Programming II

Object Oriented Programming with Java

- Advanced Topics -

Java 8: Default Methods

Alastair Donaldson

www.doc.ic.ac.uk/~afd

Imagine we are designing a collections framework

Let's keep things very simple:

```
public interface ICollection<E> extends Iterable<E> {

  public void add(E e);

  public boolean contains(E e);

}
```

Adds e to the collection

Tells us whether something that `.equals(e)` is in the collection

```
public interface IList<E> extends ICollection<E> {

  public E get(int i);

  public boolean removeFirst(E e);

}
```

Gets the `i`th element of the list, throws exception if out of bounds

Removes first element that `.equals(e)`, if it exists; returns true if something was removed

We would also have other sub-interfaces, e.g. `ISet<E> extends ICollection<E>`

Implementing an array list

We'll provide one implementation of `IList<E>`, which will represent a list as an array

Somewhat like the `ArrayList<E>` class from the real Java collections framework

To implement `IList<E>`, a class must provide:

- `public Iterator<E> iterator();`
- `public void add(E e);`
- `public boolean contains(E e);`
- `public E get(int i);`
- `public boolean removeFirst(E e);`

Required by `Iterable<E>`

Required by `ICollection<E>`

Required by `IList<E>`

ArrayListImpl<E>

(funny name so we don't confuse it with ArrayList<E>

```
public class ArrayListImpl<E> implements IList<E> {

  private static final int INITIAL_SIZE = 256;

  @SuppressWarnings("unchecked")
  private E[] data = (E[])new Object[INITIAL_SIZE];

  private int count = 0;

  // Methods required by ICollection<E>

  @Override
  public void add(E e) {
    if(count == data.length) {
      data = Arrays.copyOf(data, data.length * 2);
    }
    data[count++] = e;
  }

  // Continued on next slide
```

Initially space for 256 elements

Number of elements in the list; also next free position

If out of space, double the capacity

ArrayListImpl<E> (continued)

```
// Continued from previous slide

@Override
public boolean contains(E e) {
  for(E x : this) {
    if(bothNullOrEqual(e, x)) {
      return true;
    }
  }
  return false;
}

private boolean bothNullOrEqual(E x, E y) {
  return (x == null && y == null) ||
         (x != null && x.equals(y));
}

// Methods required by Ilist<E>

@Override
public E get(int i) {
  return data[i];
}
```

Why can we do this?

If `ArrayIndexOutOfBoundsException` is thrown, it will be propagated. Why?

// Continued on next slide

ArrayListImpl<E> (continued)

```
  // Continued from previous slide

  @Override
  public boolean removeFirst(E e) {
    for(int i = 0; i < count; i++) {
      if(bothNullOrEqual(e, data[i])) {
        for(int j = i + 1; j < count; j++) {
          data[j - 1] = data[j];
        }
        data[count - 1] = null;
        count--;
        return true;
      }
    }
    return false;
  }

  // Method required by Iterable<E>

  @Override
  public Iterator<E> iterator() {
    ... // Let's see how to implement this using an anonymous class
  }

} // End of ArrayListImpl<E>
```

Shift everything right of the element to be deleted left by one place

Why is it important to null out the last element? Why isn't it sufficient just to decrement `count`?

Implementing `iterator`

The `Iterator<E> iterator()` method should return an instance of an iterator for our list

We could write a separate class, `ArrayListImplIterator`, to achieve this

But it would make no sense for an `ArrayListImplIterator` to exist without an `ArrayListImpl`

Thus better to make the iterator class a **nested class**, or an **anonymous class**

Let us look at the anonymous class solution (see Tutorial Sheet 3 for examples with nested classes)

Implementing `iterator`

To implement `Iterator<E> iterator()` a class must provide:

- `public boolean hasNext();`
- `public E next();`

Says whether there is anything left to iterate over

Returns next element in the iteration sequence and moves the iterator on one element

Throws `NoSuchElementException` if there is no element left

Let's write an anonymous class to do the job

Implementing `iterator`

```
public class ArrayListImpl<E> implements IList<E> {

  private static final int INITIAL_SIZE = 256;

  @SuppressWarnings("unchecked")
  private E[] data = (E[])new Object[INITIAL_SIZE];

  private int count = 0;

  ... // Other methods as before

  @Override
  public Iterator<E> iterator() {

    return new Iterator<E>() { // Start of anonymous class

      private int index = 0;

      @Override
      public boolean hasNext() {
        return index < count;
      }

      // Anonymous class continued on next slide
```

State of `ArrayListImpl<E>` as before

The anonymous class can refer to fields of the enclosing class

Implementing iterator

```
      // Anonymous class continued from previous slide

      @Override
      public E next() {
        if(!hasNext()) {
          throw new NoSuchElementException();
        }
        return data[index++];
      }

    }; // End of anonymous class definition

  } // End of iterator method

} // End of ArrayListImpl<E> class
```

The anonymous class can refer to fields of the enclosing class

The iterator method on one slide

```
@Override
public Iterator<E> iterator() {

  return new Iterator<E>() {

    private int index = 0;

    @Override
    public boolean hasNext() {
      return index < count;
    }

    @Override
    public E next() {
      if(!hasNext()) {
        throw new NoSuchElementException();
      }
      return data[index++];
    }

  }; // End of anonymous class definition

}
```

Would it make any difference if this field were **public**?

An anonymous class can extend an existing class

```
class A {

  private String s;

  public A(String s) {
    this.s = s;
  }

  public void blarp() {
    System.out.println(s);
  }

}
```

```
public class AnonymousExtensionDemo {

  public static void main(String[] args) {

    A first = new A("Hello");
    A second = new A("World") {

      @Override
      public void blarp() {
        System.out.println(
            "Overridden blarp!");

        super.blarp();
      }

    };

    first.blarp();
    second.blarp();

  }

}
```

The anonymous class implicitly extends A

Prints:

```
Hello
Overridden blarp!
World
```

Local classes: not quite anonymous classes

```
class A {

  private String s;

  public A(String s) {
    this.s = s;
  }

  public void blarp() {
    System.out.println(s);
  }

}
```

Prints:

```
Hello
Overridden blarp!
World
```

```
public class LocalExtensionDemo {

  public static void main(String[] args) {

    A first = new A("Hello");

    class B extends A {

      public B(String s) {
        super(s);
      }

      @Override
      public void blarp() {
        System.out.println(
             "Overriden blarp!");
        super.blarp();
      }
    };

    A second = new B("World");

    first.blarp();
    second.blarp();

  }
}
```

Class B is declared inside method main

This is equivalent to the previous slide

Years go by…

Imagine that:

- The `ICollection<E>` and `IList<E>` interfaces become widely used
- Our `ArrayListImpl<E>` becomes widely used
- Many companies and users provide their own classes that implement the interfaces

…but: we decide we want to add some extra methods to these interfaces

Impact of adding methods to an interface

Consider adding to ICollection<E>:

- `public int count();`

Return the number of elements in the collection

Consider adding to IList<E>:

- `public boolean removeAll(E e);`

Remove all elements from the collection that are equal to `e` according to `.equals()`

After these changes, all implementations of `ICollection<E>` and `IList<E>` will **fail to compile**!

…but there is a reasonable default way to implement these methods

This implementation of `count` relies only on the `Iterable<E>` interface – it is thus applicable to any `ICollection<E>`

```
public int count() {
  int result = 0;
  for(E e : this) {
    result++;
  }
  return result;
}
```

If we added this to **every** implementation of `ICollection`, they would all compile and operate correctly

But we cannot: most of the implementations are third party

Also: why should all clients have to change their classes?

…but there is a reasonable default way to implement these methods

This implementation of `removeAll` relies only on the `IList<E>` interface – it is thus applicable to any `IList<E>`

```
public boolean removeAll(E e) {
  boolean result = false;
  while(removeFirst(e)) {
    result = true;
  }
  return result;
}
```

But again: we cannot and should not force all implementing classes to add this method

Java 8 solution: **default** methods

A Java 8 interface may include a method marked as **default**

A **default** method is **not static** and **has a body**

It is a regular method that can declare variables, create objects, and invoke other methods of the interface

Every class implementing the interface gets the default method **automatically**

A class can **override** a default method to change or replace its behaviour

ICollection<E> with default count

```
public interface ICollection<E> extends Iterable<E> {

  public void add(E e);

  public boolean contains(E e);

  public default int count() {
    int result = 0;
    for(E e : this) {
      result++;
    }
    return result;
  }

}
```

A **default method**: every implementation of `ICollection<E>` gets this version of `count`, unless it explicitly provides its own implementation

This method implicitly calls methods of `Iterable<E>`. Where?

IList<E> with default removeAll

```
public interface IList<E> extends ICollection<E> {

  public E get(int i);

  public boolean removeFirst(E e);

  public default boolean removeAll(E e) {
    boolean result = false;
    while(removeFirst(e)) {
      result = true;
    }
    return result;
  }

}
```

A **default method**: every implementation of IList<E> gets this version of removeAll, unless it explicitly provides its own implementation

Impact of the default methods on existing classes

There is (almost) no impact.

Suppose class `C` implements interface `I`
Suppose **default method** `foo` is added to `I`

- If `C` already has a `foo` method (with same signature) this **overrides** the default
- If `C` does not have a foo method, it **inherits** the default
- Old clients of `C` can interact with `C` through `I` as usual, without invoking `foo`
- New or updated clients of `C` can also invoke foo
- Later, a **specialised implementation** of `foo` can be provided in `C` if necessary

Overriding the default methods in ArrayListImpl

The default `count` in `ICollection<E>` iterates over the whole collection

Requires *O*(*N*) time, where *N* is size of collection

We can implement count in *O*(1) time for `ArrayListImpl<E>`: we know how big the list is

```
@Override
public int count() {
  return count;
}
```

This **overrides** the default implementation of `count` provided in `ICollection<E>`

Overriding the default methods in ArrayListImpl

The default `removeAll` in `IList<E>` invokes `removeFirst` once per occurrence of the element

`removeFirst` requires $O(N)$ time for `ArrayListImpl<E>`

Thus default `removeAll` requires $O(N^2)$ time for `ArrayListImpl<E>`. Quadratic time is **really bad**

```
@Override
public boolean removeAll(E e) {
  // O(N) solution here!
}
```

Challenge: can you write an $O(N)$ implementation of `removeAll` for `ArrayList<E>`? See sample code for my attempt

What is the role of default methods?

Default methods were introduced to enable **interface evolution** such that **client code does not break**

Remember this design goal when thinking about the rules for default methods

They were **not** introduced with the purpose of enabling code re-use through interfaces

…but they do allow this!

Time will tell what people do with them!

Default methods and functional interfaces

Let's write an interface to represent a **total order**

The interface will specify the following methods:

```
public boolean lessThan(E x, E y);

public boolean greaterThan(E x, E y);

public boolean lessThanOrEqual(E x, E y);

public boolean greaterThanOrEqual(E x, E y);
```

ITotalOrder<E> without default methods

```
public interface ITotalOrder<E> {

  public boolean lessThan(E x, E y);

  public default boolean greaterThan(E x, E y);

  public default boolean lessThanOrEqual(E x, E y);

  public default boolean greaterThanOrEqual(E x, E y);

}
```

Is this a **functional** interface?

No! Why not?

Observation: we could define `greaterThan` in terms of `lessThan`…

ITotalOrder<E> with default methods

```
public interface ITotalOrder<E> {

  public boolean lessThan(E x, E y);

  public default boolean greaterThan(E x, E y) {
    return lessThan(y, x);
  }

  public default boolean lessThanOrEqual(E x, E y) {
    return !lessThan(y, x);
  }

  public default boolean greaterThanOrEqual(E x, E y) {
    return !lessThan(x, y);
  }

}
```

Reminiscent of Haskell type classes

Provide `lessThan` and the rest comes for free!

Is this a **functional interface**?

Yes! Why?

Representing ITotalOrder<E> as a lambda

```
private static boolean isEven(int x, int y) {
  return (x % 2) == 0;
}

public static void main(String[] args) {

  ITotalOrder<Integer> evenThenOdd =
   (x, y) -> (isEven(x) && !isEven(y) ? true :
              (isEven(y) && !isEven(x) ? false : x < y));

  System.out.println(evenThenOdd.lessThan(4, 17));
  System.out.println(evenThenOdd.greaterThanOrEqual(4, 4));
  System.out.println(evenThenOdd.greaterThan(4, 6));
  System.out.println(evenThenOdd.lessThanOrEqual(24, 101));

}
```

This lambda takes two Integer arguments and returns a boolean

Prints:

```
true
true
false
true
```

Reminder of what the lambda means

```
ITotalOrder<Integer> evenThenOdd =
   (x, y) -> (isEven(x) && !isEven(y) ? true :
              (isEven(y) && !isEven(x) ? false : x < y));
```

is shorthand for:

```
class EvenThenOddOrder implements ITotalOrder<Integer> {

  @Override
  public boolean lessThan(Integer x, Integer y) {
    return  isEven(x) && !isEven(y) ? true :
           (isEven(y) && !isEven(x) ? false : x < y);

}

...

ITotalOrder<Integer> = new EvenThenOddOrder();
```

Note: `isEven` would have to be in scope

Can a default method be final?

In `ITotalOrder<E>` it may seem appealing to make the default methods **final**

`greaterThan(x, y)` should be exactly `!lessThan(y, x)` – implementing classes should not mess with this!

However: final default methods are **not allowed**

Reason: they may **break existing code**

Final default methods could break existing code

```
public interface I {

  public void foo();

}
```

Interface `I` requires method `foo`

```
public class C implements I {

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

  public int bar() {
    return 42;
  }

}
```

Implementation `C` provides `foo`, plus another method, `bar`

So far, all is good

Suppose `I` evolves to include `bar`

```
public interface I {

  public void foo();

  public default int bar() {
    return 53;
  }
}
```

Years later, `I` is evolved to provide a default method called `bar`

```
public class C implements I {

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

  public int bar() {
    return 42;
  }

}
```

Implementation `C` does not break: its `bar` overrides the default `bar`

`@Override` annotation deliberately omitted: `C` did not originally intend to override `bar`: there was no `bar` in `I`

All is still good

If bar were final in I this would not work

```
public interface I {

  public void foo();

  public final default int bar() {
    return 53;

}
```

Not allowed

```
public class C implements I {

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

  public int bar() {
    return 42;
  }

}
```

Implementation C would fail to compile, as **final** method cannot be **overridden**

Violates goal of default methods, which is to allow interface evolution **without breaking existing classes**

Do default methods play nicely with multiple inheritance?

Almost – there is a hypothetical problem with name clashes

Consider this:

```
public interface I {

  public void foo();

}
```

```
public interface J {

  public void bar();

}
```

```
public class C implements I, J {

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

  @Override
  public void bar() {
     ...
  }
}
```

Do default methods play nicely with multiple inheritance?

Suppose by some horrible coincidence a default method `baz` is added to **both** `I` and `J`

```
public interface I {

  public void foo();

  public default int baz() {
    return 42;
  }
}
```

```
public interface J {

  public void bar();

  public default int baz() {
    return 53;
  }
}
```

```
public class C implements I, J {

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

  @Override
  public void bar() {
     ...
  }

}
```

Breaks existing classes! Very unlikely to crop up in practice

```
Error: Duplicate default methods named bar with the parameters () and () are inherited from the types I and J
```

To avoid the error, implement baz in C

```
public class C implements I, J {

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

  @Override
  public void bar() {
     ...
  }

  @Override
  public int baz() {
    return I.super.baz() + J.super.baz();
  }

}
```

Silly implementation, but it illustrates how to use `super` to invoke a **default** method

The name of the interface associated with the method must precede `super`

Lets us select between `I`'s `baz` and `J`'s `baz`

What was the motivation for default methods?

Extending the collections framework with cool new features, without breaking existing implementations

Examples:

- A list can be turned into a `Stream`, on which operations like `filter` and `map` can be performed; conversion to a stream has a **default implementation**
- A list can be sorted by providing a comparator – a **default sorting algorithm** is used

Let us see these in action

Case study using default methods and lambdas

We'll see how to filter, sort and map a list

We'll make use of three built-in functional interfaces:

```
public interface Function<E, F> {
  public F apply(E e);
}

public interface Predicate<E> {
  public boolean test(E e);
}

public interface Comparator<E> {
  public int compare(E x, E y);
}
```

`java.util.function.Function`

`java.util.function.Predicate`

`java.util.Comparator`

`Function<E, F>` is like last lecture's `Transformer<S, T>`

Person class

```
class Person {
  private String firstname;
  private String lastname;
  private int age;

  public Person(String firstname, String lastname, int age) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.age = age;
  }

  public String getFirstname() {
    return firstname;
  }

  // getLastname() and getAge() - similar
}
```

Filtering, mapping and sorting a list

```
public class StreamsDemo {

  public static <E, F> List<F> filterThenMapThenSort(
                      List<E> in,
                      Predicate<E> p,
                      Function<E, F> f,
                      Comparator<F> c) {

    List<F> result = in.stream()
                       .filter(p)
                       .map(f)
                       .collect(Collectors.toList());

    result.sort(c);
    return result;
  }

  // Continued on next slide
```

Default methods added to the List interface

Turn the list into a stream

Eliminate elements that don't satisfy p

Map f over the remaining elements

Sort the list using the comparator

Turn the resulting stream back into a list

Let's filter, map and sort some people

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

    List<Person> people = new ArrayList<Person>();
    people.add(new Person("Ally", "Donaldson", 33));
    people.add(new Person("Poppy", "Donaldson", 4));
    people.add(new Person("Felix", "Donaldson", 1));
    people.add(new Person("Harry", "Potter", 52));
    people.add(new Person("Amazing", "Amy", 100));

    Predicate<Person> isAdult = p -> p.getAge() >= 18;

    Function<Person, String> getFirstname = (p -> p.getFirstname());

    Comparator<String> stringComparator = (s, t) -> s.compareTo(t);

    List<String> sortedFirstNamesOfAdults = filterThenMapThenSort(
          people, isAdult, getFirstname, stringComparator);

    System.out.println(sortedFirstNamesOfAdults);
  }
}
```

Lambdas

Prints: [Ally, Amazing, Harry]

Interface with default method vs. abstract class

Intended role of default methods: to support interface evolution

Default methods **defend** existing classes so that they still compile despite additions to the interface

They are sometimes called **defender methods**

It is tempting to use default methods more broadly: to capture common behaviour as we would do with an **abstract class**

This works as long as the common behaviour does not depend on state: **interfaces still cannot have** (non static final) **fields**

I don’t yet have advice for you on best practices regarding default methods

Experiment with them!

Pointers to interesting things to look at

Lambdas for event handling in GUI programs

Streams

Default method in Function and Comparator interfaces, e.g.:
- `compose` method of `Function`
- `thenComparing` method of `Comparator`

Closures in C# vs. Java
http://csharpindepth.com/articles/chapter5/closures.aspx

Lambdas in C++
http://www.cprogramming.com/c++11/c++11-lambda-closures.html

A great source I used in preparing these lectures:
http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html