Advanced Object-Oriented Programming (CSE210)

Lecture 5b: Implementing Propositional Logic

Wei Wang
CSSE

The Story So Far

- A class, Prop, to represent terms of propositional logic;
- An interface, Operator, to represent operators;
- Operators are responsible for generating a string representation of a term (Proposition).
- The goal is to generate strings containing only necessary brackets.

What's Left?

- We have the making of a solution, but the details have to be attended to:
  - for each operator, a new class implementing the Operator interface has to be declared somewhere
  - whenever a Prop instance is to be created, an instance of one of the Operator classes will have to be created — but we only need one instance of each of these classes
  - the constant MAX_PREC has to be defined somewhere.

Using the Data Representation

- We've forgotten about one of the required methods for our task: given a proposition written as a string, create the Prop instance that represents that proposition (this is 'parsing').
- The parser (Parser.java) reads a string and, if the string is a well-formed term, constructs a Prop instance to represent that term.
- In order to construct terms, the parser requires information about the precedence of operators (including MAX_PREC).
- In cases like this, it is often useful to keep all constants together in one place.

A Repository Class

- We introduce a general *repository* class, Operators, to keep all the data needed by the client class (Parser):
  - the precedence of the operators;
  - instances of the operators.

Instances of Operators

- Instances of the various operator classes have no local state, so there is no need for multiple instances of any of these classes.
- They can therefore be treated as constants:
- NB: we don't need to declare constructors with no parameters — Java provides these 'for free'.

Operators.Java

```
public static final Operator IMPLIES_OP = new ImpliesOperator();
public static final Operator AND_OP = new AndOperator();
...
```

Information-Hiding

- Why should the classes AndOperator, etc., be public?
- In fact, there is no need for these classes to be public. It would be better to hide them, so that all clients go through the repository class, Operators.
- This has the benefit of *localising* declarations, and minimising the impact of any future changes (if the implementation of an operator is changed, then only the class Operators need be modified).

Inner Classes

- It is possible to declare one class *inside* another class. These are called inner classes.
- Like any other members, inner classes can be declared public, protected, or private.

Inner Classes cont'd

```
public class Operators {
  private class AndOperator implements Operator {
    private static final int AND_PREC = 40;
    public int getPrecedence() {
      return AND_PREC;
    }
    …
  }
  …
}
```

Scope

- AndOperator is not visible outside Operators
- All members of Operators are visible within AndOperator, including private members (e.g., other private classes, such as ImpliesOperator).
- In this example, private members of one inner class are not visible within other inner classes (e.g., if AndOperator has a private member, then that will not be visible within ImpliesOperator).

But . . .

- However, we do get a compiler error:

```
public class Operators {
  public static final Operator AND_OP = new
AndOperator();
  private class AndOperator ...{
    ...
  }
  ...
}
```

```
Operators.java:230: non-static variable this
cannot be referenced from a static context
public static final Operator AND_OP = ...
```

Static Classes

- We want the constants, AND_OP, etc, to be static (we do not want multiple instances of these).
- The class AndOperator is a *member* of Operators, and in order to be referred to from a static context such as:
  - public static final Operator AND_OP = new AndOperator();
  - it must be declared static.

Static Classes cont'd

- A static inner class must not depend on the local state (e.g., non-static fields) of its enclosing class.
- It is only allowed to refer to static members of its enclosing class.
- Just as a static method is only allowed to refer to static members.
- In our example, all members of Operators are static, so this is not a problem.

Version 1.0 almost finished!

```
public class Operators {
  private static class AndOperator ...{
    private static final int AND_PREC = 40;
    ...
  }

  public static final AndOperator AND_OP = new
AndOperator();
  public static final int AND_PREC =
AND_OP.getPrecedence();
  ...
}
```

Constants and Variables

- If we complete all static inner class for OR, IMPLIES, etc
- We're getting an awful lot of inner classes, whose names we can't refer to outside class Operators.

Operators.java

```
private class TRUE_OP {
  public String toString(...)  {
    return "true";
  }
  ...
}
```

The Class with No Name

- In Prop, we don't refer to the class AndOperator.
- In fact, we *can't*, because AndOperator is declared private in Operators — the name is not in scope in Prop.
- Java allows programmers to declare classes with no names.
- These are called anonymous classes.
- The main benefit is that programs are shorter and clearer, and code can be written close to where it's used.

Anonymous Classes

- Anonymous classes are used when an instance is created that belongs to a class that implements an interface.
- In this case, the name of the interface is used as a constructor, followed by a class definition.

```
instance = new InterfaceName() {
  method-definitions
};
```

For Example

- This is equivalent to the version with AndOperator as inner class (v1.0).

Operators.java v1.1

```
public static Operator AND_OP =
  new Operator() {
    public String toString(Prop[] a, int p) {
      ...
    }

    public int getPrecedence() {
      ...
    }
  };
```

And so on…

- This is equivalent to the version with AndOperator as inner class (v1.0).

Operators.java v1.1

```
public Operator TRUE_OP =
  new Operator() {
    public String toString(Prop[] a, int p) {
      return "true";
    }

    public int getPrecedence() {
      return 0;
    }
  };
```

Another Example

In some GUI program

```
Button qB = new Button("Quit");
qB.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent e){
     System.exit(0);
    }
  });
```

Propositional Variables

- A propositional variable is like a constant (no operands), but its toString() method is going to depend upon its name.
- For example, the variable 'a should return the string "a", the variable 'b should return "b", and so on and on.
- We can't write a class for each possible variable name (there are infinitely many), but we can write a method that takes a name (a String) as parameter, and returns the desired Operator.
- The method will return an instance of an anonymous class that implements Operator in the required way.

Propositional Variables cont'd

class Operators v1.1

```
public static Operator makeVar(final String name)
{
  return new Operator() {
    public String toString(Prop[] as, int p) {
      return name;
    }

    public int getPrecedence() {
      return 0;
    }
  };
}
```

Notes

- The method is static because it doesn't use 'local state' (i.e., fields in class Operators (there are none, anyway)).
- The value returned is an instance of an anonymous class that implements Operator, with the specified methods. For example,
  - Operators.makeVar("a").getPrecedence() is 0,
  - Operators.makeVar("a").toString(new Prop[]{}, 0) is "a".
- The parameter is declared to be final — because the compiler requires it: whenever a variable is used inside an inner class, it must be declared as final.

End of Lecture 5b

- Summary
  - Inner classes
  - Static classes
  - Anonymous classes
  - Final variables

- Next:
  - Abstract classes