Scala match pattern

myjlms99@gmail.com

Match pattern syntax

Match 구문 사용

match 구문은 실제 다른 언어의 switch 구문가 유사하지만 실제 처리하는 방식은 더 다양한 매칭을 처리할 수 있도록 지원한다.

```
expression   match {
case   표현식 [가드] = > 표현식
}
```

Match pattern syntax
값 매칭하기

값 패턴 매칭

특정 값을 매칭하기 위해 실제 case 문 내에 값을 지정해서 처리한다.

```
scala> def matchText(x:Int) : String = x match {
     |     case 1 => "one"
     |     case 2 => "two"
     | }
matchText: (x: Int)String

scala> matchText(1)
res0: String = one

scala> matchText(2)
res1: String = two

scala> matchText(3)
scala.MatchError: 3 (of class java.lang.Integer)
  at .matchText(<console>:11)
  ... 29 elided
```

case 문에 지정이 안 된 경우가 발생하면 matchError가 발생

Match pattern syntax
디폴트 매칭하기

Defaults matching : 와일드 카드

매치가 안되는 경우 에러가 발생하므로 모든 것을 매칭할 수 있도록 _(와일드 카드)를 사용해서 처리한다.

case 문에 _ 을 사용해서 전체 처리를 할 수 있도록 해야 한다.

```
scala> def matchText(x:Int) : String = x match {
     |     case 1 => "one"
     |     case 2 => "two"
     |     case _ => "all"
     | }
matchText: (x: Int)String

scala> matchText(1)
res5: String = one

scala> matchText(2)
res6: String = two

scala> matchText(3)
res7: String = all
```

Defaults matching : 변수

특정 변수를 사용하면 매칭되지 않는 것을 전부 처리된다. 이때 변수는 소문자로 시작되어야 한다.

case 문에 특정 변수를 사용해서 전체 처리를 할 수 있도록 해야 한다.

```
scala> def matchText(x:Int) : String = x match {
     |     case 1 => "one"
     |     case 2 => "two"
     |     case a => "all"
     | }
matchText: (x: Int)String

scala> matchText(1)
res8: String = one

scala> matchText(2)
res9: String = two

scala> matchText(3)
res10: String = all
```

Match pattern syntax
guard 처리해서 매칭하기

Guard 처리

특정 변수 등에 매칭할 때 세부적인 가드를 지정해서 true로 일치할 때만 매칭되도록 처리가 가능하다.

case 문에 if 문을 사용해서 실제 처리하는 값들이 범위를 지정해서 처리한다,

```
scala> def matchGuard(x:Int) = x match {
     |    case y if (y> 10) => y
     |    case z if (z <= 10) => z
     | }
matchGuard: (x: Int)Int

scala> matchGuard(11)
res18: Int = 11

scala> matchGuard(10)
res19: Int = 10
```

Match pattern syntax
sequence 매칭하기

상수를 이용해서 sequence 처리

Seq 타입을 받을 경우 아무것도 없는 Nil 과 :: 연산을 이용해서 하나의 값을 처리하고 나머지는 와일드카드 처리한다.

```
scala> def matchInt(seq : Seq[Int]) = seq match {
     |    case Nil => "empty"
     |    case head :: Nil => s"one entry $head"
     |    case _ => "multiple entries"
     | }
matchInt: (seq: Seq[Int])String

scala> matchInt(List())
res20: String = empty

scala> matchInt(List(1))
res21: String = one entry 1

scala> matchInt(List(1,2))
res22: String = multiple entries
```

Sequence 타입으로 처리

Seq 타입을 case 문에 지정해서 실제 인스턴스의 구조에 따라 처리할 수 있도록 만들 수 있다.

```
scala> def matchSeq(seq : Seq[Int]) = seq match {
     |    case Seq() => "empty"
     |    case Seq(head) => s"one entry $head"
     |    case Seq(one,two) => s" two entry $one, $two "
     |    case _ => "multiple entries"
     | }
matchSeq: (seq: Seq[Int])String

scala> matchSeq(Seq())
res24: String = empty

scala> matchSeq(List(1))
res25: String = one entry 1
```

Match pattern syntax
Tuple 매칭하기

Tuple타입으로 처리

Seq 타입으로 tuple 처리를 하지말고 별도의 튜블 패턴을 만들어서 처리해야 한다.

```
scala> def matchTupe(x:Any) :String = x match {
     |    case (a,b)  => s"get $a and $b "
     |    case (a,b,c) => s"get $a, $b, and $c"
     |    case _ => "multiple tuple"
     | }
matchTupe: (x: Any)String

scala> matchTupe((1,2))
res28: String = "get 1 and 2 "

scala> matchTupe((1,2,3))
res29: String = get 1, 2, and 3
```

Match pattern syntax
타입 매칭하기

타입 매칭 처리하기

특정변수를 지정하고 그 뒤에 데이터 타입을 지정해서 특정 타입을 매칭해서 처리한다

case 문에 특정변수 다음에 타입을 지정해서 처리하면 된다.

```
scala> def matchType(x:Any) : Any = x match {
     |     case y : Int => "Int"
     |     case a : Double => "Double "
     |     case s : String => "String"
     |     case _ => "Any"
     | }
matchType: (x: Any)Any

scala> matchType(1)
res14: Any = Int

scala> matchType(10.1)
res15: Any = "Double "

scala> matchType("sss")
res16: Any = String

scala> matchType(10.1f)
res17: Any = Any
```

Match pattern syntax 혼용하기

다양한 타입 매칭 하기

최상위 타입을 지정해서 내부적으로 하위 타입을 할당해서 매칭 처리할 수 있도록 처리도 가능하다.

case 문에 값과 타입을 지정해서 처리도 가능하다.

```
scala> def matchType(x:Any) :Any = x match {
     |     case 1 => "Int"
     |     case "two" => 2
     |     case y:Int => "Int"
     |     case _ => "all"
     | }
matchType: (x: Any)Any

scala> matchType(1)
res11: Any = Int

scala> matchType("two")
res12: Any = 2

scala> matchType(3)
res13: Any = Int
```

Match pattern syntax
생성자 매칭하기

Case class를 이용해서 처리

실제 생성자를 이용해서 매칭하려면 case class로 지정된 것을 처리해서 사용해야 한다.

```
scala> case class Person(val fn: String, val ln:String)
defined class Person

scala> def matchCon(x:Person) : String = x match {
     |     case Person("D",_) => "Person D"
     |     case Person("K",_) => "Person K"
     |     case _ => "missing person"
     | }
matchCon: (x: Person)String

scala> val p1 = Person("D","D")
p1: Person = Person(D,D)

scala> val p2 = Person("K","D")
p2: Person = Person(K,D)

scala> matchCon(p1)
res30: String = Person D

scala> matchCon(p2)
res31: String = Person K
```

Case class 내의 @ 표기

실제 생성자를 이용할 때 @ 다음에 생성자를 사용해서 세부적으로 처리도 가능하다.

```
scala> case class Person(fn: String, ln:String)
defined class Person

scala> val p = Person("Dahl", "Moon")
p: Person = Person(Dahl,Moon)

scala> p match {
     |    case x @ Person("Moon",_) => println(s"Moon $x")
     |    case x @ Person("Dahl",_) => println(s"Dahl $x ")
     |    case _ => "all"
     | }
Dahl Person(Dahl,Moon)
res38: Any = ()
```

Match pattern syntax
Option 타입을 사용 매칭하기

option 타입 처리

스칼라는 타입 체크할 때 오류를 최소화하기 위해 Option 타입을 제공해서 null에 대한 예외를 처리하지 않도록 제공한다.

```
scala> val map : Map[String, String] = Map("k1" -> "val1")
map: Map[String,String] = Map(k1 -> val1)

scala> val val1 : Option[String] = map.get("k1")
val1: Option[String] = Some(val1)

scala> val val2 : Option[String] = map.get("k2")
val2: Option[String] = None

scala> println(val1)
Some(val1)

scala> println(val1.get)
val1

scala> println(val2.get)
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:349)
  at scala.None$.get(Option.scala:347)
  ... 29 elided

scala> println(val2.getOrElse(0))
0
```

None 값을 가져올 경우는 getOrElse(값)을 이용해서 초기값을 처리할 수 있도록 한다.

option 타입내의 None 처리

스칼라는 타입 체크할 때 기본 값을 어떻게 처리할 지에 대해 getOrElse, isEmpty로 지정해야 한다.

```
scala> val val3 : Option[String] = None
val3: Option[String] = None

scala> val3.getOrElse(0)
res61: Any = 0

scala> val3.getOrElse("")
res62: String = ""

scala> val3.isEmpty
res63: Boolean = true

scala> val3.get
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:349)
  at scala.None$.get(Option.scala:347)
  ... 29 elided
```

None 값을 가져올 경우는 getOrElse(값), isEmpty을 이용해서 초기값을 처리할 수 있도록 한다.

option 타입을 리스트로 처리

Option 타입을 리스트에 넣고 이 내부의 원소를 isDefined로 체크해서 None이 아닌 경우에 처리

```
scala> class E(val name:String, val gender:Option[String])
defined class E

scala> val e1 = new E("Dahl",Some("male"))
e1: E = E@2e140dd9

scala> val e2 = new E("Moon",None)
e2: E = E@3eec49ac

scala> val eList = List(e1,e2)
eList: List[E] = List(E@2e140dd9, E@3eec49ac)

scala> eList.foreach { x =>
     |    x match {
     |       case x.gender if (x.gender.isDefined) => println(x.name)
     |       case _ => println(x.name, x.gender)
     |    }
     | }
(Dahl,Some(male))
(Moon,None)
```

함수에 option 타입 처리

Option[타입]으로 지정할 경우 올 수 있는 자료형은 Some(타입)이나 None이다. Some일 경우 get으로 호출해서 내부 값을 가져온다.

```
scala> def add(x:Option[Int]) : Int = {
     |     x match {
     |         case None => 2
     |         case _ => x.get + 2
     | }
     | }
add: (x: Option[Int])Int

scala> add(None)
res50: Int = 2

scala> add(Some(5))
res51: Int = 7
```

함수에 option타입 getOrElse

Option[타입]으로 지정할 경우 올 수 있는 자료형은 Some(타입)이나 None이다. getOrElse 메소드로 실제 None이 올 경우 초기값을 처리할 수 있도록 제공한다.

```
scala> def add(a: Option[Int]) : Int = a.getOrElse(0) + 2
add: (a: Option[Int])Int

scala> add(Some(5)
     | )
res52: Int = 7

scala> add(None)
res53: Int = 2
```