Advanced Object-oriented Programming

Lecture 14
Implementing Prop
Java Implementation

Skeleton of class Prop

```java
public class Prop {
    public Prop and(Prop p1, Prop p2) { ... } // similarly for `or`, etc.
    public String printAllBrackets() { ... }
    public String toString() { ... }
}
```

We need a data representation.
What Do We Need to Represent?

A Proposition is really just
- an operator applied to some
- operands

For example, not 'a or 'b and 'c

\[
\text{or(not('a),}
\text{ and('b,
\text{ 'c
\text{ )
\text{ )}
}\]

Trees!
What Do We Need to Represent?

A Proposition is really just

- an operator applied to some
- operands

For example, not 'a or 'b and 'c

```
  or(not('a),
      and('b, 'c)
  )
```

Trees!
What Do We Need to Represent?

A **Proposition** is really just
- an **operator** applied to some
- **operands**

For example, not 'a or 'b and 'c

```
or(not('a),
    and('b,
        'c
    )
)
```

Trees!
What Do We Need to Represent?

A Proposition is really just
- an operator applied to some
- operands

For example, not 'a or 'b and 'c

Trees!
What Do We Need to Represent?

A Proposition is really just

- an operator applied to some
  - operands

For example, not 'a or 'b and 'c

Trees!
Trees in Maude

```maude
fmod BINARY_TREE{ X :: TRIV } is
  protecting INT .
  sort BTree .
  op null : -> BTree [ ctor ] .
  op _/\_ : BTree X$Elt BTree -> BTree [ ctor ] .
  op height : BTree -> Int .
  var E : X$Elt .
  vars T1 T2 : BTree .
  eq height(null) = 0 .
  eq height(T1 / E \ T2) = max(height(T1), height(T2)) + 1 .
endfm
```

Like `<x>` in Java
Trees in Maude

```maude
fmod BINARY_TREE{ X :: TRIV } is
  protecting INT .
  sort BTree .
  op null : -> BTree [ ctor ] .
  op _/\_ : BTree X$Elt BTree -> BTree [ ctor ] .
  op height : BTree -> Int .
  var E : X$Elt .
  vars T1 T2 : BTree .
  eq height(null) = 0 .
  eq height(T1 / E \ T2) = max(height(T1), height(T2)) + 1 .
endfm
```

Like `<X>` in Java
Trees in Maude

Note that null and _/_/\_ are **structural** operations (ctors)

i.e., these operations are intended to allow us to write terms that represent trees; for example, the term

\[(null / 1 \ null) / 3 \ (null / 5 \ null) / 6 \ null\]

represents a tree that might be pictured as (null subtrees aren’t shown):
Trees in Maude

Because `null` and `/_\_` are the only two structural operations, any tree is either `null`, or of the form `T1 / I \ T2` for some integer `I` and (sub)trees `T1` and `T2`.

This is why there are two equations defining `height`: one equation for each of the two possible forms of trees.

\[
\begin{align*}
\text{eq} & \quad \text{height(null)} = 0 . \\
\text{eq} & \quad \text{height(T1 / E \ T2)} = \max(\text{height(T1)}, \text{height(T2)}) + 1 .
\end{align*}
\]
Trees in Java. . .

. . . are a bit more complicated.

We’ll re-use some of the ideas from linked lists, and have nodes storing values, as well as pointers to the subtrees
Trees in Java...

...are a bit more complicated.

We’ll re-use some of the ideas from linked lists, and have nodes storing values, as well as pointers to the subtrees.
Trees in Java

```
leftSubTree  value  rightSubTree
```
Trees in Java
Trees in Java
Trees in Java
This class will give the ‘linked tree’ structure. The constructors are very similar to linked-list constructors.
Trees in Java

```java
public class BinaryTree<X> {
    private TreeNode root;

    public BinaryTree(X v) {
        root = new TreeNode(null, v, null);
    }

    public BinaryTree(TreeNode l, X v, TreeNode r) {
        root = new TreeNode(l, v, r);
    }

    // This class will give the 'linked tree' structure.
    // The constructors are very similar to linked-list constructors.
```
This class will give the ‘linked tree’ structure. The constructors are very similar to linked-list constructors.
private static class TreeNode {
    private X value;
    private TreeNode leftSubTree;
    private TreeNode rightSubTree;

    public TreeNode(TreeNode l, X v, TreeNode r) {
        value = v;
        leftSubTree = l;
        rightSubTree = r;
    }
}

Question: how is this different from doubly-linked lists?
class BinaryTree, contd.

private static class TreeNode {

    private X value;
    private TreeNode leftSubTree;
    private TreeNode rightSubTree;

    public TreeNode(TreeNode l, X v,
                     TreeNode r) {
        value = v;
        leftSubTree = l;
        rightSubTree = r;
    }
}

Question: how is this different from doubly-linked lists?
Computing Height

public int height() {
    return (root == null) ? 0 : root.height();
}

In this class, we can test if the tree is empty:
if it is, the height is zero;
if it’s not,
pass the computation of height onto the TreeNode class
Public int height() {
    return (root == null) ? 0 : root.height();
}

In this class, we can test if the tree is empty:
if it is, the height is zero;
if it's not,
pass the computation of height onto the TreeNode class.
Computing Height

In class BinaryTree

```java
public int height() {
    return (root == null) ? 0 : root.height();
}
```

In this class, we can test if the tree is empty: if it is, the height is zero; if it's not, pass the computation of height onto the TreeNode class.
Computing Height

In class BinaryTree

```java
public int height() {
    return (root == null) ? 0 : root.height();
}
```

In this class, we can test if the tree is empty:
if it is, the height is zero;
if it's not,
pass the computation of `height` onto the `TreeNode` class
In this class, we can test if the tree is empty:
if it is, the height is zero;
if it's not,
pass the computation of height onto the TreeNode class
Computing Height

private int height() {
    int leftHeight, rightHeight;
    leftHeight = (leftSubTree == null) ?
                0 : leftSubTree.height();
    rightHeight = (rightSubTree == null) ?
                  0 : rightSubTree.height();
    return Math.max(leftHeight, rightHeight) + 1;
}

Like the Maude equations: but we need to calculate the heights of the left and right subtrees.
Computing Height

in class TreeNode

private int height() {
    int leftHeight, rightHeight;
    leftHeight = (leftSubTree == null) ? 0 : leftSubTree.height();
    rightHeight = (rightSubTree == null) ? 0 : rightSubTree.height();

    return Math.max(leftHeight, rightHeight) + 1;
}

Like the Maude equations: but we need to calculate the heights of the left and right subtrees.
Computing Height

in class TreeNode

```java
private int height() {
    int leftHeight, rightHeight;
    leftHeight = (leftSubTree == null) ? 0 : leftSubTree.height();
    rightHeight = (rightSubTree == null) ? 0 : rightSubTree.height();
    return Math.max(leftHeight, rightHeight) + 1;
}
```

Like the Maude equations: but we need to calculate the heights of the left and right subtrees.
private int height() {

    int leftHeight, rightHeight;

    leftHeight = (leftSubTree == null) ? 0 : leftSubTree.height();
    rightHeight = (rightSubTree == null) ? 0 : rightSubTree.height();

    return Math.max(leftHeight, rightHeight) + 1;
}

Like the Maude equations: but we need to calculate the heights of the left and right subtrees.
We can see that tree structures can be implemented in Java using references (‘pointers’) to nodes/trees in a similar way to the implementation of linked lists. (Note also that computing values like height often involve recursion.)

However, binary trees don’t quite meet our needs for propositions, as some operators take two arguments, while others (e.g., not) take only one.
We can see that tree structures can be implemented in Java using references (‘pointers’) to nodes/trees in a similar way to the implementation of linked lists. (Note also that computing values like height often involve recursion.)

However, binary trees don’t quite meet our needs for propositions, as some operators take two arguments, while others (e.g., not) take only one.
We can see that tree structures can be implemented in Java using references (‘pointers’) to nodes/trees in a similar way to the implementation of linked lists. (Note also that computing values like height often involve recursion.)

However, binary trees don’t quite meet our needs for propositions, as some operators take two arguments, while others (e.g., not) take only one.

```
 not
  | 'a
 or
          and
             'b
             'c
```
Prop’s Fields

```java
public class Prop {
    private Operator op;
    private Prop[] operands;
    ...
}
```

**Operator** — some representation of Boolean operators; maybe as Strings?
An array of subterms, to store a variable number of operands — binary ops will require an array of length 2; unary ops an array of length 1
Prop’s Fields

public class Prop {
    private Operator op;
    private Prop[] operands;
    ...

Operator — some representation of Boolean operators; maybe as Strings?
An array of subterms, to store a variable number of operands — binary ops will require an array of length 2; unary ops an array of length 1
Prop’s Fields

```java
public class Prop {
    private Operator op;
    private Prop[] operands;
    ...
}
```

**Operator** — some representation of Boolean operators; maybe as Strings?

An array of subterms, to store a variable number of operands — binary ops will require an array of length 2; unary ops an array of length 1
Prop’s Methods

public String toString() {
    return toStringPrec(48);
}

public String toStringPrec(int prec) {
    ???
}

Recall PropLog.maude
Since the form of the string depends on the operator — infix or prefix — we should consider the candidate class Operator
Prop's Methods

```java
public String toString() {
    return toStringPrec(48);
}

public String toStringPrec(int prec) {
    ???
}
}
```

Recall PropLog.maude

Since the form of the string depends on the operator — infix or prefix — we should consider the candidate class `Operator`
Recall PropLog.maude

Since the form of the string depends on the operator — infix or prefix — we should consider the candidate class **Operator**
The candidate class `Operator` is taken seriously because these things depend upon operators:

- the number of arguments (operands) in a term;
- the form of the string representation (e.g., prefix operators such as ‘not’ come before their argument; infix operators such as ‘and’ come between their two arguments)
Class Operator

Instances of this class will correspond to ‘not’, ‘and’, etc.

fields:
   ???

methods:
   toString(Prop[] os, int p)
      — generate the string, given the array of operands os, and the precedence p of the ‘operator above’

   getPrecedence()
      — return the precedence of the operator
This decision allows us to fix class Prop:

```java
class Prop {
    private Operator op;
    private Prop[] operands;

    public Prop(Operator o, Prop[] os) {
        op = o;    operands = os;
    }

    public String toString() {
        return toStringPrec(48);
    }

    private int String toStringPrec(int prec) {
        return op.toString(operands, prec);
    }
}
```
Delegating Responsibility

The decision to make a class of operators allows us to:

- complete the class Prop straightforwardly;
- give a generic solution to the problem of string representations of terms;
- neatly separate data representation from data manipulation.
A Generic Solution?

in class Prop

public String toString() {
    op.toString(operands);
}

Rather than:

if (op.equals("not")) {
    ...
} else if (op.equals("and")) {
    ...
} else ...

The Class Operator?

There will be instances of class Operator for `not`, `and`, `or`, etc.

But each instance will need its own method for generating strings (appropriate to the operator that the instance represents).

```java
public class Operator {
    public String toString(Prop[] ps) {
        ???
    }
}
```

So how can we write one method to suit all operators?
The Class Operator?

There will be instances of class Operator for not, and, or, etc. But each instance will need its own method for generating strings (appropriate to the operator that the instance represents).

```java
public class Operator {
    public String toString(Prop[] ps) {
        ???
    }
}
```

So how can we write one method to suit all operators?
Interfaces are Generic and Particular

- Each instance of class Operator will correspond to a Boolean operator
- each instance will have its own method of generating a string representation

What is needed is an interface.

An interface specifies the methods in a class (names, return and parameter types).

Each class that implements an interface implements the methods in the interface (possibly in different ways).

But classes that use the interface can do so in a generic way (as in class Prop) — This is called polymorphism.
Interfaces are Generic and Particular

- Each instance of class Operator will correspond to a Boolean operator
- each instance will have its own method of generating a string representation

What is needed is an interface. An interface specifies the methods in a class (names, return and parameter types).

Each class that implements an interface implements the methods in the interface (possibly in different ways).

But classes that use the interface can do so in a generic way (as in class Prop) — This is called polymorphism.
Interfaces are Generic and Particular

- Each instance of class Operator will correspond to a Boolean operator
- each instance will have its own method of generating a string representation

What is needed is an interface.

An interface specifies the methods in a class (names, return and parameter types).

Each class that implements an interface implements the methods in the interface (possibly in different ways).

But classes that use the interface can do so in a generic way (as in class Prop) — This is called polymorphism.
Interfaces are Generic and Particular

- Each instance of class Operator will correspond to a Boolean operator
- Each instance will have its own method of generating a string representation

What is needed is an interface.

An interface specifies the methods in a class (names, return and parameter types).

Each class that implements an interface implements the methods in the interface (possibly in different ways).

But classes that use the interface can do so in a generic way (as in class Prop) — This is called polymorphism.
The Proposed Solution

```java
public interface Operator {
    public String toString(Prop[] os, int prec);
    public int getPrecedence();
}
```

The name `Operator` can now be used as a type; e.g.,

```java
private Operator op;
```

This means any class that is declared to implement the `Operator` interface. Any class that is declared to implement `Operator` must contain a public method called `toString` with the same return and parameter types. Similarly for `int getPrecedence()`. Note the semicolons!
The Proposed Solution

```java
public interface Operator {
    public String toString(Prop[] os, int prec);
    public int getPrecedence();
}
```

The name Operator can now be used as a type; e.g.,

```java
private Operator op;
```

This means any class that is declared to implement the interface Operator.

Any class that is declared to implement Operator must contain a public method called toString with the same return and parameter types.

Similarly for int getPrecedence()

Note the semicolons!
The Proposed Solution

```java
public interface Operator {
    public String toString(Prop[] os, int prec);
    public int getPrecedence();
}
```

The name `Operator` can now be used as a type; e.g.,

```java
private Operator op;
```

This means any class that is declared to implement the interface `Operator`.
Any class that is declared to implement `Operator` must contain a public method called `toString` with the same return and parameter types.
Similarly for `int getPrecedence()`
Note the semicolons!
The name Operator can now be used as a type; e.g.,

```java
private Operator op;
```

This means any class that is declared to implement the interface `Operator`.

Any class that is declared to implement `Operator` must contain a public method called `toString` with the same return and parameter types.

Similarly for `int getPrecedence()`

Note the semicolons!
The name `Operator` can now be used as a type; e.g.,

```java
private Operator op;
```

This means any class that is declared to **implement** the interface `Operator`. Any class that is declared to implement `Operator` **must** contain a public method called `toString` with the same return and parameter types.

**Similarly for** `int getPrecedence()`

Note the semicolons!
public interface Operator {
    public String toString(Prop[] os, int prec);
    public int getPrecedence();
}

The name *Operator* can now be used as a type; e.g.,

```java
private Operator op;
```

This means any class that is declared to implement the interface *Operator*.

Any class that is declared to implement *Operator* must contain a public method called *toString* with the same return and parameter types.

Similarly for *int getPrecedence()*

Note the semicolons!
class AndOperator implements Operator {
    private static final int ANDPREC = 40;
    public String toString(Prop[] os, int prec) {
        String s = os[0].toString(ANDPREC) + " and " + os[1].toString(ANDPREC);
        if (prec < ANDPREC) {
            s = "(" + s + ")";
        }
        return s;
    }
    ...
}
The Proposed Solution: And

class AndOperator implements Operator {
    private static final int AND_PREC = 40;
    public String toString(Prop[] os, int prec) {
        String s = os[0].toString(AND_PREC) +
                    " and " +
                    os[1].toString(AND_PREC);
        if (prec < AND_PREC) {
            s = "(" + s + ")";
        }
        return s;
    }
    ...
}
The Proposed Solution: And

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

Note that the compiler will signal an error if a class that declares that it implements an interface does not include implementations of all the methods in that interface.
The Proposed Solution: And

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

Note that the compiler will signal an error if a class that declares that it implements an interface does not include implementations of all the methods in that interface.
public class OrOperator implements Operator {
    private static final int OR_PREC = 44;

    public String toString(Prop[] os, int prec) {
        String s = os[0].toString(OR_PREC) +
                " or " +
        os[1].toString(OR_PREC);

        if (prec < OR_PREC) {
            s = "(" + s + ")";
        }

        return s;
    }

    ...
}

// getPrecedence is similar to and's
The Proposed Solution: Not

public class NotOperator implements Operator {
    private static final int NOT_PREC = 30;
    public String toString(Prop[] os, int prec) {
        String s = "not " + os[0].toString(NOT_PREC);
        if (prec < NOT_PREC) {
            s = "(" + s + ")";
        }
        return s;
    }
    ...
    // getPrecedence is similar to and's
}
In package java.awt.event

```java
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

And in, e.g., java.awt.Button

```java
public void addActionListener(
    ActionListener l
) {
    ...
}
```

Actual parameters can be any instance of any class that is declared to implement ActionListener
Other Interfaces

In package java.awt.event

```java
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

And in, e.g., java.awt.Button

```java
public void addActionListener(
    ActionListener l
) {
    ...
}
```

Actual parameters can be any instance of any class that is declared to implement ActionListener
Other Interfaces

In package java.awt.event

```java
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

And in, e.g., java.awt.Button

```java
public void addActionListener(
    ActionListener l
) {
    ...
}
```

Actual parameters can be any instance of any class that is declared to implement ActionListener
Implementing ActionListener

in some class...

private class Quitter
    implements ActionListener {
        public void ActionPerformed(ActionEvent e) {
            System.exit(0);
        }
    }

Button b = new Button("Quit");
b.addActionListener(new Quitter());
More Methods

Interfaces can contain any number of methods (and, incidentally, constants)

In package java.awt.event

```java
public interface MouseListener {
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

Any class implementing this must contain ‘concrete implementations’ of all five methods.
More Methods

Interfaces can contain any number of methods (and, incidentally, constants)

In package java.awt.event

```java
public interface MouseListener {
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

Any class implementing this must contain ‘concrete implementations’ of all five methods.
More Methods

Interfaces can contain any number of methods (and, incidentally, constants)

In package java.awt.event

```java
public interface MouseListener {
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

Any class implementing this must contain ‘concrete implementations’ of all five methods.
Summary

- Binary trees
- Trees with arbitrary branching factor (number of subtrees)
- Interfaces

Next:

inner and anonymous classes