Objects and classes

Classes

- A Java program is defined using classes

```
class SimpleExample {
  public static void main (String args[])
  {
    System.out.println ("Hello world!");
  }
}
```

- Simple examples just use one class definition with the program defined in a single `main` method

- We'll now be looking at defining programs
  - where more than one class is used,
  - classes have more than just a `main` method

Why use classes

- Class definitions let you group related data together

```
class Main {
  String names[] = new String[200];
  String numbers[] = new String[200];
}
```

could become

```
class Main {
  NameNumber info = new NameNumber[200];
}

class NameNumber {
  String name;
  String number;
}
```

Why use classes (2)

- Class definitions associate code with the data that it operates on

```
class NameNumber {
  String name;
  String number;

  /* Append "prefix" to the current number */
  void updateNumber (String prefix)
  {
    number = prefix + number;
  }
}
```

Why use classes (3)

- Classes decompose the program into smaller sections that may be re-used between different programs

- For example, the simple *Hello world* example re-uses several classes from the standard Java libraries.
  - `java.lang.String`
  - `java.io.PrintStream`
  - `java.io.BufferedWriter`

- By convention class names are spelled with an initial capital letter and have names that are nouns, e.g. `Shape`, `Car`, `PrintStream`

Objects

- A class definition can be *instantiated* to create an *object*

- Each object holds separate values for each of the fields

```
class NameNumbr {
  String name;
  String number;
}
```

  e.g. one instance of `NameNumber` could have `name` as "Tim" and `number` as 34650, another could have "Austin" and 34645

- More generally, classes define ‘kinds of thing’ whereas objects identify individual things

Object references

- A program manipulates objects by using *object references*

```
1 public static void main (String args[])
2 {
3   NameNumber nn;
4
5   nn = new NameNumber ();
6   nn.name = "Tim";
7   System.out.println (nn.name);
8 }
```

- Line 3 declares a variable of type *reference to NameNumber*
- Line 5 instantiates the class `NameNumber` to create a new object
- Line 6 sets the `name` field in the new object to refer to the string "Tim"

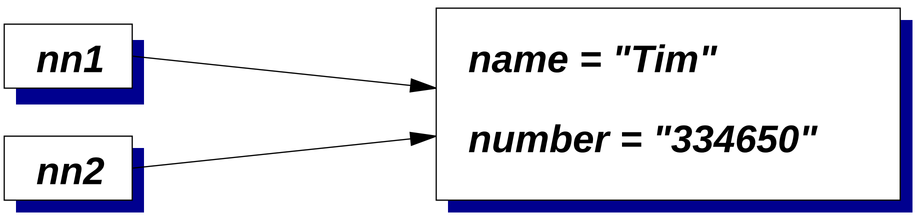
Object references (2)

- 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

```
NameNumber nn1 = new NameNumber ();
NameNumber nn2 = nn1;
```

creates two references to the same object:



- If `nn1.name` is updated then that new value can also be accessed by `nn2.name`

Object references (3)

- Using references allows linked data structures to be created

```
class List
{
   NameNumber nn;
   List       next;
}
```

- In an instance of `List` the reference contained in `next` identifies the successor in the list structure
- Similarly a `Tree` class could contain two fields of reference types

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. `List` in terms of `NameNumber` in the previous example
- 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`

Methods

- Methods define operations that may be performed on objects
- In general objects are expected to interact by calling methods on one another, rather than by accessing fields directly
- Promotes *modularity*: it allows the representation of data to be changed without updating the interface
- By convention method names are spelled with an initial loewr-case letter and have names that are verbs, e.g. `setName` or `incrementValue`

Methods (2)

- The method has access to per-object field values, e.g. if as before:

```
class NameNumber
{
  String name, number;

  void setName (String newname)
  {
    name = newname;
  }


  ...
  NameNumber n1 = new NameNumber ();
  NameNumber n2 = new NameNumber ();
  n1.setName ("Tim");
  n2.setName ("Austin");
  ...
}
```

Methods (3)

- Within the `setName` method, `name` refers to the value of that field in the current object and `newname` to the parameter passed by the caller:

```
void setName (String newname)
{
  name = newname;
}
```

- Fields within the current object can also be accessed by writing `this.fieldname`, e.g. `this.name`

- Doing so is useful where the same name is used for both a field and for a parameter or local variable:

```
void setName (String name)
{
  this.name = name;
}
```

Overloaded methods

- The same name can be used for more than one method in any class. They are said to be *overloaded*
- ...however, they must have distinct parameter types to disambiguate which one to call,

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

void setNumber (int number)
{
  this.number = new String (number);
}
```

Overloaded methods (2)

- It is insufficient to merely have distinct return types, e.g. the following code is incorrect:

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

String setNumber (String number) {
  this.number = number;
  return "Set to " + number;
}
```

- How should the following invocations behave?

```
String s = nn.setNumber ("First");
nn.setNumber ("Second");
```

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

Overloaded methods (3)

- 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

- The previous examples have used methods for initializing fields, e.g.

```
void setName (String name) {
  this.name = name;
}

...

NameNumber nn = new NameNumber ();
nn.setName ("Tim");
```

- The programmer has to remember to do all of the initialization steps
- If the class is modified with extra fields then there is no checking that they are ever set up properly
- An alternative is to use *constructors*

Constructors (2)

- Constructors define pieces of code to be executed whenever `new` is used to instantiate a class

```
class NameNumber {
  String name;
  String number;

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

- The programmer can now write

```
NameNumber nn = new NameNumber ("Tim",
                                "34650");
```

Constructors (3)

- 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`

- This typically models a *has a* relationship and often allows objects within the program to reflect the associations between the 'real' objects that they represent

- *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 (2)

- 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, e.g. `ColouredShape` may extend `Shape` and add a further field to hold the colour,
  - add new methods, e.g. `setColor` in the previous example,
  - 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

- Previous examples have used reference variables whose type is associated with a particular class:

```
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`

- This continues transitively, e.g. if `C extends B` or `D extends C`, etc

- The intent is that instances of a subclass support *at least* the operations of their superclass and can consequently be used wherever and object of the superclass is expected

Types and inheritance (2)

- A particular object may be accessed through fields of different *reference types* over the course of its lifetime

- In contrast, it is important to remember that an object's *class* is determined at its time of creation – it does not change

- *Casting* operations convert references to an object between different reference types, e.g. if `B extends A` then we can write

```
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`, but the variable `ref2` may only refer to instances of `B`

Types and inheritance (3)

- A reference to a particular object can be cast to any reference type corresponding to the object's class or (transitively) a superclass

- Cast operations are checked at compile-time (where possible) and otherwise at run-time

- Other programming languages (e.g. C++) allow unchecked casts

- In Java any definition without an explicit super-class will implicitly extend `java.lang.Object`

- Consequently reference variables of that type can refer to any object at all (and also to any arrays)

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

- Instances of the sub-class contain all of the fields of their super-class, e.g. with `NameNumberPlace`:

```
NNP nnp = new NameNumberPlace ();
nnp.name = "Tim";
nnp.number = "34650";
nnp.place = "Room A205";
```

- The field `place` is defined in the class `NameNumberPlace`, the fields `name` and `number` are *inherited* from `NameNumber`

- 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`.

Fields and inheritance (2)

- 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");
  }
}
```

Methods and inheritance

- What happens when a method is called?

```
class A {
  void f() {
    System.out.println ("A");
  }
}

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

...

A object_1 = new A();
B object_2 = new B();
A object_3 = new B();
object_1.f(); // outputs "A"
object_2.f(); // outputs "B"
object_3.f(); // outputs "B"
```

Methods and inheritance (2)

- In these examples the selected method depends on the *class* of the *target* object

```
1   class A {
2     void method1() {
3       this.method2 ();
4     }
5
6     void method2() { }
7   }

8   class B extends A {
9     void method2() { }
10  }

11  (new A()).method1();
12  (new B()).method1();
```

- Line 11 calls 2 and then 6 because it is an object of class `A`. Line 12 calls 2 and then 9 because it is an object of class `B`

Methods and inheritance (3)

- 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();
```

Constructors and inheritance

- Sub-classes *do not* inherit constructors from their superclass
  - ...because constructors are intended for initialization and this typically differs in the sub-class
- A sub-class constructor must start by either invoking another constructor in the sub-class, or a constructor in the superclass
  - This ensures that any required initialization has been performed by each class in the hierarchy
  - Doesn't guarantee termination – e.g. a valid (but perhaps not very useful) class could have one constructor that calls itself

Packages

- Previous examples have used short class names such as `NameNumber` and `ColouredShape`
  - What happens if multiple programmers want to use the same name?
- Java groups classes into *packages*. Classes within a package are typically...
  - written by co-operating programmers
  - expected to be used together
- Packages are organized hierarchically – e.g. a package for a large project may contain packages for different modules

Packages (2)

- Package names are generally prefixed by an internet DNS name, in reverse order, e.g. `com.sun` or `uk.ac.cam`
- After this prefix an organization is able to use whatever naming scheme it wants, e.g. I might use package names starting with `uk.ac.cam.cl.tlh20`
- 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`

Packages (3)

- 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 NameNumber {
    ...
}
```

- Referring to fully-qualified type names is inconvenient. Definitions in the same package can be accessed without the package name. Definitions in the `java.lang` package can always be accessed. Otherwise, the `import` keyword can be used:

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

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

- There are 11 modifiers available:
  - `final` (classes, methods, fields)
  - `abstract`, `strictfp` (classes, methods)
  - `native` (methods)
  - `static` (methods, fields, some classes)
  - `public`, `protected` and `private` (classes, methods, fields)
  - `synchronized` (methods)
  - `transient`, `volatile` (fields)

- `synchronized` and `volatile` will be considered later, along with multi-threaded applications

- `transient` will be considered in the final section, along with distributed applications

Modifers (2)

```
<class-modifiers> class NameNumber {

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

  NameNumber () {
    /* No constructor modifiers are allowed */
  }

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

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

- More than one modifier can be used with each definition – not all combinations are valid

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 = "Fixed value";
  final String B;

  FinalField () {
    B = "Other fixed value"
  }
}
```

The `abstract` modifier

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

```
1   public class A {
2     abstract int methodName ();
3   }
4
5   public class B extends A {
6     int methodName () {
7       return 42;
8     }
9   }
```

- The method declared in line 2 is `abstract` – note that no code is supplied. The method in line 6 over-rides it and provides an implementation

The `abstract` modifier (2)

- An `abstract` class cannot be instantiated. A class containing an `abstract` method is implicitly defined `abstract`

- 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 `strictfp` modifier

- A `strictfp` method is implemented at run-time using IEEE 754/854 arithmetic (see *Numerical Analysis 1*) – identical results are guaranteed on all computers

- Otherwise, the machine's own floating point arithmetic format may be used – typically faster, but results may differ. The differences are important for some numerical algorithms

- If a class is defined `strictfp` then all of the methods defined in the class are implemented using IEEE arithmetic – as are all initialization expressions and constructors

The `native` modifier

- The `native` modifier may only be applied to methods – it means that the implementation of the method is provided is some language other than Java and is not included in the class definition

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

- Typically used for access to machine-specific devices, for interacting with existing code or for performance reasons
- Limits portability: native code is machine/operating-system specific and is distributed separately from the Java `.class` files

The `static` modifier

- The `static` modifier can be applied to any method or field definition. (It can also be applied to *nested* class, 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 `NameNumber` 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

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;
  }
}
```

The `static` modifier (3)

```
1 String instanceMethod () {
2   return name + " " + instantiationCount;
3 }

4 static String staticMethod1 (Example e1) {
5   return e1.name;
6 }

7 static String staticMethod2 () {
8   return name;
9 }
```

- Line 2 is valid: an instance method can access fields in that object (`name`) and static fields in that class (`instantiationCount`)
- Line 5 is valid: the access to field `name` is accessed through an object reference (`e1`)
- Line 8 is invalid: the method is static and cannot access a non-static field (`name`)

The `static` modifier (4)

- `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, e.g.

```
class A {
  static void a() { }
  static void b() { a(); } // Calls A.a
}

class B extends A {
  static void a() { }
  static void c() {
    a();   // Calls B.a
    A.a(); // Calls A.a
    b();   // Calls A.b
  }
}
```

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
- Java provides facilities for enforcing some forms of access control
- This can be used to ensure that encapsulation is honoured and also, in some standard libraries, to ensure that untrusted downloaded code executes safely
- The access modifier applied to a constructor controls where the class (or arrays of that class) may be instantiated

Access modifiers (2)

- The `public`, `protected` and `private` modifiers provide increasing restrictions to the access granted to a class, method, field or constructor
- A fourth *default* access level applies to definitions where none of the modifiers are used
- `public` can be applied to classes, methods, fields and constructors. A `public` entity is accessible from anywhere within the Java application
- An entity with *default* access is accessible in the package within which it is defined
- A `private` entity is accessible within only the class that it is defined

The `protected` modifier

- The rules for `protected` access are more complex.
- `protected` can be applied to methods, fields and constructors. A `protected` entity is always accessible in the package within which it is defined
- Additionally, it is accessible within sub-classes (`S`) of the class (`C`) within which it is defined, so long as
  - protected constructors may only be called on the current object from the constructors of `S`,
  - protected fields may only be accessed on instances of `S` or its sub-classes,
  - protected methods may only be invoked on instances of `S` or its sub-classes

The `protected` modifier (2)

- Suppose these classes are defined in different pacakges:

```
1   public class A {
2     protected int field1;
3   }

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

- Lines 6 and 7 are correct: the field access is to an instance of `B` or one of its sub-classes
- Line 8 is incorrect: the object referred to by `a_ref` may be an instance of `A`

Access modifiers – summary

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

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
- Previous examples have used *inheritance* and *abstract classes* to move common functionality into super-classes such as `Collection` and `Set`

Interfaces (2)

- Is that form of inheritance sufficient for all uses?
  - 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 (3)

- 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 SetImplementation implements 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`

Interfaces (4)

- 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)
- A variable or field of an interface reference type (e.g. `Set` in the previous example) may refer to any instance of any class that implements that interface

Abstract classes v interfaces

- When should abstract classes be used and when should interfaces be used?
- Abstract classes allow some method implementations to be defined – no method implementations can appear in an interface
- A class can only extend a single abstract class – it can implement more than one interface
- In general an abstract class is used to model something whose implementation cannot be completely supplied. An interface is used to model a way of interacting with something

Nested classes

- A *nested* class/interface is one whose definition appears inside another class or interface
- There are four cases:
  - *inner classes* in which one class definition occurs within another
  - *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

inner classes $\subset$ nested classes

Nested classes (2)

- 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

- If permitted by the access modifiers then a nested class can be accessed by qualifying its name with that of its enclosing class, e.g. `A.B.C` in the following example:

```
class A {
  class B {
    class C {
    }
  }
}
```

Inner classes

- 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 keep track of the associated `Wheel` instances, if it wishes to do so

Static nested classes

- 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
  - For example the `ExpandedIcon` and `CollapsedIcon` classes in the *Swing* toolkit are `static` classes defined within the classes that represent *tree* user-interface components

```
class WindowsTreeUI {
  static class ExpandedIcon {
    ...
  }
}
```

Nested interfaces

- Nested interfaces are implicitly declared static
- As with ordinary interfaces they define a set of methods required of any class that implements the interface – they are never instantiated and so it does not make sense for them to be associated with an enclosing instace

Anonymous inner classes

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

```
1  class A {
2    void method1 () {
3      Object ref = new Object () {
4        void method2 () { };
5      };
6    }
7  }
```

- Lines 3–5 define an anonymous inner class that extends `Object`

- The class inherits the standard methods of `Object` and Line 4 defines the additional method `method2`

- Code in the inner class may access fields of the enclosing instance and any `final` local variables of the enclosing method

Anonymous inner classes (2)

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

```
1   interface Ifc {
2     void InterfaceMethod ();
3   }
4
5   class A {
6     void method1 () {
7       Ifc i = new Ifc () {
8         void InterfaceMethod () {
9         };
10      };
11    }
12  }
```

- The class defined in Lines 7–9 *implements* the interface `Ifc` and consequently must define an `InterfaceMethod` as specified in Line 2

Exceptions

Signalling errors

- Many methods need some way of signalling that they have gone wrong
  - e.g. that a file they wished to access did not exist, or that one of their input parameters was out of range
- One way is to print an error message and terminate the program, e.g.

```
class RealMaths {
  public static int sqrt(int x) {
    if (x < 0) {
      System.err.println ("Need +ve input");
      Ststem.exit (1);
    } else {
      ...
    }
  }
}
```

Signalling errors (2)

- This is unsatisfactory in large programs: the caller of the method may wish to recover from the error – e.g. to retry it with some other parameter. An alternative is to reserve particular result values to mean 'something has gone wrong', e.g.

```
class RealMaths {
  public static int sqrt(int x) {
    if (x < 0) {
      return -1;
    } else {
      ...
    }
  }
}
```

- This is also problematic: what if there are no 'spare' values? How does the caller know what a particular value means? They must also make an explicit test each time the method is called

Exceptions

- Java provides *exceptions* in an attempt to provide a consistent mechanism for signalling errors

- Each method in a program may either return *normally* (as in previous examples) or return *abnormally* by 'throwing' an exception

```
class RealMaths throws Exception {
  public static int sqrt(int x) {
    if (x < 0) {
      Exception e;
      e = new Exception("Need +ve input");
      throw e;
    } else {
      ...
    }
  }
}
```

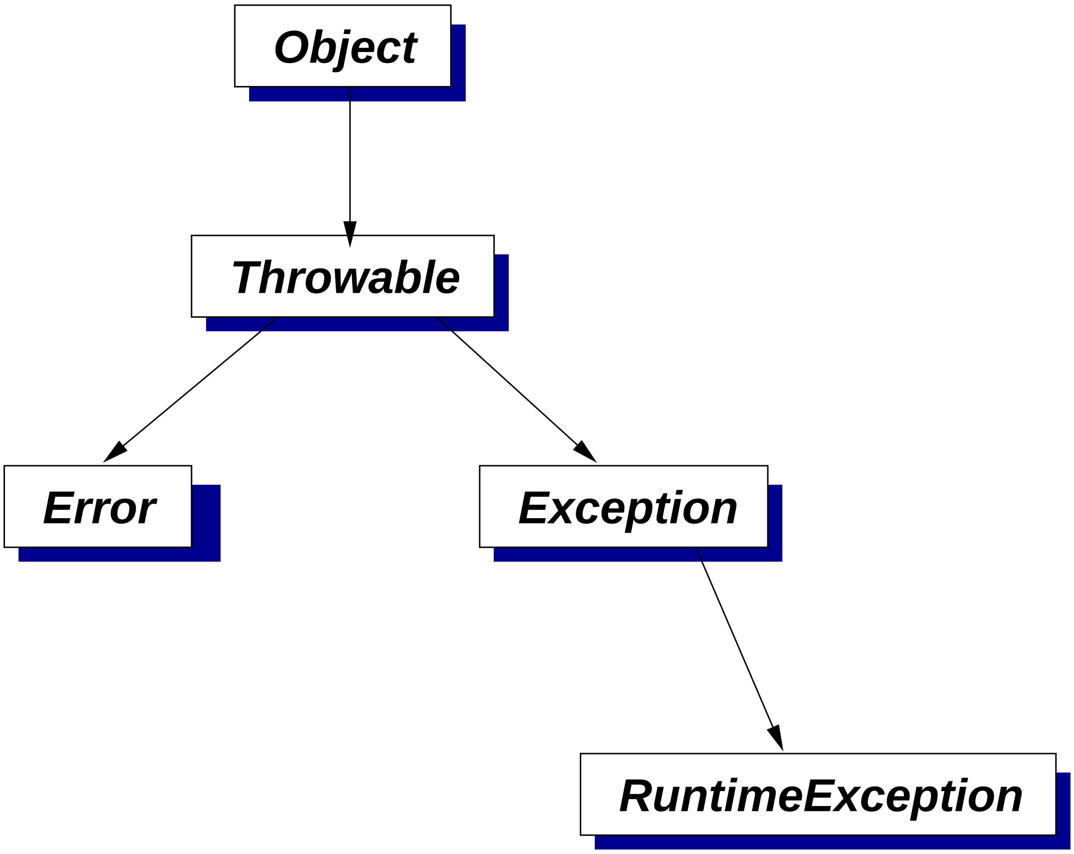
Exceptions (2)

- An exception is just an object that is instantiated from `java.lang.Throwable` or one of its sub-classes

- The standard libraries pre-define a number of further kinds of exception. These are intended to be used as follows:
  - `Error` extends `Throwable` and is intended for errors that may be generated in many places throughout an application – e.g. a division by zero
  - `Exception` also extends `Throwable` and is intended for other kinds of error
  - `RuntimeException` extends `Exception` and is intended for internal use by the runtime system

Exceptions (3)

- These classes are organized as follows:



- A programmer is free to sub-class any of these, but in practise will use `Error` or `Exception`

Throwing exceptions

- An exception is 'thrown' at the point at which a problem is detected

- As we've seen before, this is done using the `throw` keyword: anything of type `Throwable` may be thrown

- Exceptions of type `Error` or `RuntimeException` are said to be *unchecked*

- Other exceptions are *checked*, meaning that a method definition must contain a `throws` clause listing their types,
  - e.g. `throws Exception` in Line 1 of the example

Throwing exceptions (2)

- A Method's `throws` clause cannot be *more permissive* than that of any method it overrides, e.g. suppose `E1` and `E2` are checked exception classes and `E2` extends `E1`

```
class Superclass {
  void Method1 () throws E2 { ... }
}

class ValidSubclass extends Superclass{
  void Method1 () throws E2 { ... }
}

class InvalidSubclass extends Superclass{
  void Method1 () throws E1 { ... }
}
```

Catching exceptions

- When an exception is thrown, the run-time system tries to find code to *catch* it – that is, to deal with the problem that the exception represents

- Exceptions are handled by `catch` statements, e.g.

```
1  try {
2      ...
3  } catch (E2 e2) {
4      ...
5  } catch (E1 e1) {
6      ...
7  }
```

- If code in Line 2 throws an exception of type `E2` then this is handled at Line 4

- Similarly, if an exception of type `E1` is thrown then this is handled at Line 6

Catching exceptions (2)

- The run-time system finds the `catch` expression that most-closely encloses the point at which the exception was thrown

- It no handler is found then the search continues to the method's caller

```
void method3 throws Exception {
  throw new Exception ();
}

void method2 throws Exception {
  method3();
}

void method1 {
  try {
    method2 ();
  } catch (Exception e) { ... }
}
```

Catching exceptions (3)

- In the above example, `method2` is said to *propagate* the exception that is thrown in `method3` and eventually handled in `method1`
- A method's `throws` clause must include any exception types that it propagates, as well as those that it throws directly
- This allows one `try ... catch` statement to handle exceptions generated with multiple causes
- ...in practise doing so usefully requires careful programming to make sure that each different exception is handled appropriately

Cleaning up

- A futher kind of `try` statement exists to support cleaning up after an exception is thrown

```
1 try {
2   ...
3 } finally {
4   ...
5 }
```

- In this example, first of all the code in Line 2 is executed and then – whether or not this throws an exception – the code in Line 4 is executed

- This may be used to make sure that data structures are consistent

- Once the instructions in the `finally` block complete, the exception is handled as normal

(Ab)using exceptions

- Some programmers exploit the way that exceptions such as `NullPointerException` or `ArrayIndexOutOfBoundsException` are ued, e.g.

```
try {
  for (int i = 0;; i ++) {
    System.out.println (array[i]);
  }
} catch (ArrayIndexOutOfBoundsException e) {
  /* Nothing */
}
```

- This is not a good way to iterate along an array:
  - Throwing a single exception may be much slower than doing the correct array-length test
  - Exceptions caused within `println` will be silently discarded

Concurrency

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

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

- Lines 13–14 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...

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 12–13 instantiate two `Thread` objects, passing a reference to an instance of `MyCode` to them
- Lines 14–15 start 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

Thread groups

- Each thread belongs to a *thread group*
- Thread groups may be nested within one another
- A maximum priority may be specified for all of the threads within a group
- Each thread group is represented by an instance of `java.lang.ThreadGroup`. This supports operations to add and remove threads from the group and to iterate over the contents of the group

Thread scheduling

- Some aspects of threads' behaviour vary between different systems
  - The choice of exactly which thread(s) execute at any given time can depend on the operating system and on the JVM
- The Java language specification says that, in general, threads with high priorities will run in preference to those with lower priorities
- Some systems are *pre-emptive* – i.e. they switch between the threads that are eligible to run
- Other systems are *non-pre-emptive* – i.e. they only switch when the running thread yields, becomes blocked or exits

Thread scheduling (2)

- 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 control 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-pre-emptive 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

Terminating a thread

- Threads exit when their `run` method completes

- 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 ();
```

- The thread is not expected to handle `ThreadDeath` – it should be propagated all the way to the `run` method and that method will consequently exit

Terminating a thread (2)

- Beware: when using `stop` the exception may be delivered to the target thread when it is executing a `finally` block
  - This can cause data structures to be left in an inconsistent state
- `stop` is deprecated in newer version of Java: it should not be used in new programs
- The preferred alternative is to use the `interrupt` method on `java.lang.Thread`
- A thread is responsible for periodically calling `isInterrupted` on `java.lang.Thread` to test whether it has been interrupted
- If a thread is interrupted while it is blocked (e.g. during `sleep`) then that operation terminates by throwing `InterruptedException`

Terminating a thread (3)

- For example:

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

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 9 to finish. The parameter specifies a time in milliseconds (0 $\Rightarrow$ wait forever)

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

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

- Java provides several facilities for co-ordinating access to shared data structures. These can be used to implement the kind of higher level facilities described in the *Concurrent Systems* course

Shared fields

- Suppose that multiple threads access a field at the same time...

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

What happens to the field accessed by `o.x`?

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

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

Shared fields (2)

- There are also potential pit-falls when reading data from a shared field:

Thread A	*Thread B*
`o.x = true;`	`do {` `} while (o.x == false)`

Will the loop in Thread B ever end?

- Answer: it may never see the update to `o.x`, even if a long time elapses since Thread A's update

- There are complex rules about how updates made by one thread become visible to other threads (*Threads and Locks* chapter of the language spec)

Shared fields (3)

- In general the threads must to *something* to make sure that they interact correctly

- One option is to use the `volatile` modifier on a field, e.g.

```
class A {
  volatile int value;
}
```

- All updates to volatile fields are atomic

- When a method reads a value from a volatile field then it is actually read from memory every time

- A more flexible scheme is to use *locks* and *condition variables*

Locks

- Simple shared data structures can be managed using *mutual exclusion locks* ('mutexes') and the `synchronized` keyword

- 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 ensure that data remains consistent between multiple fields:

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

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

Locks (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 syncrhonized 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 (3)

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

```
synchronized void methodA () {
  methodB ();
}

synchronized void methodB () {
  ...
}
```

- 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

Synchronized regions

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

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

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

- The lock must be re-acquired at line 5

Limitations of mutexes

- Locks alone cannot solve all problems! Suppose we want a one-cell buffer with a `put` operation (store something if cell empty) and a `get` operation (read something if anything there):

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

- What can we put in line 10?

Limitations of mutexes (2)

- The problem here occurs when a thread does `getValue`, finds nothing there and wants to block until another thread does `putValue`
- We've previously seen two ways that a thread may block...
- ...but we don't want to use `Thread.suspend` because that's unsafe
- ...and can't use `synchronized` because the thread is already has a lock on the cell in question
- We could just write code that keeps testing `full` (a 'spin lock'), but at best that would be inefficient

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
  - a *notify* operation causes 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)

```
public synchronized int getValue () {
  while (!full) wait ();

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

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

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

Condition variables (3)

- Line 2 causes a thread executing `getValue` 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` – in the example they are therefore being invoked on the object referenced by `this`
- A thread can only perform a *wait* or *notify* operation on a condition variable when it holds the associated mutex
  - Observe how the `wait` and `notifyAll` invocations all occur within `synchronized` methods
- 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

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 `getValue()` 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

Graphical interfaces

Graphical interfaces

- The *Abstract Window Toolkit* (AWT) defines classes that can be used for creating graphical applications in Java
- We'll look at it for a number of reasons,
- ...firstly to introduce the facilities it provides and show how they can be used
- ...secondly because its design illustrates how many of the object-oriented features of the Java programming language can be used

Applets

- The simplest way to create graphical applications is to write them as Java *applets* and to execute them using the *appletviewer*, e.g.

```
import java.awt.*;

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

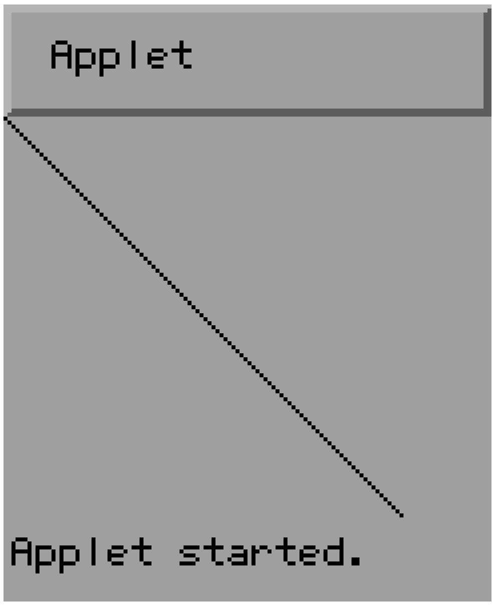
In `E1.html`:

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

Applets (2)

- Running the previous example...

```
$ appletviewer E1.html
```



Applets (3)

- In this case the *appletviewer* renders a web page defined in the file `E1.html`, including the applet defined in `E1.class` at a size of 100x100 pixels

- The applet's `paint` method is called whenever part of the window needs to be re-displayed. The `java.awt.Graphics` class defines various primitive graphical operations, such as

```
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);
```

AWT in applications

- Using applets and operations on an instance of `Graphics` is not always the best approach...

1. The environment within which applets are executed is more restrictive than for ordinary applications
2. The programmer must create and distribute an associated `html` page along with the Java `class` files
3. The appletviewer automatically creates a new top-level window on the display, with its own menus etc – this lack flexibility

- We'll now look at graphical *applications*

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

Containers

- *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 tile 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`

Containers (2)

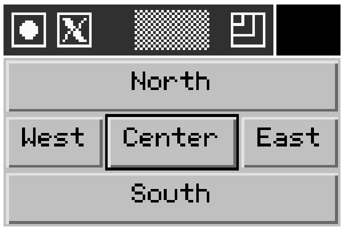
- Components are organized within a container under the control of a *layout manager* using the `setLayout` method on `Container`, e.g.

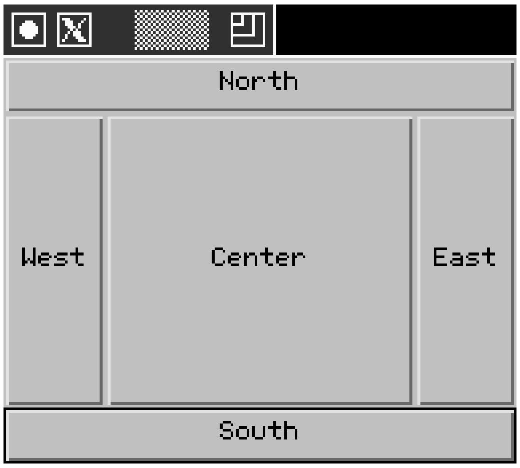
```
import java.awt.*;
public class Buttons extends Frame {
  public Buttons() {
    super();
    setLayout(new BorderLayout());
    add("North",  new Button("North"));
    add("South",  new Button("South"));
    add("East",   new Button("East"));
    add("West",   new Button("West"));
    add("Center", new Button("Center"));
  }

  public static void main (String args[]) {
    Buttons b = new Buttons();
    b.pack();
    b.show();
  }
}
```

Containers (3)

- The `pack` method on `Window` and `Frame` causes all of the subcomponents to be laid out at the 'preferred' size. If the window is re-sized then the layout managers are invoked again to decide how to render it, e.g.





Containers (4)

- `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

Containers (5)

- In practise windows are often designed using multiple containers: it's often simpler to nest containers (particularly instances of `Panel`) rather than to use the more complex `GridBagLayout` manager

- The nesting of containers defines a *spatial hierarchy* within the window

- This kind of design also promotes code re-use

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

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

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

API documentation

- These notes provide a brief overview of the main components available
- They are not intended to be complete, but should provide some pointers to where to look in the full API reference documentation

  `http://www.cl.cam.ac.uk/javadoc/jdk/docs/api/tree.html`

Buttons

- Instances of `java.awt.Button` 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

Canvas

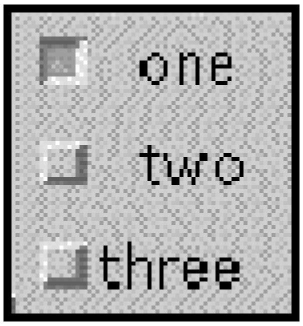
- Instances of `java.awt.Canvas` represent blank rectangular areas of the screen onto which the application can draw

- In practise `Canvas` must be sub-classed and the `paint` method over-ridden. As with the original *applet*-based examples, the paint method is passed a reference to a `Graphics` object on which it can invoke rendering operations

Checkbox

- 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"));
```



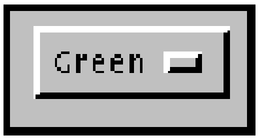
- Check boxes can be grouped together (only one may be *on* in each group) with `setCheckboxGroup`

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

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

Label

- A `java.awt.Label` object represents a single line 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

- Instances of `java.awt.List` represent scrollable lists of text items. The programmer can control how many items are visible at once. For example:

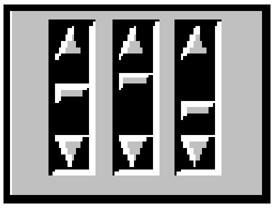
```
List lst = new List(4, false);
lst.add("La-la"); lst.add("Po");
lst.add("Dipsy"); lst.add("Tinky-winky");
```

- 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

- The `java.awt.Scrollbar` class embodies a scroll bar, 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

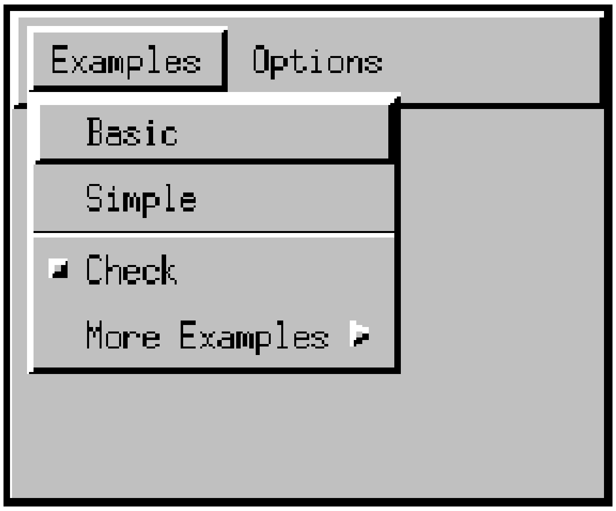
- `java.awt.TextComponent` is a super-class for `TextArea` and `TextField`
- It defines methods `getText` and `setText`, control over whether the text is editable by the user and whether some portion of the text is selected
- `TextArea` and `TextField` are multiple and single-line text regions, e.g.

```
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

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



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

Reflection

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

Reflection (2)

- These facilities are often used 'behind the scenes' in the Java libraries – we'll see some examples later when looking at remote method invocation

- They are also used in some 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 (3)

- 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 (4)

- 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 (5)

- 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 (6)

- 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

- In Java 1.1 only `public` fields are returned. Later versions of Java provide other mechanisms for granting some applications access to all fields

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

Fields and reflection (2)

```
public class ReflExample3
{
public static int field1 = 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
- The result is also returned as an object reference and may need unwrapping – e.g. invoking `intValue()` on an instance of `Integer`

RMI

Remote method invocation

- Remote method invocation (RMI) is a good example of how the reflection facilities are used in Java

- RMI allows a thread running in one JVM to make method invocations on services exported by another JVM

- This might be another JVM running on the same machine (e.g. two different processes running on `hammer.thor`) or it might be a JVM running on a completely separate machine

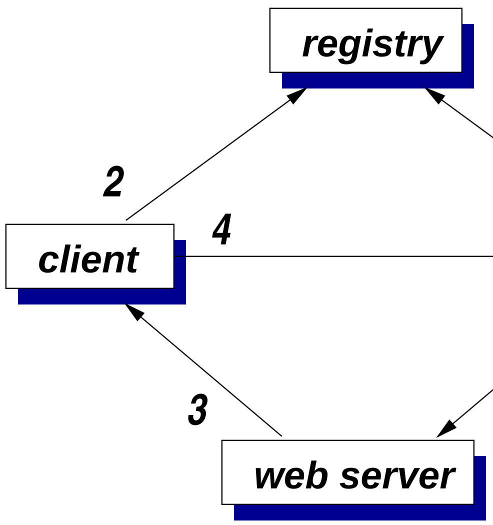
- The infrastructure that supports RMI tries to simplify the way in which it appears to the programmer: aside from some initialization code a remote invocation looks a lot like an ordinary method call

Remove method invocation (2)

- RMI is an example of a *client-server* based distributed system
- A server *exports* a number of *remote objects*
- A client *imports* references to these remote objects and, when it calls a method on one of them, the call is executed remotely on the server rather than locally on the client
- Remote objects are accessed through *remote interfaces*: these are ordinary Java interfaces that extend `java.rmi.Remote` and where all of the methods are declared to throw `java.rmi.RemoteException`

— the 'remoteness' of the invocation can't be made completely transparent, e.g. because of network failures

Overview



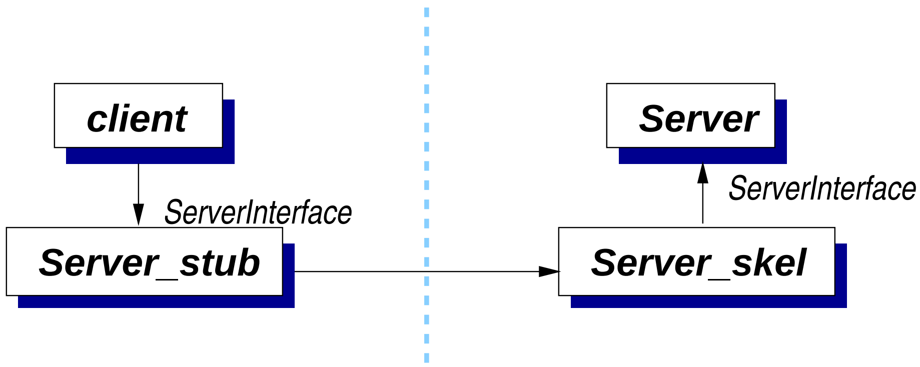
1. A server registers a reference to a remote object with the *registry* and deposits associated `.class` files with a web server

2. A client queries the registry to obtain a reference to a remote object

3. The client obtains the `.class` files associated with the remote object from a web server

4. The client makes an RMI invocation on the remote object

Overview (2)



- The remote object itself is defined as a class that `implements` the corresponding remote interface and that `extends` `java.rmi.server.UnicastRemoteObject`

- In addition to the remote interface and remote object, the programmer must create 'stubs' and 'skeletons' that are responsible for converting local method invocations into remote ones. They are generated from the class of the remote object using the `rmic` tool:

```
$ rmic Server
```

Parameters and results

- Parameters and results are generally copied when passed or returned over RMI – the reflection API is used to determine the field values etc
- Changes made to the copy do not effect the original
- However, remote objects are passed *by reference* and so both caller and callee will interact with *the same* remote object if a reference to it is passed or returned
- Note that Java only supports Remote *Method* Invocation – changes to fields must be made using `get/set` methods

File organization