**Scala**

- https://docs.scala-lang.org/
- 기본적으로 Java를 다뤄봤다는 전제하에 실용적인 측면으로만 기술.
- 2.12 이상 버전을 기준으로 기술.

Language, Scala, JVM, Object Oriented Programming, Functional Programming

**환경 설정**

- JDK 1.8 이상 설치

**Command Line**

https://www.scala-lang.org/download/

- SBT (Scala Build Tools)

Scala의 Ant/Maven/Gradle와 비교될 수 있음. 직접으로 프로젝트 생성, 배포 관여.

```
# 생성
sbt [Project Name]

# Build (Fat Jar 생성)
## build.sbt에
## addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.6")
## 추가 필요
### 생성된 jar 위치 : target/scala-[version]/*.jar
sbt assemblyty
```

- Scala Binaries (선택 사항)
  - REPL(scala), 단독 Compile(scalac)을 이용하기 위해 필요

```
# 실행
scala

scala> 1
res0: Int = 1

# 종료
scala> :quit
## 혹은
scala> :q
```

#### IDE

- Intellij에 기본적으로 SBT 내장. Intellij만 설치하면 된다.

### REPL 실행 방법

왼쪽 Project Tree에서 아무 Class에 오른쪽으로 누르고 “Run Scala Console”.

#### Hello, world!

- 가변 인자 args를 입력 받아 “Hello, world!”를 출력하는 예제 작성

#### Script

- Compile 하지 않고, Shell script 처럼 사용

```
// class, main 함수를 선언하지 않고도 가변인자 args 사용 가능!
args.foreach(print)
println()
```

```
> scala script.scala Hello , world !
Hello, world!
```

#### Compile

https://www.scala-lang.org/documentation/your-first-lines-of-scala.html

### App trait 상속 O

```
object HelloWorld extends App {
  // class, main 함수를 선언하지 않고도 가변인자 args 사용 가능!
  args.foreach(print)
  println()
}
```

```
> scalac HelloWorld.scala
```

```
> scala HelloWorld Hello , world !
```

### App trait 상속 X

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    args.foreach(print)
    println()
  }
}
```

```
> scalac HelloWorld.scala
> scala HelloWorld Hello , world !
```

#### 문법

#### Identifier

식별자 규칙 및 관례적 명명법 서술.

- 연산자 정의(Operator Defination) Method와 암시적 변환(Implicit Conversions)는 간결하고 유연한 표현을 정의할 수 있지만, 잘못 사용하면 가독성을 떨어뜨리는 결과를 낳으니 주의가 필요하다.

### Alphanumeric Identifier

1. Java와 동일하게 영숫자 혹은 '_', '$'가 가능하나 '$' Scala Compiler가 내부적으로 생성하는 식별자에 사용하는 예약 문자이므로 충돌 가능성이 있기 때문에 사용해서는 안된다.
2. 상수는 첫글자만 대문자로, Camel Case 표현을 한다. (Java 표현으로 MAX_VALUE라면, Scala는 MaxValue)

### Operator Identifier

- Java와 가장 큰 차이점 중 하나. 이를 이용하면 Operator Overloading이 가능. Method에만 적용 가능.
- +, :, ?, ~, #, +, ++, ::(List에서 요소 덧붙임), :::(List에 다른 List 추가), <?>, :→ 등 사용 가능. 이 때 길이 제한이 없다.
- Overloading 예시

```
def *** (v: Int) = math.pow(v, 3)
```

## 연산자 우선순위와 결합법칙

**:**(Cons; Construct)로 끝나는 연산자는 right-associative. 이외 나머지는 left-associative

```
// left-associative
a * b * c
= (a * b) * c
= a.*(b).*(c)

// right-associative
a ::: b ::: c
= a ::: (b ::: c)
= b.:::(c).:::(a)
```

### Mixed Identifier

Alphanumeric + Operator. 예를 들어 myvar_=. Scala Compiler가 Property를 지원하기 위해 내부적으로 생성.

### Literal Identifier

역따옴표(`)를 사용하며 Runtime에서 인식할 수 있는 문자열 생성. 이를 이용하면 예약어도 변수나 Method 이름으로 지정 가능.

```
// 아래 코드는 Compile 및 실행에 문제가 없다.

val `val` = 3;
def show(`var` : Int) = println(`var`);

show(`val`)
```

#### 변수

- val : Immutable
- var : mutable

#### Type

### Type Hierarchy



| | |
|---|---|
| **Any** | 모든 type의 supertype. equals, hashCode, toString 메서드가 정의됨. |
| **AnyVal** | value type을 표현, non-nullable. Double, Float, Long, Int, Short, Byte, Char, Unit, Boolean의 상위 타입. |
| **AnyRef** | referenced type을 표현, java.lang.Object에 대응. |
| **Nothing** | 모든 type의 subtype (i.e. bottom type)으로 value를 가지고 있지 않다. |
| **Null** | 모든 reference type의 subtype. keyword literal 'null'을 값으로 가지고 있다. |

### Literals

### Integer Literals

- C/Java와 달리 8진(0으로 시작하는 정수형)을 지원하지 않음. 10진과 16진만 지원.

### String Literals

# Raw Sring

큰따옴표 3개(""")를 이용하여 그 내부 문자열을 "날 것" 그대로 표현하는 문자열. (공백, 개행 전부 포함)

```
object Test extends App {
  // Java에서의 일부 특수문자 표현. 가독성이 좋지 않음.
  val str1 = "Hello\n\"world\"";
  println(str1)

  /**
    * Hello
    * "world"
    */


 // 공백, 개행 모두 그래도 출력
  val str2 =
    """Hello,
      "world"!
    """
  println(str2)

  /**
    * Hello
    * "world"
    */
```

```
  //|를 넣으면 문자열의 시작을 알림. 이후 stripMargin을 하면 이전 공백 제거
    val str3 =
      """          |Hello,
        |"world!"
      """.stripMargin
    println(str3)

    /**
      * Hello
      * "world"
      */
}
```

# String interpolation

문자열 내부에 표현식(expression)을 내장시키는 것. 이로써 문자열을 이어붙이지 않고 가독성이 좋아짐.

- s : “$val” 혹은 “${expression}“과 함께 쓰여 mapping.
- raw : 문자열 그대로 출력.
- f : printf 형태의 형식 지정.

| | Description | Example |
|---|---|---|
| s | “$val” 혹은 “${expression}“과 함께 쓰여 mapping. | `val str = "9 * 9 = "`<br>`println(s"$str ${9 * 9}")`<br>`/**`<br>`* 9 * 9 = 81`<br>`*/` |
| raw | 문자열 그대로 출력. | `println(raw"A\nB\\C\tD\bE")`<br>`/**`<br>`* A\nB\\C\tD\bE`<br>`*/` |
| f | printf 형태의 형식 지정.<br>${expression}%[format] | `println(f"${math.Pi}%.7f")`<br>`println(f"${3 * 3}%03d")`<br>`/**`<br>`* 3.1415927`<br>`* 009`<br>`*/` |

## Symbol literals

- 작은 따옴표(') + (알파벳 + 숫자)
- 단순 식별자가 필요할 때 사용. Java의 enum이나 Clojure의 Keyword와 비슷.
  - Scala의 Strong Type 언어이기 때문에 Weak Type 형태로 사용할 때 사용.
  - Clojure의 Keyword처럼 자기 자신을 반환
  - .name 필드로 문자열 변환 가능.

### Array

- 모든 것을 Method가 있는 객체로 다루기 때문에 Parameter에 해당하는 호출은 **()**을 사용한다.

```
val fruits = new Array[String](2)
// val fruits: String = new Array[String](2)

fruits(0) = "apple"
// fruites.update(0, "apple")

fruits(1) = "banana"
// fruites.update(1, "banana")
```

```
val fruits = Array("apple", "banana")
// val fruits = Array.apply("apple", "banana")
```

#### Operator

### Operator = Method!

Scala에서 모든 연산자는 Method!! 실질적으로 Scala는 Method를 호출한다.

```
/** Binary Operator **/

val sum = 1 + 2
// 1.+(2)

"Hello, world!" indexOf 'o'
// "Hello, world!".indexOf('o')

"Hello, world!" indexOf ('o', 5)
// "Hello, world!".indexOf('o', 5)



/** Unary Operator **/

/*** 전위 표기법 ***/
// +,0,!,~ 4가지.

val a = -2.0
// (2.0).unary_-
```

```
/*** 후위 표기법 ***/
// 인자를 취하지 않는 Method를 '.'이나 괄호 없이 호출

val str = "Hello, world!" toLowercase
// val str = "Hello, world!".toLowercase()
```

### == & eq

Java와 동일하다.
==, !=은 원시 타입에서 값(value)이 같은지 비교, eq/ne는 JVM Heap에서 참조(reference) 비교

```
scala> List(1, 2, 3) == List(1, 2, 3)
res8: Boolean = true

scala> List(1, 2, 3) eq List(1, 2, 3)
res9: Boolean = false

scala> List(1, 2, 3) ne List(1, 2, 3)
res10: Boolean = true

// 주의!
scala> ("He" + "llo") == "Hello"
res11: Boolean = true
```

### Wrapper

```
scala> 111 max 222
res0: Int = 222

scala> 111 min 222
res1: Int = 111

scala> -math.Pi abs
res2: Double = 3.141592653589793

scala> -math.Pi round
res3: Long = -3

scala> (1.0 / 0) isInfinity
res4: Boolean = true

scala> 1 to 100
res5: scala.collection.immutable.Range.Inclusive = Range 1 to 100
```

```
scala> "hello" capitalize
res6: String = Hello

scala> "Hello, World" drop 7
res7: String = World
```

#### 내장 제어 구문

**전체적으로 재귀나 내장함수(reduce, map, filter, reduce)로 대체가 가능하기 때문에 사용을 자제하자. 제어 구문은 함수형 표현에 적합하지 않다.**

- 키워드로써 break, continue는 존재하지 않는다. (함수형 언어에 어울리지 않으므로.)
    - 굳이 break를 쓰고 싶다면 scala.util.control.Breaks 에 있는 것을 사용할 수 있다… 만…
- 제어 구문들도 함수이므로 var/val에 할당할 수 있다. (이 때, 기본적으로 바로 계산한다. lazy loading이 필요하면 lazy를 맨 앞에 붙이자.)
- while은 대체로 중간 결과를 저장하는 변수 var가 필요하기 때문에 사용을 자제하자.

### for

foreach 문만 존재한다. 그러나 대체로 반복하는 경우 내장 함수(foreach, reduce, map, filter 등)이 존재하기 때문에 특별한 경우가 아니면 큰 필요성이 없다.

```
/* Collections */

//10 ~ 100
for (i <- 10 to 100)
    println(i)
//10 ~ 99 (최대값 제외)
for (i <- 3 until 10)
    println(i)

/* Iterator Guard / Filter. if 표현식이 true일 때만 반복 */
for (file <- new java.io.File(".").listFiles()
        if file.isFile
        if file.getName.endsWith(".scala"))
    println(file)

/* 중첩 및 임시 변수 Binding */
// 구구단 출력
for {x <- (1 to 9)
     y <- (1 to 9)
     result = x * y}
    println(f"$x * $y = $result%02d")
```

## for-yield

<EXPRESSION>의 값을 Collection로 반환.

```
for (<IDENTIFIER> <- <ITERATOR>) yield <EXPRESSION>
```

```
// 인자가 없기 때문에 ()를 생략한 형태.
  def timesTable = for {
              x <- (1 to 9)
              y <- (1 to 9)
              result = x * y}
    yield f"$x%2d * $y%2d = $result%2d"

  println(timesTable)
  // Vector( 1 *  1 =  1,  1 *  2 =  2, ... ,  9 *  8 = 72,  9 *  9 = 81)
```

### try-catch

- try, finally는 단 하나의 식만 포함한다면 중괄호 생략 가능.

```
  try {
    val f = new FileReader(".")
  } catch {
    case ex: FileNotFoundException => ex.printStackTrace()
    case ex: IOException => ex.printStackTrace()
  } finally println("done")
```

- 다른 제어 구조, 함수와 마찬가지로 결과로 값을 반환한다. 이 때, finally 절의 값은 버려진다.
- finally에는 값을 반환하지 말고 Side-effect(결과 완료 알림, 파일 저장등)만 이용하자.

```
// 결과:2
  // Java 형식으로 finally 절 안에 return문을 명시적으로 사용하면 try/cath의 결과를 덮어쓰게 된다.
// 잘 생각해보면 Java에서 finally는 어떤 결과로든 항상 실행되는 구문이기 때문.
  def test1(): Int = try return 1 finally return 2

// 결과:1
  // Scala 형식으로 작성하면 의도된대로 가장 처음에 만나는 값을 반환한다.
  def test2(): Int = try 1 finally 2
// 결론적으로 finally에는 값을 사용하지 말자.
```

### match

case의 대안. Java와 달리 타입에 상관없이 어떤 상수값이라도 사용 가능. 이 때, match 앞에 오는 비교대상의 타입을 기본적으로 다른다. (그렇게 되면 case로 비교하는 값들도 그 타입을 따른다.) '_'은 default에 대응하며 "완전히 알려지지 않은 값"을 의미.

```
object Client extends App {
  def matchTest(x: Any): Any = x match {
    case 1 => "one"
    case "two" => 2
    case y: Int => "scala.Int"
    case _ => 'unknown
  }

  println(matchTest(3)) // scala.Int
}
```

#### Functions

- 재귀 함수인 경우 반드시 Return Type 명시, 그외에는 생략 가능.
  - 함수 길이 긴 경우 사람이 읽기 쉽게 Return Type을 명시하는 게 좋음.
  - "return" 구문은 선택. 생략 가능.
- 본론 문장이 하나면 괄호 생략 가능
- Retury Type의 "Unit"은 Java의 void. 즉, Side-effect를 위해서만 실행하는 함수.
- parameter는 val이므로 값을 재할당할 수 없다. (예를 들어 Call-by-reference를 이용하려는 경우)
- 함수 안에 함수를 정의할 수 있다.
- 인자가 없는 메소드는 소괄호를 생략할 수 있다. 그러나 관례로서 Side-Effect가 있거나 함수 본래 작업 이외의 부가적인 작업이 존재할 경우 소괄호를 붙인다.

```
def sum(x: Int, y: Int): Int = {
  x + y
}

// 위와 같은 표현
def sum(x: Int, y: Int) = x + y
```

```
// Unit 함수. Return 값이 없음을 의미.
// Type 추론이 되므로 Return Type에 Unit 생략 가능.
def greeting(): Unit = println("Hello, world!")
```

### Local functions

함수의 중첩이 가능.

```
def test1(x: Int) = {
  def test1(y: Int) = x * y
 test1(10)
}

println(test1(5)) // 50
```

### Placeholder syntax

위치 표시자.

```
 /* 고차함수를 더 간략하게 작성하는 방법 */

  val values = 1 to 10

  values.filter((x: Int) => x > 5)

  // filter의 인자는 Int를 받도록 명시되어 있으므로 생략 가능
  values.filter(x => x > 5)

 // 위치표시자(_)를 이용하면 더 간략한 표현 가능.
  values.filter(_ > 5)

 // 아래와 같은 응용도 된다.
  values.foreach(println _) // values.foreach(x => println(x))
 // 그런데 foreach의 인자가 명확한 위치가 나타나기 때문에 생략 가능
  values.foreach(println)
  val func1 = (_:Int) + (_:Int)
  println(func1(1, 1)) // 2
```

### Partitially applied function

```
  def sum(a: Int, b: Int, c: Int) = a + b + c

  // sum 함수를 대입. 이 때, _의 의미는 sum의 모든 인자를 의미.
  val a = sum _
  println(a(1, 2, 3)) // = a.apply(1, 2, 3)

 // 아래와 같이 부분적인 인자 적용도 가능.
  val b = sum(1, _:Int, 3)
```

```
  println(b(2)) // = b.apply(2)
```

### Closure

```
  /* Java의 경우 final(immuitable)로 선언해야만 접근 가능한 것에 비해 var(mutable)도 사용 가능.*/
  /* Javascript와 동일.*/

 /* 변수 바인딩 */
  var sum = 0
  (1 to 10).foreach(sum += _) ;; 바깥 범위(scope)에 있는 sum의 값에 누적.

  println(sum) // 55

 /* 함수 바인딩 */
  def scope1(x: Int) = {
    /*def scope2(y: Int) = x + y
    scope2 _*/

 // 위와 같은 표현
    x + (_: Int)
  }

// 각 scope1에 바인딩된 바깥쪽 x를 다르게 설정
  val closure1 = scope1(1)
  val closure2 = scope1(2)

  println(closure1(10)) // 11
  println(closure2(10)) // 12
```

### Paramter & Argument

```
  /* repeated parameter. 가변 인자(variable paramter)와 동일 개념 */
  def test1 (args: String*) = args.foreach(println)
  test1("Hello", "world", "!")
  test1(Array("Hello", "world", "!"): _*)

  /* named argument. 순서 상관없이 인자의 이름으로 전달 */
  def test2 (v1: Int, v2: Int) = v1 + v2
  println(test2(v2 = 1, v1 = 9))

  /* default argument. 인자의 값이 없을 경우 default로 값을 미리 지정 */
  def test3 (v1: Int = 1, v2: Int) = v1 + v2
  println(test3(v2 = 1))
```

### Currying

함수를 1급 객체로 취급하기 때문에 가능. 다중 인수를 갖는 함수를 단일 인수를 갖는 함수들의 함수열로 바꾸는 것.

```
def curriedSum(x: Int)(y: Int) = x + y
  //def curriedSum(x: Int) = (y: Int) => x + y

  println(curriedSum(1)(2))
```

### By-name parameters

값, 함수 모두 "값"으로 평가하면서 인자를 넘기고 싶은 경우 사용.

이 역시 함수를 1급 객체로 취급하기 때문에 가능. paramter의 Lazy evalution. 성능상의 이점을 가진다.

```
  def test1(b: Boolean) = b
  test1(2 > 1)

  // by-name parameter. lazy evaluation.
  def test2_1(b: () => Boolean) = b
  test2_1(() => 2 > 1)

 // 개선
  def test2_2(b: => Boolean) = b
  test2_2(2 > 1)
```

#### Collections

- https://docs.scala-lang.org/overviews/collections-2.13/overview.html
- https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html

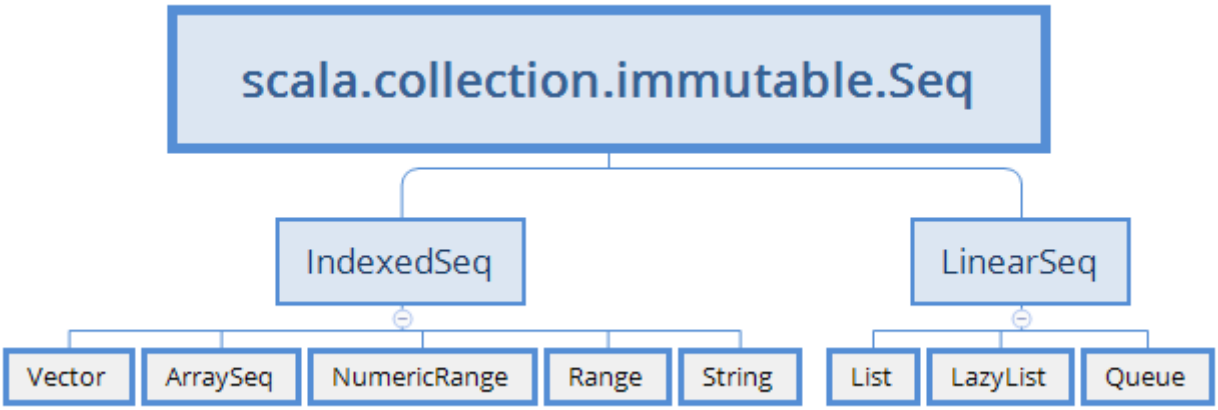
기본적으로 불변(Immutable) 상태를 가진다.

### Seq

선형 자료 구조.

- 아래 두 관점으로 적합한 자료구조를 선택.
  - "Indexing"이 필요한가?
  - "mutable state"가 필요한가?

### Immutable




- immutable.LazyList는 immutable.Stream을 대체. immutable.Stream 는 2.13부터 Deprecated.
- immutable.ArraySeq 는 2.13부터 추가되었는데, 왠만하면 성능면에서 사실상 상수 시간(Constant time)을 가지는 Vector를 사용하는게 이득.

## List

Singly Linked List. 앞부분에 추가/삭제 유리. 임의 접근은 불리.

- ::: : List를 이어 붙여 새로운 List 반환
- :: : Cons(콘즈) 라고 부르며, 새 원소를 기존 List 앞에 삽입한 새로운 List 반환

```
[새 원소] :: [기존 List]
```

- mutable 상태는 collecdtion.mutable.Buffer

```
val test1 = List(1, 2, 3)
val test1 = 1 :: 2 :: 3 :: Nil
```

```
val test1 = List(1, 2) ::: List(3, 4) ::: List(5, 6)
// List(1, 2, 3, 4, 5, 6)

val test2 = 1 :: List(2, 3)
// List(1, 2, 3)

val test3 = List(1, 2) :: List(2, 3)
// List(List(1, 2), 2, 3)
```

# List vs. Vector

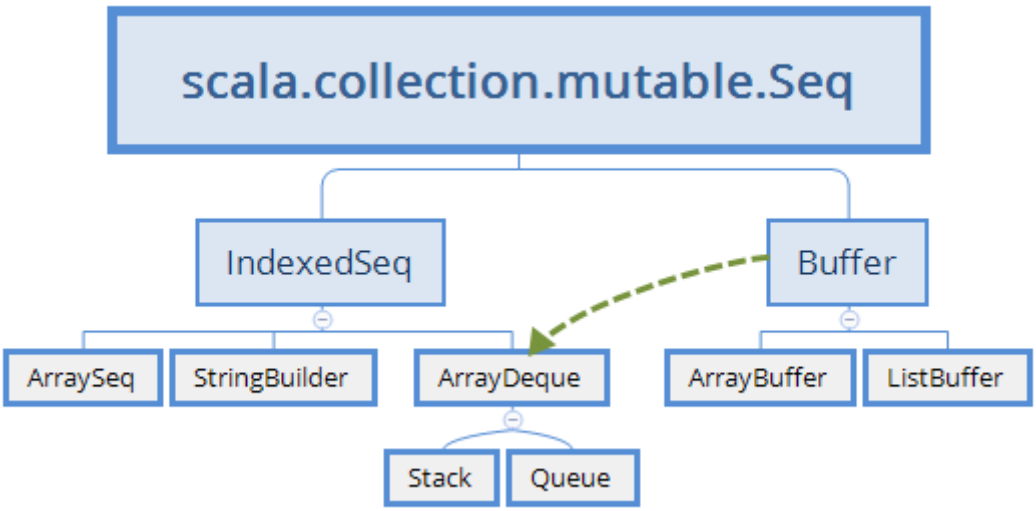
전체 Collections 성능 비교

| | head | tail | apply | update | prepend | append |
|---|---|---|---|---|---|---|
| List | C | C | L | L | C | L |
| Vector | eC | eC | eC | eC | eC | eC |

- C (Constant) : 연산에 상수 시간이 걸린다. (빠름)
- eC (Effectively Constant) : 연산에 사실상 상수 시간이 걸리는데, 이는 Vector의 최대 길이나 Hash key 분포와 같은 일부 가정에 따라 달라질 수 있다.
- L (Linear) : 연산에 선형 시간이 걸린다. 즉, Collections 크기에 비례하여 시간이 걸린다.

**결론적으로 head가 임의 위치 접근이 필요한 경우 Vector 선택.**

## Mutable




### Set

- 같은 이름의 mutable, immutable 패키지가 있으니 주의.

```
import scala.collection.mutable

val set1 = Set("a", "b")

// 원소 추가
// Collection이 val인 경우, mutable package의 import 필요!
/// set1.+=("b") 와 같은 표현
set1 += "b"
```

### Map

```
// immutable이기 때문에 원소를 추가할 수 없음. 이어붙이는 연산을 해야 함.

val map1 = Map("A" -> 1, "B" -> 2, "C" -> 3)
val map2 = Map("D" -> 4, "E" -> 5, "F" -> 6)
val map3 = map1 ++ map2

println(map3)

// 기본 구현체는 immutable.HashMap
// HashMap(E -> 5, F -> 6, A -> 1, B -> 2, C -> 3, D -> 4)
```

```
import scala.collection.mutable

val map1 = Map[String, Int]()

map1 += ("One" -> 1)
map1 += ("Two" -> 2)

map1("One")
```

```
val map1 = Map("One" -> 1, "Two", 2)
map1("One")
```

## TrieMap vs. ConcurrentHashMap

- scala.collection.concurrent.TrieMap
- java.util.concurrent.ConcurrentHashMap

둘 다 동시성을 보장하는 Thread-Safe Map.

- Reference
  - https://stackoverflow.com/questions/29499381/what-is-a-triemap-and-what-is-its-advantages-disadvantages-compared-to-a-hashmap
  - https://www.researchgate.net/publication/221643801_Concurrent_Tries_with_Efficient_Non-Blocking_Snapshots

### Tuple

엄격히 말하면 Collections에 포함되지 않음.

- Tuple22 까지, 즉 22개의 Argument까지 지원.

```
val tp = (123, "Hello", 1.23)
// val tp = Tuple3[Int, String, Double](123, "Hello", 1.23)

print(tp._1)
print(tp._2)
print(tp._3)
```

#### Class & Object

### 기초적으로 알아두어야 할 사항

- Procedure : Side-effect만을 위해서 실행되는 Method (≒ Return Type이 없는, Unit인 Method)
- static Member가 없다. 대신, Singleton Object를 제공한다.
- Java는 Field, Method, Type, Package의 4가지 Namespace를 갖지만 Scala는 Field, Method를 2가지 Namespace 갖는다.
  - Field가 Parameter없는 Method를 Override할 수 있다. (서로 동일시 한다.)
    - 이 같은 성질로 같은 Class에 같은 이름의 Field와 Method를 정의할 수 없다.
  - Package도 Field 및 Method처럼 Namespace를 공유하는 이유는 Singleton Object에서 Field와 Method를 import하기 위해서다.

### Access Modifier

| Modifier | Class | Companion | Subclass | Package |
|---|---|---|---|---|
| public (default) | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| private | Y | Y | N | N |

- Y : Field/Method 접근 가능
- N : Field/Method 접근 불가

### Class

- Java와 달리 Class 이름과 Class가 작성된 파일 이름이 같을 필요가 없으나 권장.

```
class Person {
  // member 변수와 Getter/Setter 서술
}
```

## Constructor

# Primary Constructor

- 주 생성자만이 Super Class의 생성자 호출 가능!

```
// 아래와 같이 class를 선언하면 member 변수들은 private val
// 이후 값을 변경할 수 없음.
class Person (name: String, age: Int) {
  def getName: String = name
  def getAge : Int = age
}
```

```
// 아래와 같이 class를 선언하면 member 변수들은 public val
/// val을 선언함으로써 Getter 생성. ".name", ".age"로 접근.
// 바로 멤버 변수에 접근할 수 있고, 이후 값을 변경할 수 없음.

// Class 뒤의 인자들을 "Class Parameter"라고 지칭한다.
// Scala Compiler는 내부적으로 Class Parameter를 기반으로 Primary Constructor(주 생성자)를 생성한다.
/// 기본적으로 Scala의 Constructor는 Class Parameter를 생성하지 않는다. val을 붙이면 생성한다.
class Person (val name: String, val age: Int)
```

```
// Scala Compiler는
// Class 내부에 있으면서
// Field나 Method 정의에 들어 있지 않은 코드를
// Primary Constructor에 삽입한다.

/// 객체 생성과 동시에 println 메시지 출력.
class Person (name: String, age: Int) {
  println("Created!" + name + " / " + age)
}
```

# Auxiliary Constructor

- 모든 보조 생성자는 **반.드.시.** 같은 클래스에 속한 다른 생성자를 호출하는 코드로 시작해야 한다. 결국 주 생성자를 호출하게 만드는 효과가 있다.
- 주 생성자만이 Super Class의 생성자 호출 가능!

```
class Person (val name: String, val age: Int) {
  def this(age: Int) = this("meteor", age);
```

```
  override def toString: String = name + " / " + age
}
```

## Checking Preconditions

- 유효성 검사 때문에 생성되는 try ~ catch, if ~ else 에서 벗어날 수 있다!
- 조건을 만족시키지 못하면 IllegalArgumentException 발생. 메시지가 없으면 “requirement failed”

```
class Person (name: String, age: Int) {
  require(name != null, "name is not nullable.")
  require(age > 0)
  override def toString: String = name + " / " + age
}
```

## Operator Defination

Operator Overloading.

```
class Person (val name: String, val age: Int) {
  def +(p: Person) = this.toString + "\n" + p.toString

  override def toString: String = name + " / " + age
}
```

```
object Client extends App {
  val p1 = new Person("hyuk", 30)
  val p2 = new Person("meteor", 26)
  println(p1 + p2)
}
```

## Method Overloading

Java와 동일하나 Scala는 Type을 유추하기 때문에 적합한 Method의 Parameter Type 일치 결과가 없다면 'ambiguous reference' 오류를 출력. 그리고 Overloading보다 DEFAULT PARAMETER VALUES를 이용을 권장. Method 수를 줄일 수 있다.

## 'apply' Method

'Default Method' 또는 'Injector Method'라고도 불린다. 이는 Method 이름없이 **괄호**를 사용하여 호출하는 기능을 갖고 있다.

```
val l = List(1, 2, 3)

// 같은 의미
println(s"${l.apply(1)} ${l(1)}")
```

위 예제의 'List.apply(index)' 처럼 상식적으로 이해할 수 있는, 자연스러운 연산에 적용되어야 한다.

### object

- class 대신 **object**로 시작. Java의 static class에 대한 대안.
  - Instance를 생성할 수 없다. 그러므로 당연히 Constructor도 선언 불가.
- "Singleton"으로 작동하므로 첫 접근이 되지 않는 한 인스턴스가 만들어지지 않는다.
- 대개 특정 객체에 상관없는 Side-effect Method들(대개 Service 목적의 Util 성질을 가진 공통 사용 Method)을 담아두고 사용하는 Method 사용.

```
// Companion class
class Person (val name: String, val age: Int)

// Companion object
object Person {
  def print(person: Person) :Unit = println(person.name + " / " +
person.age)
}
```

```
object Client extends App {
  val person: Person = new Person("Amy", 11)
  Person.print(person)
}

/* Amy / 11 */
```

## Companion object

어떤 class **이름**이 같은 object. 그 피대상인 class는 "Companion class"라고 한다. Companion object와 'apply' Method를 이용하여 Companion class의 **"Factory Pattern"**을 적용할 때 사용. 이 때, object의 field들이 "private"으로 선언되어 있더라도 접근 가능.

Companion class와 Companion object는 **반드시 같은 소스 파일**에 있어야 한다!

```
object DBConnection {
  private val url = "jdbc://localhost"
  private val user = "franken"
  private val password = "berry"
```

```
  def apply() = new DBConnection()
}

class DBConnection {
  private val props = Map(
    "url" -> DBConnection.url,
    "user" -> DBConnection.user,
    "password" -> DBConnection.password
  )
}

// val connection = DBConnection()
```

### case class

- 자동으로 유용한 Method들 생성. (Kotlin의 'data class'와 동일한 개념)
  - Data를 관리하는데 유용. 즉, DTO로 유용.
- 자동으로 Companion Object 생성. “new” 필요하지 않음.
- 기본적으로 생성자의 Parameter들은 'val'.
- Compile하면 '**[소스이름].class(class)**', '**[소스이름]$.class(object)**'이 생성된다.
- 자동으로 생성되는 Methods

| 이름 | 위치 | 설명 |
|---|---|---|
| apply | object | case class를 Instance로 만드는 Factory method |
| unapply | object | Instance의 field들을 Tuple로 추출하여 **Pattern Matching**에 case class instance를 사용할 수 있도록 함 |
| copy | class | **Deep copy!** |
| equals | class | Reference가 아닌 구조적 동등성(모든 필드가 일치)으로 비교. '=='로도 호출 할 수 있다. |
| hashCode | class | Instance의 field들의 Hash code 반환. hash 기반 collections에서 동등성에 이용. |
| toString | class | Field들을 String으로 전환 |

```
object Practice extends App {
  val person = Person("Luke", 33)
  val copy = person.copy()
  // Equals?
  println(person == copy)

  person.name = "James"
  person.age = 11
  // Deep copy?
  println(copy)

 // Pattern Matching
  val p = person match {
    case Person(_, 11) => "Original!"
    case Person(_, 33) => "Copy!"
```

```
  }
  println(p)
}

case class Person(var name: String, var age: Int)
```

- Reference
  - https://docs.scala-lang.org/overviews/scala-book/case-classes.html
  - https://www.oreilly.com/library/view/scala-cookbook/9781449340292/ch04s15.html

#### Composition and Inheritance

## Override

Java의 경우 1.5 기준으로 @Override를 명시할 수 있으나 필수사항이 아닌데 반면, Scala는 강제한다. 이로 인해 "우연한 Override"로 "fragile base class"가 생성되는 문제를 줄여준다.

```
class Person (val name: String, val age: Int) {
  override def toString: String = name + " / " + age
}
```

## Abstract Class

Java와 비교하여

- method에 abstract를 붙이면 안된다.
  - abstract는 class 혹은 abstract class를 mix-in한 trait의 method만 붙일 수 있다.
  - 본체가 없으면 abstract method로 인식한다. (이는 trait도 동일)
- 본체가 있는 concrete method를 구현할 수 있다.
- **Field가 파라미터 없는 메소드를 Override 할 수 있다.**

```
abstract class Abstract {
  /* abstract가 필요하지 않다. */
  def values: Array[Int]
  def sum = values.sum
}

class Detail(conts: Array[Int]) extends Abstract {
 /* 아래 동작은 같다. */
  //override def values: Array[Integer] = conts
  val values: Array[Int] = conts
}
```

## Parameter field 정의

같은 이름의 생성자 Paramter와 Field를 동시 정의하는 단축 표기. 특정 Field에 할당하는 이름 중복을 피하기 위한 Boilerplace 제거.

```
abstract class Abstract {
  def values: Array[Int]
}

// 아래와 같이 같은 목적이지만 중복된 이름을 피하기 위해 불필요한 부분이 작성된다.
// 할당과 동시에 바로 getter 생성
class Detail(vals: Array[Int]) extends Abstract {
  override def values: Array[Int] = vals
}

// 이를 아래와 같이 작성할 수 있음.
class Detail2(var values: Array[Int]) extends Abstract
```

## Super class의 Constuctor 호출

Java의 경우 상속을 받으면 반드시 부모 Class의 생성자와 같은 형태의 생성자를 호출하고 super()를 하는 과정을 아래와 같이 간략화할 수 있다.

```
abstract class Abstract {
  def values: Array[Int]
  def sum = values.sum
}

class Detail(val values: Array[Int]) extends Abstract
/*
class Detail(vals: Array[Int]) extends Abstract {
  override def values: Array[Int] = vals
}
*/

/* Super class(Detail)의 생성자 호출.*/
class MoreDetail(x: Int) extends Detail(Array(x)) {
}
```

#### Trait

Java의 Interface의 역할을 하는 한 단위.

- 두 가지를 제외하고 Class가 할 수 있는 일을 다 할 수 있다. (Field 정의, 본체가 있는 concrete

method 구현등등)
    - 할 수 없는 두 가지 : 생성자 Parameter 구현, super를 이용한 Method 정적 바인딩
- Mix-in 대상은 Trait뿐만이 아니라 Class, Abstract Class 모두 다 된다.
- 첫 번째 상속/구성할 대상이 Class든 Trait든 무조건 extends를 사용한다. 단, Class와 Trait를 혼합사용(Mix-in)할 경우 Class를 먼저 기술한다. 이후 Trait를 with를 이용하여 Mix-in 한다.
    - 즉, Java의 Interface와 달리 Class도 상속이 가능하다! 이는 해당 부모 Class의 Method를 Override가 필요한 상황에서 사용한다.

```
object Client extends App {
  val x:D = new D
  x.func
 // "안녕, 세상아!"
}

trait T1 {
  def func(): Unit
}

trait T2 {
  private val greeting = "Hello, world!"
  def func() = println(greeting)
}

class C1 {
}

class D extends C1 with T1 with T2 {
  override def func(): Unit = println("안녕, 세상아!")
}
```

**Stackable Modifications**

```
object Client extends App {
  val queue = new BasicIntQueue with Doubling
  queue.put(10)

  println(queue.get())
  // 20
}

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
```

```
  private val buf = new ArrayBuffer[Int]
  override def get(): Int = buf.remove(0)
  override def put(x: Int) = buf += x
}

trait Doubling extends IntQueue {
  // trait에서 super로 접근할 때는 override abstract가 필요.
  override abstract def put(x: Int): Unit = super.put(x * 2)
}
```

#### 여러 Trait를 mix-in 했을 때 우선 순위

가장 오른쪽에 있는 Trait의 효과부터 적용된다.

```
trait Base {
  override def toString: String = "Base"
}


trait A extends Base {
  override def toString: String = s"A->${super.toString}"
}

trait B extends Base {
  override def toString: String = s"B->${super.toString}"
}

trait C extends Base {
  override def toString: String = s"C->${super.toString}"
}


class MixIn extends A with B with C {
  override def toString: String = s"MixIn->${super.toString}"
}

// println(new MixIn())
/// MixIn->C->B->A->Base
```

#### Type parameterization

#### Bounded Type

Type Parameter를 특정 Class나 거의 Sub type 또는 Base type으로 제한하는 방법.

### Upper bound

Type을 Base type 또는 Sub type 중 하나로 제한한다.

https://docs.scala-lang.org/tour/upper-type-bounds.html

```
<identifier> <: <upper bound type>
```

```
object Practice extends App {
  def check[A <: Parent](u: A): Unit = {
    println(s"${u.name}")
  }

  // Runtime Error
  // inferred type arguments [GrandParent] do not conform to method check's
type parameter bounds [A <: Parent]
  check(new GrandParent("Jake"))

  check(new Parent("Fred"))
  check(new Child("Mike"))
}

class GrandParent(val name: String)
class Parent(name: String) extends GrandParent(name)
class Child(name: String) extends Parent(name)
```

### Lower bound

Type을 해당 Type으로 제한하거나 또는 해당 Type이 확장되는 Base type 중 하나로 제한한다.

```
<identifier> >: <lower bound type>
```

#### Type variance

Type parameter를 **덜 제한적**으로 만드는 방법. 타입 가변성(Type variance)은 Type parameter가 Base type이나 Sub type을 충족하도록 적응하는 방법을 지정한다.

| Scala | Java | 설명 | 예시 |
|---|---|---|---|
| T | T | Invariance (무공변성) | **X[T]**는 **X[T]**. X[$T_{sub}$] 혹은 X[$T^{super}$]가 X[T]를 대체할 수 없음. |
| +T | ? extends T | Covariance (공변성) | **X[$T_{sub}$]**는 **X[T]**의 Sub type |
| -T | ? super T | Contravariance (반공변성) | **X[$T^{super}$]**는 **X[T]**의 Sub type |

- Effective Java에서 나오는 PECS (Producer extends, Consumer super) 원리와 같은 개념.
  - Producer는 getter, Consumer는 setter
  - Scala의 Function1 혹은 Java의 Function을 보면 쉽게 유추 가능.
    - 첫번째 인자 **-T1**는 Producer, 두번째 인자 **+R**는 반환값으로 Consumer.
      - 반환해야 하는(생산하는) 값이 Super type인데, 인자로 받은(제공하는) 값이 Sub type이면 Super type의 Method를 호출할 수 없음.
      - 이런 이유로 T1은 동등한 레벨의 타입이거나 상위 타입(Super type)이어야 한다.

#### Invariance

무공변성.

Type parameter의 기본값. Type-parameterized class의 Instance는 오직 같은 Parameterized type class만 호환된다.

```
class Item[A]

class Person
class Employee extends Person
class HomeMinusEmployee extends Employee


object Practice extends App {
  // Okay!
  val p: Person = new Employee()
  val i1: Item[Person] = new Item[Person]()

  // Error!
  val i2: Item[Person] = new Item[Employee]()
}
```

#### Covariance

공변성.

Type parameter가 부모 Type으로 변할 수 있는 성질. (Sub parameterized-type → Base parameterized-type) Type parameter 앞에 **+**를 붙인다.

```
class Item[+A]

class Person
class Employee extends Person
class HomeMinusEmployee extends Employee

object Practice extends App {
```

```
  // Okay!
  val p: Person = new Employee()
  val i1: Item[Person] = new Item[Person]()

  // Okay!
  val i2: Item[Person] = new Item[Employee]()
}
```

### Contravariance

반공변성.

#### Abstract types

```
import java.io._
import scala.io.Source

//
https://github.com/deanwampler/programming-scala-book-code-examples/blob/mas
ter/src/main/scala/progscala3/typelessdomore/AbstractTypes.scala
// 위 소스를 응용한 예제
abstract class BulkReader {
  type In        // 구체적인 타입을 지정하지 않음.
  val source: In // 바로 위에서 지정한 Type인 "In"을 사용
  def read: String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  override def read: String = source
}

class FileBulkReader(val source: File) extends BulkReader {
  type In = File
  override def read: String = {
    val source1 = Source.fromFile(source)
    try source1.mkString finally source1.close()
  }
}
```

#### Parameterized types vs. Abstract types

어떤 상황에서 무엇이 적절한가?

- Parameterized types : Type parameter가 Parameterized types와 관련이 없는 경우

    - List[A]에서 A는 String, Int 무엇이든 상관 없음.

- Abstract types : 타입 멤버가 객체의 **동작**과 일치하는 경우
    - 위 예시를 보면 "읽는다"라는 동작이 일치한다.

#### Implicit Parameters and Conversions

##### Implicit Parameters

Currying 혹은 Lazy 된 함수의 인자를 명시적으로 지정하지 않아도 자신의 네임스페이스의 기본값을 사용하는 방법. 변수나 함수 매개변수에 "implicit"를 앞에 붙인다.

"Dependency Injection"의 시각으로 볼 수 있다.

```
object Printer {
  def print(num: Double)(implicit format: String) =
println(format.format(num))
}

// fmt가 주입된다.
object Practice extends App {
  implicit val fmt = "%.7f"
  Printer.print(1.11)
}
```

##### Implicit Class

암시적 Type 변환. 어느 Instance의 Type이 특정 Instance의 Type인 것처럼 자동 전환하여 그 특정 Instance의 Method를 가진 것처럼 보이게 할 수 있다.

C#, Kotlin의 Extension Methods과 비슷한 기능.

서로를 고려하지 않고 독립적으로 개발된 소프트웨어/라이브러리를 하나로 묶으려고 할 때 사용.

- 생성 규칙
    - object/class/trait 안에 정의되어야 한다.
    - Implicit를 선언하지 않는 인자를 받아야 한다.
    - 같은 namespace에서 object/class/trait 이름이 같아서는 안된다.
        - case Class는 사용할 수 없다. (자동으로 생성되는 Companion Object 때문)

```
object Client extends App {
  val uc = new UnaryCalculator(2)
  val op1 = uc ** 3
  //val op1 = uc.**(3)
```

```
  val op2 = 3 ** uc // Error!
//val op2 = {3}.**(uc), 정수 3은 ** Method를 가지고 있지 않음.
  println(op1 + " / " + op2)
}

class UnaryCalculator (val v: Int) {
  def ** (`pow`: Int) = math.pow(v, `pow`)
}
```

```
object Client extends App {
  implicit def intToUnaryCalculator (initVal: Int) = new
UnaryCalculator(initVal)

  val uc = new UnaryCalculator(2)
  val op1 = uc ** 3
  //val op = uc.**(3)

  val op2 = 3 ** uc
  //val op = {3}.**(uc)

  println(op1 + " / " + op2) // 8.0 / 9.0
}

class UnaryCalculator (val v: Int) {
  def ** (`pow`: UnaryCalculator) = math.pow(v, `pow`.v)
}
```

From:
https://ledyx.xyz/ - **Ledyx WIKI**

Permanent link:
**https://ledyx.xyz/doku.php?id=language:scala&rev=1599070152**

Last update: **2026/10/24 07:03**