**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 {...}
  ...
}`

**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 ListIterator getIterator()
    {
        return new ListIterator(head);
    }
    public class ListIterator // inner class to ClosedList
    {
        private Cell current;
        private Cell previous; // keep track of previous
        public ListIterator(Cell head)
        {
            current = head;
            previous = null;
        }
        ...
    }
}

ClosedList.Iterator: remove()

Defined to remove the value just produced by next().

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

Aside: Converting a list to a String (e.g. for printing entire list at once)

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();
}

Test Program

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 ClosedList
{
    public data field: int[] seq()
    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
    ClosedListIterator it = L.iterator();
    for( int i = 0; i < 3; i++ )
    {
        System.out.println("skipping " + it.next());
    }
    // remove 2 items
    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());
    }
    // 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

<table>
<thead>
<tr>
<th>Test Program Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial list contents: 0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>skipping 0</td>
</tr>
<tr>
<td>skipping 1</td>
</tr>
<tr>
<td>skipping 2</td>
</tr>
<tr>
<td>removing 3</td>
</tr>
<tr>
<td>removing 4</td>
</tr>
<tr>
<td>List contents after removing two: 0 1 2 5 6 7 8 9</td>
</tr>
<tr>
<td>skipping 5</td>
</tr>
<tr>
<td>skipping 7</td>
</tr>
<tr>
<td>inserting 10</td>
</tr>
<tr>
<td>inserting 20</td>
</tr>
<tr>
<td>inserting 30</td>
</tr>
<tr>
<td>List contents after inserting three: 0 1 2 5 6 7 10 20 30 8 9</td>
</tr>
</tbody>
</table>

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)

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.

UML = Unified Modeling Language
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.

```java
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);
    }
}

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)
```

Additional variable for the derived class.

```java
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);
}
```

Program Output

```
Balance: $100.0
Balance: $150.0
Balance: $129.0
Balance: $118.0
Balance: $108.0
```

Multiple derived classes

Program continued: Bank

Link to the complete program: Bank

Additional service charge

Service charge added

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

```java
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
```

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 `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`, `subList`, `toArray`, `toString`, `trimToSize`

- Methods of `AbstractList`:
  - `iterator`, `listIterator`

- 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)
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 ≈ Base Class
- Implementor ≈ 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.