No cascading aborts or deadlock.

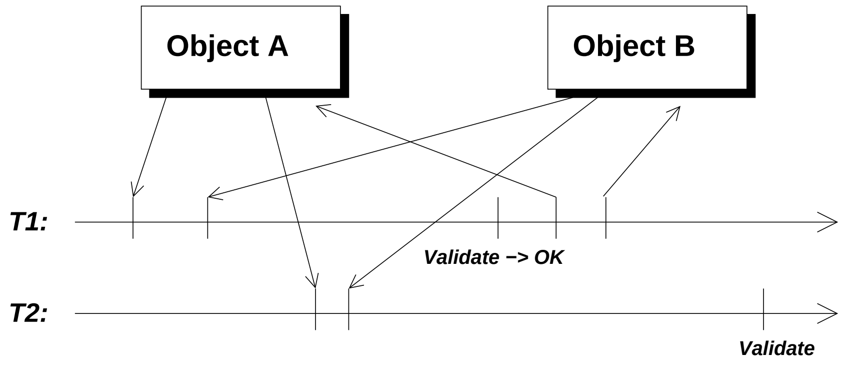
But conflicts force transactions to retry.

Isolation—OCC (3)

- The first step avoids unrepeatable reads, e.g. `T2` has seen `T1`'s update to `B` but has 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 (1)

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 will 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 will 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 that 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 the corresponding row.

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 has 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:
  - If shadow is earlier/equal: part of a consistent snapshot at the start time (`B`, `E` both OK here).
  - If shadow is later: it might 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 have 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 schemes are prone to deadlocking.
   - Use when contention is likely and deadlock is avoidable. Use strict 2PL if transactions are short or cascading aborts are 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 might 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 might allow more flexibility than TSO.

Exercises (1)

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.

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 might 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 (2)

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?

Lecture 20: Crash recovery and logging

Previous lecture

- Enforcing isolation
- Two-phase locking
- Timestamp ordering
- Optimistic concurrency control

Overview of this lecture

- Logging
- Crash recovery
- Checkpoints

Persistent storage

We assume a **fail-stop** model of crashes in which

- the contents of main memory (and above in the memory hierarchy) is lost; and
- 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 of them simultaneously is tolerable (*Part II—Distributed Systems*); or
- ensure that enough information is written to non-volatile storage in order to recover the state after a restart.

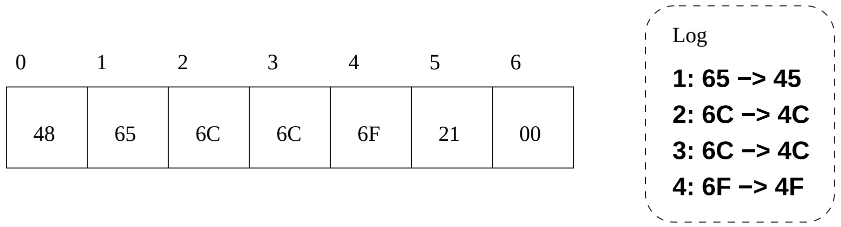
Can we just write object state to disk before every commit? (e.g. invoking `flush()` on any kind of Java `OutputStream`)

- No, not directly: the failure might occur part-way through the disk write (particularly for large amounts of data). We would end up with inconsistent (corrupted) data on the disk.

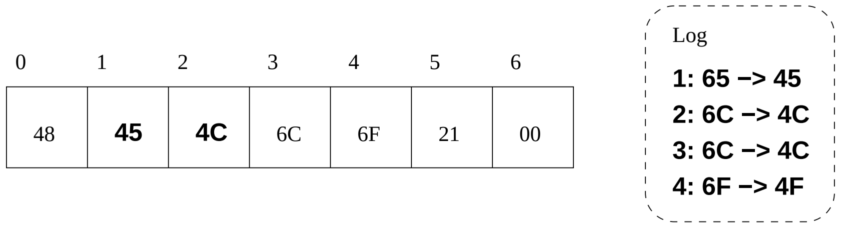
Persistent storage—logging (1)

We could split the update into stages.

1. Write details of the proposed update to a **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.



2. Proceed through the log making the updates.



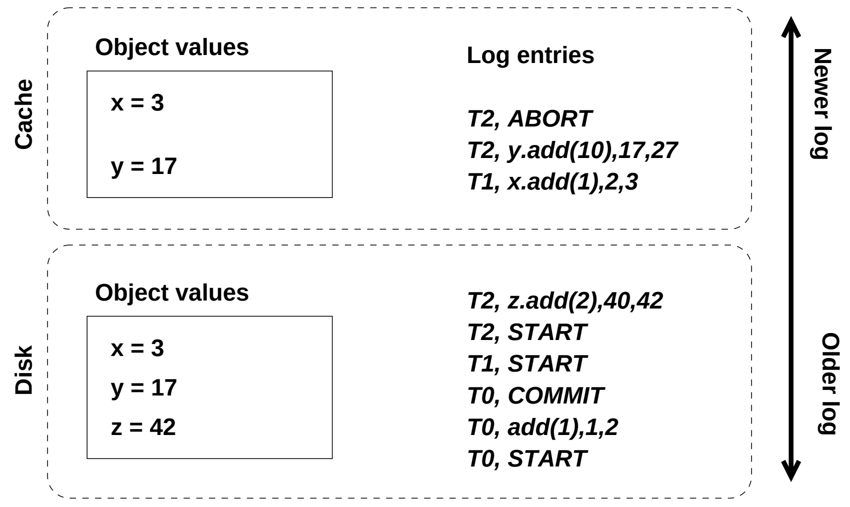
Crash during phase 1 $\Rightarrow$ no updates performed.

Crash during phase 2 $\Rightarrow$ re-check log, either undo (so no change) 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, are 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 to youngest).

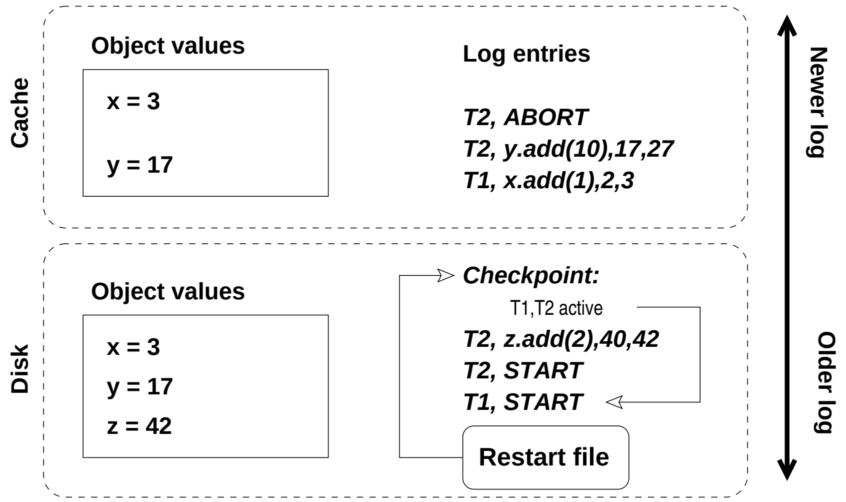
- Note the need for an **idempotent** record of an update—e.g. for `add` we keep the new and old values as well as the difference.
- The old value lets us undo a transaction that is either logged as aborted...
- ...or for which the log stops before we know its outcome.

The naive recovery algorithm can be inefficient.

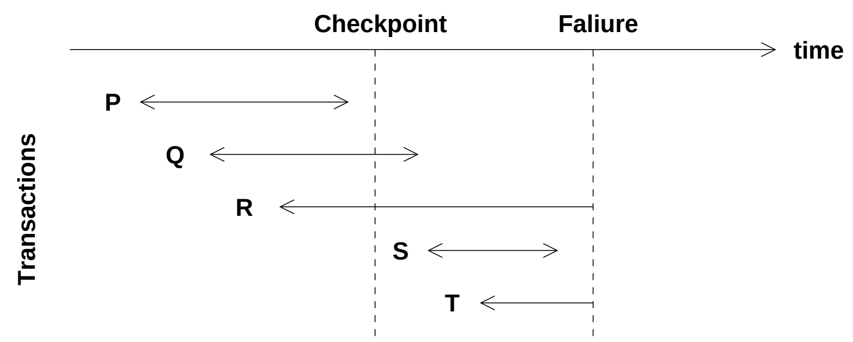
Persistent storage—logging (4)

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; and
- 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

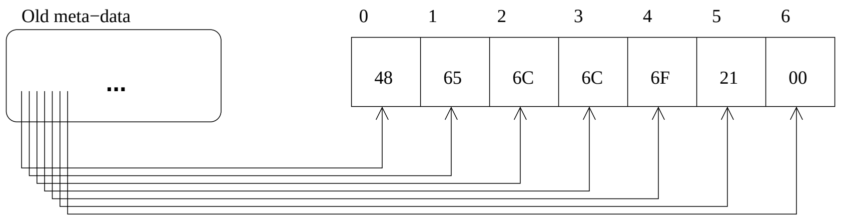
Persistent storage—logging (6)

A general algorithm for recovery:

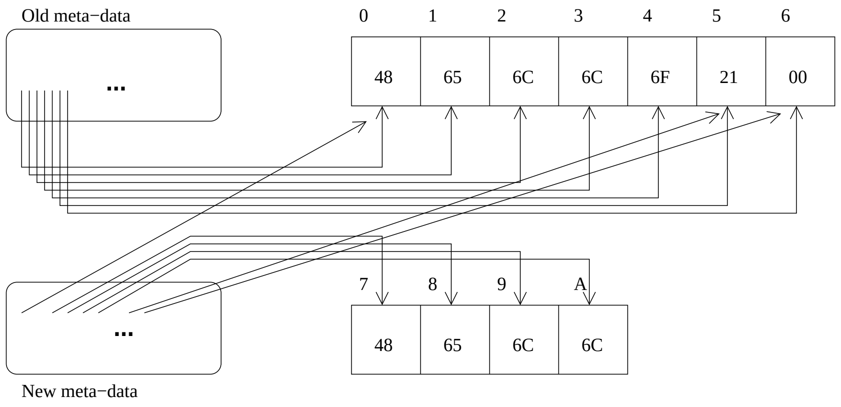
- The **recovery manager** keeps UNDO and REDO lists.
- Initialise UNDO with the set of transactions active at the last checkpoint.
- REDO is initially empty.
- Search forwards from the (most recent) checkpoint record:
  - Add transactions that `start` to the UNDO list.
  - Move transactions that `commit` from the UNDO list to the REDO list.
- Then work backwards through the log from the end to the checkpoint record:
  - UNDOing the effects of transactions on the UNDO list.
- Then work forwards through the log from the checkpoint record:
  - REDOing the effects 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

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 might 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.

2. Suppose that you wish a (non-networked) computer game to maintain a high-score table 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.

Lecture 21: Java 1.5.0—What's new?

Previous lecture

- Logging
- Crash recovery
- Checkpoints

Overview of this lecture

- Generics
- Wildcards
- Bounded wildcards
- Generic methods

Generics

- **Generics** are new to JDK 1.5.
- Generics provide parametric polymorphism in Java.
- Generics have similar syntax to C++ *templates* and there are similarities in semantics but also important differences.
  - Similarities to/differences from C++ templates is not examinable.
- There are resources on the web.
  - `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`
  - Google for “Generics” and “Tutorial”.
- Generics are aimed (primarily) at fixing annoying typing/casting problems with the `java.util` package `Collection` classes.
- The `java.util` package has been re-developed to be generics-aware.

Generics—why?

- The cast to Integer in this code snippet is typical of Java code pre-1.5, and is annoying. The programmer knows that the list contains Integers because (s)he put them there. The type system is getting in the way (and adding run-time overhead).

  ```
  List listOfInts = new LinkedList();
  listOfInts.add(new Integer(42));
  Integer i = (Integer) listOfInts.iterator().next();
  ```

- The problem is that the compiler can only guarantee that the iterator's next item will be an instance of java.lang.Object so the cast is needed to perform a run-time check on the type of the object returned by next().

- Generics allow us to encode sufficient extra information in the source code that the compiler can relax the requirement for the programmer to cast to the subtype.

Generics—syntax (1)

```
List<Integer> gil = new LinkedList<Integer> ();
gil.add(new Integer(42));
Integer i = gil.iterator().next();  // whohoo!
```

- The use of triangle-brackets (< and >) provide the parameters for the generic class `LinkedList`.
  - They are the names of types, `java.lang.Integer` in this case.
  - Think of them as providing a special version of `List` that is specific to `Integer`s.
- Java permits there to be several parameters, not just one.
- The parameter types cannot be primitive types like `int` or `bool`.
- The parameter types cannot be arrays either.
- The data type `LinkedList<Integer>` is a type in its own right.

Generics—syntax (2)

- How do we write parametically polymorphic definitions?

```
public interface List<T> {
  void add(T item);
  Iterator<T> iterator();
}

public interface Iterator<S> {
  S next();
  boolean hasNext();
}
```

- The **formal type parameters** in triangle-brackets (`T` and `S`) provide the parameters.
- We can use `T` and `S` as if they were the names of declared types.
  - `add` swallows arguments of type `T` (or a subclass).
  - `next` returns object references to things of type `S`.
  - `List` uses `T` to specialise another generic type—making a flavour of `Iterator` that operates on things of (base) type `T`.
  - There are important restrictions on how we can use the formal type parameters.
- "Ordinary" methods are permitted too—`hasNext()`.

Generics—syntax (3)

What about classes, inheritance, implementing interfaces...?

```
class MySubClass<T>
 extends MySuperClass<T>
 implements I1<T>, I2<String,T>
{
  ...
}
```

- Single-inheritance is permitted from generic and non-generic superclasses.
  - No multiple-inheritance.
- It is permitted to use `MySubClass`'s formal type parameter in the name of the class that it extends and in the names of interfaces it implements.
  - They are **the same** `T`.
- Thinking of `T` as being text-substituted for (say) Integer is naive.
- Java compiles a generic class once, into a single class file.
  - It does not "have copies" of it for `T=Integer`, `T=String`
- When we invoke a method in Java, the formal arguments are replaced by the actual arguments for the call. Similarly, when we instantiate a parameterised type, the formal type parameters are replaced by the actual type parameters.
- This is unlike C++ templates.

Generics—Naming conventions

- The names of type parameters are usually single upper-case letters.
- The often abbreviate words like *Type* or *Element*.
- Short, capitalised names make it easier to identify the generic type parameters in a body of code.
- The `java.util` package uses `E` (for Element) in many of the definitions of the `Collection` classes/interfaces.

How are generic types related?

- If I have a class `A` and a class `B` that extends `A`, how are `List<A>` and `List<B>` related?

  ```
  List<String> ls = new ArrayList<String>();
  List<Object> lo = ls;   // error!
  ```

- This gives a compile-time error.
- In general, `List<A>` is not a supertype of `List<B>` (and `List<B>` is not a subtype of `List<A>`).

  If the generic types were related, this would be valid:

  ```
  lo.add(new Object());
  String s = ls.get(0);
  ```

- We have turned an `Object` into a `String` by aliasing the object references `ls` and `lo`. This is not permitted in Java 1.5.
- Not what you intuitively expect.

Wildcards (1)

Because `Collection<Object>` is not the superclass of other `Collections`, we cannot write a method that is able to operate on a collection of "anythings". The type of the argument supplied to the method would not match that declared type of the parameter.

- Instead, we can use **wildcards**:

```
void printThemAll(Collection<?> c) {
  for (Object e : c) System.out.println(e);
}
```

- The type of argument c is pronounced "collection of unknowns".
- But, we cannot insert into c because the type of the collection is unknown.

```
void insert(Collection<?> c) {
  c.add(new Object());
}
```

- This is an error at compile-time.

Wildcards (2)

- There is a new use of the *extends* keyword:

  ```
  class A<? extends T, List<T>> {...}
  ```

- And a new use of the *super* keyword:

  ```
  class A<T, List<? super T>> {...}
  ```

- These are known as **bounded** wildcard parameters.

Generic methods

- The solution to the problem of not being able to insert into collections of unknowns is to use **generic methods**.

```
interface Collection<E> {

  public <T>
   boolean containsAll(Collection<T> c);

  public <T extends E>
   boolean addAll(Collection<T> c);

  /* this is also valid...
  public <T, S extends T>
   void copy(List<T> dest, List<S> src);
  */
}
```

Erasures

This code is valid and will compile...

```
public String canYouBelieveIt(Float f) {
  List<Integer> li = new LinkedList<Integer>();
  List nongenericli = li;
  nongenericli.add(f);
  return li.iterator().next();
}
```

- The call to `add()` generates an compile-time unchecked warning.
- And if we execute the code, it fails precisely when an argument of the wrong type is encountered.
  - We were warned.
- This code demonstrates an *erasure* of generic type information.
- The type safety of the Java Virtual Machine is never at risk, even if the code compiles with unchecked warnings.

Generic classes are shared

Because a generic class is compiled only once and multiple type-specialised versions are created dynamically at runtime, this code prints `true` rather than doing as many people expect—printing `false`.

```
List<Float> lf = new LinkedList<Float>();
List<Integer> li = new LinkedList<Integer>();
System.out.println(lf.getClass() == li.getClass());
```

- For the same reason, `static` fields and methods of a generic class are also shared between all instances of all type-specialised versions.
- Thus it is not permitted to refer to the formal type parameters in a static method or initializer, or in the definition or initializer of a static variable.
- And the keyword `synchronized` will disallow far more concurrent activity than you expected!

Casts

- We can write casts that the compiler cannot check at compile-time. A run-time check is used.

  ```
  Collection<String> css =
   (Collection<String>) c_something;
  ```

  This code compiles with a compile-time unchecked warning.

- `instanceof` is not valid with generic types:

  ```
  Collection cs = new LinkedList<Float>();

  // The following line is illegal:
  if (cs instanceof Collection<String>) {...}
  ```

Generics and array

- In Java 1.5 it is illegal to attempt to declare an array whose element type is a type variable or a parameterised type unless it is an unbounded wildcard type.
- This is necessary to ensure that we never get failures at runtime that were neither
  - caught at compile-time and rejected; nor
  - noticed at compile-time as unchecked warnings.
- So this is valid Java...

```
List<?>[] l = new LinkedList<?>[10];
Object o = l;
Object[] oa = (Object []) o;

List<Float> lf = new LinkedList<Float>();
lf.add(new Float(1.414));
oa[1] = lf;

String s = (String) l[1].get(0);
```

  ... although it does give a run-time error. At least the cast is explicit.

Exercises

1. Building is the superclass of House and Shop, and Bungalow is a subclass of House. Create methods `void addItem(...)` and `String printOut(...)` that can be used in conjunction with the following code to produce sensible textual representations of the lists.

```
Shop s = new Shop();
Bungalow b = new Bungalow();

Collection<Building> street =
  new LinkedList<Building>();
Collection<House> houses =
  new LinkedList<House>();
addItem(s, street);
addItem(b, street);
addItem(b, houses);

System.out.println(printOut(street));
System.out.println(printOut(houses));
```