Java interfaces and generics

Lecturer: Arran Stewart

Outline

We discuss:

- What are Java interfaces?
  - And how can we use them when implementing abstract data type (ADTs)?
- What are Java generics?
  - And how can we use them to avoid duplicating code?

Interfaces and generic types

- We have seen a little of the use of two features of Java that are very useful for implementing abstract data types: *interfaces* and *generic types*.
- We will now examine these in more detail.

Interfaces in Java

- An *interface* in Java is a little like a class.
- It can contain constants (like the `public static int CAPACITY` constant we saw in the `Stack` class in week 1).
- However, whereas a normal class can contain *methods*, an interface contains only method *signatures* – the "type" of the method.
  - For instance, the `isEmpty()` method of our `Stack` class has the signature:
    ```
    public boolean isEmpty();
    ```

Interfaces in Java

- Also, the purpose of interfaces is quite different to classes.
- Interfaces let us specify the methods *should* have, if we want them to conform to that interface.
- Unlike classes, interfaces do not have constructors – we can never *instantiate* an interface directly.
- But, we can create other classes that conform to that interface – they are said to *implement* it.

Interfaces example

- For instance – we might have an interface `StackADT`, representing the behaviour that any class *should* have, if we want to call it a stack.
- What behaviours are those? We said any stack should support the operations of *push*ing on an item, *pop*ping off on item, letting us inspect the *top*, and checking whether the stack *is empty*.

Interfaces example

- So an interface for a `StackADT` would look like this:

```
public interface StackADT {
    public void push(int a);
    public int pop();
    public int top();
    public boolean isEmpty();
}
```

- It details all the methods that a class *must* implement, if we want to say it is of the `StackADT` type.
- Note that the method signatures end in a semicolon.
- Interfaces are an excellent way of specifing *abstract* data types.

Using an interface

- If we want to say that a class we have written is of type `StackADT`, how do we do so?
- We use the keyword *implements*.
- If we wanted to say our `Stack.java` class from week 1, which uses arrays, is of type `StackADT`, then our class might look something like this:

```
public class Stack implements StackADT {
  // field declarations go here
  // constructor goes here

  // implement the "push" method:
  public void push (int a) {
    stack[top] = a;
    top = top+1;
  }

  // ... rest of class
```

Interfaces and software development

- Recall that in our discussion of abstract data types, we said that they help improve *modularity*, and make it easier for multiple developers to work on a single software system.
- In Java, abstract data types (ADTs) will often be represented by interfaces, and the concrete data structures that implement those ADTs will be classes.
- If someone is writing an ADT implementation – then as long as we have the *interface*, we can start making use of that ADT in our code, before the implementation is even finished.
- And if the implementation later needs to be changed – this can be done without our code having to be re-written.

Generic types

Generic Types

- A *generic type* is a sort of a "template" for creating other types.
- For instance, suppose for some reason we needed a "Box" class – a little like a linked list node, but which just holds a value.
- A Box for holding an `int` might look like this:

```
class IntBox {
  int value;
  public IntBox(int v)         { this.value = v; }
  public void setValue(int v) { this.value = v; }
  public int  getValue()       { return value; }
}
```

Box for a String

- And a Box for holding a `String` might look like this:

```
class StringBox {
  String value;
  public StringBox(String v)      { this.value = v; }
  public void setValue(String v) { this.value = v; }
  public String  getValue()       { return value; }
}
```

More general boxes

- But this is very repetitive – we are writing similar code over and over, for boxes that hold different types of things.

More general boxes

- But this is very repetitive – we are writing similar code over and over, for boxes that hold different types of things.
- Can we make our code more general?

More general boxes

- But this is very repetitive – we are writing similar code over and over, for boxes that hold different types of things.
- Can we make our code more general?
- One possibility is – instead of writing boxes to store ints, Strings, and so on, we could create a box that stores `Objects`.

More general boxes

- But this is very repetitive – we are writing similar code over and over, for boxes that hold different types of things.
- Can we make our code more general?
- One possibility is – instead of writing boxes to store ints, Strings, and so on, we could create a box that stores `Objects`.
- `Object` is the most general class in Java – all other classes are *sub-classes* of `Object`.

More general boxes

- But this is very repetitive – we are writing similar code over and over, for boxes that hold different types of things.
- Can we make our code more general?
- One possibility is – instead of writing boxes to store ints, Strings, and so on, we could create a box that stores `Objects`.
- `Object` is the most general class in Java – all other classes are *sub-classes* of `Object`.
- Our code might be:

```
class ObjectBox {
  Object value;
  public ObjectBox(Object v)      { this.value = v; }
  public void setValue(Object v) { this.value = v; }
  public Object  getValue()       { return value; }
}
```

More general boxes

- When using our `ObjectBox`, we would write code like this:

```
ObjectBox b = new ObjectBox(9);
b.setValue(13);
System.out.println( b.getValue() ); // prints 13
```

More general boxes

```
ObjectBox b = new ObjectBox(9);
b.setValue(13);
System.out.println( b.getValue() ); // prints 13
```

But now, we have no way of knowing exactly what *type* of value is stored in a box – and we may accidentally change the type without realizing:

```
// somewhere else, we write:
b.setValue("cat");
// and later try to fetch an int out of the box
int myInt = (Integer) b.getValue(); // a runtime error!
```

NB: when we store an `int` in our `ObjectBox`, it automatically gets converted to a class called `Integer` by Java; because primitive types like `int`, `byte` and `char` etc. are *not* classes, and are not sub-types of `Object`. And when we fetch the value out, we have to tell the compiler we think the `Object` it holds is an `Integer`.

More general boxes

- The problem is that if we store an `Object` in a box, we never can be sure *exactly* what sort of `Object` it is.
- But Java generics give us a way of getting the best of both worlds –
  we only need write *one* version of the class; but we can use it with multiple types, *and* know exactly what sort of type we have.
- Here is a version of the box class using Java generics:

```
public class Box<T> { // T stands for "Type"
  T value;
  public Box(T v)           { this.value = v; }
  public void setValue(T v) { this.value = v; }
  public T  getValue()       { return value; }
}
```

Java generics

- The "T" is a bit like an empty placeholder in a template, which we can fill in with some other type.

```
public class Box<T> { // T stands for "Type"
  T value;
  public Box(T v)           { this.value = v; }
  public void setValue(T v) { this.value = v; }
  public T  getValue()      { return value; }
}
```

Java generics

- The "T" is a bit like an empty placeholder in a template, which we can fill in with some other type.

```
public class Box<T> { // T stands for "Type"
  T value;
  public Box(T v)           { this.value = v; }
  public void setValue(T v) { this.value = v; }
  public T  getValue()      { return value; }
}
```

- And we could use our new, generic Box class like this:

```
Box<String> b = new Box<String>("cat");
b.setValue("dog");
System.out.println( b.getValue() ); // prints "dog"
```

Compile-time checking

- If we ever try to store an incorrect type in our variable . . .

```
b.setValue( 3 );
```

Compile-time checking

- If we ever try to store an incorrect type in our variable . . .

```
b.setValue( 3 );
```

- . . . we will get an error at *compile* time, to let us know we have made a mistake:

```
error: incompatible types: int cannot be converted to S
    b.setValue( 3 );
```

Terminology

- We say that a generic class is a class that is *parameterized* over types.
- We know that we can call a *method* with parameters ...

  ```
  myStack.push( 3 )
  ```

- ... and in a similar way, we can make use of a generic type, by supplying a *type* parameter, to be substituted into the "placeholder".