Nested Classes

Nested Classes

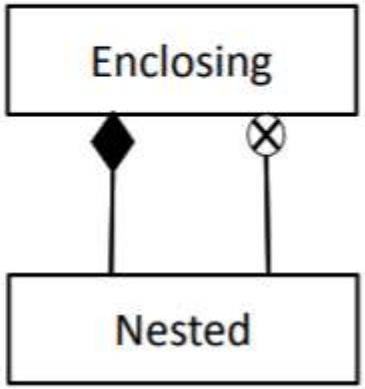
- A class defined inside of another class.

- Nested classes enable you to:
  - logically group classes that are only used in one place
  - increase the use of encapsulation
  - create more readable and maintainable code.

Inner class syntax

```
// outer (enclosing) class
public class Enclosing {
    ...

    // inner (nested) class
    private class Nested {
        ...
    }
}
```



- Only this outer class/object can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName**`.this`

What Happens…

- When we compile code containing inner classes?
  - Class files are made for each inner class, but the naming convention is different

```
LinkedList$ListNode.class
LinkedList$ListIterator.class
```

Types

- Inner class – non-static nested class
  - Local class
  - Anonymous class
- Static inner class

Static Nested class

Here are a few points to remember about static nested classes:

- As with static members, these belong to their enclosing class, and not to an instance of the class
- They can have all types of access modifiers in their declaration
- They only have access to static members in the enclosing class
- They can define both static and non-static members

```java
public class Enclosing {

    private static int x = 1;

    public static class StaticNested {

        private void run() {
            // method implementation
        }
    }

    @Test
    public void test() {
        Enclosing.StaticNested nested = new Enclosing.StaticNested();
        nested.run();
    }
}
```

Inner class (Non-static)

Next, here are a few quick points to remember about non-static nested classes:

- They are also called inner classes
- They can have all types of access modifiers in their declaration
- Just like instance variables and methods, inner classes are associated with an instance of the enclosing class
- They have access to all members of the enclosing class, regardless of whether they are static or non-static
- They can only define non-static members

Local class

Local classes are a special type of inner classes – in which **the class is defined inside a method** or scope block.

Let's see a few points to remember about this type of class:

- They cannot have access modifiers in their declaration
- They have access to both static and non-static members in the enclosing context
- They can only define instance members

Local classes can access only final/effectively final members of the surrounding method or block.

```
public class NewEnclosing {

    void run() {
        class Local {

            void run() {
                // method implementation
            }
        }
        Local local = new Local();
        local.run();
    }

    @Test
    public void test() {
        NewEnclosing newEnclosing = new NewEnclosing();
        newEnclosing.run();
    }
}
```

Anonymous Class

Anonymous classes can be used to define an implementation of an interface or an abstract class without having to create a reusable implementation.

Let's list a few points to remember about anonymous classes:

- They cannot have access modifiers in their declaration
- They have access to both static and non-static members in the enclosing context
- They can only define instance members
- They're the only type of nested classes that cannot define constructors or extend/implement other classes or interfaces

Example – Inner class

- Every class that implements Iterable interface appropriately, can be used in the enhanced For loop (for-each loop).

```
for(Item item: customDataStructure) {
    // do stuff
}
```

- To implement an iterable data structure, we need to:

1. Implement Iterable interface along with its methods in the said Data Structure
2. Create an Iterator class which implements Iterator interface and corresponding methods.

Pseudocode

```
class CustomDataStructure implements Iterable<> {

    // code for data structure
    public Iterator<> iterator() {
        return new CustomIterator<>(this);
    }
}
class CustomIterator<> implements Iterator<> {

    // constructor
    CustomIterator<>(CustomDataStructure obj) {
        // initialize cursor
    }
    // Checks if the next element exists
    public boolean hasNext() {
    }
    // moves the cursor/iterator to next element
    public T next() {
    }
    // Used to remove an element. Implement only if needed
    public void remove() {
        // Default throws UnsupportedOperationException.
    }
}
```