Composition

Computer applications often involve complex structured data.

One very important data structuring mechanism is composition — one piece of data is composed of sub-parts, it has a second piece of data as a component. E.g. a Stock has a symbol, an Investment has a Stock.

The more interesting cases are when an object is composed of other objects.

In math, corresponding structuring mechanism is Cartesian product.

In Java, objects are the main composition mechanism.

E.g. Using Investment

```java
import type.lang.*;
import type.lib.*;
public class InvestmentTest
{
    public static void main(String[] args)
    {
        Stock stk1 = new Stock("NT");
        IO.println("stk1= " + stk1.toString());
        Investment inv1 = new Investment(stk1, 100,
                                           stk1.getPrice());
        IO.println("inv1= " + inv1.toString());
        IO.println("inv1.getQty()= " + inv1.getQty());
        IO.println("inv1.getBookValue()= " +
                    inv1.getBookValue());
        IO.println("inv1.getStock().toString()= " +
                    inv1.getStock().toString());
        IO.println("inv1.getStock().getSymbol()= " +
                    inv1.getStock().getSymbol());
        Stock stk2 = new Stock("BMO");
        IO.println("stk2= " + stk2.toString());
        Investment inv2 = new Investment(stk2, 100,
                                           stk2.getPrice());
        IO.println("inv2= " + inv2.toString());
        IO.println("inv1.equals(inv2)= " +
                    inv1.equals(inv2));
        Stock stk3 = inv1.getStock();
        IO.println("stk3= " + stk3.toString());
        stk3.setSymbol("RY");
        IO.println("reset stk3’s symbol to RY");
        IO.println("stk3= " + stk3.toString());
        IO.println("inv1.getStock()= " +
                    inv1.getStock().toString());
    }
}
```

Can use UML class diagrams to represent relationships between classes including composition. E.g. Investment, VehicleTransfer.

Other terms used for the composition/has-a relationship include aggregation, part-of, association. Can make a finer analysis.
As we see, Investment constructor uses Stock object passed as component of the Investment and getStock accessor method returns a reference to this component. This allows a user to change the state of the Investment object with strange results.

Investment could protect against this by setting its component to a copy of the passed Stock in the constructor and returning a reference to a copy of its component in getStock. Then, only methods belonging to Investment could change the Stock component.

In general, to ensure that users cannot change components of an object without using the object’s methods, one must make a deep copy of components passed to/from the object, i.e. a copy where subcomponents and subsubcomponents are also copied.

Collections

In many case, an object has a whole collection of components, e.g. a Portfolio has a collection of Investments, a Course has a collection of Students.

In math, the corresponding structuring mechanism is sets.

In Java, there are several mechanisms to deal with collections, in particular, arrays and the Vector class. More about these later. Some classes, e.g. Portfolio, hide/encapsulate the mechanism used.

Can use UML class diagrams to represent having a collection of components.

Iteration over Collections

Often you need to do some operations on each element of a collection. This is called iterating over the collection.

Objects that have collections of components provide mechanisms for this.

E.g. the Portfolio class provides two groups of methods for iterating over the Investments in the portfolio:

getInvestment(i) together with getCount() and getNext() together with getFirst().

g getNext() can be viewed as an iterator that is initialized by getFirst().
A Portfolio Can be Modified in Strange Ways

```java
import type.lang.*;
import type.lib.*;
public class PortfolioTest2
{
    public static void main(String[] args)
    {
        Portfolio ptf1 = new Portfolio("My e.g. Portfolio", 10);
        IO.println(ptf1.toString());
        Stock stk1 = new Stock("NT");
        ptf1.add(new Investment(stk1, 100, stk1.getPrice()));
        stk1 = new Stock("BMO");
        ptf1.add(new Investment(stk1, 50, stk1.getPrice()));
        stk1 = new Stock("RY");
        ptf1.add(new Investment(stk1, 100, stk1.getPrice()));
        IO.println(ptf1.toString());
        for(Investment inv = ptf1.getFirst(); inv != null; inv = ptf1.getNext())
        { IO.println(inv); }
        IO.println();
        Investment inv1 = ptf1.getFirst();
        stk1 = inv1.getStock();
        stk1.setSymbol("RY");
        IO.println("Changed 1st investment to RY");
        IO.println(ptf1.toString());
        for(Investment inv = ptf1.getFirst(); inv != null; inv = ptf1.getNext())
        { IO.println(inv); }
    }
}
```
import type.lang.*;
import type.lib.*;
public class PortfolioDeepCopy
{
    public static void main(String[] args)
    {
        Portfolio ptf1 = new Portfolio("My e.g. Portfolio",10);
        Stock stk1 = new Stock("NT");
        ptf1.add(new Investment(stk1,100,stk1.getPrice()));
        stk1 = new Stock("BMO");
        ptf1.add(new Investment(stk1,50,stk1.getPrice()));
        stk1 = new Stock("RY");
        ptf1.add(new Investment(stk1,100,stk1.getPrice()));
        IO.println("\nPortfolio ptf1 is:");
        IO.println(ptf1.toString());
        for(Investment inv = ptf1.getFirst(); inv != null; inv = ptf1.getNext())
        {
            IO.println(inv);
        }
        // Make deep copy
        Portfolio ptf1c = new Portfolio(ptf1.getName(),10);
        for(Investment inv = ptf1.getFirst(); inv != null; inv = ptf1c.getNext())
        {
            Stock stkc = new Stock(inv.getStock().getSymbol());
            Investment invc = new Investment(stkc,
                                              inv.getQty(),inv.getBookValue());
            ptf1c.add(invc);
        }
        IO.println("\nMade deep copy ptf1c of ptf1:");
        IO.println(ptf1c.toString());
        for(Investment inv = ptf1c.getFirst(); inv != null; inv = ptf1c.getNext())
        {
            IO.println(inv);
        }

        // Change 1st investment in ptf1 to RY
        Investment inv1 = ptf1c.getFirst();
        stk1 = inv1.getStock();
        stk1.setSymbol("RY");
        IO.println("\nChanged 1st investment in ptf1 to RY");
        IO.println("\nptf1 is now:");
        IO.println(ptf1c.toString());
        for(Investment inv = ptf1c.getFirst(); inv != null; inv = ptf1c.getNext())
        {
            IO.println(inv);
        }
    }
}

The **Vector** Class

The Vector class provides a generic mechanism for maintaining a collection of objects, e.g. a set of favorite Stocks.

Elements are ordered starting from index 0. A Vector can grow and shrink as needed. New elements can be inserted in arbitrary positions and the remaining elements will shift to higher indices.

See the Vector class’s API for available methods.

As we can see in the following e.g., a variety of mechanisms are available for iterating over the elements of a Vector.

Note also that the elementAt(i) method's return type is Object. If you want to assign the returned element to a variable of a more specific type, e.g. Stock, you must cast it. Only then can you use methods belonging to the more specific class on the object.
```java
import type.lang.*;
import type.lib.*;
import java.util.*;

public class VectorTest
{
    public static void main(String[] args)
    {
        Vector hotPicks = new Vector();
        IO.println(hotPicks.toString());
        Stock stk1 = new Stock("NT");
        hotPicks.add(stk1);
        stk1 = new Stock("BMO");
        hotPicks.add(stk1);
        stk1 = new Stock("RY");
        hotPicks.add(stk1);
        IO.println(hotPicks.toString());
        int hotPicksSize = hotPicks.size();
        for(int i = 0; i < hotPicksSize; i++)
        {
            Stock stk2 = (Stock) hotPicks.elementAt(i);
            stk2.refresh();
            IO.println(stk2.toString() + " " + stk2.getPrice());
        }
        IO.println();
        hotPicks.insertElementAt(new Stock("ALI"),1);
        Iterator hpIter = hotPicks.iterator();
        while(hpIter.hasNext())
        {
            Stock stk3 = (Stock) hpIter.next();
            stk3.refresh();
            IO.println(stk3.toString() + " " + stk3.getPrice());
        }
        hotPicks.remove(0);
        hotPicks.remove(new Stock("BMO"));
        Enumeration hpEnum = hotPicks.elements();
        while(hpEnum.hasMoreElements())
        {
            Stock stk4 = (Stock) hpEnum.nextElement();
            stk4.refresh();
            IO.println(stk4.toString() + " " + stk4.getPrice());
        }
    }
}
```