Introduction to Java

Perdita Stevens, University of Edinburgh

April 2010

Agenda

- Defining a class: attributes, methods
- Creating instances, message passing
- Static members
- Basic data types and control structures
- Interfaces, inheritance, dynamic binding
- Exceptions
- *Inner classes*
- *Genericity, and collection classes*
- *Streams, I/O*
- *Events and threads*
- *Building GUIs with Swing*
- *The rest of the Java SE libraries*

Java is...

- imperative
- object-oriented
- class-based
- statically typed
- “write once, run anywhere” virtual machine based
- open source (mostly, since 2006)
- a big language
- well supported by tools, libraries etc.

Learning more Java

The Sun, oops, Oracle Java tutorial trails are good. (Most code examples in these lectures are taken from them.)

`http://java.sun.com/docs/books/tutorial/`

Reference: `http://java.sun.com/reference/`

Zillions of good books...

Defining a class

```
public class Foo {
  private String s;
  public Foo() {s = "Hello World";}
  public void wibble(String t) {s = t;}
}
```

To note already

Public class Foo will be defined in file Foo.java

- we'll discuss packages later

Terminology: attributes/data members; operations/methods

Constructors are special methods that build an object of this class – more later.

Blocks surrounded by $\{...\}$, statements terminated by ; etc. as in C.

Message passing

Objects communicate by sending messages (*not* by calling methods, because you generally aren't supposed to know exactly which method will be executed as a result of a particular message). E.g.

```
void someMethod(Foo f) {
  f.wibble("Goodbye cruel world");
}
```

Here `f` is an object reference. Another part of the program may have another reference to the same object, and it will see the wibbled version now.

Referring to objects

Objects are always referred to by reference. E.g.

```
(new Foo()).wibble("Goodbye cruel world");
```

or if you have an object `bar` which understands message `mung()` and returns a `Foo`,

```
bar.mung().wibble("Goodbye cruel world");
```

– but be aware of the Law of Demeter!

Special object reference: `this`

If you need a copy?

You have to make it explicitly:

```
Foo another = (Foo)oldfoo.clone();
```

but here be dragons. Foo must implement the Cloneable interface appropriately, you must know how deep the cloning is, etc. Read up on this carefully before using.

Access control 1

The basic two:

- public – any code can access this
- private – only code in this class can access this

`public` can be applied to class, interface, attribute, method...

`private` is only for class members

Access control 2

The other two:

- no modifier, called "package-private" – only code in the same package can access this
- protected – code can access this if it is in this package OR in a subclass of this class

`package-private` can be applied to class, interface, attribute, method... `protected` is only for class members

How to choose access modifiers

Attributes should be private (except constants)

A class's public methods define its interface: choose them with care.

If you need to write a method, but it doesn't belong in the interface, should it be private, package-private or protected? Opinions differ...

Memory management

Unlike C and C++, Java has garbage collection: that is, the programmer does not need to allocate and deallocate memory explicitly.

However, other resources e.g. database handles may still need to be managed in a similar way.

Creating a new object

```
Foo myFoo = new Foo(1);
```

requires that class Foo defines an accessible constructor taking an integer argument,

```
Foo(int j) {...}
```

The compiler will try to make a no-parameter constructor, if the programmer doesn't provide one.

(Could you ever want a private constructor? Yes, see Factory pattern, later.)

Destroying an object

Beauty of garbage collection: you don't. Once an object is no longer accessible it is eligible for garbage collection.

What if clean-up is needed, e.g. releasing resources? You can write a finaliser:

```
protected void finalize() throws Throwable {
  //do whatever should be done
}
```

BUT delays reclaiming the object's memory: possible hazard.

Null pointer errors

Single most common Java programming error.

How do they happen...? Lots of ways, e.g.

```
Obect o; // o is null
if(...) {
  // do something that initialises object reference o
}
... use o ...
```

Hopefully, in future everyone will have static analysis tools to identify these :-)

In the meantime: always initialise objects.

(A NPE causes an unchecked exception to be raised.)

Static members

One already very familiar example:

```
public static void main(String[] args) }
```

Meaning:

- attribute: one copy per class, not one per instance.
- operation: not invoked on any particular instance

Can be very useful, but overuse is a pitfall for non-OO programmers.

Basic types in Java

Java has both int and Integer, bool and Boolean, i.e. boxed and unboxed forms of basic types.

Lower case: primitive data types

Upper case: classes

Primitive data types

byte, short, int, long, float, double

boolean

char

e.g.

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
```

String

String is not a primitive data type but it is special.

```
String s = "this is a string";
```

Strings are immutable. (Contrast StringBuilder/StringBuffer)

Lots of convenience fns e.g. + for concatenation:

```
s = "Hello," + " world" + "!"
```

(But I/O is painful, e.g.

```
System.out.println(s);
```

)

Wrappers for primitive types

Class equivalent for each primitive type; compiler can often box and unbox automatically, e.g.

```
Integer x, y;
x = 12;
y = 15;
System.out.println(x+y);
```

The classes provide constants and conversion methods.

Arrays

```
class ArrayDemo {
  public static void main(String[] args) {
    int[] anArray; // declares an array of integers
    anArray = new int[10]; // allocs mem for 10 integers
    anArray[0] = 100; // initialize first element
...

    System.out.println("Element at index 0: "
                       + anArray[0]);
...
  }
}
```

Or:

```
int[] anArray = {100, 200, 300, 400, 500, 600,
                 700, 800, 900, 1000};
```

Array sanity checking

An array has a fixed length and any attempt to access an element outside these bounds produces an exception at runtime (`ArrayIndexOutOfBoundsException`)

In an array of type T you can only store instances of T or of subtypes of T. An attempt to store something else produces `ArrayStoreException`.

NB String is not the same as char array. Neither ends with a null character.

enum types

Not in Java originally but there since Java 5. Basic declaration:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Gold-plated though! Not only type safe, but also fully-fledged classes: can define methods etc.

Control structures

are all very unsurprising to anyone who knows C.

Let's just zip through them...

if

```
void applyBrakes(){
  if (isMoving){
    currentSpeed--;
  }
}
```

Can omit braces round a single-statement then-clause – but this is widely considered bad practice.

```
if (...) { ... } else { ... }
```

just as you'd expect.

switch

```
int month = 8;
switch (month) {
  case 1:  System.out.println("January"); break;
  ...
  default: System.out.println("Invalid month.");break;
}
```

Works for enum types as well as for integers.

for

```
for(int i=1; i<11; i++){
  System.out.println("Count is: " + i);
}
```

Enhanced for

```
class EnhancedForDemo {
  public static void main(String[] args){
    int[] numbers = {1,2,3,4,5,6,7,8,9,10};
    for (int item : numbers) {
      System.out.println("Count is: " + item);
    }
  }
}
```

Works for Collections as well as arrays.

while

```
while (count < 11) {
  System.out.println("Count is: " + count);
  count++;
}
```

```
do {
  System.out.println("Count is: " + count);
  count++;
} while (count <= 11);
```

Inheritance, dynamic binding

Recall: implementation inheritance Bad, type hierarchy Good.

So let's talk about interfaces and implementing interfaces first, since you should use that most.

Interfaces

```
public interface Fooable {
  public void foo(); // NB no {}
}

public class MyConcreteFooable implements Fooable {
  public void foo() { //actually do something }
  // other stuff not mentioned in interface
}
```

A class can implement several interfaces.

Type hierarchy

Interfaces can extend one another.

```
public interface GoldFooable extends Fooable {
  public void goldfoo();
}
```

Typechecking works as you expect: if a context expects a T and the compiler can check that actually what will arrive is an instance of $S < T$, all is well; other way round, not.

NB implementation classes need have nothing to do with one another.

Inheritance

```
public class SpecialCustomer extends Customer {
... some overridden methods ...
... some new methods...
}
```

All classes extend Object (no need to state this).

Single inheritance only.

Anything not mentioned is inherited directly from the parent.

Overriding

A subclass can override a superclass's method: that is, provide a specialised version of it. Should behave consistently – remember Liskov Substitution Principle and Design by Contract.

No syntactic clue that something's being overridden unless you use an annotation:

```
@Override
void foo() {...}
```

Overriding method usually has same signature and access as the one it's overriding, but may

- specialise the return type
- widen the access

Dynamic binding

Inheritance works properly.

That is, the version of a method that's invoked depends on the actual runtime class of the object, not on what the context knew about the class of the object.

Abstract

Abstract class cannot be instantiated, but serves as a basis for extending.

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

final

A final class cannot be extended.

A final method cannot be overridden.

Use final if you do not want to trust future developers not to break things!

Overloading

Can have multiple methods with the same name in the same class, distinguished by type. Compiler determines which to use.

Do not confuse with *overriding* – in fact, overloading is permitted precisely when neither method signature could override the other.

Something of a bad smell – suggests there may be a missing abstract type.

Nested classes

As well as attributes and methods, Java classes can have classes as members.

- Static nested class: normal class just put inside another class for packaging convenience
- Inner class: a class whose objects can only exist within the world provided by an instance of the outer class

Also local classes, anonymous classes.

Commonest context for nested class

```
//An example of using an inner class.
public class MyClass extends Applet {
  ...
  someObject.addMouseListener(new MyAdapter());
  ...
  class MyAdapter extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
    ...//Event listener implementation goes here...
    }
  }
}
```

Exceptions

An exception is an exceptional event: normal control flow of the program does not apply.

Two common uses:

- Genuinely exceptional events ("there's a bug", "that file doesn't exist", ...)
- As a dressed-up GOTO to jump out of complex logic.

The first use is safer and more reputable.

Try: preparing to handle exceptions

Even if your code never throws exceptions you'll need to be able to deal with exceptions that may arise from library code.

```
private Vector vector;
private static final int SIZE = 10;
PrintWriter out = null;
try {
  System.out.println("Entered try statement");
  out = new PrintWriter(new FileWriter("OutFile.txt"));
  for (int i = 0; i < SIZE; i++) {
    out.println("Value at: " + i + " = "
                + vector.elementAt(i));
  }
}
catch and finally statements . . .
```

If no handler, the exception goes up the stack.

Catching exceptions

```
try {

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "
                       + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: "
                       + e.getMessage());
}
```

Finally

A try block may get exited several ways, which is a pain for clean-up code like closing files. A finally block lets you write clean-up code in one place: it is always* executed when a try block exits.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

Checked and unchecked exceptions

Declare what exceptions may be raised in a method, and not handled there, like this:

```
public void writeList()
        throws IOException,
                ArrayIndexOutOfBoundsException {
```

For checked exceptions like IOException you *must* do this, or the code won't compile. ("Catch or specify")

For unchecked exceptions like ArrayIndexOutOfBoundsException, you may do it.

Creating exceptions

Exceptions are just instances of classes that inherit from Exception. You can define your own.

If your class inherits from RuntimeException or Error, it will be unchecked, otherwise checked.

Use checked exceptions if clients can do anything to recover.

Packages

Groups of related classes can be packaged together:

```
package uk.co.bitwise.someapp.somepackage;
public class Foo ...
```

so as to manage the namespace.

It looks like a hierarchy, but it isn't!

Use either using fully qualified names or by importing, e.g.

```
import uk.co.bitwise.someapp.somepackage.Foo;
import uk.co.bitwise.someapp.someotherpackage.*;
...

uk.co.bitwise.someapp.yetanotherpackage.Bar =
  new uk.co.bitwise.someapp.yetanotherpackage.Bar();
```

Compiling a Java application

Absolute simplest case:

```
javac Classname.java
```

The compiler will simple-mindedly (re)compile anything else that needs doing.

For anything non-trivial, though, need more, usually Ant (or make).

Running a Java application

java Classname

(NB not filename!)

Classname had better have a `public static void main(String[] args)`

There is a class loader that will load classes as required, finding them on the CLASSPATH, which includes .

If it fails to find a class you'll get a `java.lang.NoClassDefFoundError`.

Applets

run in the user's browser, provided that has Java enabled.

Main class must extend `java.applet.Applet` or more likely the Swing subclass of that, `javax.swing.JApplet`.

Must follow applet life cycle – see the applet section of Deployment tutorial for details.

Debug using appletviewer (easier than in a browser).