Similar, yet Different, Concepts

- Inner classes
- Inheritance

- Both are based on hierarchical ("tree-like") structures
Inner Classes

- In Java (and C++), classes can be nested within one another.
- Objects in the inner class has available instance variables and methods of the outer class.
Ways to Construct ClosedList

- `class Cell {...}
class ClosedList {...}

- `class ClosedList {
  class Cell {...}
  ...
  }

Inner Class
Interpretation of Identifiers

- In an inner class, the innermost meaning of an identifier applies.

```java
class ClosedList {
    String identity;
    class Cell {
        String identity;
        ...
    }
    ...
}
```

Inner identity is used.

Outer identity is used.
Usage

- Normally one or more objects of the inner class are created for a given object of the outer class.

- Objects of the inner class only make sense in the context of a supporting object of the outer class.
Exporting Inner Objects

- Inner objects can be used outside, understanding that they are always relative to the object in which they are contained.
Example: List Iterator

- We want to define an Iterator for a ClosedList.
- For read-only Iteration, the Iterator class can be defined outside the ClosedList class.
- For modification, such as `remove()`, it is sometimes necessary for the Iterator to change part of the list itself.
Example: List Iterator (2)

- By making the ListIterator an inner class, we can:
  - Use data elements defined in the ClosedList.
  - Avoid exposing those data elements to the world at large.
  - Use Iterators outside ClosedList.
ClosedList.Iterator

class ClosedList
{
    private Cell head;
    private Cell tail;
    ...
    public Iterator getIterator()
    {
        return new Iterator(head);
    }
}

public class Iterator // inner class to ClosedList
{
    private Cell current;
    private Cell previous; // keep track of previous

    public Iterator(Cell head)
    {
        current = head;
        previous = null;
    }
    ...
    Can export Iterator to outside!
ClosedList.Iterator: remove()
Defined to remove the value just produced by next().

```java
public void remove()
{
    if (previous == null)
    {
        head = head.getNext();
    } else
    {
        previous.setData(current.getData()); // reuse
        previous.setNext(current.getNext()); // previous
        current = previous; // lose current
    }
}
```

head is defined in outer class!
Aside: Converting a list to a String
(e.g. for printing entire list at once)

```java
public String toString()
{
    StringBuffer buffer = new StringBuffer();
    Iterator it = new Iterator(head);
    if( it.hasNext() )
    {
        buffer.append(it.next()); // first element
    }

    while( it.hasNext() )
    {
        buffer.append(" "+it.next()); // leave space
    }

    return buffer.toString();
}
```
class ClosedListTest
{
    public static void main(String arg[])
    {
        int numItems = 10;
        ClosedList L = new ClosedList();

        for( int i = 0; i < numItems; i++ )
        {
            L.enqueue(new Integer(i));
        }

        ClosedList.Iterator it = L.getIterator();
        System.out.println("removing " + it.next());
        it.remove(); // remove first item
    }
}
class ClosedListTest
{
    public static void main(String arg[])
    {
        int numItems = 10;
        ClosedList L = new ClosedList();

        // add 10 items to L
        for( int i = 0; i < numItems; i++ )
        {
            L.enqueue(new Integer(i));
        }

        System.out.println("Initial list contents: " + L);

        // starting from the beginning, skip 3 items
        ClosedList.Iterator it = L.getIterator();
        for( int i = 0; i < 3; i++ )
        {
            System.out.println("skipping " + it.next());
        }

        System.out.println("removing " + it.next());
        it.remove();
        System.out.println("removing " + it.next());
        it.remove();
        System.out.println("List contents after removing two: " + L);

        for( int i = 0; i < 3; i++ )
        {
            System.out.println("skipping " + it.next());    // ignore value
        }

        // insert 3 items
        for( int i = 0; i < 3; i++ )
        {
            int value = 10*(i+1);
            System.out.println("inserting " + value);
            it.insert(new Integer(value));
        }

        System.out.println("List contents after inserting three: " + L);
    }
}
Test Program Output

Initial list contents: 0 1 2 3 4 5 6 7 8 9
skipping 0
skipping 1
skipping 2
removing 3
removing 4
List contents after removing two: 0 1 2 5 6 7 8 9
skipping 5
skipping 6
skipping 7
inserting 10
inserting 20
inserting 30
List contents after inserting three: 0 1 2 5 6 7 10 20 30 8 9
Inheritance

- “Inheritance” is a way of building one class on top of another.
  - The original class is called the **base class**, or **parent class**.
  - The new class is called the **derived class**, or **child class**.
Diagrammatic Notation (UML)

class Professor extends class Employee

UML = Unified Modeling Language
Inherited Capabilities

- **Extension:**
  The derived class can potentially use all data components and methods from the base class, and add more of its own.

- **Over-Riding:**
  It can also selectively re-define or “over-ride” methods of the same name.
Purposes of Inheritance

- Use the same concepts and code for many classes (base-class concepts and code shared by derived classes):
  - Work economy
  - Intellectual economy
- Tie together similar classes:
  - Increases the utility of methods that use such classes.
Extension = Java Inheritance

- In Java, the keyword for “inherits from” is **extends**

- The derived class **extends** the base class.

- Extension allows over-riding as well; there is no separate keyword for over-riding.
Extension Example

- **class** Account defines a basic bank account

- **class** CheckingAccount defines a special account for check-writing
class Account
{
    Money balance;

    Account(Money initialBalance)
    {
        balance = initialBalance;
    }

    void deposit(Money amount)
    {
        balance = balance.add(amount);
    }

    boolean withdraw(Money amount)
    {
        if (balance.lessThan(amount))
            return false;
        balance = balance.subtract(amount);
        return true;
    }

    void showBalance(PrintStream out)
    {
        out.println("Balance: " + balance);
    }

    Only allow withdrawal if sufficient funds; return boolean to indicate success or failure.
class CheckingAccount extends Account
{
Money serviceCharge;
}

CheckingAccount(Money initialBalance, Money serviceCharge)
{
  super(initialBalance);
  this.serviceCharge = serviceCharge;
}

boolean cashCheck(Money amount)
{
  return withdraw(amount.add(serviceCharge));
}

(continued next page)
(program continued)

public static void main(String arg[])
{
    CheckingAccount myCheckingAccount =
            new CheckingAccount(new Money(10000),
            new Money(100));

    myCheckingAccount.showBalance(System.out);

    myCheckingAccount.deposit(new Money(5000));
    myCheckingAccount.showBalance(System.out);

    myCheckingAccount.cashCheck(new Money(2000));
    myCheckingAccount.showBalance(System.out);

    myCheckingAccount.cashCheck(new Money(1000));
    myCheckingAccount.showBalance(System.out);

    myCheckingAccount.withdraw(new Money(1000));
    myCheckingAccount.showBalance(System.out);
}

(Link to the complete program: Bank)
Multiple derived classes

Account

CheckingAccount
- Service charge added

SavingsAccount
- No service charge
- Interest accruing
Which methods can be over-ridden?

- In order to be over-ridden, a method must be declared either:
  - public
  - protected

  in the base class.
Inheritance Examples from Java Libraries

- class java.lang.Object
- class java.util.AbstractCollection (implements java.util.Collection)
- class java.util.AbstractList (implements java.util.List)
- class java.util.AbstractSequentialList
- class java.util.LinkedList (implements java.util.List)
- class java.util.ArrayList (implements java.util.List)
- class java.util.Vector (implements java.util.List)
- class java.util.Stack
- class java.util.AbstractList (extends java.util.AbstractSequentialList)
- class java.util.LinkedList (implements java.util.List)
- class java.util.ArrayList (implements java.util.List)
- class java.util.Vector (implements java.util.List)
Implications of Inheritance

- The preceding diagram means, for example, that to find all methods for class `java.util.Stack`, you may wish to look at:
  - `java.util.Vector`
  - `java.util.AbstractList`
  - `java.util.AbstractCollection`
  - `java.lang.Object`
Methods of java.util.Stack

- Methods of Stack proper:
  - boolean empty()
  - Object peek()
  - Object pop()
  - Object push(Object item)
  - int search(Object o)

- Methods of AbstractList:
  - iterator, listIterator

- Methods of Vector:
  - add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toString, trimToSize

- Methods of Object (not otherwise over-ridden):
  - finalize, getClass, notify, notifyAll, wait
Testing where Object is in Hierarchy

- `instanceof` operator

```java
Object ob = ...;
if( ob instanceof Vector ) ...  
if( ob instanceof Stack ) ...
```
Casting

Object

up-casting (always safe)

down-casting (use `instanceof` to check before)

Vector

Stack

If you down-cast and you are wrong, you will get a `ClassCastException`, which can terminate your program.
class Object

- Object is the ancestor of all classes
- Some methods of Object:
  - boolean equals(Object)
  - Class getClass(): returns a Class object
  - String toString()

- Method of class Class:
  - String getName()
Implementing an Interface is similar to Inheritance

- Interface $\approx$ Base Class
- Implementor $\approx$ Derived Class

- By declaring methods to use the Interface rather than the Implementor class as an argument, more *generality* is afforded to that method.
Example

- `java.util.Iterator` is standard
- Make `ClosedList.Iterator` implement `java.util.Iterator`
- Any other code accepting a `java.util.Iterator` can now use our: `ClosedList.Iterator`
- We can still do everything we did before.
Morals

- When possible, write your methods to use **Interfaces** rather than classes.

- Define Interfaces for your classes, on which others can depend.

- Ideally, define the Interfaces first.