Nested Classes

- Besides fields and methods, a Java class can also contain other classes.

```
class EnclosingClass {
  ...
  class ANestedClass {
    ...
  }
}
```

- The rules for defining such classes are similar to fields and methods.
  - Access specifier:
    * A class defined inside of another class can be public, protected, package private, or private.
  - Scope specifier:
    * A class defined inside of another class can be static or non-static.

Static Nested Classes

- When it is defined as static, it is called a *static nested* class.

```
class EnclosingClass {
   ...
   static class AStaticNestedClass {
      ...
   }
}
```

  - The members (i.e. fields, methods, classes) of a static nested class can access only static members of the *outer* class.
    - ∗ The enclosing class is called the outer class.

- Usage:
  - Static nested classes are used mostly to avoid name clash and to promote *information hiding*.

Inner Classes

- When it is non-static, it is called an *inner* class.

```
class EnclosingClass {
  ...
  class AnInnerClass {
    ...
  }
}
```

  – An inner class is a nested class whose instance exists within an instance of its enclosing class and has direct access to the instance members of its enclosing instance.

Inner Classes (cont.)

```
class EnclosingClass {
  ...
  class AnInnerClass implements IMyInterface {
    ...
  }

  IMyInterface myMethod()
  {
    return new AnInnerClass();
  }
}
```

Inner Classes (cont.)

- Usage:
  - Event listeners for Java GUI components are implemented as inner classes.
  - In the state design pattern, the states of an object are often implemented as inner objects.
    - Since an inner object has access to its outer object (the context), there is no need to have set and get methods for the state.

Anonymous Inner Classes: An Example

```
public class LRStruct
{
  ...
  public final String toString()
  {
    return _head.toString(this);
  }
  ...
```

Anonymous Inner Classes: An Example (cont.)

```
abstract class ANode
{
  ...
  String toString(LRStruct owner)
  {
    return (String)owner.execute(
      new IAlgo()
      {
        public Object emptyCase(LRStruct host, Object input)
        {
          return "()";
        }

        public Object nonEmptyCase(LRStruct host, Object input)
        {
          return "(" + host.getFirst() + host.getRest().execute(
            new IAlgo()
          {
            public Object emptyCase(LRStruct h, Object inp)
            {
              return ")";
            }

            public Object nonEmptyCase(LRStruct h, Object inp)
            {
              return " " + h.getFirst() +
                     h.getRest().execute(this, null);
            }
          }, null);
      }
    }, null);
}
```