**Inner Classes**

- In Java (and C++), a class can be **nested within** another class.
- Each object in the inner class exists relative to an object of the outer class.
- Objects of the inner class have 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;
  }
  ...
}
```

**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 were created.
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 variables in the container, such as the head or tail.

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

```java
class ClosedList_iterator {  
  private Cell head;  
  private Cell tail;  
  private Cell current;  
  private Cell previous;  // keep track of previous  
  public ClosedList.Iterator(Cell head) {  
    current = head;  
    previous = null;  
  }  
  public void remove() {  
    if (previous == null) {  
      head = head.getNext();  
    } else {  
      previous.setData(current.getData());  // reuse previous  
      previous.setNext(current.getNext());  // previous current = previous;  // lose current  
    }  
  }  
}
```

Test Program

```java
class TestClosedList {  
  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  
  }  
}
```
**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)**

![Diagram showing class Professor extends class Employee]

**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.  
  (It is a good idea to keep the same approximate meaning.)

**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

```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);
    }
}
```

```java
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));
    }
}
```

**Program continued**

```java```
public static void main(String args[]) {
    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**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance: $100.0</td>
<td>Balance: $150.0</td>
</tr>
<tr>
<td>Balance: $129.0</td>
<td>Balance: $118.0</td>
</tr>
<tr>
<td>Balance: $108.0</td>
<td>Balance: $108.0</td>
</tr>
</tbody>
</table>

(Click to the complete program: Bank)

### Multiple derived classes

- **Account**
- **CheckingAccount**
- **SavingsAccount**

- **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 abound in Java Libraries

class java.lang.Object
class java.util.AbstractCollection (implements java.util.Collection)
extends class java.util.LinkedList (implements java.util.List)
class java.util.ArrayList (implements java.util.List)
class java.util.Stack

class java.util.AbstractList (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:
  - Object push(Object item)
  - Object pop()
  - boolean empty()
  - Object peek()
  - int search(Object o)

- Methods of AbstractList:
  - iterator, listIterator

- Methods of Object (not otherwise over-ridden):
  - finalize, getClass, notify, notifyAll, wait

Methods of AbstractList:

- Methods of AbstractList:
  - remove, listIterator

Methods of Vector:

- add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, tête, remove, removeAll, removeAll, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toString, trimToSize

Testing where Object is in Hierarchy

- instanceof operator

Object ob = ...;
if (ob instanceof Vector) ... if (ob instanceof Stack) ...

More than one can be true!

Casting

Object

Stack

Vector

up-casting (always safe)
down-casting (use instanceof to check before)

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 the "most derived" Class object
  - String toString()

- Method of class Class:
  - String getName()
  - So getClass().getName() will get you the class name of the object.
### 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.

### Abstract Classes

- A class is **abstract** if it is not intended to be instantiated directly; rather, objects in derived classes are instantiated.
- Each derived-class object implicitly entails an underlying base-class object.
- In Java, an abstract class is so-declared: `abstract class MyClass { . . }`

### Abstract Classes in Java

- An abstract class is so-declared: `abstract class MyClass { . . }`
- A class declared abstract cannot be instantiated directly.

### Abstract Methods

- A method is **abstract** if there is no code for it in the abstract class; instead the meaning of the method is obtained from over-riding in derived classes.
- Only abstract classes can contain abstract methods.

### Abstract Class Example: Tree

- Consider the following type of Tree:
  - A Tree can be an Atom
  - A Tree can be a Composite: a pair of Sub-Trees (each of which is a tree in its own right).
- Type Tree is abstract:
  - We never create a tree directly.
  - We only create an Atom or a Composite.
Tree in Java

abstract class Tree
{
  object value;
  Leaf(Object value)
  { this.value = value; }
}

class Leaf extends Tree
{
  object value;
  Leaf(Object value)
  { this.value = value; }
}

class Composite extends Tree
{
  Tree left;
  Tree right;
  Composite(Tree left, Tree right)
  { this.left = left; this.right = right; }
}

Adding a Method to Tree

- Add method
  
  int leafCount();

to Tree

leafCount in the base class

abstract class Tree
{
  object value;
  public abstract int leafCount();
}

class Leaf extends Tree
{
  object value;
  Leaf(Object value)
  { this.value = value; }
  int leafCount()
  { return 1; }
}

class Composite extends Tree
{
  Tree left;
  Tree right;
  Composite(Tree left, Tree right)
  { this.left = left; this.right = right; }
  int leafCount()
  { return left.leafCount() + right.leafCount(); }
}

leafCount in the Derived Classes

Abstract Class vs. Interface

- An abstract class can still contain data and methods; an interface cannot.
- It is common for an abstract class to define methods with the intention that they be overridden differently by each derived class.

Multiple Inheritance

- Some languages allow one class to derive from multiple base classes; Java does not.
- The nearest thing would be a class deriving from a single basic class and implementing an interface at the same time.