Advanced Object-oriented Programming

Lecture 16

Abstract Classes
Abstract Classes for Generic Solutions

There is still scope for making our Operators class more concise.

The `getPrecedence()` methods all have the same form.

For ‘and’:

```java
in class AndOperator

public int getPrecedence() {
    return AND_PREC;
}
```

For ‘or’:

```java
in class OrOperator

public int getPrecedence() {
    return OR_PREC;
}
```

e tc.
Abstract Classes for Generic Solutions

There is still scope for making our Operat ors class more concise.

The `getPrecedence()` methods all have the same form.

For ‘and’:

```java
public int getPrecedence() {
    return AND_PREC;
}
```

For ‘or’:

```java
public int getPrecedence() {
    return OR_PREC;
}
```

etc.
Digression: Generic vs Particular

Consider a tool that lets a user draw shapes on a canvas: circles, squares, rectangles, ovals, etc.

Suppose the design of such a tool has reached a stage where the following classes have been identified:

- **Canvas** — the ‘drawing area’
- **Color** — to represent different colours for shapes
- **AbstractShape** — to represent the drawn shapes, with subclasses:
  - **Circle**
  - **Square**, etc.
Consider a tool that lets a user draw shapes on a canvas: circles, squares, rectangles, ovals, etc.

Suppose the design of such a tool has reached a stage where the following classes have been identified:

- **Canvas** — the ‘drawing area’
- **Color** — to represent different colours for shapes
- **AbstractShape** — to represent the drawn shapes, with subclasses:
  - **Circle**
  - **Square**, etc.
Digression: Generic vs Particular

Consider a tool that lets a user draw shapes on a canvas: circles, squares, rectangles, ovals, etc.

Suppose the design of such a tool has reached a stage where the following classes have been identified:

- **Canvas** — the ‘drawing area’
- **Color** — to represent different colours for shapes
- **AbstractShape** — to represent the drawn shapes, with subclasses:
  - **Circle**
  - **Square**, etc.
Generic vs Particular

The AbstractShape class is to have the following fields:

- Color color
- Point anchorPoint

where anchorPoint represents some location representing the position of the shape on the canvas. This might be the centre of a Circle, the top-left corner of a Square, etc.
Generic vs Particular

The AbstractShape class is to have the following methods:

- `void move(int dx, int dy)` — to move a shape on the canvas
- `void draw(Canvas c)` — to actually draw the shape on the canvas
Generic vs Particular

The tool will store all the drawn shapes in an array:

```java
AbstractShape[] shapes;
```

Whenever necessary (e.g., refreshing a screen), all the shapes on the canvas can be drawn by

```java
for (int i=0; i < shapes.length; i++) {
    shapes[i].draw(theCanvas);
}
```

Thus all the shapes can be drawn in a *generic* way.
The tool will store all the drawn shapes in an array:

```java
AbstractShape[] shapes;
```

Whenever necessary (e.g., refreshing a screen), all the shapes on the canvas can be drawn by

```java
for (int i=0; i < shapes.length; i++) {
    shapes[i].draw(theCanvas);
}
```

Thus all the shapes can be drawn in a *generic* way.
Generic vs Particular

However, actually drawing the shapes is not generic: the code for drawing a Circle will be very different from the code for drawing a Square, and so on. (Sound familiar?)

We might include a ‘dummy’ method in AbstractShape, which is overridden in the subclasses Circle, Square, etc.
Generic vs Particular

However, actually drawing the shapes is not generic: the code for drawing a Circle will be very different from the code for drawing a Square, and so on. (Sound familiar?)

We might include a ‘dummy’ method in AbstractShape, which is overridden in the subclasses Circle, Square, etc.
public class AbstractShape {
    private Color color;
    private Point anchorPoint;

    public void move(int dx, int dy) {
        anchorPoint.move(dx, dy);
    }

    public void draw(Canvas c) { }
}
This dummy method will be overridden in the subclasses. 

E.g.,

```java
public class Circle extends AbstractShape {
    private int radius;

    public void draw(Canvas c) {
        // code to draw the circle
    }
}
```
The method `Circle.draw()` *overrides* the method `AbstractShape.draw()` if `shapes[i]` happens to be an instance of `Circle`, then

```java
shapes[i].draw(theCanvas);
```

executes the code in `Circle.draw()`.

Similarly, if `Square` overrides `AbstractShape.draw()`, and `shapes[i]` happens to be an instance of `Square`, then the `Square.draw()` code is executed.

This run-time determination of which code to execute is sometimes called *dynamic binding*. 
The method `Circle.draw()` **overrides** the method `AbstractShape.draw()` if `shapes[i]` happens to be an instance of `Circle`, then

```
shapes[i].draw(theCanvas);
```

executes the code in `Circle.draw()`.

Similarly, if `Square` overrides `AbstractShape.draw()`, and `shapes[i]` happens to be an instance of `Square`, then the `Square.draw()` code is executed.

This run-time determination of which code to execute is sometimes called **dynamic binding**.
The method \texttt{Circle.draw()} \textit{overrides} the method \texttt{AbstractShape.draw()}

if \texttt{shapes[i]} happens to be an instance of \texttt{Circle}, then

\begin{verbatim}
shapes[i].draw(theCanvas);
\end{verbatim}

executes the code in \texttt{Circle.draw()}. \hfill

Similarly, if \texttt{Square} overrides \texttt{AbstractShape.draw()}, and \texttt{shapes[i]} happens to be an instance of \texttt{Square}, then the \texttt{Square.draw()} code is executed.

This run-time determination of which code to execute is sometimes called \textit{dynamic binding}. \hfill
Having different classes, and dynamic binding allows us to avoid horrible code like:

```java
if (shapes[i] instanceof Circle) {
    Circle c = (Circle)(shapes[i]);
    // code to draw the circle
} else if (shapes[i] instanceof Square) {
    Square s = (Square)(shapes[i]);
    // code to draw the square
} else // etc., etc.
```
Clearly (I hope!) the generic code is better than the non-generic code.
But having a ‘dummy’ `AbstractShape.draw()` method is rather inelegant.

This kind of situation arises often enough for Java (and other OO languages) to have introduced the notion of `abstract class`. 
Abstract Classes

An **abstract class** is a class that contains an *abstract method*.

An **abstract method** is a method declaration with no body; for example:

```java
public void draw(Canvas c);
```

If a class contains an abstract method, it must declare itself and the method **abstract**: 

```java
abstract public void draw(Canvas c);
```
Abstract Classes

An **abstract class** is a class that contains an *abstract method*. An **abstract method** is a method declaration with no body; for example:

```java
public void draw(Canvas c);
```

If a class contains an abstract method, it must declare itself and the method **abstract**: 

```java
public abstract void draw(Canvas c);
```
Abstract Classes

An **abstract class** is a class that contains an *abstract method*. An **abstract method** is a method declaration with no body; for example:

```java
public void draw(Canvas c);
```

If a class contains an abstract method, it must declare itself and the method **abstract**:
public abstract class AbstractShape {
    private Color color;
    private Point anchorPoint;

    public void move(int dx, int dy) {
        anchorPoint.move(dx, dy);
    }

    public abstract void draw(Canvas c);
}

NB - you cannot create an instance of an abstract type.
‘Concrete’ Subclasses

Just as before, we want the subclasses of AbstractShape to contain the actual code for drawing circles, squares, etc.

```java
public class Circle extends AbstractShape {
    private int radius;
    public void draw(Canvas c) {
        // code to draw the circle
    }
}
```

NB — this class is not `abstract` because it *does* provide a ‘concrete implementation’ of the abstract method `draw()`. Class Circle is not `abstract`, so we *can* create instances of Circle.
‘Concrete’ Subclasses

Just as before, we want the subclasses of `AbstractShape` to contain the actual code for drawing circles, squares, etc.

```java
public class Circle extends AbstractShape {
    private int radius;

    public void draw(Canvas c) {
        // code to draw the circle
    }
}
```

NB — this class is not `abstract` because it does provide a ‘concrete implementation’ of the abstract method `draw()`. Class `Circle` is not `abstract`, so we can create instances of `Circle`. 
‘Concrete’ Subclasses

Just as before, we want the subclasses of `AbstractShape` to contain the actual code for drawing circles, squares, etc.

```java
public class Circle extends AbstractShape {
    private int radius;
    public void draw(Canvas c) {
        // code to draw the circle
    }
}
```

NB — this class is not abstract because it does provide a ‘concrete implementation’ of the abstract method `draw()`. Class `Circle` is not abstract, so we can create instances of `Circle`. 
Polymorphism

The code

```java
for (int i=0; i < shapes.length; i++) {
    shapes[i].draw(theCanvas);
}
```

works just as before.

This is an example of how inheritance, abstract classes and dynamic binding allow for **polymorphic code** i.e., code that works for a number of classes in a generic way.
Polymorphism

The code

```java
for (int i=0; i < shapes.length; i++) {
    shapes[i].draw(theCanvas);
}
```

works just as before.

This is an example of how inheritance, abstract classes and dynamic binding allow for **polymorphic code** i.e., code that works for a number of classes in a generic way.
Remember Interfaces?

Interfaces are completely abstract classes, but interfaces and abstract classes are treated differently in Java.

- The members of an interface can only be either:
  - constants (`final`)
  - abstract methods
- The keyword `abstract` is not used in interfaces.
- A class can implement any number of interfaces.

```java
class DoItAll implements ActionListener, MouseListener, ComponentListener {
    ...
}
```

But can extend only one class, abstract or not.
Remember Interfaces?

Interfaces are *completely abstract* classes, but interfaces and abstract classes are treated differently in Java.

- The members of an interface can only be either:
  - constants (‘final’)
  - abstract methods
- the keyword `abstract` is not used in interfaces
- a class can implement any number of interfaces

```java
class DoItAll implements ActionListener, MouseListener, ComponentListener {
    ...
}
```

but can extend only one class, abstract or not.
Remember Interfaces?

Interfaces are *completely abstract* classes, but interfaces and abstract classes are treated differently in Java.

- The members of an interface can only be either:
  - constants (‘final’)
  - abstract methods
- the keyword `abstract` is not used in interfaces
- a class can implement any number of interfaces

```java
class DoItAll implements ActionListener, MouseListener, ComponentListener {
    ...
}
```

but can extend only *one* class, abstract or not
Remember Interfaces?

Interfaces are *completely abstract* classes, but interfaces and abstract classes are treated differently in Java.

- The members of an interface can only be either:
  - constants (‘final’)
  - abstract methods
- the keyword `abstract` is not used in interfaces
- a class can implement any number of interfaces

```java
class DoItAll implements ActionListener, MouseListener, ComponentListener {
    ...
}
```

but can extend only *one* class, abstract or not
Abstract Classes for Generic Solutions: Back to Operators

The `getPrecedence()` methods of our operator classes were all very similar.

Why not capture this similarity of form with an abstract class? This class will:

- store the precedence value as a field,
- provide a concrete implementation of `getPrecedence()`,
- but no implementation of `toString()` (that is why it is abstract).
Abstract Classes for Generic Solutions: Back to Operators

The `getPrecedence()` methods of our operator classes were all very similar.

Why not capture this similarity of form with an `abstract` class? This class will:

- store the precedence value as a field,
- provide a concrete implementation of `getPrecedence()`,
- but no implementation of `toString()` (that is why it is abstract).
Abstract Classes for Generic Solutions: 
Back to Operators

The `getPrecedence()` methods of our operator classes were all very similar.

Why not capture this similarity of form with an `abstract` class? This class will:

- store the precedence value as a field,
- provide a concrete implementation of `getPrecedence()`,
- but no implementation of `toString()` (that is why it is abstract).
Abstract Classes for Generic Solutions: Back to Operators

The `getPrecedence()` methods of our operator classes were all very similar.

Why not capture this similarity of form with an abstract class? This class will:

- store the precedence value as a field,
- provide a concrete implementation of `getPrecedence()`,
- but no implementation of `toString()` (that is why it is abstract).
Abstract Classes for Generic Solutions: Back to Operators

The `getPrecedence()` methods of our operator classes were all very similar.

Why not capture this similarity of form with an `abstract` class? This class will:

- store the precedence value as a field,
- provide a concrete implementation of `getPrecedence()`,
- but no implementation of `toString()` (that is why it is abstract).
In the **Operators** class:

```java
private static abstract class Op
    implements Operator {

    private int precedence;

    public Op(int p) {
        precedence = p;
    }

    public int getPrecedence() {
        return precedence;
    }
}
```
The class is **private** because we don’t want other classes outside **Operators** making instances of **Op**: all uses of operators go through the **Operators** class.

The class is **abstract** because it doesn’t implement **toString()**. (it ‘inherits’ the abstract method by declaring that it implements **Operator**)

The field **precedence** is **private**. It might seem a good idea to make it **protected**, because it will be useful in subclasses of **Op** (esp. in the **toString()** methods). But there’s no need: the **getPrecedence()** method is public.

The class has a constructor — even though we can’t create instances of the class! Nevertheless, the functionality is useful (see later).
The class is **private** because we don’t want other classes outside **Operators** making instances of **Op**: all uses of operators go through the **Operators** class.

The class is **abstract** because it doesn’t implement **toString()**. (it ‘inherits’ the abstract method by declaring that it implements **Operator**)

The field **precedence** is **private**. It might seem a good idea to make it **protected**, because it will be useful in subclasses of **Op** (esp. in the **toString()** methods). But there’s no need: the **getPrecedence()** method is public.

The class has a constructor — even though we can’t create instances of the class! Nevertheless, the functionality *is* useful (see later).
Notes

- The class is `private` because we don’t want other classes outside `Operators` making instances of `Op`: all uses of operators go through the `Operators` class.

- The class is `abstract` because it doesn’t implement `toString()`. (it ‘inherits’ the abstract method by declaring that it implements `Operator`)

- The field `precedence` is `private`. It might seem a good idea to make it `protected`, because it will be useful in subclasses of `Op` (esp. in the `toString()` methods). But there’s no need: the `getPrecedence()` method is public.

- The class has a constructor — even though we can’t create instances of the class! Nevertheless, the functionality is useful (see later).
Notes

- The class is **private** because we don’t want other classes outside **Operators** making instances of **Op**: all uses of operators go through the **Operators** class.

- The class is **abstract** because it doesn’t implement **toString()**. (it ‘inherits’ the abstract method by declaring that it implements **Operator**)

- The field **precedence** is **private**. It might seem a good idea to make it **protected**, because it will be useful in subclasses of **Op** (esp. in the **toString()** methods). But there’s no need: the **getPrecedence()** method is public.

- The class has a constructor — even though we can’t create instances of the class! Nevertheless, the functionality is useful (see later).
Using the Abstract Class

To use the class `Op` in creating the constants `AND_OP`, etc., we could explicitly declare a concrete subclass of `Op` (i.e., with an implementation of `toString()`):

```java
in class Operators

public static class AndOperator
    extends Op {

    public AndOperator(int prec) {
        super(prec);
    }

    public String toString(Prop[] ps, int prec) {
        ...
    }
}
```
As before

Inheriting the `getPrecedence()` method

Even though `Op` is abstract, and we can’t create instances of the class, we *can* call the constructor to set the (private) precedence field. Odd, maybe, but useful.
public static class AndOperator
    extends Op {
    public AndOperator(int prec) {
        super(prec);
    }
    ...
}

As before
Inheriting the `getPrecedence()` method
Even though `Op` is abstract, and we can’t create instances of the class, we can call the constructor to set the (private) precedence field. Odd, maybe, but useful.
public static class AndOperator extends Op {
    public AndOperator(int prec) {
        super(prec);
    }

    ...
}

As before
Inheriting the `getPrecedence()` method
Even though `Op` is abstract, and we can’t create instances of the class, we can call the constructor to set the (private) precedence field. Odd, maybe, but useful.
in class Operators$AndOperator

```java
public String toString(Prop[] ps, int prec) {
    String s = ps[0].toString(getPrecedence()) + " and " + ps[1].toString(getPrecedence());
    if (prec < getPrecedence()) {
        s = "(" + s + ")";
    }
    return s;
}
```

We can’t refer to the private field precedence; we can refer to the public method `getPrecedence()`
public String toString(Prop[] ps, int prec) {

    String s = ps[0].toString(getPrecedence()) + " and " + ps[1].toString(getPrecedence());
    if (prec < getPrecedence()) {
        s = "(" + s + ")";
    }
    return s;
}

We can’t refer to the private field precedence; we can refer to the public method getPrecedence()
Using the Abstract Class

Or we can create an instance of an *anonymous* concrete subclass (with the concrete implementation of `toString()`).

```java
public static final Operator AND_OP
    = new Op(ANDPREC) {
        public String toString(Prop[] ps, int prec) {
            String s = ps[0].toString(getPrecedence())
                + " and "
                + ps[1].toString(getPrecedence());
            if (prec < getPrecedence()) {
                s = "(" + s + ")";
            }
            return s;
        }
    };
```
Using the Abstract Class

*Or* we can create an instance of an *anonymous* concrete subclass (with the concrete implementation of `toString()`).

```java
in class Operators

```public static final Operator AND_OP
 = new Op(AND_PREC) {
    
    public String toString(Prop[] ps, int prec) {
        String s = ps[0].toString(getPrecedence())
               + " and "
               + ps[1].toString(getPrecedence());
        if (prec < getPrecedence()) {
            s = "(" + s + ")";
        }
        return s;
    }
};```
Creating an instance of an anonymous class that extends `Op`
Passing the parameter to the `Op` constructor (so setting the `precedence` field)
We must provide a concrete implementation of the abstract method(s)
Creating an instance of an anonymous class that extends `Op`

Passing the parameter to the `Op` constructor
(so setting the `precedence` field)

We must provide a concrete implementation of the abstract method(s)
Creating an instance of an anonymous class that extends `Op`.

Passing the parameter to the `Op` constructor (so setting the `precedence` field).

We must provide a concrete implementation of the abstract method(s)
Notes

We can create an instance of an (anonymous) class that implements an interface using the pattern:

```java
new InterfaceName() {
    method-definitions
}
```

We can do a similar thing with abstract classes, using the pattern:

```java
new AbstractClassName(actualParametersToConstructor) {
    method-definitions
}
```
We can create an instance of an (anonymous) class that implements an interface using the pattern:

```java
new InterfaceName() {
    method-definitions
}
```

We can do a similar thing with abstract classes, using the pattern:

```java
new AbstractClassName(actualParametersToConstructor) {
    method-definitions
}
```
Why Stop Here?

The concrete implementations of `toString()` for the binary operators ‘and’ and ‘or’ are very similar (‘implies’ has a subtle difference...).

We introduce a class `BinOp` as a subclass of `Op` that will provide a concrete implementation of the `toString()` method for binary operators.

This avoids the need to rewrite the `toString()` method for ‘and’ and ‘or’ — clearly, this is only worth doing if we have enough binary operators (but remember we’re looking at a very simple language).
Why Stop Here?

The concrete implementations of `toString()` for the binary operators ‘and’ and ‘or’ are very similar (‘implies’ has a subtle difference...).

We introduce a class `BinOp` as a subclass of `Op` that will provide a concrete implementation of the `toString()` method for binary operators.

This avoids the need to rewrite the `toString()` method for ‘and’ and ‘or’ — clearly, this is only worth doing if we have enough binary operators (but remember we’re looking at a very simple language).
The concrete implementations of `toString()` for the binary operators ‘and’ and ‘or’ are very similar (‘implies’ has a subtle difference...).

We introduce a class `BinOp` as a subclass of `Op` that will provide a concrete implementation of the `toString()` method for binary operators.

This avoids the need to rewrite the `toString()` method for ‘and’ and ‘or’ — clearly, this is only worth doing if we have enough binary operators (but remember we’re looking at a very simple language).
Infix Binary Operators

private static class BinOp extends Op {
    private String opName;
    BinOp(int prec, String s) {
        super(prec);
        opName = s;
    }

    // toString() to come here
}

inheriting `getPrecedence()`, and setting the `precedence` field storing the name of the operator
Infix Binary Operators

In class Operators

```java
private static class BinOp extends Op {
    private String opName;
    BinOp(int prec, String s) {
        super(prec);
        opName = s;
    }
    // toString() to come here
}

inheriting getPrecedence(), and setting the precedence field
storing the name of the operator
```
private static class BinOp extends Op {
    private String opName;
    BinOp(int prec, String s) {
        super(prec);
        opName = s;
    }
    // toString() to come here
}

inheriting getPrecedence(), and setting the precedence field storing the name of the operator
Infix Binary Operators

```java
public String toString(Prop[] as, int prec) {
    String s = as[0].toString(getPrecedence())
    + " " + opName + " "
    + as[1].toString(getPrecedence());
    if (prec < getPrecedence()) {
        s = "(" + s + ")";
    }
    return s;
}
```

Generalising the code for ‘and’ and ‘or’
Using BinOp

Now we can just declare:

```
public static final Operator AND =
    new BinOp(AND_PREC, "and");

public static final Operator OR =
    new BinOp(OR_PreC, "or");
```
Implies is a Special Case

The `toString()` method can be overridden ‘on the fly’, using the notation for anonymous classes.

```java
public static final Operator IMPLIES =
    new BinOp(IMPLIESPREC, "implies")
    {
        public String toString(Prop[] as, int p)
        {
            // code goes here
        }
    };
```
And On And On. . .

It would also be useful to have another subclass of $\text{Op}$ for constants, including variables. . . .
Encapsulation

We say that a class \textit{encapsulates} an abstract data type if it implements that data type correctly.

Recall that an abstract data type consists of a set of values (the data values), together with certain operations for manipulating those values.

A class correctly implements the data type if it:

- provides a representation for the data values,
- provides methods for the data type operations, and
- only those \textit{operations} are provided.
We say that a class **encapsulates** an abstract data type if it implements that data type correctly.

Recall that an abstract data type consists of a set of values (the data values), together with certain operations for manipulating those values.

A class correctly implements the data type if it:
- provides a representation for the data values,
- provides methods for the data type operations, and
- only those operations are provided.
Encapsulation

Only the specified operations are available as methods, and the implementation details are hidden by making the fields private.

The same considerations are applicable to the new features we have seen: inner, static, and anonymous classes.

Their scope and scope-modifiers can and should be used to ensure encapsulation.
The class **Prop** is intended to represent an abstract data type of Boolean terms. (They are *implemented* as tree structures.)

This is a simple data structure: the only method required is to get a string representation with brackets only where necessary.

How successful have we been?
That’s All, Folks!

Summary

- Abstract Classes
- Overriding
- Anonymous Classes

Next:

Things fall apart