OPENING NOTES

Inheritance provides some benefits that we value highly: flexibility, extensibility, modifiability.

Inheritance also imposes some costs on a language and programs.

- Since methods are resolved at run-time, there is a look-up cost.
- Inheritance hierarchies add complexity to a program.

Many other costs mentioned by Budd (program size and message-passing overhead) are not costs of inheritance so much as costs associated with a way of implementing an OO language. These costs can be avoided at least in part by good language design (C++'s pre-processor).

CLOSING NOTES

We have covered Budd, Chapters 1 and 3-11.

All is fair game—text and class room material.

We won't worry much about 10.4 or 10.5.3 until after the exam.

An Opening Problem

Fill in the blank:

```
public class Tree
{
   private int  value;
   private Tree leftTree;
   private Tree rightTree;

   public Tree( int nodeValue, Tree leftChild, Tree rightChild )
   {
      value     = nodeValue;
      leftTree  = leftChild;
      rightTree = rightChild;
   }

   public int height()
   {
      // THE BLANK
   }
}
```

A Possible Solution

In Data Structures, we learned that recursive algorithms are the natural solution for many non-linear structures. So:

```
public int height()
{
   return 1 + Math.max( leftTree.height(),
                        rightTree.height() );
}
```

But there is the small matter of the base case. How does the recursion terminate?

We could store a `null` pointer at the leaves of the trees, which would allow us to create `Tree`s in the following way?

```
public static void main( String args[] )
{
   Tree root;

   root = new Tree(
            5,
            new Tree( 4,
                     new Tree( 3,
                              null,
                              new Tree( 2, null, null ) ),
                     new Tree( 1, null, null ) ),
            new Tree( 8,
                     new Tree( 5, null, null ),
                     new Tree( 7,
                              null,
                              new Tree( 6, null, null ))));

   System.out.println( root.height() );
}
```

A Possible Solution, Continued

Now, to handle the base cases, we can test for `null` children and return an answer:

```
public int height()
{
   if ( (leftTree == null) && (rightTree == null) )
      return 0;
   else if ( leftTree == null )
      return 1 + rightTree.height();
   else if ( rightTree == null )
      return 1 + leftTree.height();
   else
      return 1 + Math.max( leftTree.height(),
                           rightTree.height() );
}
```

And we have a working solution.

This is a form of **recursion** for an object-oriented program. When a `Tree` receives a `height()` message, it responds by doing a computation that involves sending a `height()` message to its instance variables, `leftTree` and `rightTree`.

So, the same message is sent, but to different objects. In an OO program, recursion is often a just a special case of **delegation**, in which the object delegates (most of) the responsibility for responding to a message to its instance variables. Object recursion delegates using the same message.

Improving Our Solution

We have a working solution, but...

(Don't you hate it when I say that?)

Won't I have to write a big selection statement like the one above for lots and lots of `Tree` methods? They are all recursive.

How can I say "it" once and only once?

Whenever you see a selection statement, ask yourself, "Am I combining the responsibilities of two objects here?" Remember: what distinguishes objects is how they behave.

In our `Tree` problem, we really have two kinds of tree: empty trees and non-empty trees. They behave differently in response to a `height()` message. An empty tree knows that it does not add any height to a branch, and a non-empty tree knows that its height is 1 more than the height of its longer child.

But in our solution, we used a `null` pointer—not an object that can respond to message—at the leaves of a `Tree`.

```
root = new Tree( 5,
                 new Tree( 4,
                           new Tree( 3,
                                     null,
                                     new Tree( ... ) ),
                 ... );
```

Since `null` is a "dead" data value, our `Tree`s must test for and handle `null`s themselves.

Why not try to take advantage of inheritance to help us solve this problem?

A New and Improved Solution

```
public interface Tree
{
   public int height();
}

public class EmptyTree implements Tree
{
   public int height()
   {
      return -1;
   }
}

public class NonEmptyTree implements Tree
{
   private int  value;
   private Tree leftTree;
   private Tree rightTree;

   public NonEmptyTree( int nodeValue,
                        Tree leftChild, Tree rightChild )
   {
      value     = nodeValue;
      leftTree  = leftChild;
      rightTree = rightChild;
   }

   public int height()
   {
      return 1 + Math.max( leftTree.height(),
                           rightTree.height() );
   }
}
```

```
public class TreeTester
{
   public static void main( String args[] )
   {
      Tree      root;
      EmptyTree aLeaf = new EmptyTree();

      root = new NonEmptyTree(
             5,
             new NonEmptyTree(
                 4,
                 new NonEmptyTree(
                     3,
                     aLeaf,
                     new NonEmptyTree( 2, aLeaf, aLeaf ) ),
                 new NonEmptyTree( 1, aLeaf, aLeaf ) ),
             new NonEmptyTree(
                 8,
                 new NonEmptyTree( 5, aLeaf, aLeaf ),
                 new NonEmptyTree(
                        7,
                        aLeaf,
                        new NonEmptyTree( 6, aLeaf, aLeaf ) ) ) );

      System.out.println( root.height() );
   }
}
```

A Short Exercise

Implement an `equals()` method for our pared-down `Ball` class.

```
public class Ball
{
   private Rectangle location;
   private Color     color;

   ...

   public void paint (Graphics g)
   {
      g.setColor (color);
      g.fillOval (location.x, location.y,
                  location.width, location.height);
   }
}
```

Possible Solutions

```
public class Ball
{
   private Rectangle location;
   private Color     color;

   public boolean equals( Ball anotherBall )
   {
      return ( location == anotherBall.region()   ) &&
             ( color    == anotherBall.getColor() );
   }

   public Color getColor() { return color; }

   ...
}

public class Ball
{
   private Rectangle location;
   private Color     color;

   public boolean equals( Object anotherObject )
   {
      if ( !( anotherObject instanceof Ball ) )
         return false;

      Ball target = (Ball) anotherObject;

      return ( location.equals( target.region()   ) ) &&
             (    color.equals( target.getColor() ) );
   }

   public Color getColor() { return color; }

   ...
}
```
Show a demo with unequal, equal, and identical?

An Important Implication of Inheritance

In order to allow for substitutability, Java must allow **polymorphic variables**: variables defined to be of one type (a class) but to which we can assign values of another type (a subclass).

But: polymorphic variables make it impossible for the compiler to determine the size of an object. Only the Java virtual machine can determine that, at run-time after an actual object has been created.

But: compilers want to know the sizes of objects for managing procedure calls on the run-time stack.

Implication: Use pointers to objects as the basic “values” for variables. All pointers are the same size, which allows the compiler to make its decisions early, and they can point to any object, which allows the programmer to use substitutable objects whenever she wants.

The ultimate implication of inheritance, then, is the advantage of using **reference semantics** in Java, rather than value semantics and explicit pointers.

The irony: Java programmers can’t and don’t use pointers.
All variables really hold pointers.

Implications of Reference Semantics

What does

x = y;

mean?

x and y are references to objects, so it could mean:

Let x refer to what y refers to.

But the objects that x and y refer to have values, so it could mean:

Let the object x refers to
have the same value as
the object y refers to.

The former is the assignment of *references*.
The latter is the assignment of *values*.
Both are useful in different contexts.

Java has to commit to a single meaning, though.
It performs assignment of references.

*** *Note the impact that this has on parameter passing.* ***

But the language should also support assignment of values.
How does Java do it?

Assignment and Cloning

The `Cloneable` interface...

The `clone()` method...

Shallow and deep cloning...

Who is responsible for the decision?

Done as Demo

Exercise 1

Modify the `Shape` class (page 176) to implement the `Cloneable` interface.

```
public class Shape
{
   protected int x;
   protected int y;

   public Shape (int ix, int iy)
   {
      x = ix;
      y = iy;
   }

   public String describe ()
   {
      return "parent shape " + x + " " + y;
   }

   public int getX ()
   {
      return x;
   }

   public int getY ()
   {
      return y;
   }
}
```

Solution 1

```
public class Shape implements Cloneable
{
   // ... all existing data and methods, plus:

    public Object clone ()
   {
       Shape s = new Shape ( getX(), getY() );
       return s;
   }
}
```

Moved to Session 17

Exercise 2

Modify the `Box` class (page 180) so that

(1) a `Box` holds a value that is `Cloneable`,
(2) a `Box` is `Cloneable`, and
(3) when cloned, a `Box` makes a deep copy of itself.

```
public class Box
{
   private int value;

   public Box ()
   {
      value = 0;
   }

   public void setValue(int v)
   {
      value = v;
   }

   public int getValue ()
   {
      return value;
   }
}
```

Solution 2

```
public class Box implements Cloneable
{
   private Cloneable value;

   public Box ()
   {
      value = null;
   }

   public void setValue(Cloneable v)
   {
      value = v;
   }

   public Cloneable getValue ()
   {
      return value;
   }

    public Object clone ()
    {
       Box result = new Box();
        result.setValue( getValue().clone() );
       return result;
    }
}
```

A sample use:

```
         Shape inner = new Shape( 10, 20 );
         Box outer = new Box();
         outer.setValue( inner );
         Box secondBox = (Box) outer.clone();

         Box newOuter = new Box();
         newOuter.setValue( outer );
         Box thirdBox = (Box) newOuter.clone();
```

.

The Object Recursion Pattern

The Problem

We would like to add a behavior to an object that requires solving essentially the same problem for its instance variables.

A Tempting Solution that Fails

Treat the instance variables as atomic units.

This solution fails when an instance variable is not atomic!

For example, in the case of assignment or cloning (§11.3), we end up with two objects having instance variables that point to the same object.

For example, in the case of equality testing (§11.4), our code will return false because two instance variables refer to distinct objects, even if their values are the same.

Why Does This Problem Matter?

Java uses a reference semantics. With reference semantics, questions of identity and equality can only be answered by looking at the parts of an object, too. Many other problems require looking at “the parts of the parts”.

The Solution

Use polymorphism to implement a recursive method.

Each composed object will send the same message to its parts and then assemble its answer from the answers of its parts.

Each atomic object will simply construct and return an answer without sending the recursive message.

.