SCALA PATTERN MATCHING

Pattern matching is the second most widely used feature of Scala, after function values and closures. Scala provides great support for pattern matching for processing the messages.

A pattern match includes a sequence of alternatives, each starting with the keyword **case**. Each alternative includes a **pattern** and one or more **expressions**, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions. Here is a small example, which shows how to match against an integer value:

```
object Test {
   def main(args: Array[String]) {
      println(matchTest(3))

   }
   def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
many

C:/>
```

The block with the case statements defines a function, which maps integers to strings. The match keyword provides a convenient way of applying a function *likethepatternmatchingfunctionabove* to an object. Following is a second example, which matches a value against patterns of different types:

```
object Test {
   def main(args: Array[String]) {
      println(matchTest("two"))
      println(matchTest("test"))
      println(matchTest(1))

   }
   def matchTest(x: Any): Any = x match {
      case 1 => "one"
      case "two" => 2
      case y: Int => "scala.Int"
      case _ => "many"
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
2
many
one

C:/>
```

The first case matches if x refers to the integer value 1. The second case matches if x is equal to the string"two". The third case consists of a typed pattern; it matches against any integer and binds the selector value xto the variable y of type integer. Following is another form of writing same

match...case expressions with the help of braces {...}:

```
object Test {
   def main(args: Array[String]) {
      println(matchTest("two"))
      println(matchTest("test"))
      println(matchTest(1))

   }
   def matchTest(x: Any){
      x match {
         case 1 => "one"
         case "two" => 2
         case y: Int => "scala.Int"
         case _ => "many"
      }
   }
}
```

Matching Using case Classes:

The **case classes** are special classes that are used in pattern matching with case expressions. Syntactically, these are standard classes with a special modifier: **case**. Following is a simple pattern matching example using case class:

```
object Test {
   def main(args: Array[String]) {
    val alice = new Person("Alice", 25)
    val bob = new Person("Bob", 32)
    val charlie = new Person("Charlie", 32)

      for (person <- List(alice, bob, charlie)) {
         person match {
            case Person("Alice", 25) => println("Hi Alice!")
            case Person("Bob", 32) => println("Hi Bob!")
            case Person(name, age) =>
               println("Age: " + age + " year, name: " + name + "?")
         }
      }
   }
   // case class, empty one.
   case class Person(name: String, age: Int)
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Hi Alice!
Hi Bob!
Age: 32 year, name: Charlie?

C:/>
```

Adding the case keyword causes the compiler to add a number of useful features automatically. The keyword suggests an association with case expressions in pattern matching.

First, the compiler automatically converts the constructor arguments into immutable fields *vals*. The val keyword is optional. If you want mutable fields, use the var keyword. So, our constructor argument lists are now shorter.

Second, the compiler automatically implements **equals, hashCode,** and **toString** methods to the class, which use the fields specified as constructor arguments. So, we no longer need our own toString methods.

Finally, also, the body of **Person** class is gone because there are no methods that we need to

define!