More Java

Perdita Stevens, University of Edinburgh

May 2010

Agenda

- Concurrency and threads
- Building GUIs with Swing
- Event handling in Swing GUIs
- Genericity, and collection classes
- Streams, I/O
- The rest of the Java SE libraries

Concurrency

For more information see the Concurrency tutorial under the Essential Java Classes heading. (Most code examples here are quoted from there.)

A JVM usually runs as just one OS *process*, but there will be multiple *threads*.

Multicore programming is a whole other issue...

Threads

```
public class HelloRunnable implements Runnable {
  public void run() {
    System.out.println("Hello from a thread!");
  }
  public static void main(String args[]) {
    (new Thread(new HelloRunnable())).start();
  }
}
```

You create a Runnable object, use it to instantiate a Thread object, and start the thread. This will send run() to its Runnable.

Threads: things to note

Use of static methods. E.g. to sleep, you don't send a thread object the message sleep: you go

```
Thread.sleep(4000); // time in milliseconds
```

which causes *the current* thread to sleep.

sleep may be interrupted by another thread, will raise checked InterruptedException if so. NB you are responsible for allowing interrupts of your own code, e.g.

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        //We've been interrupted: no more crunching.
        return;
    }
}
```

Non-static thread methods

`t.start()` – already seen: start a thread

`t.interrupt()` – interrupt a running thread, which will find out this has happened when its code checks the interrupted status, or when a Thread method raises InterruptedException

`t.join()` – wait for a thread to finish

`t.isAlive()` – check whether a thread has finished

Interference

Threads share access to objects.

Good: this is how they collaborate.

Bad: can cause interference.

Synchronised

```
public synchronized void increment() {
  c++;
}
```

Any other thread wanting to invoke *any* synchronised method on this object will block.

Every object has one associated lock (IOW any object can be used as a lock). Synchronised method implicitly grabs this's lock on entry, releases it on (any) exit.

Explicit use of locks

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

General rule: put as little as possible inside synchronised blocks, to avoid unnecessary blocking.

For more info

and more advanced concurrency features, see the rest of the Concurrency tutorial, and other documentation of the `java.util.concurrent` package.

Building GUIs

Java provides the Java Foundation Classes (JFC) part of Swing. (AWT now best regarded as a failed experiment.)

You build a GUI either by writing explicit Java code, or using a GUI builder such as that in NetBeans, or any of several for Eclipse or whatever else you use. JFormDesigner is worth a look.

For details see the Swing tutorial: `http://java.sun.com/docs/books/tutorial/uiswing/TOC.html` - here only a few words.

Basic components just as you'd expect...

Acting on Swing GUI events: ActionListener

```
public class MultiListener ... implements ActionListener {
    //... where initialization occurs:
    button1.addActionListener(this);
    button2.addActionListener(this);
    button2.addActionListener(new Eavesdropper(bottomTextArea));
  }

  public void actionPerformed(ActionEvent e) {
    topTextArea.append(e.getActionCommand() + newline);
  }
}

class Eavesdropper implements ActionListener {
    //...
  public void actionPerformed(ActionEvent e) {
    myTextArea.append(e.getActionCommand() + newline);
  }
}
```

Threads in Swing applications

Three kinds of threads:

- initial thread, just to kick things off
- event dispatch thread, for interacting with Swing components
- worker threads, for non-trivial computation (SwingWorker)

....

Genericity, and collection classes

Recall the term *polymorphism*.

Code is polymorphic if it works on things of more than one type.

We've met *object-oriented polymorphism*: a client that works with things of type C will also work with things of type $B < C$ (subtype/subclass of C).

At least it should, and the compiler will let it try – we discussed Liskov substitutivity, the point of which is to ensure that it does.

But this is not the only kind of polymorphism...

The collection class problem

Imagine we're trying to write collection classes – classes that allow us to work with sets, bags, lists etc. of objects.

We want to write the code for add, remove etc. just once and have it work for all kinds of collections.

Might try: writing a base class Set, which is independent of any application class we might want to store in a set, so that Set's add method takes an Object as argument, Set's remove method returns an Object, etc.

Trouble with this is that it gives no support for Sets that should only contain Customers – lots of class casting required, possibility of exceptions.

Collections using OO polymorphism?

Next try:

Can we then write a new class SetOfCustomer that inherits from Set?

add would have to take a Customer

remove would have to return a Customer

Nope: fails typing rules, and fails Liskov substitutivity (not every context that works with Sets will work with SetOfCustomers, since the context might try to put a non-Customer into the set).

Parametric polymorphism

The point is that actually we don't expect to have to write different code for the different kinds of Set – all we're after is for the compiler to check consistent use of the content-type.

In other words our Set class should be *parametric* over its content type.

Early versions of Java had no way to express this.

Other languages had type mechanisms that handled this; even C++ had templates.

Now Java has it too.

Writing a generic class

```
public class Box<T> {
  private T t; // T stands for "Type"
  public void add(T t) {
    this.t = t;
  }
  public T get() {
    return t;
  }
}
```

Here T is a formal type parameter – code inside the class is written so that T could be any class.

Using a generic class

```
public class BoxDemo3 {
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add(new Integer(10));
    Integer someInteger = integerBox.get(); // no cast!
    System.out.println(someInteger);
  }
}
```

More on generics

Things we haven't really talked about:

- methods can also be generic
- the formal type parameters can be bounded: e.g. if generic code won't work with *any* class, but only with subclasses of `Number`, the type parameter can be `T extends Number`.
- wildcards, e.g.

  ```
  Cage<? extends Animal> someCage = ...;
  Comparator<? super E> c = ...;
  Iterator<?> it = ...;
  ```
- type erasure (no generics survive to bytecode)

See the tutorial for detail on all of this.

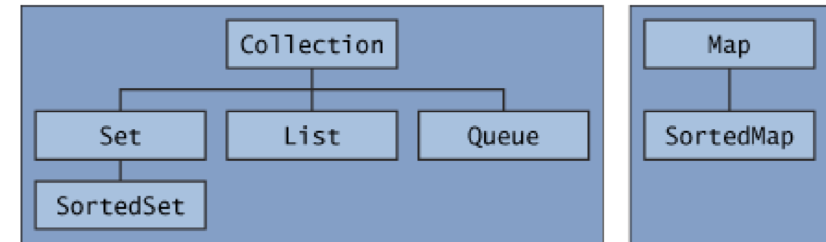
Warning

Parametrically polymorphic libraries can be very useful, and once in a blue moon writing your own parametrically polymorphic classes can be useful.

But it's also easy to make a horrible mess, especially by mixing genericity and inheritance.

If in doubt, don't!

Core collection interfaces



All generic. Various classes implementing these interfaces: pick an appropriate one. E.g.

```
List<String> list = new ArrayList<String>();
```

list is now empty with initial capacity 10. Rest of code need only know it's a List of String.

Iterating over collections

```
for (Object o : collection)
    System.out.println(o);
```

Or use an iterator, which every Collection can provide:

```
static void filter(Collection<?> c) {
  for (Iterator<?> it = c.iterator(); it.hasNext(); )
    if (!cond(it.next()))
      it.remove();
}
```

Sorting things generically?

Problem: How can you have a SortedSet that contain arbitrary objects? How do you sort them?

Solution: make the classes implement the Comparable interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

or create a Comparator object and use this to initialise the collection...

Hold your hats!

```
import java.util.*;
public class EmpSort {
  static final Comparator<Employee> SENIORITY_ORDER =
               new Comparator<Employee>() {
  public int compare(Employee e1, Employee e2) {
    return e2.hireDate().compareTo(e1.hireDate());
  }
  };

// Employee database
static final Collection<Employee> employees = ... ;

public static void main(String[] args) {
  List<Employee>e = new ArrayList<Employee>(employees);
  Collections.sort(e, SENIORITY_ORDER);
  System.out.println(e);
  }
}
```

Streams and I/O

We've seen many times:

`System.out.println(e);`

What's going on here and how does it generalise?

`System` is a JDK class that provides lots of useful static stuff...

`System.out` is an output stream of class PrintStream (stdout)

`println` prints a whole line, with newline.

`print` prints without a newline; `format` works like C's `printf`.

(Guess what `System.err` is?)

Kinds of streams

Lots, e.g. buffered vs non-buffered, for bytes and for characters. Generally you create a simple one and pass that to the constructor of a more complex one, e.g.

```
java.io.BufferedReader inputStream =
  new BufferedReader(new FileReader("xanadu.txt"));
```

See tutorial for details.

Scanning example

```
import java.io.*;
import java.util.Scanner;

public class ScanXan {
  public static void main(String[] args) throws IOException {
    Scanner s = null;
    try {
      s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
      while (s.hasNext()) {
        System.out.println(s.next());
      }
    } finally {
        if (s != null) {
          s.close();//closes underlying stream
        }
      }
  }
}
```

Getting input from the user

There is `System.in` but it's a byte stream not a character stream, so you normally need something like:

```
InputStreamReader cin = new InputStreamReader(System.in);
```

– or look up System.Console...

Object serialization, basic idea

Object I/O requires the object's class to implement the interface `Serializable`.

`ObjectOutputStream` has method `writeObject`

`ObjectInputStream` has method `readObject` (returns Object: cast it back to the expected type, get ClassCastException if this doesn't work)

(Today we're only talking about serialization onto byte streams - in August we'll talk about XML.)

Object serialization, how to do it

`Serializable` is just a marker interface – implementing it doesn't require the class write to actually do anything (except check that serialization will make sense!)

The object streams are filter streams, wrapping concrete node streams – e.g., you instantiate an ObjectOutputStream with a FileOutputStream to save objects to a file.

In the simplest cases everything Just Works – the streams have the logic to write out everything needed to reconstitute the object, and to do so.

Beyond simple cases see `http://java.sun.com/developer/technicalArticles/Programming/serialization/` `http://java.sun.com/j2se/1.6.0/docs/guide/serialization/spec/serialTOC.html`

Esp. pay attention to `serialVersionUID`!

Things to remember

- Always close your streams after use (except the standard system ones).
- A common cause of strange behaviour is needing to flush a buffer (`mystream.flush()`) (although some have the option to auto-flush).

Java SE stuff we haven't mentioned

- regular expressions
- handling files and directories
- security, e.g. use of `SecurityManager`
- package applications using JAR files
- RMI
- reflection
- internationalisation
- Java beans
- JAXP/JAXB

etc. etc. See `http://java.sun.com/javase/6/docs/`