TOUR OF SCALA

INNER CLASSES

In Scala it is possible to let classes have other classes as members. As opposed to Java-like languages where such inner classes are members of the enclosing class, in Scala such inner classes are bound to the outer object. Suppose we want the compiler to prevent us, at compile time, from mixing up which nodes belong to what graph. Path-dependent types provide a solution.

To illustrate the difference, we quickly sketch the implementation of a graph datatype:

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node): Unit = {
      if (!connectedNodes.exists(node.equals)) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

This program represents a graph as a list of nodes (`List[Node]`). Each node has a list of other nodes it's connected to (`connectedNodes`). The `class Node` is a *path-dependent type* because it is nested in the `class Graph` . Therefore, all nodes in the `connectedNodes` must be created using the `newNode` from the same instance of `Graph` .

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
val node3: graph1.Node = graph1.newNode
node1.connectTo(node2)
node3.connectTo(node1)
```

We have explicitly declared the type of `node1` , `node2` , and `node3` as `graph1.Node` for clarity but the compiler could have inferred it. This is because when we call `graph1.newNode` which calls `new Node` , the method is using the instance of `Node` specific to the instance `graph1` .

If we now have two graphs, the type system of Scala does not allow us to mix nodes defined within one graph with the nodes of another graph, since the nodes of the other graph have a different type. Here is an illegal program:

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
node1.connectTo(node2)      // legal
val graph2: Graph = new Graph
val node3: graph2.Node = graph2.newNode
node1.connectTo(node3)      // illegal!
```

The type `graph1.Node` is distinct from the type `graph2.Node` . In Java, the last line in the previous example program would have been correct. For nodes of both graphs, Java would assign the same type `Graph.Node` ; i.e. `Node` is prefixed with class

`Graph`. In Scala such a type can be expressed as well, it is written `Graph#Node`. If we want to be able to connect nodes of different graphs, we have to change the definition of our initial graph implementation in the following way:

```
class Graph {
  class Node {
    var connectedNodes: List[Graph#Node] = Nil
    def connectTo(node: Graph#Node): Unit = {
      if (!connectedNodes.exists(node.equals)) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

← previous next →