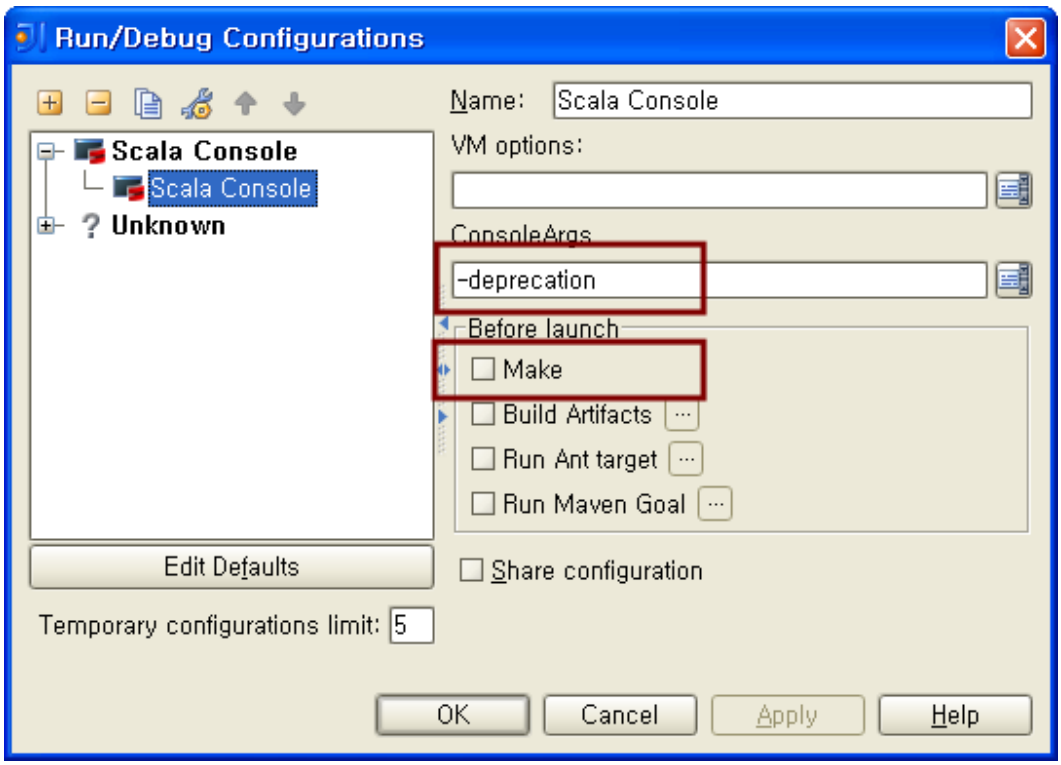
pattern matching

- intro
  - scala 2.8.final이 나왔어요
  - deprecated option



  - ctrl + shift + F9 개별 컴파일

- 나름의 정의: 강화된 (switch)문?

- 다음과 같이 사용

```
scala> val input = "A"
input: java.lang.String = A
scala> input match {
     |   case "C" => "씨!"
     |   case "B" => "비!"
     |   case "A" => "에이!"
     | }
res7: java.lang.String = 에이!

```

- 평가는 위에서 아래로, break는 자동으로
- case와 모두 일치 하지 않으면?
  - scala.MatchError 발생
  - 대비책 -> 기타 케이스에 대해 case _ 사용
- 리터럴 (literal)

  :[wikipedia 정의] 소스 코드에서 특정 '값'을 나타내기 위한 표기

  - cf1. function literal (= anonymous function)

```
//함수를 저장
scala> val func1:(Int,Int)=>Int = (x,y)=> x + y
func1: (Int, Int) => Int = <function2>
```

    - wiki 의 function literal 소제목 확인 > higher order function 과 관련있음
    - Java는 7에서 지원 예정(Project Lambda)
  - cf2. object literal (=javascript object notation)

```
// JSON 표기 2가지 예
{ name:"Ron Jeffries", age:19}  // 표기 1
{ name:"JeongHoon Byun",        // 표기 2
  age:18,
  doStudy: function() { ... }
}

```

- 패턴별 정리

  :[요령] 학습의 시작 단계에선 각각의 패턴을 암기하는 것이 좋다.

  1. 와일드카드 패턴

```
expr match {
    case BinOp(_, _, _) => println(expr +"is a binary operation")   // 이름 없는 변수
    case _ => println("It's something else")                        // 기타 case
}
```

  2. 상수 패턴

```
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"      // 싱글턴 오브젝트
  case _ => "something else"        // 와일드 카드
}
```

  3. 변수 패턴

     - 전달된 내용을 변수에 대입

```
expr match {
  case 0 => "zero"
  case somethingElse => "not zero: "+ somethingElse
}
```

     - <예시> 다음 상수 패턴은 변수 패턴과 혼동의 여지가 있다

```
scala> import math.Pi
import math.Pi
scala> someValue match {
     |   case Pi => "pi"
     |   case _ => "?"
     | }
```

```
res13: java.lang.String = ?

```

```
scala> someValue match {
     |   case pi => "pi"
     |   case _ => "?"
     | }
<console>:14: error: unreachable code
         case _ => "?"
                   ^
```

- 상수명은 꼭 '대문자'로 시작. -> 변수 패턴 참고
- this. 등으로 구체적으로 지시하면 상수명 대소문자 문제 없음

4. Typed 패턴

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => 1
}
```

```
scala> 1 match {
     |   case _:Int => "int"
     |   case _ => "?"
     | }
res8: java.lang.String = int
```

5. Tuple 패턴

- <예시>

```
scala> val tuple2 = ( 1, "ABC")
tuple2: (Int, java.lang.String) = (1,ABC)
scala> tuple2 match {
     |   case ( x:Int, "ABC") => "ABC 앞에 " + x
     |   case _ => "?"
     | }
res6: java.lang.String = ABC 앞에 1

```

- cf. Tuple 대입

```
scala> val (name, age) = ("안세원", 20)
name: java.lang.String = 안세원
age: Int = 20
```

6. Sequence 패턴(List,Array)
   - val list = List(1,2,3,4,5) 이라 가정
   - _* : 나머지 표지

```
scala> list match {
     |   case List( first, second, _* ) => first + "-" + second
     |   case _ => "?"
     | }
```

```
res5: java.lang.String = 1-2
```

- cons 연산자 사용

```
scala> List(1,2,3,4) match {
     |   case head::tail => "꼬리 =" + tail
     |   case _ => "?"
     | }
res4: java.lang.String = 꼬리 =List(2, 3, 4)
```

- namedVar@_* 와 같이 변수명 바인딩 가능

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
}
```

- 무관한 quiz. 다음 함수 정의에서 parts를 가변 인자로 바꿔라a

```
def apply( parts: String):String =
  parts.reverse.mkString(".")
```

7. Constructor 패턴
   - 패턴 매칭을 강력하게 만드는 기법
   - 마치 생성자를 기반으로 접근하는 듯한 모양새라 Constructor 패턴이라 불림
   - 실제 생성자를 이용하는 것은 아니다

```
scala> class SomeClass
defined class SomeClass
scala> val a = SomeClass()
<console>:5: error: not found: value SomeClass
       val a = SomeClass()
               ^
scala> val a = new SomeClass()
a: SomeClass = SomeClass@3ca56f
scala> a match {
     |   case SomeClass() => "some!"
     |   case _ => "?"
     | }
<console>:9: error: not found: value SomeClass
         case SomeClass() => "some!"
              ^
```

   - case class를 사용하면 날로 먹을 수 있다

```
scala> case class SomeClass()
defined class SomeClass
scala> val b = SomeClass()
b: SomeClass = SomeClass()
scala> b match {
```

```
      |   case SomeClass() => "some !"
      |   case _ => "?"
      | }
res1: java.lang.String = some !
```

        - unapply 메서드에 대한 이해가 필요
            - 뒤에 곧 나온다
    8. XML
        - 14장으로 미루자! (yaho!)
- guard
    - case 에 if 를 달면 된다.

```
case n:Int if n < 0 => "0 보다 작다"
```

- case classes
    - 이전 설명 참조
    - 다음 3가지를 알아서 구현

        1. factory mathod (apply)
        2. val
        3. toString (+4. pattern matching (unapply))
    - case class를 패턴 매칭에 쓸 수 있다.
        - 예시

```
//REPL에 다음과 같이 입력
case class Product(code:String, price:Int)
def describeProduct(input:Product):String = {
  input match {
    case Product("맥북", price) if price > 1000000 => "비싼맥북!"
    case Product("콜라", price) => price+ "원짜리 콜라"
  }
}

```

        - quiz
            - describeProduct(Product("콜라", 1200)) ?
            - describeProduct(Product("맥북", 800000) ?
            - describeProduct(Product("맥북", 2100000) ?
        - 패턴 매칭에 쓰려면 꼭 case class로 만들어야 하나? NO! Extractor!

- sealed classes
    - sealed
    - 결국은 case의 coverage에 대한 이야기
        - 컴파일러에게 case의 모든 경우를 알려주고 100%를 커버하지 않으면 경고
    - <사용법> 상속 계층의 최상위에 sealed 키워드 사용
        - 상속한 자식 클래스들이 전체 coverage가 됨
    - <예제>: 혈액형

```
sealed abstract class BloodType
case class AType() extends BloodType
case class BType() extends BloodType
```

```
case class OType() extends BloodType
case class ABType() extends BloodType
scala> val a:BloodType = AType()
a: BloodType = AType()
scala> a match {
     |   case AType() => "A!"
     | }
<console>:10: warning: match is not exhaustive!
missing combination         ABType
missing combination          BType
missing combination          OType
       a match {
       ^
res2: java.lang.String = A!
```

- List 도 sealed abstract로 선언됨

```
scala> List(1,2,3,4) match {
     |   case head::tail => "꼬리 =" + tail
     | }
<console>:6: warning: match is not exhaustive!
missing combination            Nil
       List(1,2,3,4) match {
           ^
res3: java.lang.String = 꼬리 =List(2, 3, 4)
```

- extractor
  - apply 와 쌍을 이루는 개념

```
object EMail {
    // injection 메서드 (optional)
    def apply(user: String, domain: String) = user +"@"+ domain
    // extraction 메서드
    def unapply(str: String): Option[(String, String)] = {
        val parts = str split "@"
        if (parts.length == 2) Some(parts(0), parts(1)) else None
    }
}
```

  - 다음 결과를 유심히 관찰하자

```
scala> EMail("lee","gmail.com")         //EMail.apply 와 같다
res1: java.lang.String = lee@gmail.com
scala> EMail.unapply("seok@naver.com")
res3: Option[(String, String)] = Some((seok,naver.com))
```

        - Email객체와 관련없는 문자열에 대한 처리가 이뤄졌음
            - extractor vs case classes 참고
            - case classes는 클래스가 담고 있는 data에 대한 표현
    - apply & unapply

구성요소--(apply)--> 객체 --(unapply)-->구성요소

- case 구문에는 unapply가 사용된다.

```
"seok@naver.com" match {
  case EMail("seok","naver.com") =>     // EMail.unapply 호출
    println("혹시 석종일?");
  case _ =>
    println("석호필인가... 잘 모르겠다");
}
```

- extractor signature (장표 참고)
    1. def unapply(...): Boolean
        - extractor 인자 개수 0개인 case문에 해당함

```
object UpperCase {
  def unapply(s: String): Boolean =
    s.toUpperCase == s
}
```

    2. def unapply(...): Option[T]
        - 인자 개수 1개 "
    3. def unapply(...): Option[(T,U)]
        - 인자 개수 (tuple 원소 개수)개
    4. def unapplySeq(...): Option[Seq[T]]
        - 인자 개수 0~n개 **동시** 대응 가능
        - 위 1~3의 unapply 메서드는 signature 때문에 한 클래스에 여러 개 구현할 수 **없음**

```
// email에 사용하는 도메인 명(ex. gmail.com)
object Domain {
  def apply(parts: String*): String =
    parts.reverse.mkString(".")
  def unapplySeq(whole: String): Option[Seq[String]] = {
    println("unapplySeq")
    Some(whole.split("\\.").reverse)
  }
}

```

        - 인자 개수 가변
        - case 구문에서 _* 사용 가능
    5. 패턴을 이용한 대입
        - Tuple을 이용한 대입. 기억하나요?

```
scala> val (name, age) = ("lee", 30)
name: java.lang.String = lee
age: Int = 30
```

        - 예시

```
scala> val Domain(top, sub@_*) = "a.b.c"
unapplySeq
top: String = c
sub: Seq[String] = WrappedArray(b, a)
```

6. 패턴의 복합적인 사용

```
scala> "www.NAVER.com" match {
     |   case Domain(_, first@UpperCase(), "www") =>
     |     first + "!!";
     |   case _ => "?"
     | }
unapplySeq
res6: java.lang.String = NAVER!!
```

- <예제>: Person의 extractor
  - PASIV code
    - (Personal Asset Simple Indentification Value)
    - 이름,나이, 재산정도가 csv형태로 연결된 문자열

```
"박성철,28,재산많음"    // 예 1
"이건희,50,재산보통"    // 예 2
```

- <예제>: HTTP Request의 extractor (상상만 해보자)

정규식

- 스칼라에서 extractor를 사용하는 좋은 예
- refresher
- + 숫자 한개 이상(* => 0개 이상)
- 스칼라에서 정규식 선언하기

```
import scala.util.matching.Regex
// raw string 사용
val Decimal = new Regex("""(-)?(\d+)(\.d*)?""")
// ?
val Decimal = """(-)?(\d+)(\.d*)?""".r


```

- "".r ?
  - scala.runtime.RichString 참고
- extractor 활용

```
val str = "8월 우리집 전기료 43000원으로 저번달 대비 31.2% 상승"
for (Decimal(s,i,d) <- Decimal findAllIn str) {
  println("부호:" + s + " 정수:" + i + " 소숫점이하:" + d)
}
```

5