Lecture 7 Supplement: A Review of Nested Classes

Member Inner Classes

```
public class Member {
    private String name = "Joe";
    private Pair p = new Pair();
    {
        p.first = 4;
        p.second = 5;
        System.out.println(p);
    }
    private void printHello() {
        System.out.println("Hello " + name);
    }
    class Pair {
        int first;
        int second;
        Pair() {
            printHello();
        }
        public String toString() {
            return "(" + first + ", " + second+ ")";
        }
    }
    public static void main(String[] args) {
        new Member();
    }
}
```

- Accessible only when an instance of enclosing class exists, and when inner class has been instantiated
- Have access to private members of enclosing class
- May be declared private, public, etc
- Cannot contain static variables or methods
- Enclosing class can access private members of inner class, relative to an existing reference
- Best Practice: Should be accessed only by the enclosing class

Static Nested Classes

```
public class Static {
    private String name = "Joe";
    private Pair p = new Pair();
    {
        p.first = 4;
        p.second = 5;
        System.out.println(p);

    }
    private  void printHello() {
        System.out.println("Hello" + name);
    }
    static class Pair {
        int first;
        int second;
        Pair() {
            //no access
            //printHello();
        }
        public String toString() {
            return "(" + first + ", " + second+ ")";
        }
    }
    public static void main(String[] args) {
        (new Static()).printHello();
    }
}
```

- Considered a "top-level class", packaged inside another class
- May be instantiated even if enclosing class has not been instantiated
- No access to non-static members of enclosing class without a reference (with a reference, can access private members)
- Direct access to static members
- May be declared private, public, etc
- May contain static and non-static members
- Enclosing class can access private members of nested class, relative to an existing reference

Local Inner Classes

```
public class Local {
    private String name = "Joe";
    public void printPair(int x, int y) {
        class Pair {
            int first;
            int second;
            Pair() {
                printHello(name);
            }
            public String toString() {
                return "(" + first + ", "
                           + second+ ")";
            }
        }
        Pair p = new Pair();
        p.first = x;
        p.second = y;
        System.out.println(p);
    }
    private void printHello(String n) {
        System.out.println("Hello " + n);
    }
    public static void main(String[] args) {
        (new Local()).printPair(11, 3);
    }
}
```

- Always defined within a method body – never accessible from outside the method in which it is defined
- Has access to all members of enclosing class
- Has access to local variables, but may not modify them (they are "effectively final")
- Access specifiers (public, private…) may not be used in definition of a local inner class
- Local inner classes provide *strong encapsulation* – no other method in the enclosing class (or anywhere else) can access it.

Anonymous Inner Classes

```
public class Anonymous {
    interface IPair {
        public void printHello();
    };

    private String name = "Joe";
    public void printPair(int x, int y) {
        (new IPair() {
            int first = x;
            int second = y;

            public String toString() {

                return "(" + first + ", " +
                            second + ")";
            }
            public void printHello() {
                name = "Tom";
                System.out.println("Hello " +
                     name + "\n" + this);

            }
        }).printHello();
    }
    public static void main(String[] args) {
        (new Anonymous()).printPair(11, 3);
    }
}
```

- Defines and instantiates, at the same time, a class, without giving it a name
- The syntax can be used to define a subclass of a given class or an implementation of a given interface
- Main usage: when class definition involves few lines of code and the class needs to be defined only once (example: attaching a handler to a button)