Inheritance - Assignment 5

- Expr objects
  - What they look like?

- Inheritance hierarchy
  - Inheriting instance variables and methods
    - How to do method lookup?
    - Polymorphism
  - Abstract classes

- Complex objects
  - Recursive methods
  - Structural equality
Expr Objects

- **Examples of expressions**
  - 1, 2 + 3, (4 + 51) * 14, 16 - 1, -3, -(6 - 4)

- **Operators:** +, -, *, /, % (unary minus)
  - Each operator takes one or two Expr operands
  - Can be simple constants (e.g., 1, 50, 3) or subexpressions themselves, as %3 or (4 + 51) the first operand in (4 + 51) * 14 (see below)
Expr Objects - Structure

(4 + 51) * 14

Times_Expr (Plus_Expr (Const_Expr, Const_Expr), Const_Expr)

operand1

operand2
Inheritance Hierarchy - Instance Variables

Object

Expr operator

Const.Expr

Unary.Expr

operand

Uminus

Binary.Expr

operand1

operand2

Plus.Expr

Times.Expr

Minus.Expr

Divide.Expr
Inheriting Instance Variables

- A Times_Expr object consists of an operator (in Expr), and operand1, operand2 (in Binary_Expr)

  - operator
  - operand1
  - operand2

  Must be initialized in Expr
  Must be initialized in Binary_Expr

Every Times_Expr object is also a Binary_Expr object and an Expr object, since Times_Expr class extends Binary_Expr and Binary_Expr class extends Expr
Example I : $1 + 2$

Plus.Expr object:

```
operator
operand1
operand2
```

or pictorially:

```
+   
|    
|    
1 --+-- 2
```
Example II : \((4 + 51) \times 14\)

Time.Expr object:

\[
\begin{array}{c}
\times \\
\text{Plus.Expr} \\
\text{Const.Expr}(14)
\end{array}
\]

or pictorially:

\[
\begin{array}{c}
\times \\
+ \\
4 \\
51 \\
14
\end{array}
\]
Inheritance Hierarchy - Inheriting Methods

Object

Expr

operator

Const_Expr

Unary_Expr

operand
c

operator

operand

getOperand()

Operand

getOperand()

commute()

Plus_Expr

Minus_Expr

operand1

operand2

Binary_Expr

getFirstOperand()

getSecondOperand()

Times_Expr

Divide_Expr

commute()
Inheritance

• Class Times_Expr extends class Binary_Expr which extends class Expr

• If times is a Times_Expr object, where do we find methods which can be invoked on times?
  – times.commute()
  – times.getFirstOperand(), times.getSecondOperand()
Method Lookup

- Without inheritance, method must be in class of receiver
- With inheritance, method used is in class of receiver or its “closest” ancestor class
  - Method lookup starts in class of receiver and proceeds up the tree until first method of same name is found
    - `commute()` is in Times.Expr
    - `getFirstOperand(), getSecondOperand()` are in Binary.Expr
Abstract Classes

- Can define methods (and implementations) in an abstract class which can be inherited by subclasses
- Can also contain instance variables to be inherited by subclasses
- Abstract classes in Assignment 5: \textit{Expr, Unary\_Expr, Binary\_Expr}
  - Non-abstract classes are at leaves of the \textit{Expr} inheritance tree
Abstract Classes

• Useful when you want to define only part of an implementation

• Abstract classes
  – Abstract methods are signatures of promised methods to be provided in subclasses of the abstract class
    – Can provide these through definition or inheritance
  – No objects can be created as instances of an abstract class
    – Because abstract method implementations don’t exist
Assignment 5: Expressions

Diagram:

```
Object
  | Expr
  +------------------------+------------------------+
  | Const_Expr             | Unary_Expr             |
  | operand: c             | operator: operand      |
  | equals()               | equals()               |
  | eval()                 | eval()                 |

Unary_Expr
  +------------------------+------------------------+
  | operand: operand1      | operand: operand2      |
  | equals()               | equals()               |
  | eval()                 | eval()                 |

Binary_Expr
  +------------------------+------------------------+------------------------+
  | operand: operand1      | operand: operand2      |
  | equals()               | equals()               |
  | getFirstOperand()      | getSecondOperand()     |
  | eval()                 | eval()                 |
```

Abstract methods:

- `eval()`
- `equals()`
Eval()  

- Abstract in Expr, only signature provided  
- Implementation provided in Const_Expr, Plus_Expr, Times_Expr, Minus_Expr, Divide_Expr  
- Provides a way to evaluate an Expr object
Constructors with Inheritance

- With inheritance, within a constructor for a subclass object, constructors for the superclass are implicitly called by system.
- If instance variable data needs initialization in a superclass, can use `super` to explicitly call constructor of superclass with initialization values.
Constructors - Example

```java
public abstract class Expr extends Object
    { Expr(String s) // constructor
        { operator = s; }
    }

public abstract class Unary_Expr
    { Unary_Expr(Expr e, String s)
        { super(s); operand = e; }
    }

public class Uminus
    { Uminus (Expr e, String s)
        { super(e,s); }
    }

Uminus u = new Uminus (new Const_Expr(3),"%")
```
Super

- **Super** acts as a reference to an object as an instance of its superclass
- The reference to `super` in the `Unary_Expr` class constructor, means call the `Expr` constructor with argument `String s`.
  - Implicitly, when a subclass object is created, the constructor of the superclass is called before anything else is done in the subclass constructor
  - If arguments are needed, `super(<args>)` is used to call the superclass constructor explicitly.
Objects

• Simple objects have instance variables of primitive types

• Complex objects have instance variables which themselves are objects
  – e.g., Expr objects with instance variables that are other Expr objects
  – Why needed? allows for all possible kinds of subexpressions:
    \[ 1 + 2, \ 1 + (3 + 4), \ 1 + (2 * 5), \ 1 + \%4, \text{ etc.} \]

• Requires us to define operands as Expr’s
Equals( )

- Tests **structural equality**
  - Two `Expr` objects are **structurally equal** if their operand(s) are structurally equal and they have the same operator
  - i.e., `Plus_Expr` objects can only be equal to other `Plus_Expr` objects
  - e.g., `2 + 1` is equal to `2 + 1`, but not to `1 + 2`;
    `2*3 + 4` is equal to `(2*3) + 4`, but not to `(2*2)+6`
- Provided by inheritance for all kinds of binary or unary expressions, defined in `Const_Expr`
Equals()}

- Equals() is example of a useful recursive function on Expr objects

- Const_EXPR objects are equal to other Const_EXPR objects representing the same integer value
  - 2 equals 2, 2 not equal to 5

- Unary_EXPR objects are equal only to other Unary_EXPR objects, if their operands are equal and their operator is the same
  - %1 equal to %1 but not equal to %(1*1)
Equals() 

- Binary_Expr objects are equal if both are Binary_Expr objects, their first operands are equal, their second operands are equal and their operators are equal
- Remember this is structural equality NOT equal in value (such as 1 + 3 and 5 + %1)
- Can think of it as “sliding” one expression tree over another and “matching” shape and nodes
- Example of polymorphism, where a function can take parameters of different types
Equals( )

in Const_Expr:

public boolean equals(Expr other)
{
    if (!(other instanceof Const_Expr)) return false;
    else return (this.c == (other.eval()));
}

instanceof is a way of checking the runtime class membership of an object. The expression returns true when other is a Const_Expr object and false otherwise;

method checks that other is a Const_Expr object and if so, checks its value versus the value of the receiver object

2 equals? 3
equals( )

in Unary_Expr:

public boolean equals(Expr other)
{
  if (!(other instanceof Unary_Expr)) return false;
  else if
    ((other.getOperator()).equals(this.getOperator())) &&
    (operand.equals(((Unary_Expr) other).getOperand()))
    return true;
  else return false;
}

this

<table>
<thead>
<tr>
<th>% equals? %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

other

<table>
<thead>
<tr>
<th>% equals?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

+ 3

4
in Binary_Expr:

```java
public boolean equals(Expr other) {
    if (!(other instanceof Binary_Expr)) return false;
    else if (!((this.getOperator()).equals(other.getOperator())))
        return false;
    else return ((this.getFirstOperand()).equals((Binary_Expr)other).getFirstOperand())
        && (this.getSecondOperand()).equals((Binary_Expr)other).getSecondOperand());
}
```

*other* can be a Const_Expr, Plus_Expr, Times_Expr, Divide_Expr, Minus_Expr, or Uminus
Equals() in Binary.Expr

equals?

\[ + \]
\[ 1 \quad 2 \]
\[ 1 \quad 3 \]

equals?

\[ + \]
\[ 1 \quad 2 \]
\[ 1 \quad 3 \]

\[ + \]
\[ 1 \quad 2 \]
\[ 1 \quad 3 \]

\[ * \]
\[ 1 \quad 3 \]

\[ + \]
\[ 1 \quad 2 \]
\[ 1 \quad 4 \]
Equals( ) in Binary_Expr

1. see both are Binary_Expr’s then check operators same

2. check first operands same through call which in this case calls equals in Const_Expr
3. check second operands through call to equals() in Unary_Expr.
returns false since 2nd Expr is not unary!

Methods we called in example (in order):

equals() in Binary_Expr
getOperator()
getFirstOperand()
equals() in Const_Expr
getSecondOperand()
equals() in Binary_Expr