G51PRG: Introduction to Programming Second semester

Lecture 6

Natasha Alechina

School of Computer Science & IT

nza@cs.nott.ac.uk

Previous lecture

- ***final*** keyword
- Casting objects
- More on polymorphism
- Object class

Example of usefulness of Object data type

- So far I talked a lot about how useful it is to have everything extending Object, and being able to use polymorphism and to cast objects….
- Here is a little example of how this is actually used.
- Hash table is a very useful data structure. Hashtable class is part of Java class libraries.
- Hashtables store any kind of Objects.
- So, there is just one Hashtable class which can be used in programs which need to store various kinds of objects.

Hashtable class

- Hash table holds data items indexed by keys .
- Key is used to access the value, just as an array index is used to access the corresponding element in the array.
- Hash table keys can be of any reference type (for example, Strings).

hash(key1)	**(key1, value1)**
hash(key2)	**(key2, value2)**

Example

indices	buckets: lists of (key,value) pairs
hash(“john”)	**(john, 9150001)**
hash(“adam”)	**(adam, 9510010)**

Hashtable methods

- **`Object put(Object key, Object value)`**
- **`Object get(Object key)`** - returns the value
- **`boolean containsKey(Object key) -`** returns true or false depending on whether there is an item with such key in the table
- **`boolean containsValue(Object value) -`** returns true or false depending on whether there is such a value in the table
- **`Object remove(Object key) -`** removes item with the specified key

Example Hashtable

```
Hashtable telephones = new Hashtable();
// put some telephones in a table
telephones.put(“john”, “9150001”);
telephones.put(“adam”, “9510010”);
String tel1 =
(String) telephones.get(“john”);
// will return John’s telephone number
// note that get() returns an Object so
// need to cast to String!
```

What about storing basic types?

- The trouble with Java Collections (such as Hashtable) is that they are designed to store Objects.
- Hashtable works with ***any*** objects: Strings, Points, Persons,… . But it does not have methods to store integers or doubles.
- There is a workaround though - you can use Wrapper classes.

Envelopes, or wrapper classes

- We can’t cast between basic types and objects.
- What do we do if we need to use a basic type where an object is required?
- Define a new class and put the basic type value inside it as a field. That's what "wrappers" or "envelopes” do: make an object out of a basic type.
- There are Java classes for all basic types: byte, float and so on: Boolean, Character, Byte, Short, Integer, Long, Float, Double.

Two purposes of wrapper classes

- Make it possible to use basic types with classes which handle objects. For example, we cannot store ints or use them as keys in Hashtable, but can store an object of type Integer.
- Provide home for useful methods associated with the basic type. For example, Integer class has method

```
static int parseInt(String s)
```

- which given a string “2002” returns number 2002.

Some common methods of wrapper classes

- A constructor which takes the primitive type and creates an object of the type class (e.g. **`Character(char c)`**);
- **`xxxxValue()`** (where xxxx is the primitive type, e.g. **`Character.charValue()`** and **`Boolean.booleanValue()`**) returns the value of the basic type stored in the object.

Telephone example

- If we wanted to store telephones as ints:

```
Hashtable tels = new Hashtable();
tels.put(“john”, new Integer(9150001));
tels.put(“adam”, new Integer(9510010));
```

- Which way to get tel number as an int is correct?

```
int tel1 = (tels.get(“john”)).intValue();
int tel2 = (Integer) tels.get(“john”);
int tel3 =
((Integer) tels.get(“john”)).intValue();
```

Abstract classes and interfaces

- There are many more useful data structures in Java class libraries.
- They are organised in a hierarchy of classes.
- Before I can really explain how they work I need to introduce ***abstract classes*** and ***interfaces*** in Java.

Abstract methods

- Sometimes when defining a class one wants to guarantee that a certain method exists but cannot provide implementation for this method which would work for all classes extending the given one.
- A method can be declared as ***abstract*** without providing implementation.

Example: Number class

- The abstract class Number is the superclass of classes Byte, Double, Float, Integer, Long, and Short.
- Subclasses of Number must provide methods doubleValue(), floatValue(), intValue(), longValue() to convert the represented numeric value to double, float, int, and long. These methods are declared abstract in Number.
- You just assume that each Number can be converted to double, float, etc. but for each particular type those methods will be different.

Abstract class

- A class which contains abstract methods must be declared abstract .
- It is not possible to create instances of an abstract class.
- A class which does not contain abstract methods can also be declared abstract. This is done to make it impossible to create instances of a class.
- Abstract classes are used to keep the relevant code together at the right place in the class hierarchy, and make it easier to define subclasses.
- Every concrete (non-abstract) class which extends an abstract class should provide implementation for all abstract methods.

Example: Benchmark class

- Taken from Arnold and Gosling.
- Implementation of the benchmark() method depends on the code to be tested.

```
abstract class Benchmark {
 abstract void benchmark();
 public long repeat(int count){
  long start = System.currentTimeMillis();
  for (int i = 0; i < count; i++){
   benchmark();
  }
  long finish = System.currentTimeMillis();
  return ((finish - start)/count);
 }}
```

Interfaces

- Do not confuse them with interfaces in the sense of graphical user interfaces!
- An interface is a type which contains only abstract methods and related constants, classes and interfaces.
- Any class which implements an interface is guaranteed to provide its methods.
- Interfaces provide only design whereas classes provide both design and implementation.
- Instances of a class which implements an interface I can be treated as being of type I.
- A class can implement several different interfaces (a bit of multiple inheritance through the back door).

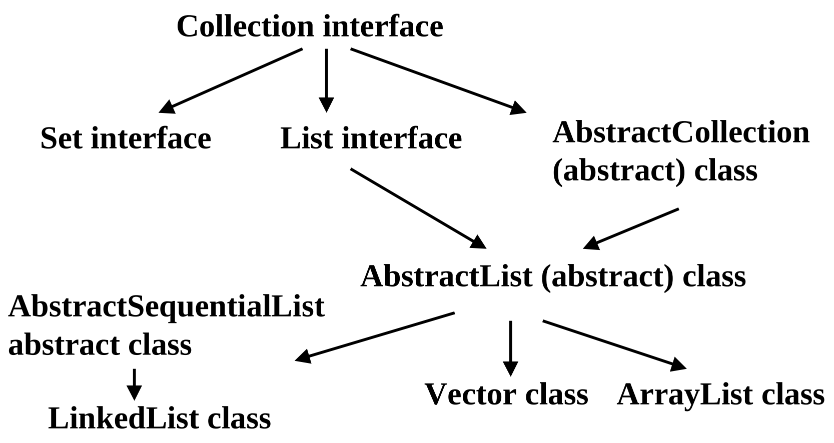
Members of an interface

- Methods are always abstract (no implementation), so abstract keyword is omitted.
- Methods are always public.
- Methods are never static, because static methods are class-specific.
- Fields are always static and final (constants used to define methods).

Interfaces and classes

- Interfaces are in many respects similar to classes.
- Interfaces are types; an object can be declared to be of type I, where I is an interface.
- Interfaces have members, just as classes do.
- Interface can extend another interface.
- Main difference is: interfaces cannot be instantiated, and they are more abstract than abstract classes.

Example: Java Collections (part of)


Collection interface
Set interface
List interface
AbstractCollection (abstract) class
AbstractList (abstract) class
AbstractSequentialList abstract class
LinkedList class
Vector class
ArrayList class

Example: Java Collections

- Methods all Collections must implement: add(Object o); contains(Object o); remove(Object o); size();...
- Methods all Lists must implement: indexOf(Object o); Object get(int index);...
- AbstractList - abstract class to extend when implementing an array-type (random access to any index) list. Some methods (to support iteration through the list) already implemented using abstract methods get(int index) and size().
- AbstractSequentialList - abstract class to extend when implementing a list with sequential access (starting at the head of the list and visiting elements in sequence). Here get() and set() implemented, iterator() and size() abstract.

Why is it useful to implement List

- Some utility methods exist which work for all Collections.
- For example, a method which can sort any data structure of type List.
- Not a separate sorting method for Vectors, a separate method for ArrayLists, a separate method for LinkedLists, but a method for any class implementing List.
- In general utility methods for Collections are held in a class from java.util package (need to import it this package to use Colections!). The class is called Collections.

Java.util.Collections.sort(List list)

- **`public static void sort(List list)`**

This method sorts elements in the list in ascending order using ***natural ordering*** of elements in the list.

- If we are sorting a list of numbers, we know what natural ordering means: the less than relation <.
- What do we do about an arbitrary list of arbitrary things? How do we compare them and decide which one should be before the other?
- In order for the method to work, things in the list must be guaranteed to implement **`compareTo()`** method.
- They way to achieve this in Java is to require that they implement **`Comparable`** interface.

Comparable interface

- **`public int compareTo(Object o)`**
- This is the only method in this interface.
- It returns a negative integer if current object is before **`o`** in the natural order, 0 if they are the same, and positive integer if it is after **`o.`**
- Strings implement Comparable (compareTo() supports lexicographic ordering of Strings).
- Numbers implement Comparable (compareTo() supports ordering of numbers).

Example: Integers

```
public int compareTo(Object o){
   return (this.intValue() -
            ((Integer) o).intValue());
}
```

- This is how we could have implemented compareTo() for Integers.
- Note that we need to cast o to Integer.
- Often people return -1 if this object is less than o, 0 if they are the same, 1 if this is greater than o.

Example: sorting a Vector

For any objects which implement Comparable, in this case Integers:

```
Vector myVector = new Vector();
myVector.add(new Integer(5));
myVector.add(new Integer(3));
myVector.add(new Integer(7));
Collections.sort(myVector);
// now myVector is sorted in natural
// order of Integers
```

Example: iterating through a Vector

- Here is a simple minded iteration not using an **`Iterator`** object.

```
Vector myVector = new Vector();
myVector.add(new Integer(5));...
for(int i = 0; i < myVector.size(); i++){
  System.out.println(
  ((Integer)myVector.get(i)).intValue());
}
```

Summary and further reading

- Abstract classes and interfaces allow Java programmers to implement methods at the right place in the class hierarchy and re-use code.
- I covered general principles of extending classes, implementing interfaces, and using methods polymorphically, but only a tip of the iceberg in Collection classes and other library methods.
- If you are interested look at Iterators and Comparators.
- For class hierarchies and interfaces, read

http://java.sun.com/docs/books/tutorial/java/javaOO/subclasses.html

http://java.sun.com/docs/books/tutorial/java/interpack/interfaces.html