Introduction

Introduction

This course covers three main topics:

- modular design of applications and systems software, using the facilities of the Java programming language as examples,
- the need for and implementation of concurrency control and communication in inter-process and intra-process contexts and
- the concept of transactions and their implementation and uses.

Concurrent Systems and Applications replaces the *Further Java* and *Concurrent Systems* courses this year

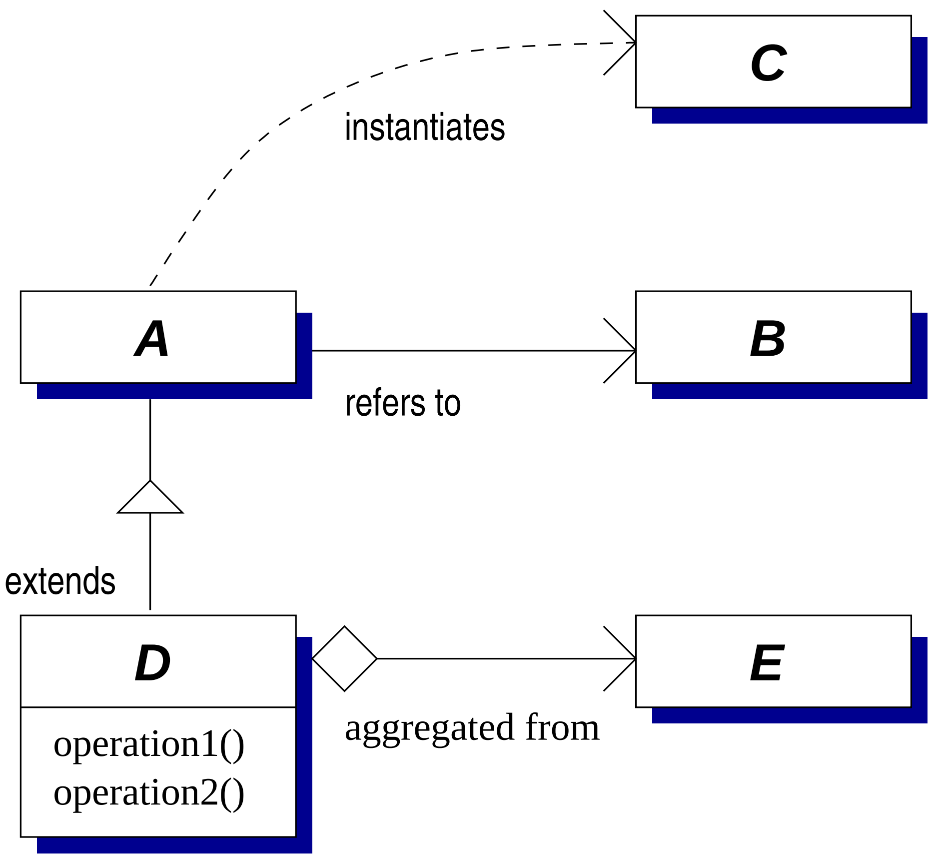
Where possible concrete examples and source code will be used to illustrate topics in concurrent systems

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

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

Notation

Many examples are illustrated using UML-style class diagrams in which nodes represent classes and edges between them denote different kinds of relationship between those classes



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

Concurrency

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

- Between the system as a whole and its user, external devices, etc.

- Between applications running at the same time on a computer – whether through context switching by the OS or by genuine concurrency on a multi-processor machine

- Explicitly between multiple threads within an application

- Implicitly within an application, e.g. when receiving call-backs through a user-interface tool-kit

- Other 'housekeeping' activities within an application, e.g. garbage collection

Books

These course notes are not intended as a complete reference text – either to the subject or for practical programming in Java. For the latter, local documentation is available on the web. `http://www-uxsup.csx.cam.ac.uk/java/jdk-1.2.2/docs` is most relevant.

- Bacon, J. (1997). *Concurrent Systems*. Addison-Wesley (2nd ed.)
- Bracha, G., Gosling, J., Joy, B. & Steele, G. (2000). *The Java Language Specification*. Addison-Wesley (2nd ed.). `http://java.sun.com/docs/books/jls/`
- Lea, D. (1999). *Concurrent Programming in Java*. Addison-Wesley (2nd ed.)
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns*. Addison-Wesley

Outline

- Objects and classes
  - Terminology
    - class, object, method
  - Structuring programs
    - encapsulation, inheritance
    - abstract classes, interfaces
    - design patterns

- GUIs (as examples of the above)
  - AWT, Swing

- Reflection and serialization
  - Class, Method, Field classes
  - newInstance(), clone()
  - Serializable, Externalizable

- Multi-threaded programming
  - Thread / process distinction
  - Thread control in Java
    - Creation
    - Why terminating threads is hard in this context
    - interrupt(), wait(), suspend(), resume()
  - Thread scheduling (uni- and multi-processor techniques)

Outline (2)

- Communication within processes
  - Scenario: a shared address space
  - General problems:
    * Safety, liveness
  - Simple atomic operations
  - Mutual-exclusion
    * Condition variables
    * Deadlock detection and avoidance
  - Building other protocols (multi-reader etc)
  - Alternatives

- Communication between processes
  - Scenario: separate address spaces (and machines?)
  - General problems:
    * Parallel execution
    * Independent failures (of processes or links)
    * Maintaining consistent state
    * No global time
  - Naming
  - Access control
  - IDLs and marshalling to portable data formats
  - Java RMI case study
  - Message passing
    * Sockets API

Outline (3)

- Transactions
  - Scenario: controlled through a transaction manager
  - Correctness criteria: serialisability, linearizability
  - ACID properties
  - Optimistic / pessimistic schemes
  - 2 phase-locking
  - Timestamp ordering
  - Cascading aborts
  - Logging

- Further Java topics
  - Memory models + volatile fields
  - Class loaders
  - Finalizers

Objects and classes

Object-oriented programming

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

For example, we could define:

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

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

  String getName () {
    return name;
  }

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

Object-oriented programming (2)

This example shows a number of concepts:

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

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

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

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

Object references

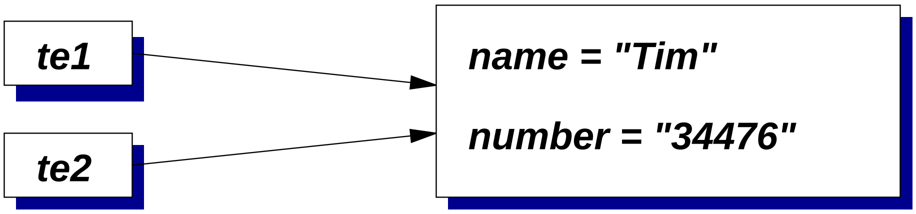
- A program manipulates objects through *object references*

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

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

```
TelephoneEntry te1 = new TelephoneEntry ("Tim",
                                          "34476");
TelephoneEntry te2 = te1;
```

  creates two references to the same object:



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

Composition

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

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

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

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

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

Overloaded methods

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

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

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

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

String doSomething (String c) {
  System.getRuntime().exec(c);
  return "OK";
}

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

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

Overloaded methods (2)

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

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

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

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

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

Constructors

- Using constructors makes it easier to ensure that all fields are appropriately initialized
- If the constructor signature changes (e.g. an extra parameter is added) then other classes using the old signature will fail to compile: the error is detected earlier
- As with methods, constructors can be overloaded
- 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

Inheritance

- Previous examples have defined new classes in terms of existing ones using *composition* – e.g. a class `Bus` containing a field of type `Engine`

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

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

- A programmer defines inheritance using the `extends` keyword, e.g.

```
class NameNumberPlace extends NameNumber
{
  String place;
}
```

Types and inheritance

- Reference types in Java are associated with particular classes:

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

- Such reference variables can also refer to any object of a sub-class of the one named, e.g. if `B extends A` then the field `anotherA` could refer to an instance of `B`

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

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

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

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

Arrays and inheritance

- If `B extends A` then how are `B[]` and `A[]` related?

- As might be expected, `B[]` is a sub-type of `A[]`, creating a subtle problem:

```
1    A[] array1;
2    B[] array2;
3    array1 = new A[2];
4    array2 = new B[2];
5    array1[0] = new B(); // A[] <- B, ok
6    array2[0] = new B(); // B[] <- B, ok
7    array1[1] = new A(); // A[] <- A, ok
8    array2[1] = new A(); // B[] <- A, fails
```

- Line 8 fails at run-time: `array2` refers to an object that is an *array of references to things of type B* and so an object of class `A` is incompatible

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 writing `super.name` rather than `this.name`.

- For example:

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

class B extends A {
  String x;
  int y;

  void f () {
    x = "Field defined in B";
    y = 42;        // B
    super.x = 17; // A
    super.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 {
  int f () { }
}

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

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

- 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

Methods and inheritance (2)

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

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

- As with fields, the `super` keyword is used:

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

Packages

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

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

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

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

class TelephoneEntry { ... }
```

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

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

Modifiers

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

```
<class-modifiers> class NameNumber {

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

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

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

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

The `final` modifier

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

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

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

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

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

The `abstract` modifier

- Used on class and method definitions. An `abstract` method is one for which the class does not supply an implementation (and hence cannot be instantiated), e.g.

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

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

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

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

The `static` modifier

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

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

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

- Similarly, `static` methods are not associated with a current object – unqualified 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 String;
  }

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

Access modifiers

- Previous examples have relied on the programmer being careful when implementing encapsulation
  - e.g. to interact with classes through their methods rather than directly accessing their fields
- Access modifiers can be used to ensure that encapsulation is honoured and also, in some standard libraries, to ensure that untrusted downloaded code executes safely

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

The `protected` modifier

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

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

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

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

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

Other modifiers

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

- 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

- There are three other modifiers to be covered later:
  - `synchronized` and `volatile` are used in multi-threaded applications
  - `transient` is used with the serialization API

Interfaces

- There are often groups of classes that provide different implementations of the same kind of functionality
  - e.g. the *collection* classes in Java 1.2 – `HashSet` and `ArraySet` provide set operations, `ArrayList` and `LinkedList` provide list-based operations

- In that example there are some operations available on all *collections*, further operations on all *sets* and a third set of operations on the `HashSet` class itself

- *inheritance* and *abstract classes* can be used to move common functionality into super-classes such as `Collection` and `Set`
  - Each class can only have a *single* super-class, so should `HashSet` extend a class representing the hashtable aspects of its behaviour, or a class representing the set-like operations available on it?

- More generally, it is often desirable to separate the definition of a standard programming *interface* (e.g. set-like operations) from their *implementation* using an actual data structure (e.g. a hash table)

Interfaces (2)

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

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

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

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

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

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

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

Nested classes

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

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

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

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

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

Nested classes (2)

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

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

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

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

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

Anonymous inner classes

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

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

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

```
interface Ifc {
  void InterfaceMethod ();
}

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

GUIs in Java

Graphical interfaces

- The *Abstract Window Toolkit* (AWT) and *Java Foundation Classes* (JFC) provide facilities that can be used for creating graphical applications in Java
- We'll look at them for a number of reasons:
- ...firstly to introduce the facilities provided and show how they can be used
- ...secondly because their design illustrates many of the object-oriented features of the Java programming language

These examples are intended to show the overall structure of these libraries, not to be a thorough reference

Graphical interfaces (2)

The Java Foundation Classes (JFC) extends the original AWT with (e.g):

- Swing components
- Pluggable look and feel
- Accessibility API
- Java 2D rendering API
- Drag and drop

API specs are available on-line (http://www.java.sun.com/products/jfc) but the emphasis here is how these facilities differ architecturally from AWT

AWT GUI components each had *peers* responsible for their display, e.g. an instance of `java.awt.Scrollbar` has an instance of a class implementing `java.awt.peer.ScrollbarPeer` as its peer.

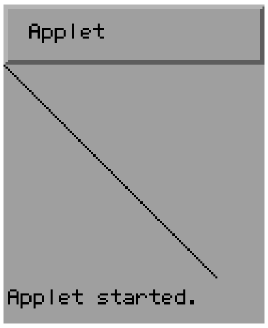
Graphics

- In both AWT and Swing, basic rendering primitives are available on instances of `Graphics`, e.g. using Java applets:

```
import java.awt.*;

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

In `E1.html`:

```
<html><body>
<applet code="E1.class" width=100 height=100>
</applet>
</body></html>
```



Graphics (2)

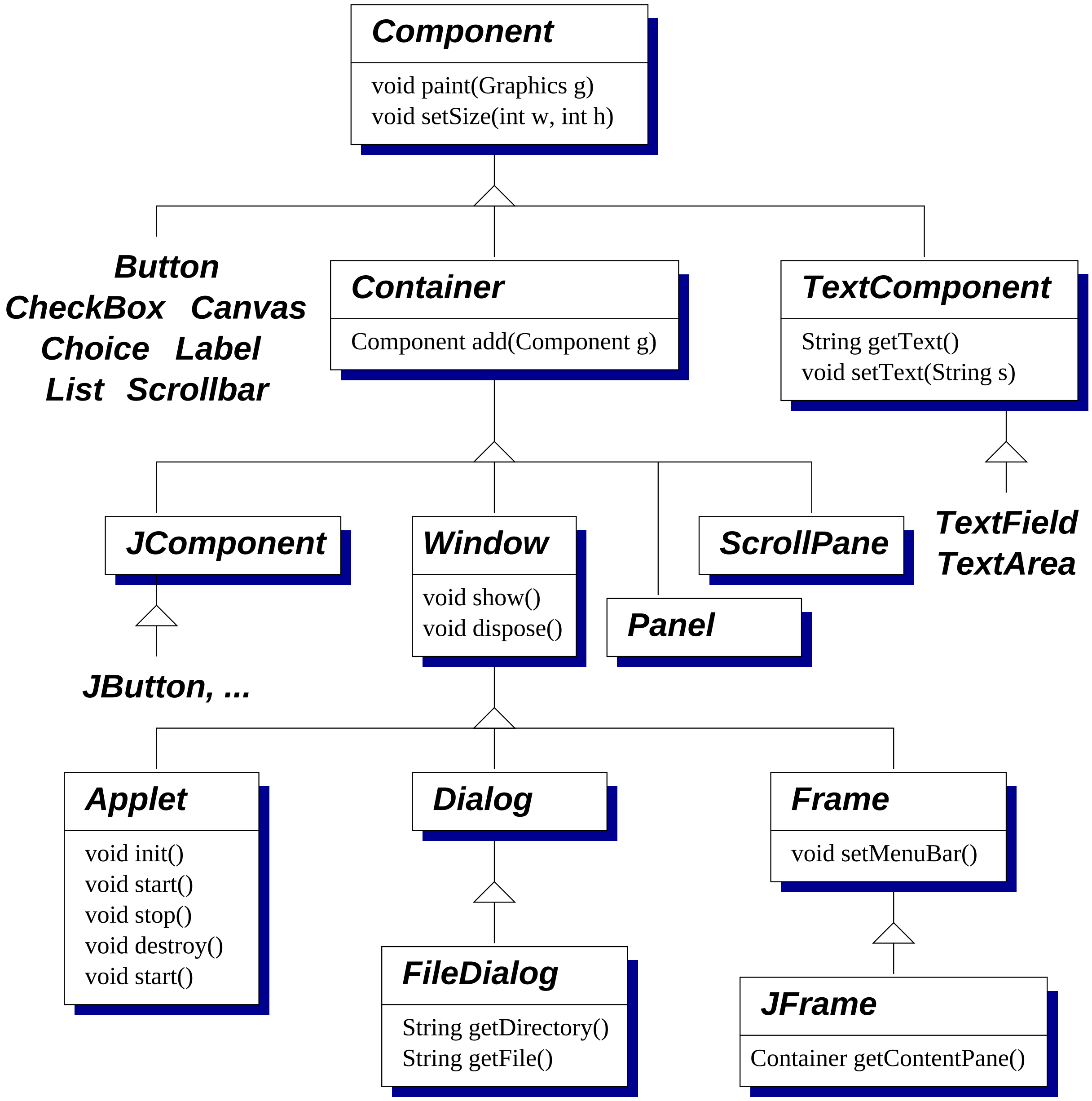
- Here the rendering is performed by making invocations on an object of type `Graphics`

- Simple primitives are available, e.g.

```
void setColor (Color c);

void copyArea (int x, int y, int w,
               int h, int dx, int dy);

void drawLine (int x1, int y1,
               int x2, int y2);

void drawArc (int x, int y, int w, int h,
              int start, int end);
```

- More abstractly, an instance of `Graphics` represents the component on which to draw (more on those later), a translation origin, the clipping mask, the current font, the drawing mode etc

Component hierarchy



Components

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

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

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

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

Components (2)

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

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

- Containers are used to model top-level windows – for example `java.awt.Window` (a plain window, without title bar or borders) and `java.awt.Frame` (a 'decorated' window with a title bar etc)

- Other containers allow the programmer to control how components are organized – in the simplest case `java.awt.Panel`

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

Swing

- The `java.awt.peer.*` definitions are interfaces defining operations that may be performed on the peer
- A particular implementation of AWT provides classes implementing these interfaces, typically using native code
- Hence the look and feel follows that of the underlying system

What are the problems here? Should portability include the exact mode of interacting with applications? What about devices without existing graphical toolkits, or non-graphical forms of input?

Swing GUI components are rendered in Java – i.e. drawn using invocations on `java.awt.Graphics`.

- Exact control over appearance
- Flexibility over look and feel

Swing (2)

Superficially there's a clear correspondence between Swing components in `javax.swing` and AWT components in `java.awt`

`Component`	`JComponent`
`Button`	`JButton`
`List`	`JList`
??	`JProgressBar`
??	`JPasswordField`

`JComponent` extends `Container` (and transitively `Component`)

But there's no direct relationship between (e.g.) `JButton` and `Button`

Containers

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

```
import java.awt.*;
public class Buttons extends Frame {
  public Buttons() {
    super();
    setLayout(new BorderLayout());
    add("North",  new Button("North"));
    add("South",  new Button("South"));
    add("East",   new Button("East"));
    add("West",   new Button("West"));
    add("Center", new Button("Center"));
  }
  public static void main (String args[]) {
    Buttons b = new Buttons();
    b.pack();  b.setVisible(true);
  }
}
```

- For Swing, change `Frame` → `JFrame` and obtain the container using `getContentPane()`, e.g.

```
getContentPane().setLayout( ... );
```

Containers (2)

- `BorderLayout`, shown above, contains up to 5 components

- `CardLayout` treats each component in the container as a card. 1 card is visible at a time. Methods `first` and `next` flip through them

- `FlowLayout` lays out components in horizontal left-to-right lines – e.g. for button bars. A new line is started when the current one becomes full

- `GridLayout` places components on a rectangular grid of equal-sized cells, e.g. `setLayout (new GridLayout (3,2))` creates a 3x2 grid

- `GridBagLayout` is a more flexible layout manager: the rectangular cells may vary in size and instances of `GridBagConstraints` are used to describe how particular cells scale

Usually nesting containers to define a *spatial hierarchy* is preferable to using a complex layout manager: it promotes re-use of the nested components

Receiving input

- An *event-based* mechanism is used for delivering input to applications

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

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

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

Receiving input (2)

- Note that each kind of listener is represented by an *interface* definition rather than by a class. The programmer can therefore define one class that implements all of the listeners that they are interested in (typically the main part of the program, or the class representing a particular part of the UI)

- Instances of `AWTEvent` have a `getSource()` method that returns the component generating the event, 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`

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

Receiving input (3)

- All components can generate:

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

- Containers can generate `ContainerEvent` when components are added or removed
- Windows can generate `WindowEvent` when opened, closed, iconified etc

Input using inner classes

- Anonymous inner classes can be used as an effective way of handling some forms of input, e.g.

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

- Recall that this defines an anonymous inner class that *implements* the `ActionListener` interface

- This idiom is useful because the inner class is able to access the fields and methods of the enclosing instance

Input using inner classes (2)

- A further idiom is to define inner classes that extend *adapter* classes from the `java.awt.event` package. The adapters provide 'no-op' implementations of the associated interfaces – e.g. `MouseMotionAdapter` for `MouseMotionListener`

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

```
addMouseMotionListener
          (new MouseMotionAdapter () {
  public void mouseDragged (MouseEvent e)
  {
    ...
  }
};
```

- The anonymous inner class *extends* the `MouseMotionAdapter` class

Button / JButton

- Instances of `java.awt.Button` (`javax.swing.JButton`) represent labelled buttons:

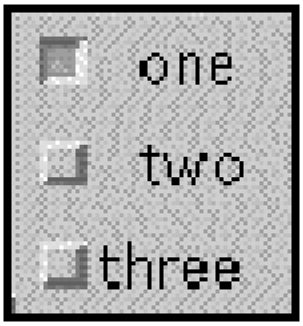
```
Button b = new Button ("Quit");
add (b);
```

- Input is delivered using `ActionEvent` supporting `getActionCommand` (defaults to the button's label) and `getModifiers` (e.g. if `SHIFT`/`CTRL`/`ALT` were pressed). An `ActionListener` has a single `actionPerformed` method

Checkbox / JCheckBox

- A check box is a graphical component that can be in either an *on* or and *off* state – mouse clicks change between the states. For example:

```
setLayout(new GridLayout(3, 1));
add(new Checkbox("one", null, true));
add(new Checkbox("two"));
add(new Checkbox("three"));
```



- Check boxes can be grouped together (only one may be *on* in each group) with `setCheckboxGroup`

- An `ItemListener` receives input events through an `itemStateChanged` method

Label / JLabel

- Label objects represent single lines of read-only text – i.e. changeable by the application, but not able to be edited directly by the user. For example:

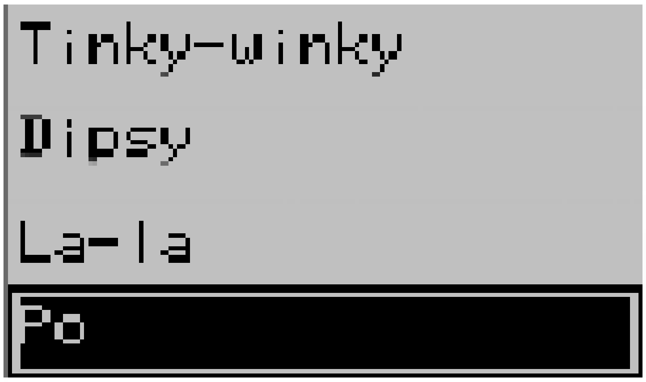
```
setLayout(new FlowLayout(FlowLayout.CENTER,
          10, 10));
add(new Label("Hi There!"));
add(new Label("Another Label"));
```

- The text string maybe passed to the constructor, or controlled using `setText` and `getText` methods

List / JList

- Instances of these classes represent scrollable lists of text items. The programmer can control how many items are visible at once. For example:

```
List lst = new List(4, false);
lst.add("Tinky-winky"); lst.add("Dipsy");
lst.add("La-la"); lst.add("Po");
```

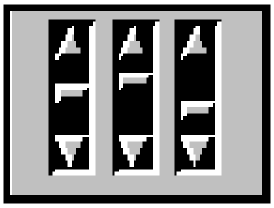


- Input can be received using an `ItemListener` (when entries are selected or deselected) and/or an `ActionListener` (when the user double-clicks on an entry)

Scrollbar / JScrollBar

- These classes embody scroll bars, for example used in a colour selector:

```
new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 255);
```



- The parameters control the orientation, initial value, ‘bubble size’, minimum and maximum values

- An `AdjustmentListener` receives input events via a `adjustmentValueChanged` method

TextComponent / JTextComponent

- These are super-classes for `TextArea` (`JTextArea`) and `TextField` (`JTextField`)

- They define methods `getText` and `setText`, control over whether the text is editable by the user and whether some portion of the text is selected

- `TextArea` and `TextField` are multiple and single-line text regions, e.g.

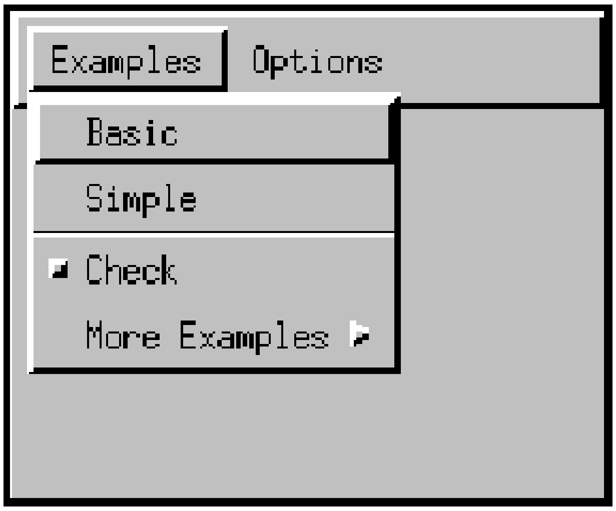
```
TextField tf = new TextField ("Hello!", 20);
TextArea ta = new TextArea ("Goodbye!", 5, 20);
add("North", tf); add("South", ta);
```

Menus

- Menus can be defined using the `Menu`, `MenuBar`, `MenuComponent`, `MenuContainer`, `MenuItem` and `MenuShortcut` classes

- In AWT menus are not components, but can be bound to instances of `Frame` using the `setMenuBar` method

- In Swing `JMenu` sub-classes `JMenuItem`, `AbstractButton` and hence `JComponent`

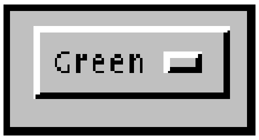


- Input is received using an `ActionListener` on a menu item (typically the same listener would be used for many items)

Choice

- An instance of `java.awt.Choice` represents a pop-up menu of choices – the current choice is displayed as the title of the menu. For example:

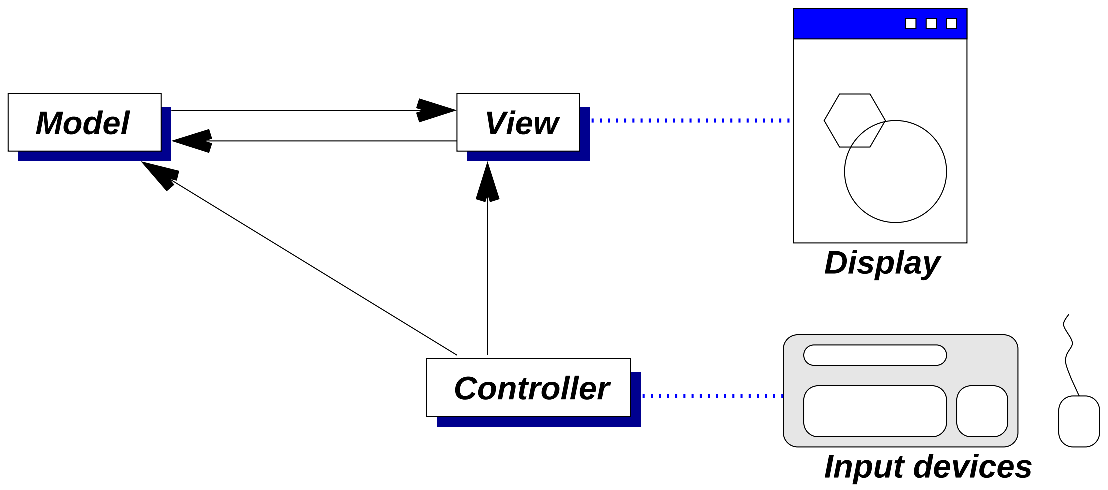
```
Choice ColorChooser = new Choice();
ColorChooser.add("Green");
ColorChooser.add("Red");
ColorChooser.add("Blue");
```



- An `ItemListener` is used for input events

Swing internals

Swing components use a 'model-view-controller' architecture (derived from Smalltalk-80)



This separates three aspects of the component:

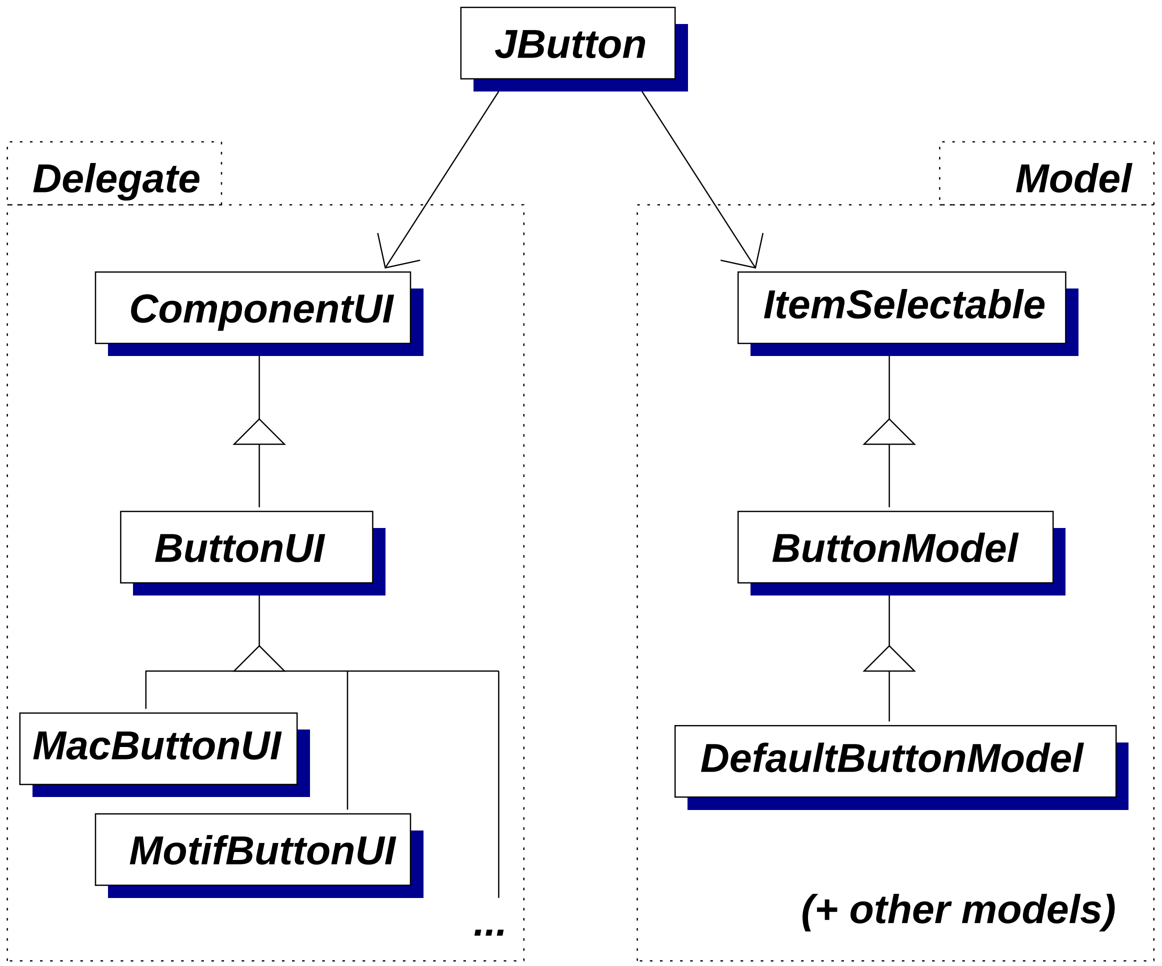
- The *view*, responsible for rendering it to the display
- The *controller*, responsible for receiving input
- The *model*, the underlying logical representation

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

Model-View-Controller (2)

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

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



Accessibility

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

Pacakge: `javax.accessibility`

User interface components should implement `Accessible`, defining a single method

```
public AccessibleContext
getAccessibleContext()
```

This is implemented by the Swing components

Accessibility (2)

An instance of `AccessibleContext` describes and is used to interact with a particular UI component. It defines methods to retrieve associated instances of

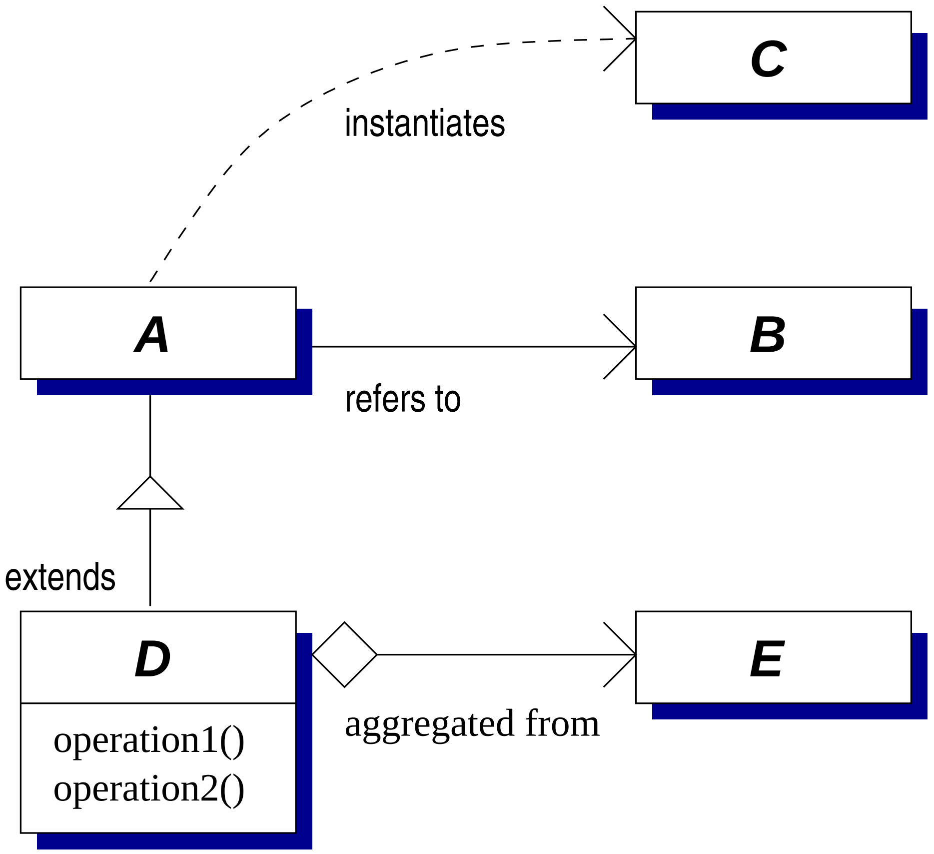
- `AccessibleAction` – representing operations that may be done on the component, named by strings
- `AccessibleComponent` – represents the current visual appearance of the component. Allows colours, fonts, focus settings to be over-ridden
- `AccessibleSelection` – e.g. items in a menu, table or tabbed pane
- `AccessibleRole` – in terms of generic roles such as `SCROLL_PANE` or `SLIDER`
- `AccessibleState` – e.g. `CHECKED`, `FOCUSED`, `VERTICAL`
- `AccessibleText` – represents textual information
- `AccessibleValue` – represents numerical values (e.g. scroll bar positions)

Design patterns

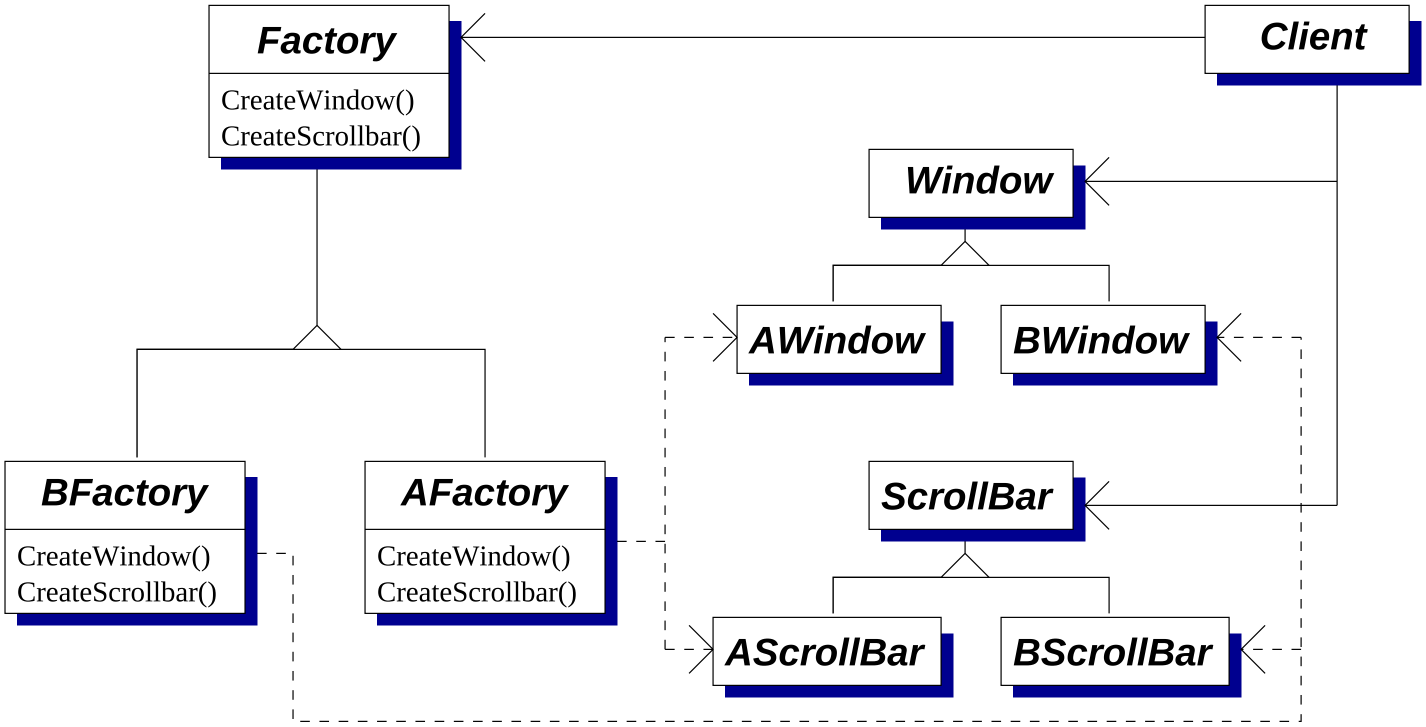
Design patterns

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

- common terminology for describing program organization in terms of inter-related classes
- examples of how to structure programs for flexibility and re-use



Abstract factory pattern



Abstract factory pattern (2)

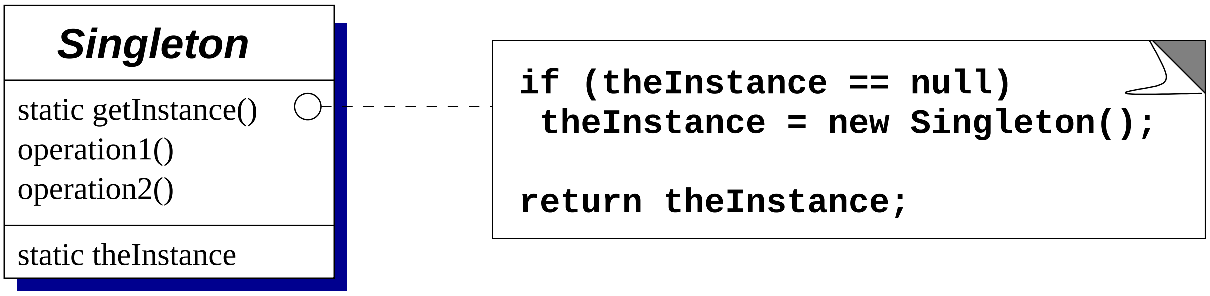
- The *Client* invokes operations on an instance of abstract class *Factory*
- Code for these methods is provided by one of a number of sub-classes, e.g. *AFactory* or *BFactory*
- The factory class instantiates objects on behalf of the client from one of a family of related classes, e.g. *AFactory* instantiates *AWindow* and *AScrollBar*

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

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

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

Singleton pattern



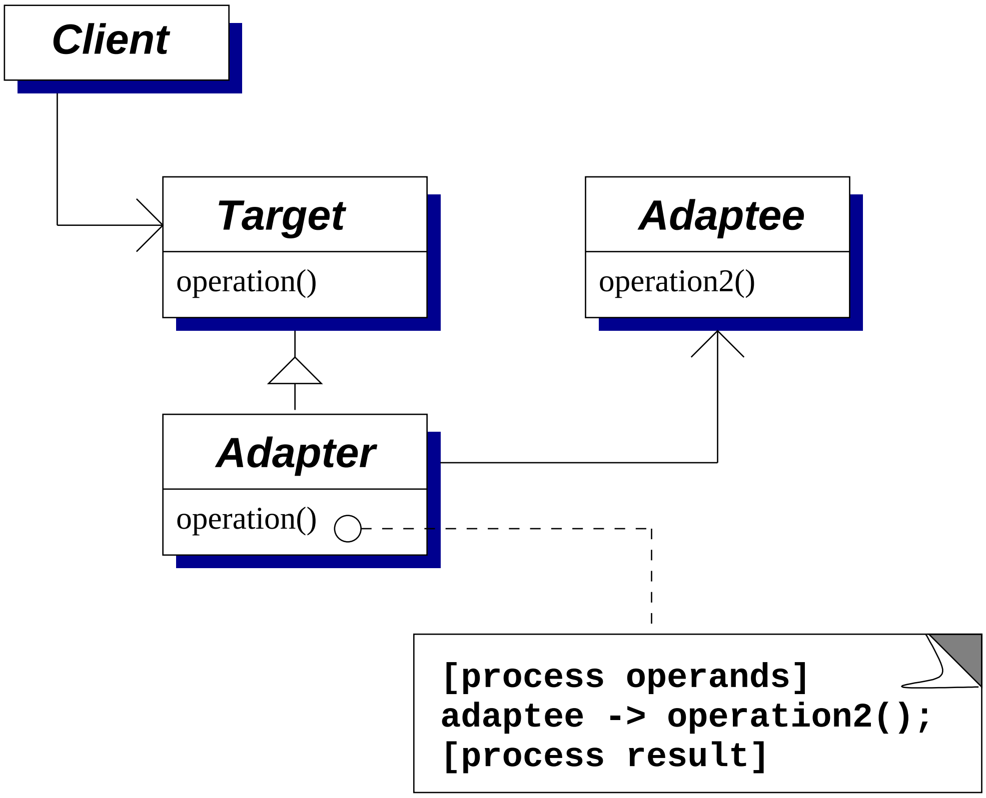
- The Singleton pattern solves the *Highlander problem*: there can be only one instance of a particular class
  - e.g. of a factory class in the previous Abstract Factory pattern
- Clients invoke `getInstance()` to retrieve the unique instance. The first invocation triggers instantiation of a (private) constructor

✔ More flexibile than a suite of static methods (allows sub-classing)

✔ Constraint can be relaxed in a single place – e.g. if a pool of instances are to be used

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

Adapter pattern

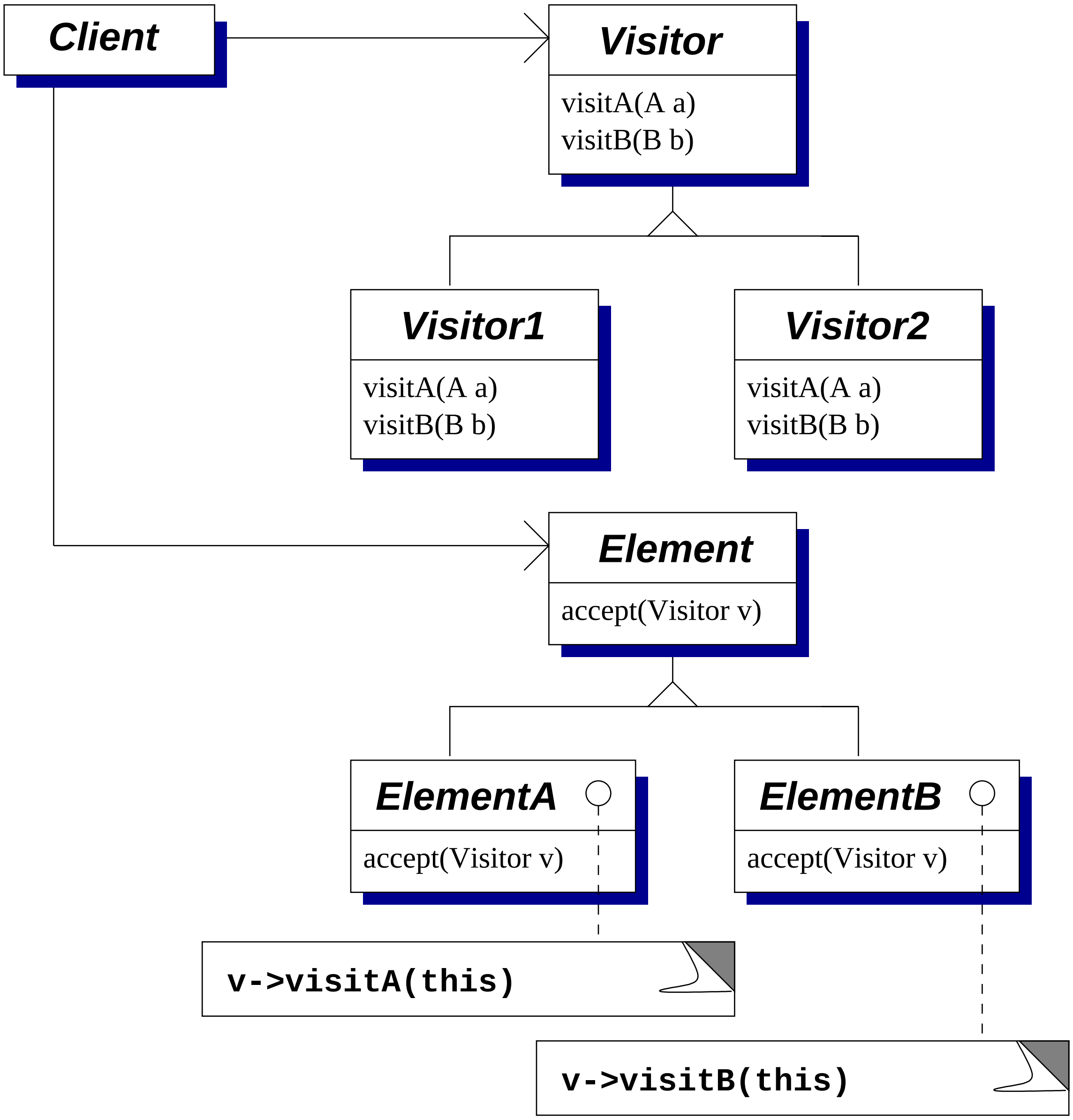


- The *Client* wishes to invoke operations on the *Target* interface which the *Adaptee* does not implement
- The *Adapter* class implements the *Target* interface, in terms of operations the *Adaptee* supports

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

Visitor pattern

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



Visitor pattern (2)

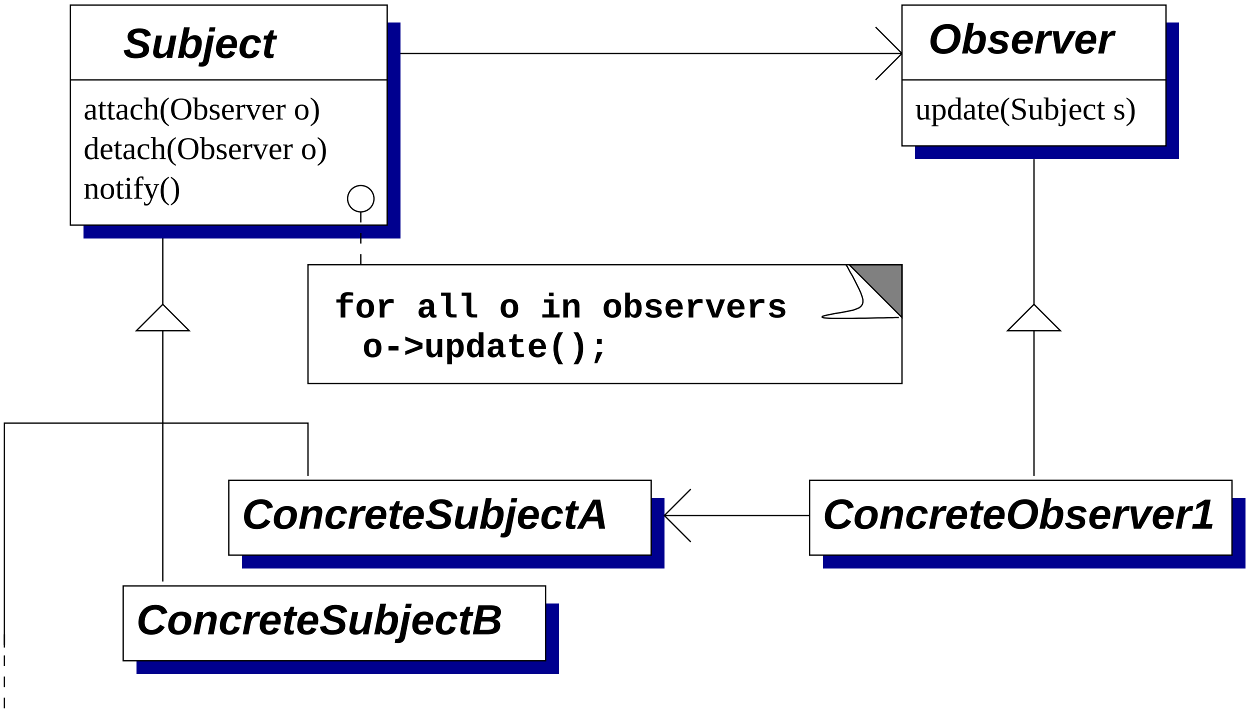
- The data structure is built from instances of *ElementA*, *ElementB*, etc., all sub-classes of *Element*
- These classes, or a separate object structure class, provide some mechanism for traversing the data structure
- The abstract *Visitor* class defines operations corresponding to each sub-class of *Element*
- A concrete sub-class of *Visitor* is constructed for each kind of operation on the data structure

✔ The methods implementing a particular operation are kept together in a single sub-class of *Visitor*

✔ Operations can be added and updated without changing the data structure's definition

✘ As with the Abstract Factory pattern, changing the data structure requires changes to many classes

Observer pattern



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

Reflection and serialization

Reflection

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

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

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

- It is *not* possible to obtain or change the source code of methods
  - ...but at the end of the course we'll look at class loaders that provide a mechanism for dynamically loading *new* class definitions

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

Reflection (2)

- Reflection is provided by a number of classes in the `java.lang` and `java.lang.reflect` packages. Each class models one aspect of the Java programming language
- An instance of `Class` represents a Java class definition. The `Class` associated with an object is obtained by the `getClass()` method defined on it
- An instance of `Field` represents a field definition, obtained from the `Class` object by `getFields()`
- Instances of `Method` and `Constructor` represent method and constructor definitions, similarly obtained by `getMethods()` and `getConstructors()`
- Similarly for `getSuperclass()` and `getInterfaces()`

Reflection (3)

- For example:

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

- Line 5 creates a new instance of `ReflExample`
- Line 6 obtains the `Class` object for that instance
- Line 7 obtains the name of that class

Reflection (4)

- We could do the same in reverse:

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

Reflection (5)

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

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

- We could have named any class on the command line
- By default a 0-argument constructor is called (and must exist)
- Specific constructors are also modelled by `Constructor` objects and define a `newInstance` method

Fields and reflection

- We can invoke `getFields()` on a `Class` object to obtain an array of the fields defined on that class
- As a shortcut we can also use `getField(...)`, passing the name required, to obtain information about an individual field
- If there is a security manager then its `checkMemberAccess` method must permit general access for `Member.PUBLIC` and `checkPackageAccess` must permit reflection within the package
- Only `public` fields are returned by `getFields()`
- A general `getDeclaredFields()` method provides full access (subject to a `checkMemberAccess` test for `member.DECLARED`)
- Given an instance of `Field` we can use...
  - `Class getDeclaringClass ()`
  - `String getName ()`
  - `int getModifiers ()`
  - `Class getType ()`

Fields and reflection (2)

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

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

Fields and reflection (3)

- For example,

```
$ java ReflExample3 ReflExample3 field1
17

$ java ReflExample3 ReflExample3 field2
42

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

- There are similar methods for setting the value of the field

Methods and reflection

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

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

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

  is equivalent to making the call

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

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

Methods and reflection (2)

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

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

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

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

Serialization

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

One use for this is automatically saving/loading data structures

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

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

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

Serialization (2)

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

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

or

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

- A real example must consider exceptions as well
- Fields with the `transient` modifier applied to them are not saved or restored

java.io.Serializable

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

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

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

The definition of `Serializable` is trivial:

```
public interface Serializable {
}
```

java.io.Serializable (2)

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

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

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

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

If present these are called to save/restore object state.

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

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

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

java.io.Externalizable

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

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

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

It is invoked using the normal rules of method dispatch

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

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

Multi-threaded programming

Threads and processes

Recall the two rôles of an operating system:

- to *securely multiplex resources*, i.e.
  - protect applications from each other, yet
  - share physical resources between them
- to provide an abstract *virtual machine*, e.g.
  - time-shareing CPU to provide virtual processors,
  - allocating and protecting memory to provide per-process virtual address spaces,
  - present h/w independent virtual devices.
  - divide up storage space by using filing systems.

In the introduction we talked about *explicit concurrency* in applications – how does this relate to these tasks of the OS?

Threads and processes (2)

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

For each process have a *process control block* (PCB):

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

For each thread have a *thread control block* (TCB):

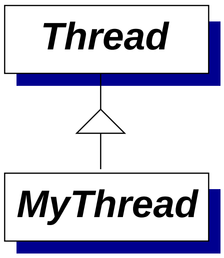
- Thread state
- Context slot (perhaps in h/w)
- Refs to user (and kernel?) stack
- Scheduling parameters (e.g. priority)

Concurrency

- Previous examples have been implemented using a single thread that runs the `main` method of a program
- Java supports *lightweight* concurrency within an application – multiple threads can be running at the same time
- Can simplify code structuring and aid interactive response – e.g. one thread deals with user interaction, another thread deals with computation
  - Easier to add additional tasks as new threads?
- Can benefit from multi-processor hardware
  - e.g. the `hammer.thor` machine has 4 processors
- Implementation schemes vary substantially. We'll look at how multiple threads are available to the Java programmer, then at how they may be implemented and managed by the OS

Creating threads

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




```
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 (2)

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

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

- Line 13 actually start the two threads executing

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

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

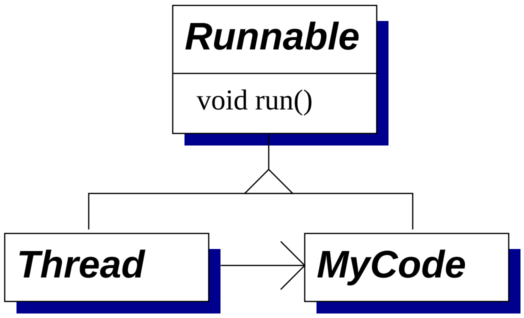
  etc...

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

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

Creating threads (3)

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




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

Creating threads (4)

- As before, Lines 2–8 define the code that the new threads will execute

- Lines 11–12 instantiate two `Thread` objects, passing a reference to an instance of `MyCode` to them as their *target*

- Line 13 starts these two threads executing

- Note that here the `run` methods of the two threads are being executed on the *same* `MyCode` object, whereas two separate `MyThread` objects were required

- The second way of creating threads is more complex, but also more flexible

- Generally, fields in the class containing the `run` method will hold per-thread state – e.g. which part of a problem a particular thread is tackling

Creating threads (5)

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

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

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

- As before, line 9 actually starts the thread executing

Terminating a thread

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

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

- Beware: when using `stop` the exception may be delivered to the target thread when it is executing a `finally` block
- `stop` is deprecated in newer version of Java: it should not be used in new programs, in favour of the `interrupt()` method on `java.lang.Thread`
- A thread is responsible for periodically calling `isInterrupted()`

Terminating a thread (2)

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

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

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

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

Join

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

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

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

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

Priority controls

- Methods `setPriority` and `getPriority` on `java.lang.Thread` allow the priority to be controlled
- A number of standard priority levels are defined: `MIN_PRIORITY`, `NORM_PRIORITY`, `MAX_PRIORITY`
- The programmer can also try to influence thread scheduling using the `yield` method on `java.lang.Thread`. This is a hint to the system that it should try switching to a different thread – note how it was used in the previous examples
  - In a non-preemptive system even low priority threads may continue to run unless they periodically `yield`
- Selecting priorities becomes complex when there are many threads or when multiple programmers are working together

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

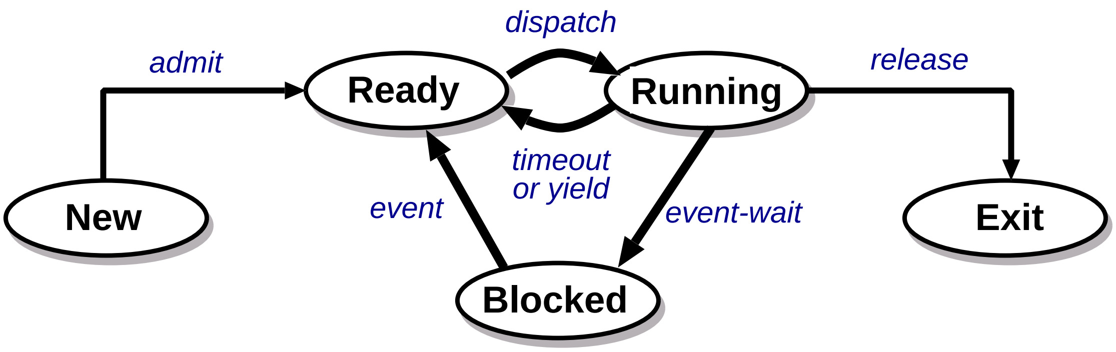
Thread scheduling

- The choice of exactly which thread(s) execute at any given time can depend both on the operating system and on the JVM
- Some systems are *preemptive* – i.e. they switch between the threads that are eligible to run. Typically these are systems in which the OS supports threads directly, i.e. maintaining separate PCBs and TCBs
- Other systems are *non-preemptive* – i.e. they only switch when the running thread yields, becomes blocked or exits. Typically these systems implement threads within the JVM
- The Java language specification says that, in general, threads with high priorities will run in preference to those with lower priorities

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

Thread scheduling (2)

For now assume a five-state model with threads supported by the OS:

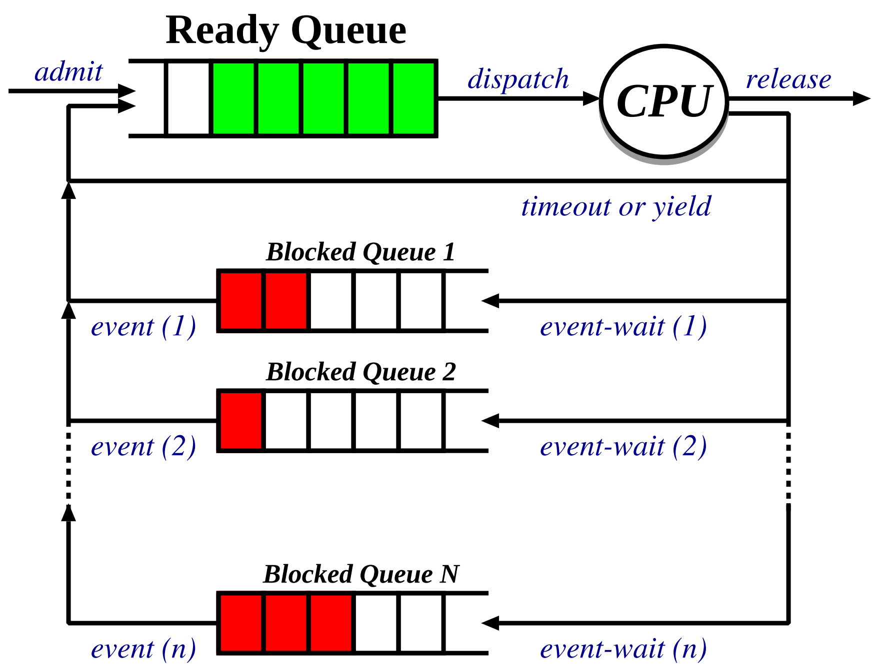


- The operating system must:
  - decide if a new thread should be admitted
  - wake up blocked threads when appropriate
  - clean up after threads terminate
  - choose amongst runnable threads $\Rightarrow$ *schedule*

- Typical scheduling objectives:
  - Maximise CPU utilisation
  - Maximise throughput
  - Minimise average response time

- Also want to minimise overhead (space + time).

Scheduler data structures



Inside scheduler maintain TCBs according to state:

- Runnable ⇒ "`current_thread`"
- Ready ⇒ on ready queue
- Blocked ⇒ on a blocked queue

Sometimes there will be multiple current threads (e.g. a multi-processor system) or multiple ready queues (e.g. for different thread priorities)

When do we schedule?

Can choose a new thread to run when:

1. a running thread blocks (`running` → `blocked`)
2. a timer expires (`running` → `ready`)
3. a waiting thread unblocks (`blocked` → `ready`)
4. a thread terminates (`running` → `exit`)

- A non-preemptive system schedules on 1, 4 only:
  - ✔ simple to implement
  - ✘ open to denial of service
  - ✘ poor priority concept
  - ✘ doesn't extend cleanly to MP
- Most modern systems use *preemptive* scheduling:
  - ✔ solves above problems
  - ✘ introduces concurrency problems. . .

Static priority scheduling

- All threads are not equal ⇒ associate a *priority* with each, e.g.

Highest	Interrupt handlers
	Device handlers
	Pager and swapper
	Other OS daemons
	Interactive jobs
Lowest	Batch jobs

- Scheduling decision simple: just select runnable thread with highest priority. (Be careful when seeing any system that refers to explicit priority numbers: lower numbers often denote higher priority)

- Problem: how to resolve ties?
  - round robin with time-slicing
  - allocate quantum to each thread in turn.
  - Problem: biased towards CPU intensive jobs.
    * per-thread quantum based on usage?
    * ignore?

- Problem: starvation. . .

Dynamic priority scheduling

- Use same scheduling algorithm, but allow priorities to change over time
- e.g. simple aging:
  - threads have a (static) *base priority* and a dynamic *effective priority*
  - if thread starved for k seconds, increment effective priority
  - once thread runs, reset effective priority
- e.g. computed priority:
  - First used in Dijkstra's THE
  - time slots: $\ldots, t, t+1, \ldots$
  - in each time slot t, measure the CPU usage of thread j: u^j
  - priority for thread j in slot $t+1$: $p^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$
  - e.g. $p^j_{t+1} = p^j_t/2 + ku^j_t$
  - penalises CPU bound $\rightarrow$ supports I/O bound.

Example: 4.3BSD Unix

- Priorities 0 (high) – 127 (low), user processes ≥ 50, round robin within priorities, quantum 100ms.

- Priorities are based on usage and "nice" value:

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{nticks} + 2 \times nice_j$$

  gives the priority of process j at the beginning of interval i, where $nice_j \in [-20, 20]$. i.e. penalizes (recently) CPU bound processes in favour of I/O bound ones.

- $CPU_j(i)$ is incremented every tick in which process j is executing, and decayed each second using:

$$CPU_j(i) = \frac{2 \times load_j}{(2 \times load_j) + 1} CPU_j(i-1) + nice_j$$

- $load_j(i)$ is the sampled average length of the run queue in which process j resides, over the last minute of operation

- so if e.g. load is 1 $\Rightarrow \sim$ 90% of 1 seconds CPU usage "forgotten" within 5 seconds

Example: Windows 2000

- Hybrid static/dynamic priority scheduling:
  - Priorities 16–31: "real-time" (static priority)
  - Priorities 1–15: variable (dynamic priority)
  - Priority 0: system (zero page thread)

- Default quantum 2 ticks ($\sim$20ms) on Professional, 12 ticks ($\sim$120ms) on Server.

- Threads have *base* and *current* ($\geq$ base) priorities.
  - On return from I/O, current priority is *boosted* by driver-specific amount
  - Subsequently, current priority decays by 1 after each completed quantum
  - Also get boost for GUI threads awaiting input
  - Yes, this is true: `HKLM\SYSTEM\CurrentControlSet\Control\Priority-Control`

- On Professional also get *quantum stretching*:
  - "...performance boost for the foreground application" (window with focus)
  - All threads in the foreground application get double or triple quantum
  - (More ad-hoc scheduler hacks to come)

Real-time systems

- Produce correct results *and* meet predefined deadlines.

- "Correctness" of output related to time delay it requires to be produced, e.g.
  - nuclear reactor safety system
  - JIT manufacturing
  - video on demand

- Typically distinguish hard (HRT) and soft real-time (SRT):

  HRT: output utility = 100% before the deadline, 0 (or less) after the deadline.

  SRT output utility = 100% before the deadline, (100 - kt)% if t seconds late.

- Building such systems is all about *predictability*.

- It is *not* about speed.

Real-time scheduling

- Basic model:
  - consider set of tasks T_i, each of which requires s_i units of CPU time before a (real-time) deadline of d_i
  - often extended to cope with *periodic* tasks: require s_i units every p_i units

- Best-effort techniques give no predictability
  - in general priority specifies *what* to schedule but not *when* or *how much*.
  - i.e. CPU allocation for thread t_i, priority p_i depends on all other threads at t_j s.t. $p_j \geq p_i$.
  - with dynamic priority adjustment becomes even more difficult.

$\Rightarrow$ need something different

- Three main approaches:
  1. static off-line scheduling
  2. static priority algorithms
  3. dynamic priority algorithms

Static off-line scheduling

Advantages:

- Low run-time overhead
- Deterministic behavior
- System-wide optimization
- Resolve dependencies early
- Can prove system properties

Disadvantages:

- Inflexibility
- Low utilisation
- Potentially large schedule
- Computationally intensive

In general, off-line scheduling only used when determinism is the overriding factor, e.g. MARS

Static priority algorithms

Most common is Rate Monotonic (RM)

- Assign static priorities to tasks at off-line (or at 'connection setup'), high-frequency tasks receiving high priorities
- the tasks processed with no further rearrangement of priorities required ($\Rightarrow$ reduces scheduling overhead)
- optimal, static, priority-driven alg. for preemptive, periodic jobs: i.e. no other static algorithm can schedule a task set that RM cannot schedule
- Admission control: the schedule calculated by RM is always feasible if the total utilisation of the processor is less than $ln2$
- for many task sets RM produces a feasible schedule for higher utilisation (up to $\sim 88\%$); if periods harmonic, can get 100%
- Predictable operation during transient overload

Dynamic priority algorithms

Most popular is Earliest Deadline First (EDF):

- Scheduling pretty simple:
  - keep queue of tasks ordered by deadline
  - dispatch the one at the head of the queue
- EDF is an optimal, dynamic algorithm:
  - It may reschedule periodic tasks in each period
  - If a task set can be scheduled by any priority assignment, it can be scheduled by EDF
- Admission control: EDF produces a feasible schedule whenever processor utilisation is $\leq$ 100%
- Problem: scheduling overhead can be large
- Problem: if system overloaded, all bets are off

Multimedia scheduling

- Increasing interest in multimedia applications (e.g. video conferencing, mp3 player, 3D games)

- Challenges OS since require presentation (or processing) of data in a timely manner

- OS needs to provide sufficient *control* so that apps behave well under contention

- Main technique: exploit soft real-time scheduling

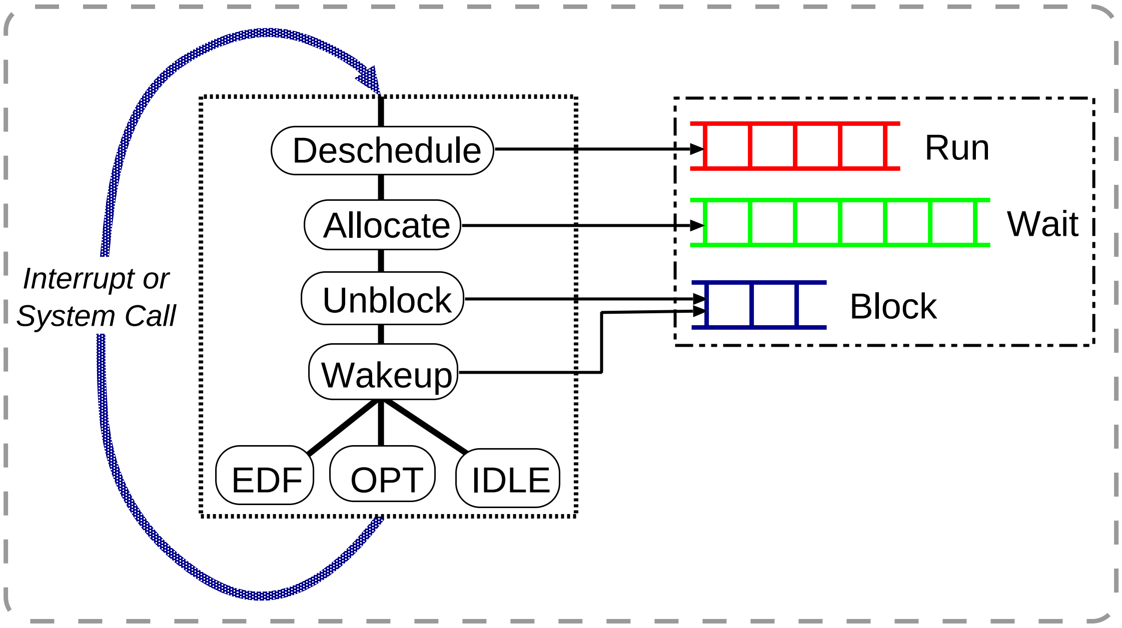
- Effective since:
  - The value of multimedia data depends on the timeliness with which it is presented or processed

  $\Rightarrow$ Real-time scheduling allows applications to receive sufficient and timely resource allocation to handle their needs even when the system is under heavy load
  - Multimedia data streams are often somewhat tolerant of information loss

  $\Rightarrow$ informing applications and providing *soft* guarantees on resources are sufficient.

- Still ongoing research area

Example: Atropos (CUCL)

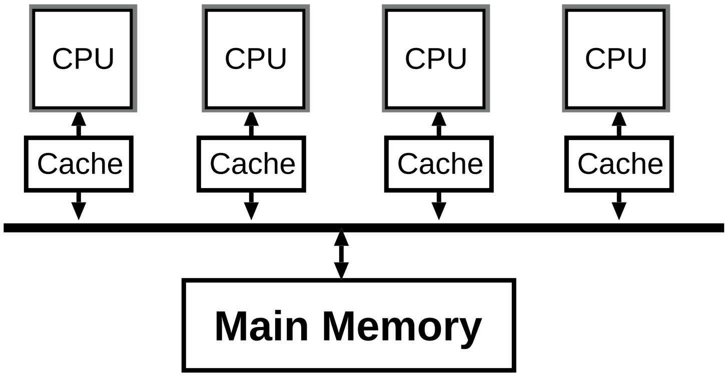


- Task requirements described by (p, s, x) triples
- System performs admission control
- Use a variant of EDF
- Expose CPU via activations
- Actual scheduling is easy ($\sim$200 lines C)

Multi-processors

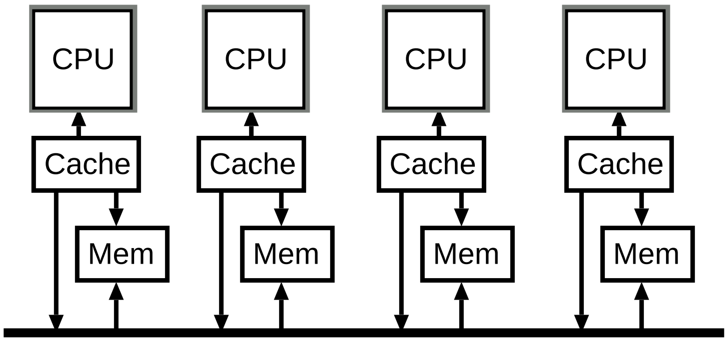
Two main kinds of [shared-memory] multi-processor:

1. Uniform Memory Access (UMA), aka SMP



- all (main) memory takes the same time to access
- scales only to 4, 8 processors

2. Non-Uniform Memory Access (NUMA)



- rarer and more expensive
- can have 16, 64, 256 CPUs . . .

Multi-processor operating systems

Multi-processor OSes may be roughly classed as either *symmetric* or *asymmetric*.

- Symmetric Operating Systems:
  - identical system image on each processor
    $\Rightarrow$ convenient abstraction
  - all resources directly shared
    $\Rightarrow$ high synchronisation cost
  - typical scheme on SMP (e.g. Linux, W2K).
- Asymmetric Operating Systems:
  - partition functionality among processors
  - better scalability (and fault tolerance?)
  - partitioning can be static or dynamic
  - common on NUMA (e.g. Hive, Hurricane)
  - NB: asymmetric $\not\Rightarrow$ trivial "master-slave"
- Also get hybrid schemes, e.g. Disco:
  - (re-)introduce *virtual machine monitor*
  - can fake out SMP (but is this wise?)
  - can run multiple OSes simultaneously. . .

Multi-processor scheduling

- Objectives:
  - Ensure all CPUs are kept busy
  - Allow application-level parallelism

- Problems:
  - Preemption within critical sections:
    - ∗ thread $\mathcal{A}$ preempted while holding spinlock
    - ⇒ other threads can waste many CPU cycles
    - ∗ similar situation with producer/consumer threads (i.e. wasted schedule)
  - Cache pollution:
    - ∗ if thread from different application runs on a given CPU, lots of compulsory misses
    - ∗ generally, scheduling a thread on a new processor is expensive
    - ∗ (can get degradation of factor or 10 or more)
  - Frequent context switching:
    - ∗ if number of threads greatly exceeds the number of processors, get poor performance

Multi-processor scheduling (2)

Consider basic ways in which one could adapt uniprocessor scheduling techniques:

- Central queue:
  - ✔ simple extension of uniprocessor case
  - ✔ load-balancing performed automatically
  - ✘ n-way mutual exclusion on queue
  - ✘ inefficient use of caches
  - ✘ no support for application-level parallelism
- Dedicated assignment:
  - ✔ contention reduced to thread creation/exit
  - ✔ better cache locality
  - ✘ lose strict priority semantics
  - ✘ can lead to load imbalance

Are there better ways?

Multi-processor scheduling (3)

- Processor affinity:
  - modification of central queue
  - threads have *affinity* for a certain processor
    $\Rightarrow$ can reduce cache problems
  - but: load balance problem again
  - make dynamic? (cache affinity?)

- 'Take' scheduling:
  - pseudo-dedicated assignment: idle CPU "takes" task from most loaded
  - can be implemented cheaply
  - nice trade-off: load high $\Rightarrow$ no migration

- Co-scheduling / gang scheduling:
  - Simultaneously schedule "related" threads.
    $\Rightarrow$ can reduce wasted context switches
  - Q: how to choose members of gang?
  - Q: what about cache performance?

Example: Mach

- Basic model: dynamic priority with central queue.

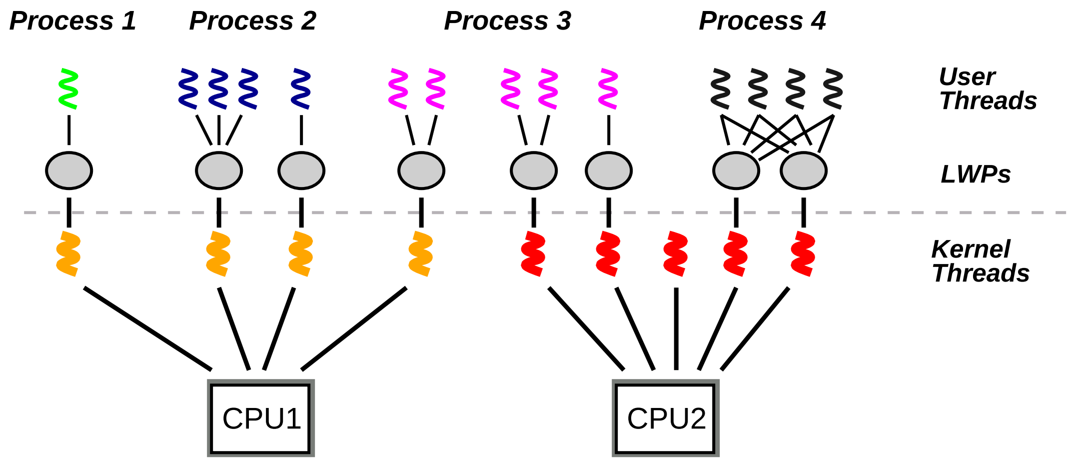
- Processors grouped into disjoint *processor sets*:
  - Each processor set has 32 shared ready queues (one for each priority level)
  - Each processor has own local ready queue: absolute priority over global threads

- Increase quantum when number of threads is small
  - 'small' means #threads $<$ ($2\times$ #CPUs)
  - idea is to have a sensible *effective quantum*
  - e.g. 10 processors, 11 threads
    * if use default 100ms quantum, each thread spends an expected 10ms on runqueue
    * instead stretch quantum to 1s $\Rightarrow$ effective quantum is now 100ms.

- Applications provide *hints* to improve scheduling:
  1. discouragement hints: mild, strong and absolute
  2. handoff hints (aka "yield to") — can improve producer-consumer synchronization

- Simple gang scheduling used for allocation.

MP thread architectures



Hybrid schemes combine user and kernel threads, e.g. *three-level scheduling* in Solaris 2:

- 1 kernel thread $\leftrightarrow$ 1 LWP $\leftrightarrow$ n user threads
- user-level thread scheduler $\Rightarrow$ lightweight & flexible
- LWPs allow potential multi-processor benefit:
  - more LWPs $\Rightarrow$ more scope for true parallelism
  - LWPs can be *bound* to individual processors $\Rightarrow$ could in theory have user-level MP scheduler
  - kernel scheduler is relatively cache agnostic (although have processor sets ($\neq$ Mach's))

Overall: either *first-class threads* (Psyche) or *scheduler activations* probably better for MP

Communication in concurrent systems

Concurrency

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

Two main scenarios:

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

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

Safety

In each of these environments we must ensure that the system remains *safe* – i.e. that ‘nothing bad happens’

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

Liveness

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

Standard problems include:

- *Deadlock* – a circular dependency between processes holding resources and processes requiring them. Typically the resources will be access to mutual-exclusion 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
- *Starvation* – a thread is waiting for some resource but never receives it – e.g. a thread with a very low scheduling priority
- *Distribution failures* – of nodes or network connections in a distributed system

Communication within processes

Shared data

- Most useful multi-threaded applications will share data between threads
- Sometimes this is straightforward e.g. data passed to a thread through fields in the object containing the `run` method
- More generally, threads may share state through...
  - `static` fields in mutually-accessible classes, e.g. `System.out`
  - objects to which multiple threads have references
- What happens to field `o.x`:

Thread A	*Thread B*
`o.x = 17;`	`o.x = 42;`

- Most fields accesses are atomic – the value read from `o.x` after those updates will be either 17 or 42
- The only exceptions are numeric fields of type `double` or type `long` – some third value may be read in that case

Locks in Java

- Simple shared data structures can be managed using *mutual exclusion locks* ('mutexes') and the `synchronized` keyword to control access to *critical sections* within an application

- 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 take out a mutual exclusion lock on the object
- If the lock is already held by another thread then the caller is blocked until the lock becomes available
- Locks operate on a *per-object* basis – that is, only one synchronized method can be called on a particular object at any time
  - ...similarly, it is OK for multiple threads to be calling the same method, so long as they do so on different objects
- Locks are *re-entrant*, meaning that a thread may call one synchronized method from another
- If a `static synchronized` method is called then the thread must acquire a lock on the *class* rather than on an individual *object*
- The `synchronized` modifier cannot be used directly on classes or on fields

Locks in Java (3)

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

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

- The `synchronized` region at line 2 acquires a lock on the object referred to by `x`, performs the `println` operation at line 3 and then releases the lock at line 4

- The lock must be re-acquired at line 8

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

Priority inversion

- All priority-based schemes can potentially suffer from *priority inversion*:
- e.g. consider low, medium and high priority threads called P_l, P_m and P_h respectively.
  1. First P_l starts, and acquires a lock $\mathcal{L}$.
  2. Then other two processes start.
  3. P_h runs since highest priority, tries to lock $\mathcal{L}$ and blocks.
  4. Then P_m gets to run, thus preventing P_l from releasing $\mathcal{L}$, and hence P_h from running.
- Usual solution is *priority inheritence*:
  - associate with every lock $\mathcal{L}$ the priority P of the highest priority process waiting for it.
  - then temporarily boost priority of *holder* of the lock up to P.
  - can use handoff scheduling to implement.
- Windows 2000 “solution”: priority boosts
  - checks if $\exists$ ready thread not run $\geq$ 300 ticks.
  - if so, doubles quantum & boosts priority to 15
- What happens in Java?

Deadlock

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

Thread P

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

Thread Q

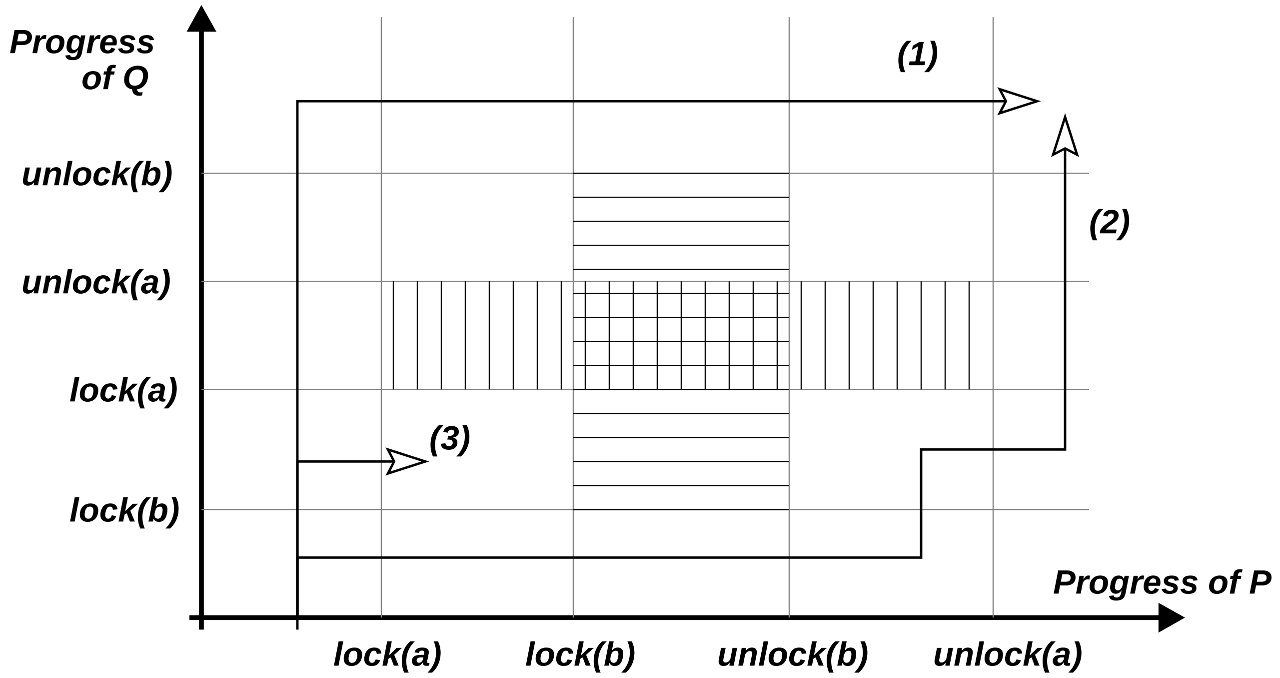
```
synchronized (b)
  synchronized (a)
  {
    ...
  }
```

- If P locks both `a` and `b` then it can complete its operation and release both locks, thereby allowing Q to acquire them
- Similarly, Q may acquire both locks, then release them and then allow P to continue

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

Deadlock (2)

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



The shaded areas indicate (left and right) that one thread is blocked by the other waiting to lock `a` and (above and below) to lock `b`

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

Deadlock is inevitable on path `(3)`

Deadlock (3)

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

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

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

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

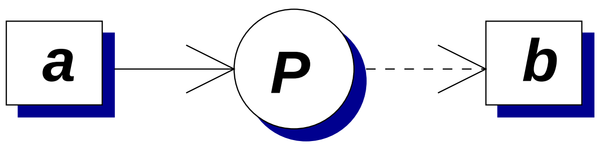
4. Circular wait – a cylce of threads exist such that each holds a lock requested by the next process in the cycle, and that request has been refused

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

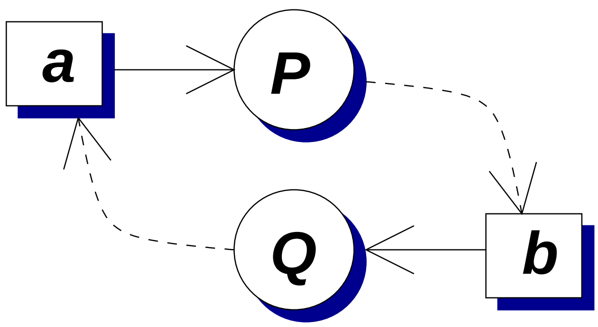
Object allocation graphs

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

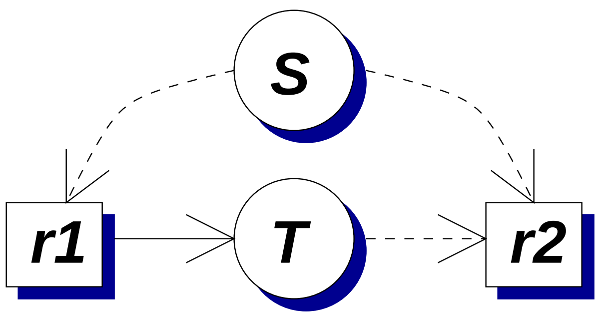
P has acquired object `a` and is requesting object `b`:



P holds `a` and Q holds `b`:



Should `r2` be allocated to S or T?



Deadlock detection

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

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

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

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

1. Select an unmarked row i such that $R_i \leq W$ – i.e. a thread whose requests can be met. Terminate if none

2. Set $W = W + A_i$, mark row i, and repeat

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

- inevitable (must happen in some possible execution path)

- possible (may happen in some path)

Deadlock detection (2)

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

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

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

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

Deadlock avoidance

A conservative approach:

- Require that each process identifies the maxmium set of resources that it may ever lock, $C_{(i,j)}$
- When thread i requests a resource then construct a hypothetical allocation matrix A' in which it has been made and a hypothetical request matrix B' in which every other process makes its maximum request
- If A' and B' do not indicate deadlock then the allocation is safe

✔ It does avoid deadlock – may be preferable to deadlock recovery

✘ Need to know maximum requests

✘ Run-time overhead

✘ What if there are no safe states?

✘ Objects are instantiated dynamically...

Deadlock avoidance (2)

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

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

- Do not hold resources while waiting – e.g. acquire all necessary locks at the same time (not possible as a primitive operation in Java if > 1 lock)

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

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

Limitations of mutexes

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

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

- What can we put in line 10? We could just write code that keeps testing `full` (a 'spin lock'), but at best that would be inefficient (also need to only hold the lock 6–9)

Limitations of mutexes (2)

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

Suspending threads

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

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

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

- This is because the use of `suspend` can lead to deadlocks if the target thread is holding locks. It also risks race conditions if multiple thread use them against the same target

- They should be avoided even if the program does not explicitly take out locks: many systems use additional locks in their implementation

Condition variables

- Java provides *condition variables* for this kind of situation
- There is an individual condition variable associated with each mutex
- Condition variables support two kinds of operation:
  - a *wait* operation causes the current thread to block
  - *notify* operations that cause blocked thread(s) to continue
- In each case there are important details about when the operation can be used and, in the case of *notify*, which blocked thread(s) are woken. We'll first look at an example and then at the details

Condition variables (2)

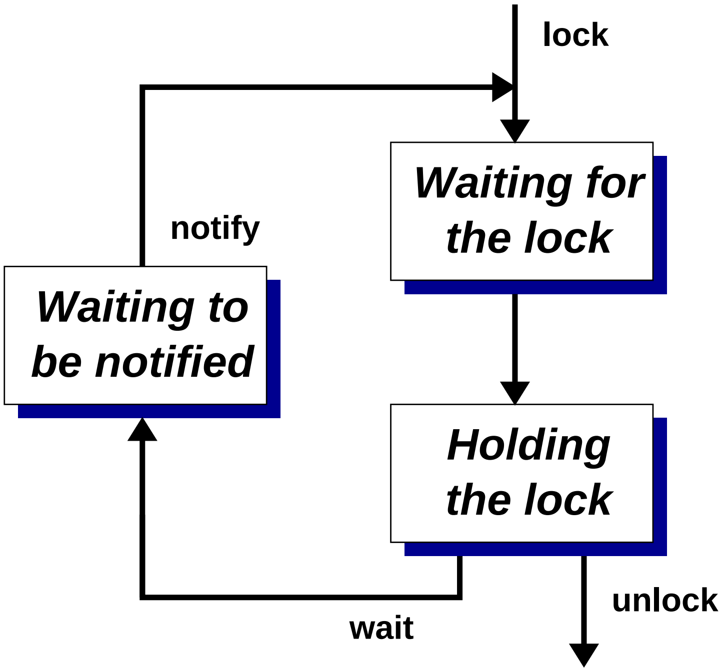
```
class Cell {
  private int value;
  private boolean full = false;

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

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

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

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

Condition variables (3)

- Line 2 causes a thread executing `removeValue` to block on the condition variable until the cell is full
- Line 4 updates the object to mark it empty
- Line 5 notifies any threads currently blocked on the condition variable
- Similarly, line 10 causes a thread executing `PutValue` to block on the condition variable until the cell is empty
- Lines 12–13 update the fields to mark the cell full and store the value in it
- Line 14 notifies any threads currently blocked on the condition variable

Condition variables (4)

- `wait`, `notify` and `notifyAll` are operations defined on `java.lang.Object`
- A thread can only perform a *wait* or *notify* operation on a condition variable when it holds the associated mutex
- When a thread *waits* it atomically releases the mutex and becomes blocked on the condition variable
- When a thread is *notified* it must re-acquire the mutex before it continues



Condition variables (5)

- There are two forms of *notify* operation:
  - `notifyAll()` notifies every thread blocked on the condition variable
  - `notify()` notifies exactly one thread if any are blocked

- Both require care to ensure correct use. With `notifyAll()` the programmer must ensure that every thread blocked on the condition variable can continue safely
  - e.g. line 2 in the example surrounds the `wait()` operation with a `while` loop – if a 'getting' thread is notified when there is no work for it, then it just waits again

- `notify()` selects arbitrarily between the waiting threads: the programmer must therefore be sure that they do not mind which thread is chosen

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

Condition variables (6)

- The example shows a common way of using condition variables

- `notifyAll()` is always used and the application contains extra code (the `while` loop) so that threads `wait()` again if they are woken prematurely

- Although only one thread is expected to continue, `notify()` would be incorrect here because the choice *does* matter: suppose there are some threads blocked in `removeValue()` and some threads blocked in `putValue()`

- When it is safe to use, some programmers prefer `notify()` because it may be more efficient

- If using other programming languages then take care that their versions of these operations behave in the same way

N-slot buffer

```
class NSlotBuffer {
  int spacesFree = SIZE; int spacesUsed = 0;
  Object empty = new Object ();
  Object full = new Object ();

  void insert (int x) {
    synchronized (full) {
      while (spacesFree == 0) full.wait ();
      spacesFree --;
      ...
    }
    synchronized (empty) {
      spacesUsed ++; empty.notify ();
    }
  }

  int remove () {
    synchronized (empty) {
      while (spacesUsed == 0) empty.wait ();
      spacesUsed --;
      ...
    }
    synchronized (full) {
      spacesFree ++;  full.notify ();
    }
  }
}
```

N-slot buffer (2)

This example illustrates a couple of points,

- Firstly, it generalizes the previous one into allow up to `SIZE` `insert()` operations to be performed without intervening invocations of `remove()`

- Secondly, it shows how multiple objects can be used to indicate different conditions

  Each Java object has an associated mutex and condition variable, so instances of `java.lang.Object` are often used for this purpose

  Remember that a thread must acquire a lock on the object before invoking `wait()`, `notify()` or `notifyAll()`

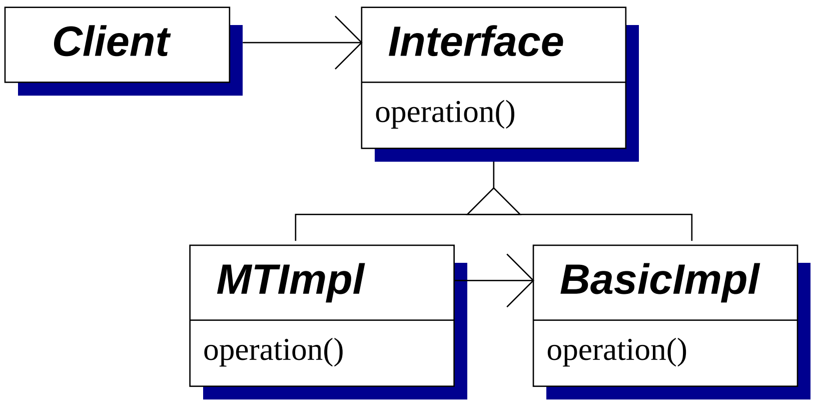
  Do `insert(...)` and `remove(...)` still need to be `synchronized`?

Design

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

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

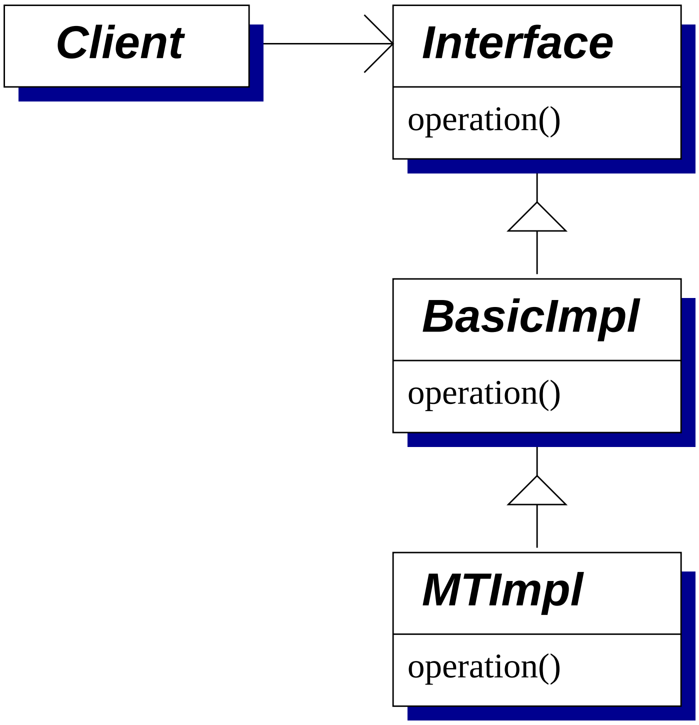
  One option is based on delegation and the Adapter pattern,



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

Design (2)

- How does that compare with:



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

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

✘ Separate sub-classes are needed for e.g. `MTImpl2`

✘ Composition of wrappers is fixed at compile time

Design (3)

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

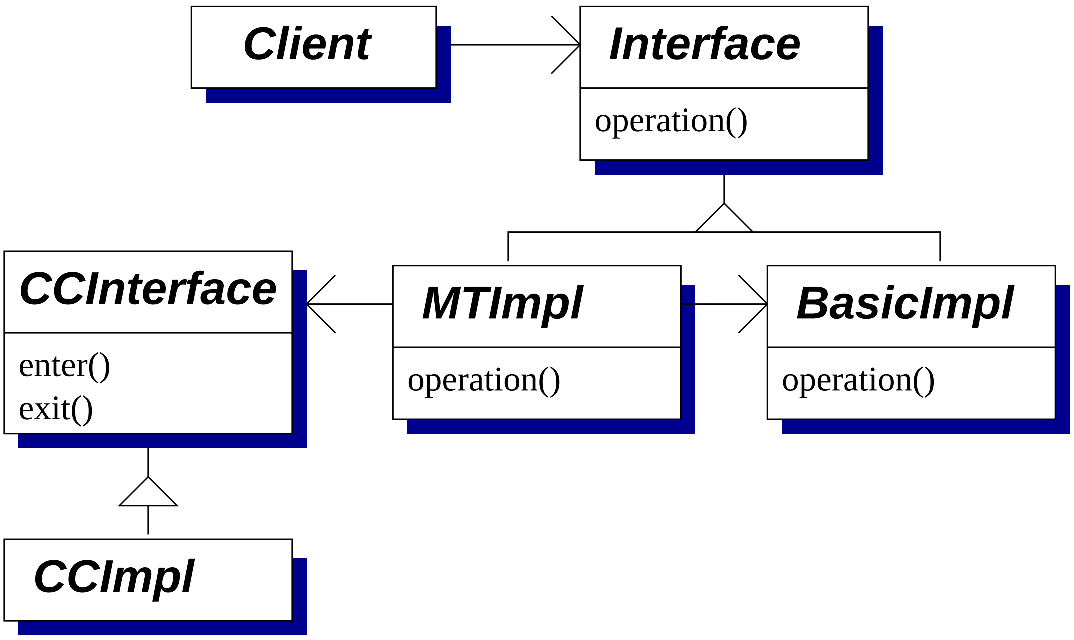
1. An *entry protocol* responsible for concurrency control – usually waiting until it is safe for the operation to continue

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

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

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

Design (4)



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

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

✔ Concurrency-control protocols can be shared

✔ Only a single `MTImpl` class is needed per data structure interface

Multiple readers, single writer

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

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

A reader can enter when `numWriters` is zero. A writer can enter when both fields are zero:

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

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

Multiple readers, single writer (2)

The exit protocols are more straightforward:

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

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

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

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

Is that efficiency likely to be a problem?

Could `notify()` be used instead?

- 1999 Paper 4 Q3

Giving writers priority

We may want to ensure writes are made as soon as possible – how can that be implemented?

```
class MRSWImpl2 implements MRSW {
  int numReaders = 0;
  int numWriters = 0;
  int waitingWriters = 0;

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

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

First-come first-served ordering

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

```
class FCFSImpl implements CCInterface {
  int currentTurn = 0;
  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 ();
  }
}
```

- 2001 Paper 4 Q3

First-come first-served ordering (2)

✔ The implementation is simple

✘ `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 must depend on the context the class is being used in, 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

- *No undetected failures:* would `long`s ever overflow here?

Primitives for concurrency

These examples have used the language-level *mutexes* and *condition variables* exposed in Java.

Semaphores provide simpler operations on which the language-level features could be based. 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 it is less than zero
- `V` (sometimes called *signal*) increments the value and then, if it is zero or less, selects a blocked thread and unblocks it

Programming with semaphores

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

```
class Mutex {
  CountingSemaphore sem;

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

  acquire () {
    sem.P();
  }

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

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

Programming with semaphores (2)

```
class CondVar {
  int numWaiters = 0;
  Mutex cv_lock = new Mutex();
  CountingSemaphore cv_sleep =
          new CountingSemaphore (0);

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

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

Programming with semaphores (3)

Why doesn't Java just provide semaphores?

- They can be implemented using mutexes and condition variables
- Using semaphores directly is intricate – the programmer must ensure `P()`/`V()` are paired correctly
- Many OS provide mutexes and condition variables directly

There are other general problems both with semaphores and with the facilities in Java:

- `wait()` and `notify()` still need care from the programmer
- The usual interfaces do not provide `isLocked()`, `tryToLock()`, `tryToLockUntil(...)` or `lockAny(...)` operations

— how could these be implemented?

Implementing semaphores

Uni-processor only: disable thread switches during the implementation of `P()` and `V()`

All systems: rely on atomic primitive operations provided by the processor to implement simple spin-locks

```
P()    sem.lock()
       sem.val -= 1
       if sem.val<0 block thread
       sem.unlock()

V()    sem.lock()
       sem.val += 1
       if sem.val<=0 unblock a thread
       sem.unlock()
```

Each semaphore has an assocaited value, boolean lock field and blocked-thread queue. The block operation

1. adds the current thread to the blocked-thread queue
2. updates the thread control block (TCB)
3. unlocks the semaphore

Implementing semaphores (2)

How are `lock` and `unlock` implemented?

Almost all processors have atomic operations such as `tas` (test-and-set), `cas` (compare-and-swap) or `ll/sc` (load-linked / store-conditional, not covered here)

```
lock()      tas lock.val, 0, 1
            if failed

unlock()    mov 0 -> lock.val
```

This *spin lock* is not a good solution in general:

✘ uni-processor non-preemptive case...

✘ threads waiting to acquire the lock are continually attempting `tas` operations – they do not block

Spin locks tend to be used when blocking is unlikely or for only short durations, so the time spent spinning is much less than the time taken to block and unblock a thread

Mutexes without hardware support

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

- 'Bakery' algorithm due to Lamport (1974)

```
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 as a tie-break

Event counts and sequencers

The bakery algorithm suffers the same efficiency concerns as a spin-lock using `tas`.

What happens if `n` changes?

However, a similar algorithm can easily be built from *event count* and *sequencer primitives*, proposed as an alternative to semaphores

- An event count is represented by a positive integer, initialized to zero, supporting the following atomic operations:
  - `advance()` – increment the value by one, returning the new value
  - `read()` – return the current value
  - `await(i)` – wait until the value is greater than or equal to `i`
- A sequencer is again represented by a positive number, initialized to zero, supporting a single atomic operation:
  - `ticket()` – increment the value by one, returning the old value

Event counts and sequencers (2)

- Mutual exclusion is easy: a thread takes a ticket entering a critical region and then invokes `await` to receive its turn

- 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

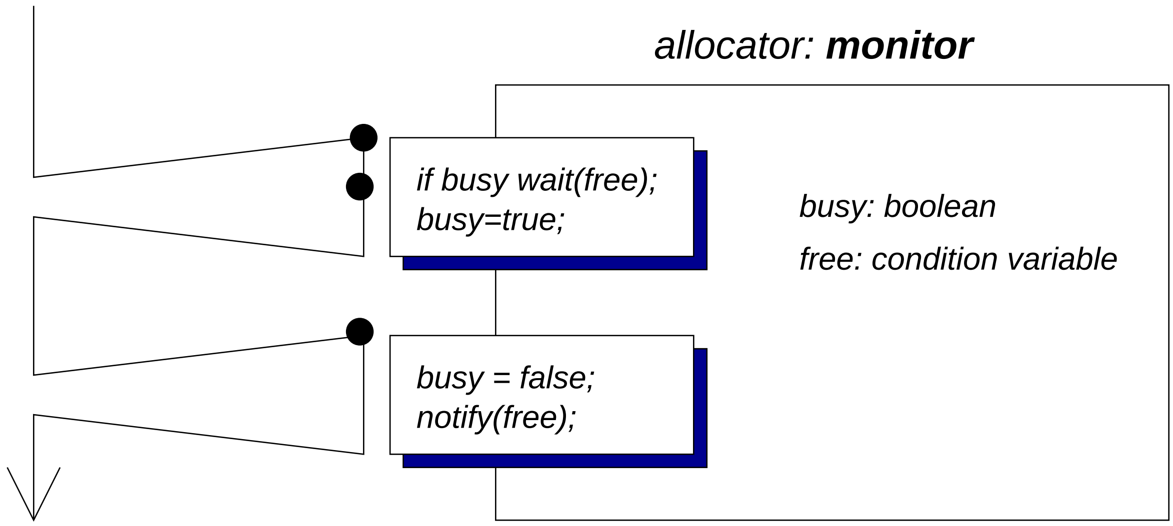
- A general N-slot buffer is more difficult. Two sequencers are used to order producers and consumers to ensure slots are read/written in order

- Note that many operations on event counts and sequencers have a straightforward implementation using `cas` – as before with semaphores, care is needed to avoid missed wake-ups between `await` and `advance`

  `cas x,y,z` atomically compares the contents of location `x` against the value `y`: if they match then `z` is written to `x`, otherwise `x` is unchanged. It's a primitive operations on x86, IA-64 and SPARC processors

Monitors

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



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

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

Active objects

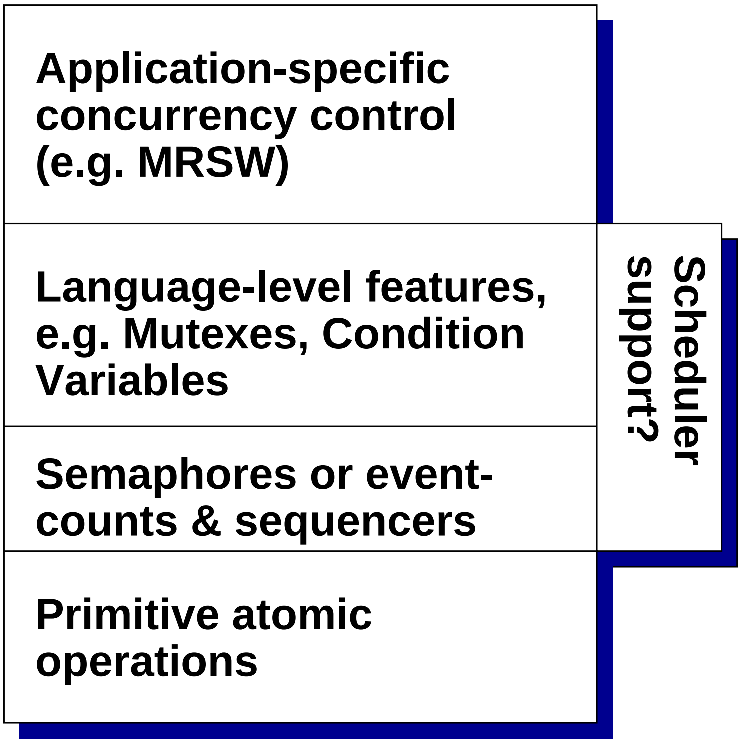
An *active object* achieves mutual exclusion between operations by (at least conceptually) having a dedicated thread that performs them on behalf of external callers, e.g.

```
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

Recap



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
- Whether mutexes, condition variables are provided directly as primitives