Functional programming with Scala

Functions

Djaouida ZAOUCHE, Romain DUJOL

CY TECH – ING2

2020 – 2021

Declaration

Declaration

Main statement

Declaration : `def`

Syntax

```
def functionName(typedParameter*)[:returnType] =
  {
    functionBody
  }
```

- *functionName* must start with a **lowercase** letter.
- **Type is required for each input parameter.**
- Return type may be infered so it may not be required.
- The body is a block so its type is the type of last statement.

Declaration

Examples

```
scala> def sum2(a : Int, b : Int) : Int = a + b
sum2: (a: Int, b: Int)Int
```

Declaration

Examples

```
scala> def sum2(a : Int, b : Int) : Int = a + b
sum2: (a: Int, b: Int)Int

scala> def sum2(a : Int, b : Int) = a + b
sum2: (a: Int, b: Int)Int
```

Declaration

Examples

```
scala> def sum2(a : Int, b : Int) : Int = a + b
sum2: (a: Int, b: Int)Int

scala> def sum2(a : Int, b : Int) = a + b
sum2: (a: Int, b: Int)Int

scala> def sayHello(name : String) = println(s"Hello $name !")
sayHello: (name: String)Unit

scala> sayHello("world")
Hello world !
```

Declaration

Examples

```
scala> def sum2(a : Int, b : Int) : Int = a + b
sum2: (a: Int, b: Int)Int

scala> def sum2(a : Int, b : Int) = a + b
sum2: (a: Int, b: Int)Int

scala> def sayHello(name : String) = println(s"Hello $name !")
sayHello: (name: String)Unit

scala> sayHello("world")
Hello world !
```

String interpolator `s"..."`

Replaces any `$`-prefixed expression with its `toString` evaluation

```
scala> println(s"1 + 1 is ${1 + 1}.")
1 + 1 is 2.
```

Declaration

Parameters

Default value for parameter

Example : sayHello

```
scala> def sayHello(name : String = "world") = println(s"Hello $name !")
sayHello: (name: String)Unit
```

Default value for parameter

Example : sayHello

```
scala> def sayHello(name : String = "world") = println(s"Hello $name !")
sayHello: (name: String)Unit

scala> sayHello()
Hello world !
```

Default value for parameter

Example : sayHello

```
scala> def sayHello(name : String = "world") = println(s"Hello $name !")
sayHello: (name: String)Unit

scala> sayHello()
Hello world !

scala> sayHello("everybody")
Hello everybody !
```

Default value for parameter(s) ?

Default value for more than one parameter is NOT recommended.

Example

```
scala>  def greet(fr : Boolean = false, name : String = "world") =
          {
            val greeter = if (fr) "Bonjour" else "Hello"
            println(s"$greeter $name !")
          }
greet: (fr: Boolean, name: String)Unit
```

Default value for parameter(s) ?

Default value for more than one parameter is NOT recommended.

Example

```
scala>  def greet(fr : Boolean = false, name : String = "world") =
          {
            val greeter = if (fr) "Bonjour" else "Hello"
            println(s"$greeter $name !")
          }
greet: (fr: Boolean, name: String)Unit

scala> greet()
Hello world !
```

Default value for parameter(s) ?

Default value for more than one parameter is NOT recommended.

Example

```
scala>  def greet(fr : Boolean = false, name : String = "world") =
          {
            val greeter = if (fr) "Bonjour" else "Hello"
            println(s"$greeter $name !")
          }
greet: (fr: Boolean, name: String)Unit

scala> greet()
Hello world !

scala> greet("everybody")
<console>:13: error: type mismatch;
 found   : String("everybody")
 required: Boolean
       greet("everybody")
             ^
```

Parameter evaluation : *by value* vs *by name*

Evaluation **by value** (default behaviour)

- Parameter is always evaluated at function call.
- Parameter is evaluated once.

Parameter evaluation : *by value* vs *by name*

Evaluation **by value** (default behaviour)

- Parameter is always evaluated at function call.
- Parameter is evaluated once.

Evaluation **by name** (add **=>** before type)

- Parameter is evaluated at each actual use.
- Parameter may be evaluated more than once.

Parameter evaluation : *by value* vs *by name*

Evaluation **by value** (default behaviour)

- Parameter is always evaluated at function call.
- Parameter is evaluated once.

Evaluation **by name** (add **=>** before type)

- Parameter is evaluated at each actual use.
- Parameter may be evaluated more than once.

Why should we consider an evaluation by name ?

Particularly useful for lengthy or throwable evaluations

Parameter evaluation : *by value* vs *by name*

Example (throwable evaluation)

```
scala> 1/0 == 0
java.lang.ArithmeticException: / by zero
  ... 32 elided
```

Parameter evaluation : *by value* vs *by name*

Example (throwable evaluation)

```
scala> 1/0 == 0
java.lang.ArithmeticException: / by zero
  ... 32 elided

scala> def and2(x : Boolean, y : Boolean) = if (x) y else false
and2: (x: Boolean, y: Boolean)Boolean

scala> and2(false, 1/0 == 0)
java.lang.ArithmeticException: / by zero
  ... 32 elided
```

Parameter evaluation : *by value* vs *by name*

Example (throwable evaluation)

```
scala> 1/0 == 0
java.lang.ArithmeticException: / by zero
  ... 32 elided

scala> def and2(x : Boolean, y : Boolean) = if (x) y else false
and2: (x: Boolean, y: Boolean)Boolean

scala> and2(false, 1/0 == 0)
java.lang.ArithmeticException: / by zero
  ... 32 elided

scala> def and2(x : Boolean, y : => Boolean) = if (x) y else false
and2: (x: Boolean, y: => Boolean)Boolean

scala> and2(false, 1/0 == 0)
res3: Boolean = false
```

Parameter evaluation : *by value* vs *by name*

Example (throwable evaluation)

```
scala> 1/0 == 0
java.lang.ArithmeticException: / by zero
  ... 32 elided

scala> def and2(x : Boolean, y : Boolean) = if (x) y else false
and2: (x: Boolean, y: Boolean)Boolean

scala> and2(false, 1/0 == 0)
java.lang.ArithmeticException: / by zero
  ... 32 elided

scala> def and2(x : Boolean, y : => Boolean) = if (x) y else false
and2: (x: Boolean, y: => Boolean)Boolean

scala> and2(false, 1/0 == 0)
res3: Boolean = false
```

Logical operators are designed that way

```
scala> false && (1/0 == 0)
res4: Boolean = false
```

Parameter evaluation : *by value* vs *by name*

Example (lengthy evaluation)

```
scala> def slow(x : Boolean) = { Thread.sleep(9000) ; x }
slow: (x: Boolean)Boolean
```

Parameter evaluation : *by value* vs *by name*

Example (lengthy evaluation)

```
scala> def slow(x : Boolean) = { Thread.sleep(9000) ; x }
slow: (x: Boolean)Boolean

scala> def and2(x : Boolean, y : Boolean) = if (x) y else false
and2: (x: Boolean, y: Boolean)Boolean

scala> and2(false, slow(true))
res0: Boolean = false (At least nine seconds later)
```

Parameter evaluation : *by value* vs *by name*

Example (lengthy evaluation)

```
scala> def slow(x : Boolean) = { Thread.sleep(9000) ; x }
slow: (x: Boolean)Boolean

scala> def and2(x : Boolean, y : Boolean) = if (x) y else false
and2: (x: Boolean, y: Boolean)Boolean

scala> and2(false, slow(true))
res0: Boolean = false (At least nine seconds later)

scala> def and2(x : Boolean, y : => Boolean) = if (x) y else false
and2: (x: Boolean, y: => Boolean)Boolean

scala> and2(false, slow(true))
res1: Boolean = false (Immediately)
```

Parameter evaluation : *by value* vs *by name*

Example (lengthy evaluation)

```
scala> def slow(x : Boolean) = { Thread.sleep(9000) ; x }
slow: (x: Boolean)Boolean

scala> def and2(x : Boolean, y : Boolean) = if (x) y else false
and2: (x: Boolean, y: Boolean)Boolean

scala> and2(false, slow(true))
res0: Boolean = false (At least nine seconds later)

scala> def and2(x : Boolean, y : => Boolean) = if (x) y else false
and2: (x: Boolean, y: => Boolean)Boolean

scala> and2(false, slow(true))
res1: Boolean = false (Immediately)
```

Logical operators are designed that way

```
scala> false && slow(true)
res2: Boolean = false (Immediately)
```

Other features

`require` (for preconditions, not functional-compliant)

- Syntax : `require(`*condition*`, `*message*`)`
- If *condition* is not met, throws **`IllegalArgumentException`** with *message* as message.

Other features

`require` (for preconditions, not functional-compliant)

- Syntax : `require(`*condition*`, `*message*`)`
- If *condition* is not met, throws **`IllegalArgumentException`** with *message* as message.

`@tailrec` annotation for *functions*

- Needs import : **`import scala.annotation.tailrec`**
- Indicates that annoted function is tail-recursive
- Allows compiler to optimize execution

First-class citizen

Every function is a “first-class citizen”

One of keystones of functional programming paradigm

What does that mean ?

A function has the same rank (“citizenship”) as any other object, so it can be used as :

- a value ;
- a variable ;
- a parameter in another function ;
- a result of another function ;
- . . .

First-class citizen

Function as a value

Function as a value

Examples

```
scala> val nextInt = (x : Int) => x + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res0: Int = 3
```

Function as a value

Examples

```
scala> val nextInt = (x : Int) => x + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res0: Int = 3

scala> val nextInt : Int => Int = x => x + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res1: Int = 3
```

Function as a value

Examples

```
scala> val nextInt = (x : Int) => x + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res0: Int = 3

scala> val nextInt : Int => Int = x => x + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res1: Int = 3

scala> val nextInt : Int => Int = _ + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res2: Int = 3
```

Function as a value

Recursive **val**-declared function needs **explicit** return type

```
scala> val fact : Int => Int =
           x => if (x <= 0) 1 else x * fact(x - 1)
fact: Int => Int = <function1>
```

Function as a value

Recursive **val**-declared function needs **explicit** return type

```
scala> val fact : Int => Int =
           x => if (x <= 0) 1 else x * fact(x - 1)
fact: Int => Int = <function1>

scala> fact(5)
res0: Int = 120
```

Function as a value

Recursive **val**-declared function needs **explicit** return type

```
scala> val fact : Int => Int =
           x => if (x <= 0) 1 else x * fact(x - 1)
fact: Int => Int = <function1>

scala> fact(5)
res0: Int = 120
```

Wildcard _ is a placeholder for each new parameter

$\Longrightarrow$ **As much parameters as _ wildcards**

```
scala> val add2 : (Int, Int) => Int = _ + _
add2: (Int, Int) => Int = <function2>
```

Function as a value

Recursive **val**-declared function needs **explicit** return type

```
scala> val fact : Int => Int =
           x => if (x <= 0) 1 else x * fact(x - 1)
fact: Int => Int = <function1>

scala> fact(5)
res0: Int = 120
```

Wildcard _ is a placeholder for each new parameter

$\Longrightarrow$ **As much parameters as _ wildcards**

```
scala> val add2 : (Int, Int) => Int = _ + _
add2: (Int, Int) => Int = <function2>

scala> add2(2, 3)
res0: Int = 5
```

How does it work ?

Let's get ahead a litte bit

Any **`val`**-declared function is an object with an `apply` method with same signature so that

$$function(parameters) \equiv function.\texttt{apply}(parameters)$$

How does it work ?

Let's get ahead a litte bit

Any **`val`**-declared function is an object with an `apply` method with same signature so that

function`(`*parameters*`)` ≡ *function*`.apply(`*parameters*`)`

```
scala> val nextInt : Int => Int = x => x + 1
nextInt: Int => Int = <function1>

scala> nextInt(2)
res0: Int = 3

scala> nextInt.apply(2)
res1: Int = 3
```

Differences with `def`-declared... methods

Example

```
scala> def nextInt(x : Int) = x + 1
nextInt: (x: Int)Int
```

Differences with **def**-declared... methods

Example

```
scala> def nextInt(x : Int) = x + 1
nextInt: (x: Int)Int

scala> nextInt.apply(2)
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^
```

Differences with **def**-declared... methods

Example

```
scala> def nextInt(x : Int) = x + 1
nextInt: (x: Int)Int

scala> nextInt.apply(2)
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^


scala> nextInt
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^
```

Differences with **def**-declared... methods

Example

```
scala> def nextInt(x : Int) = x + 1
nextInt: (x: Int)Int

scala> nextInt.apply(2)
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^

scala> nextInt
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^
```

Conversion to a value : f (_)

```
scala> val nextInt_val = nextInt(_)
nextInt_val : Int => Int = <function1>
```

Differences with **def**-declared... methods

Example

```
scala> def nextInt(x : Int) = x + 1
nextInt: (x: Int)Int

scala> nextInt.apply(2)
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^

scala> nextInt
<console>:13: error: missing argument list for method nextInt
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `nextInt _` or `nextInt(_)` instead of `nextI
       nextInt
       ^
```

Conversion to a value : f (_)

```
scala> val nextInt_val = nextInt(_)
nextInt_val : Int => Int = <function1>

scala> nextInt_val.apply(2)
res0: Int = 3
```

Nesting functions

Nesting functions are allowed

- One can declare a function with **`def`** or **`val`** with another function.
- Scope of nested function is limited to defining function.

Nesting functions

Nesting functions are allowed

- One can declare a function with **def** or **val** with another function.
- Scope of nested function is limited to defining function.

Classic example : tail-recursion encapsulation

```
scala> def factorial(n : Int)
         {
           def factorialTR(n : Int, acc : Int) =
             if (n <= 0) acc else factorialTR(n - 1, acc * n)
           factorialTR(n, 1)
         }
factorial: (n: Int)Int
```

Nesting functions

Nesting functions are allowed

- One can declare a function with **def** or **val** with another function.
- Scope of nested function is limited to defining function.

Classic example : tail-recursion encapsulation

```
scala> def factorial(n : Int)
         {
           def factorialTR(n : Int, acc : Int) =
             if (n <= 0) acc else factorialTR(n - 1, acc * n)
           factorialTR(n, 1)
         }
factorial: (n: Int)Int

scala> factorialTR
<console>:13: error: not found: value factorialTR
       factorialTR
       ^
```

First-class citizen

Function as a parameter/result

Function as a parameter

Example

```
scala> def applyTwice(f : Int => Int, x : Int) = f(f(x))
applyTwice: (Int => Int)Int
```

Function as a parameter

Example

```
scala> def applyTwice(f : Int => Int, x : Int) = f(f(x))
applyTwice: (Int => Int)Int
```

Passing anonymous function value

```
scala> applyTwice(_ + 1, 2)
res0: Int = 4
```

Function as a parameter

Example

```
scala> def applyTwice(f : Int => Int, x : Int) = f(f(x))
applyTwice: (Int => Int)Int
```

Passing anonymous function value

```
scala> applyTwice(_ + 1, 2)
res0: Int = 4
```

*Passing **val**-declared function*

```
scala> val nextInt = (x : Int) => x + 1
nextInt: Int => Int = <function1>

scala> applyTwice(nextInt, 2)
res1: Int = 4
```

Function as a parameter

Example

```
scala> def applyTwice(f : Int => Int, x : Int) = f(f(x))
applyTwice: (Int => Int)Int
```

Passing anonymous function value

```
scala> applyTwice(_ + 1, 2)
res0: Int = 4
```

*Passing **val**-declared function*

```
scala> val nextInt = (x : Int) => x + 1
nextInt: Int => Int = <function1>

scala> applyTwice(nextInt, 2)
res1: Int = 4
```

*Passing **def**-declared function*

```
scala> def nextInt(x : Int) = x + 1
nextInt: (x: Int)Int

scala> applyTwice(nextInt, 2)
res2: Int = 4
```

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>
```

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>

scala> plusTwo(2)
res0: Int = 4
```

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>

scala> plusTwo(2)
res0: Int = 4

scala> twice(_ + 1)(2)
res1: Int = 4
```

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>

scala> plusTwo(2)
res0: Int = 4

scala> twice(_ + 1)(2)
res1: Int = 4
```

Full **`def`**-declared version

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>

scala> plusTwo(2)
res0: Int = 4

scala> twice(_ + 1)(2)
res1: Int = 4
```

Full **def**-declared version

```
scala> def twice(f : Int => Int)(x : Int) = f(f(x))
twice: (f: Int => Int)(x: Int)Int
```

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>

scala> plusTwo(2)
res0: Int = 4

scala> twice(_ + 1)(2)
res1: Int = 4
```

Full **def**-declared version

```
scala> def twice(f : Int => Int)(x : Int) = f(f(x))
twice: (f: Int => Int)(x: Int)Int

scala> val plusTwo = twice(_ + 1)(_)
plusTwo: Int => Int = <function1>
```

Function as a result and curryfication

Example

```
scala> def twice(f : Int => Int) : Int => Int = x => f(f(x))
twice: (f: Int => Int)Int => Int

scala> val plusTwo = twice(_ + 1)
plusTwo: Int => Int = <function1>

scala> plusTwo(2)
res0: Int = 4

scala> twice(_ + 1)(2)
res1: Int = 4
```

Full **def**-declared version

```
scala> def twice(f : Int => Int)(x : Int) = f(f(x))
twice: (f: Int => Int)(x: Int)Int

scala> val plusTwo = twice(_ + 1)(_)
plusTwo: Int => Int = <function1>

scala> twice(_ + 1)(2)
res2: Int = 4
```

Partially applied functions

One can build a function from another function with some (but not all) parameters fixed.

Partially applied functions

One can build a function from another function with some (but not all) parameters fixed.

Example

```
scala> val transvect2 : (Int, Int, Int) => Int = _ + _ * _
transvect2: (Int, Int, Int) => Int = <function3>
```

Partially applied functions

One can build a function from another function with some (but not all) parameters fixed.

Example

```
scala> val transvect2 : (Int, Int, Int) => Int = _ + _ * _
transvect2: (Int, Int, Int) => Int = <function3>

scala> val add2 = (transvect2(_ : Int, 1, _ : Int)
add2: (Int, Int) => Int = <function2>
```

Partially applied functions

One can build a function from another function with some (but not all) parameters fixed.

Example

```
scala> val transvect2 : (Int, Int, Int) => Int = _ + _ * _
transvect2: (Int, Int, Int) => Int = <function3>

scala> val add2 = (transvect2(_ : Int, 1, _ : Int)
add2: (Int, Int) => Int = <function2>

scala> val nextInt = add2(_ : Int, 1) or transvect2(_ : Int, 1, 1)
nextInt: Int => Int = <function1>
```

Partially applied functions

Partially applied functions

One can build a function from another function with some (but not all) parameters fixed.

Example

```
scala> val transvect2 : (Int, Int, Int) => Int = _ + _ * _
transvect2: (Int, Int, Int) => Int = <function3>

scala> val add2 = (transvect2(_ : Int, 1, _ : Int)
add2: (Int, Int) => Int = <function2>

scala> val nextInt = add2(_ : Int, 1) or transvect2(_ : Int, 1, 1)
nextInt: Int => Int = <function1>

scala> nextInt(2)
res0: Int = 3
```

Generic/polymorphic. . . method actually

Example

- When used, parameter type may be infered.

```
scala> def isNone[T](o : Option[T]) = (o == None)
isNone: [T](o: Option[T])Boolean
```

Generic/polymorphic. . . method actually

Example

- When used, parameter type may be infered.

```
scala> def isNone[T](o : Option[T]) = (o == None)
isNone: [T](o: Option[T])Boolean

scala> isNone(None)
res0: Boolean = true
```

Generic/polymorphic. . . method actually

Example

- When used, parameter type may be infered.

```
scala> def isNone[T](o : Option[T]) = (o == None)
isNone: [T](o: Option[T])Boolean

scala> isNone(None)
res0: Boolean = true

scala> isNone(Some(42))
res0: Boolean = false
```

Generic/polymorphic. . . method actually

Example

- When used, parameter type may be infered.

```
scala> def isNone[T](o : Option[T]) = (o == None)
isNone: [T](o: Option[T])Boolean

scala> isNone(None)
res0: Boolean = true

scala> isNone(Some(42))
res0: Boolean = false

scala> isNone(Some("42"))
res0: Boolean = false
```

Generic/polymorphic. . . method actually

Example

- When used, parameter type may be infered.

```
scala> def isNone[T](o : Option[T]) = (o == None)
isNone: [T](o: Option[T])Boolean

scala> isNone(None)
res0: Boolean = true

scala> isNone(Some(42))
res0: Boolean = false

scala> isNone(Some("42"))
res0: Boolean = false
```

- Generic **val**-declared functions are not possible.
  But one can use **val** for fixed type.

```
scala> val isNone_Int = isNone[Int](_)
isNone_Int: Option[Int] => Boolean = <function1>
```

Generic/polymorphic. . . method actually

Example

- When used, parameter type may be infered.

```
scala> def isNone[T](o : Option[T]) = (o == None)
isNone: [T](o: Option[T])Boolean

scala> isNone(None)
res0: Boolean = true

scala> isNone(Some(42))
res0: Boolean = false

scala> isNone(Some("42"))
res0: Boolean = false
```

- Generic **val**-declared functions are not possible.
  But one can use **val** for fixed type.

```
scala> val isNone_Int = isNone[Int](_)
isNone_Int: Option[Int] => Boolean = <function1>

scala> isNone_Int(Some(42))
res1: Boolean = true
```

First-class citizen

Pattern matching

Pattern matching : `match`

General syntax

expression **match**

```
{
  case pattern1 => result1
  case pattern2 => result2
  ...
  case _ => defaultResult
}
```

Pattern matching : `match`

General syntax

expression **match**
{
case *pattern*$_1$ **=>** *result*$_1$
case *pattern*$_2$ **=>** *result*$_2$
...
case _ => *defaultResult*
}

Features

- *expression* is matched against any *pattern*$_i$:

Pattern matching : `match`

General syntax

expression **match**
{
case $pattern_1$ **=>** $result_1$
case $pattern_2$ **=>** $result_2$
...
case _ => *defaultResult*
}

Features

- *expression* is matched against any $pattern_i$:
  - if there is a match, corresponding $result_i$ is returned ;

Pattern matching : `match`

General syntax

expression **`match`**

```
{
  case pattern1 => result1
  case pattern2 => result2
  ...
  case _ => defaultResult
}
```

Features

- *expression* is matched against any $pattern_i$:
  - if there is a match, corresponding $result_i$ is returned ;
  - otherwise *defaultResult* is returned

Pattern matching : `match`

General syntax

expression **match**
{
case *pattern*$_1$ **=>** *result*$_1$
case *pattern*$_2$ **=>** *result*$_2$
...
case _ => *defaultResult*
}

Features

- *expression* is matched against any *pattern*$_i$:
  - if there is a match, corresponding *result*$_i$ is returned ;
  - otherwise *defaultResult* is returned
- type of result is least common supertype of all expressions *result*$_1$, ..., *defaultResult*

Pattern matching rules

Exhaustiveness

- **All possibilities should be covered.**
- Wildcard _ handles all remaining possibilities in one pass

Pattern matching rules

Exhaustiveness

- **All possibilities should be covered.**
- Wildcard _ handles all remaining possibilities in one pass

Sequentiality

- Pattern are treated **in the same order as written**.
- Only the **first matching pattern** is considered.

Pattern matching rules

Exhaustiveness

- **All possibilities should be covered.**
- Wildcard _ handles all remaining possibilities in one pass

Sequentiality

- Pattern are treated **in the same order as written**.
- Only the **first matching pattern** is considered.

Uniqueness

- An identifier **ony appears one in a pattern**.
- `if` guard may be used to explicit links.

Pattern matching rules : rules

Example

```
scala> (1, 2, 3) match
         {
           case (1, 2, _) => 2
           case (1, _, _) => 1
           case  _        => 0
         }
res0: Int = 2
```

Pattern matching rules : rules

Example

```
scala> (1, 2, 3) match
        {
          case (1, 2, _) => 2
          case (1, _, _) => 1
          case  _        => 0
        }
res0: Int = 2

scala>  (1, 2, 3) match
         {
           case (1, _, _) => 1
           case (1, 2, _) => 2
           case  _        => 0
         }
<console>:15: warning: unreachable code
           case (1, 2, _) => 2
                       ^
res1: Int = 1
```

Pattern matching rules : uniqueness and `if` guard

Example

```
scala> (2, 2) match
         {
           case (x, x) => x
           case  _     => 0
         }
<console>:14: error: x is already defined as value x
           case (x, x) => x
                    ^
```

Pattern matching rules : uniqueness and `if` guard

Example

```
scala> (2, 2) match
         {
           case (x, x) => x
           case  _     => 0
         }
<console>:14: error: x is already defined as value x
           case (x, x) => x
                    ^

scala> (2, 2) match
         {
           case (x, y) if x == y => x
           case  _               => 0
         }
res0: Int = 2
```