Nested Classes

Motivation Example

```
public class List {

  private Node _head = new Node(null, null);
  private Node _tail = _head;
  private int _size;

  public void add(Object obj) {
    _tail.setNext(new Node(obj, null));
     _size++;
    _tail = _tail.getNext();
  }

  public Object remove(Object obj) { … }

  public boolean contains(Object obj) {
    Node n = _head.getNext();

    while (n != null) {
      if (n.getValue().equals(obj))
        return true;
    }

    return false;
  }
}
```

```
public class Node {
  private Node _next;
  private Object _value;

  public Node(Object val, Node n) {
    _next = n;
    _value = val;
  }

  public Object getValue() { return _value; }

  public void setNext(Node n) { _next = n; }

  public Node getNext() { return _next; }
}
```

- Node only makes sense in the context of List
- Node is not fully encapsulated due to List
- What do we need to add to *List* to be able to built an iterator?
  - Node getHead()

Nested Class

- Nested class: a class defined within another class

```
[modifier] class OuterClass {
    code
    [modifiers] class InnerClass [extends BaseClassToInner] [implements SomeInterface[, MoreInterfaces, ...]] {
        fields and methods
    }
}
```

- The outer class can refer to any member of the nested class
- The nested class can have any access level:
  - public, private, protected or package-private
- Reference an object of the nested class outside the outer class must be prefixed with the name of the outer class and ‘.’
  - OuterClass.NestedClass nestedClassReference;

Why use nested classes?

- **Increases encapsulation**
  - Develop an iterator for *List* without exposing implementation details in the interface of *List*
- **Group together classes that are only used in one place**
- **Can lead to more readable and maintainable code**

Two types of nested classes

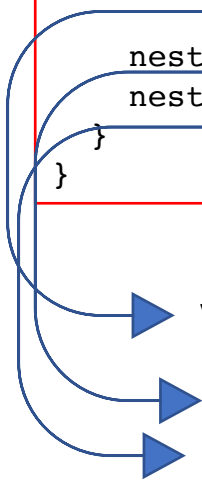
- Static nested classes
  - Similar to a static member of a class
  - Can refer to any static member of the outer class (even private)
  - Can use non-static members of the outer class (private or not) only through an object reference of the outer class

- Non-static nested classes (also called inner classes)
  - Similar to a non-static member of a class
  - Each instance of a nested class is associated with an instance of the outer class
  - It can access any member of the outer class:
    - Static or non-static, private or not

Static Nested Class Example

```
public class OuterClass {
  private int _x;
  private static int _y;

  private int getX() { return _x; }

  private static int getY() { return _y; }

  static class NestedClass {
    public void incY() {
      int y = getY();
      _y = y + 1;
    }

    public void incX(OuterClass oc) {
      int x = oc.getX();
      oc._x = x + 1;
    }

  }
}
```

```
public class Test {
  public static void main(String args[]) {
    OuterClass.NestedClass nested =
                              new OuterClass.NestedClass();

    nested.incY();
    nested.incX(new OuterClass());
  }
}
```



Value of _y of OuterClass? 0

Value of _y of OuterClass? 1!

Value of _x of new OuterClass() ? 1!

Static Nested Class – List implementation 1

```
public class List {

 private static class Node {
  private Node _next;
  private Object _value;

  private Node(Object val, Node n) {
   _next = n;
   _value = val;
  }

  private  Object getValue() { return _value; }
  private void setNext(Node n) { _next = n; }
  private Node getNext() { return _next; }
 }

 private Node _head = new Node(null, null);
 private Node _tail = _head;
 private int _size;
```

```
  public void add(Object obj) {
   _tail.setNext(new Node(obj, null));
    _size++;
   _tail = _tail.getNext();
  }

  public Object remove(Object obj) { … }

  public boolean contains(Object obj) {
   Node n = _head.getNext();

   while (n != null) {
    if (n.getValue().equals(obj))
     return true;
   }

   return false;
  }
} // end of List class
```

Static Nested Class – List implementation 2

```
public class List {

 private static class Node {
  private Node _next;
  private Object _value;

  private Node(Object val, Node n) {
   _next = n;
   _value = val;
  }
 }

 private Node _head = new Node(null, null);
 private Node _tail = _head;
 private int _size;
```

```
  public void add(Object obj) {
   _tail._next = new Node(obj, null);
    _size++;
   _tail = _tail._next;
  }

  public Object remove(Object obj) { … }

  public boolean contains(Object obj) {
   Node n = _head._next;

   while (n != null) {
    if (n._value.equals(obj))
     return true;
   }

   return false;
  }
} // end of List class
```

Nested classes

- And what about the initial getHead() method?
- Have we solved this problem?

Inner Classes

- An instance of the inner class have to be created in the context of an instance of the outer class
  - Inside a non-static method of the outer class
    - The outer class instance is stored in *this*
  - In the rest of the code, the reference must be explicitly given:
    - OuterClass outerClassRef = …;
      OuterClass.InnerClass innerClassRef = outerClassRef.new InnerClass();
- A inner class object can know its outer class object inside any method of the inner class:
  - *OuterClass.this*
- Cannot specify static members
  - Unless constant variable declarations

Inner Class Example

```
public class OuterClass {
  private int _x;
  private static int _y;

  private int getX() { return _x; }

  private static int getY() { return _y; }

  public InnerClass create() { return new InnerClass(); }

  class InnerClass {
    public void incY() {
      int y = getY();
      _y = y + 1;
    }

    public void incX() {
      int x = getX();
      _x = x + 1;
    }
  }
}
```

```
public class Test {
  public static void main(String args[]) {
    OuterClass outerClassRef = new OuterClass();
    OuterClass.InnerClass nested = outerClassRef.create();
    OuterClass.InnerClass nested2 = outerClassRef.new InnerClass();

    nested.incY();
    nested.incX();
  }
}
```

Inner Class – List implementation 1

```
public class List {

 private static class Node {
  private Node _next;
  private Object _value;
  // equal
 }  // Node

 private class ListIterator implements java.util.Iterator<Object> {
  private Node _current = _head._next;

   public boolean hasNext() {
     return _current != null; }

   public Object next() {
    Object res = _current.getValue();
    _current = _current.getNext();
    return res;
   }
 }  // ListIterator

 private Node _head = new Node(null, null);
 private Node _tail = _head;
 private int _size;
```

```
  public void add(Object obj) {
   _tail.setNext(new Node(obj, null));
    _size++;
   _tail = _tail.getNext();
  }

  public Object remove(Object obj) { … }

  public boolean contains(Object obj) {
   Node n = _head.getNext();

   while (n != null) {
    if (n.getValue().equals(obj))
     return true;
   }

   return false;
  }

  public java.util.Iterator<Object> iterator() {
   return new ListIterator();
  }
} // end of List class
```

Inner Classes - Local and Anonymous Classes

- Two additional types of inner classes:

- Local classes
  - An inner class declared within the body of a method

- Anonymous classes
  - A local class without naming the class

Local Classes

- A local class can be defined inside any block
- A local class has access to any member of its enclosing class
  - + local variables of the enclosing block that are declared final
  - + final parameters of the enclosing block
  - + local variables and parameters that are *effectively final*

Example

```
public class Log File {
   private String _id;

   void localMethod(String msg) {
      long logID = 2999993984;
      //local class
      class LocalClass {
         public void display() {
            System.err.println('Inside the local class: ' + logID);
            System.err.ptintln(”id: ” + _id + “ message: “ + msg);
         }
      }
      //create new instance of local class
      LocalClass local = new LocalClass();
      local.display();
  }
}
```

Anonymous Classes

- Make your code more concise
- Declare and instantiate a class at the same time
- They are like local classes except that they do not have a name
- May be defined at any point where an object reference is needed
- Use an anonymous class if you need to use a local class only once

Syntax

new *InterfaceToImplement* () {
  attributes
  interface methods
  other methods
} [;]

class SomeName implements InterfaceToImplement {
  attributes
  interface methods
  other methods
}
new SomeName() [;]

new *ClassToExtend* ()
[implements someInterface] {
  attributes
  overriden methods
  other methods
} [;]

class SomeName extends ClassToExtend [implements someInterface] {
  attributes
  overriden methods
  other methods
}
new SomeName() [;]

new *ClassToExtend* (*parameter list matching base class constructor*) {
  attributes
  overriden methods
  other methods
} [;]

Class declaration similar to previous

new SomeName(*parameter list*) [;]

Example

```
class BaseClassForAnonymous {

  protected int _x;

  public BaseClassForAnonymous(int x) {
    _x = x;
  }

  public void print() {
    System.out.println("BaseClassForAnonymous x = " + _x);
  }
}
```

What is printed?

AnonymousDerivedClass.print
BaseClassForAnonymous x = 6

```
public class OuterClass {

  OuterClass (int v) {
    go(
      new BaseClassForAnonymous(v) {
        public void print() {
          System.out.println("AnonymousDerivedClass.print”);
          _x++;
          super.print();
        }
      }
    );
  }

  public void go(BaseClassForAnonymous bc) {
    bc.print();
  }

  public static void main(String[] args) {
    new OuterClass(5);
  }
}
```

Anonymous Class – List implementation

```
public class List {

 private static class Node {
  private Node _next;
  private Object _value;
  // equal
 }  // Node

 private Node _head = new Node(null, null);
 private Node _tail = _head;
 private int _size;

 public void add(Object obj) {
  _tail.setNext(new Node(obj, null));
   _size++;
  _tail = _tail.getNext();
 }

public Object remove(Object obj) { … }
```

```
 public boolean contains(Object obj) {
  Node n = _head.getNext();

  while (n != null) {
   if (n.getValue().equals(obj))
    return true;
  }
  return false;
 }

 public java.util.Iterator<Object> iterator() {
  return new java.util.Iterator<Object> () {
   private Node _current = _head._next;

    public boolean hasNext() { return _current != null; }

    public Object next() {
     Object res = _current.getValue();
     _current = _current.getNext();
     return res;
    }
  };
 }
} // end of List class
```