Scala

Data types and Variables

Basic Data Types

- Int
- Float
- Double
- Boolean
- Char
- String

3

Variables

- [private] var name[: type] = value
- var is the keyword of Scala that is used to define a variable
- Variables must be initialized during their definition
- The type of the variable is implicitly inferred from the initialization value if the type is not specified
- Variables are public if the keyword private is not specified

4

Variables: Examples

- var sentence = "Hello world"
  - Implicit inference of the date type of sentence
- var sentence: String = "Hello world"
  - Explicit definition of the date type of sentence
- sentence is a public variable of type String and its initial value is "Hello world"

5

Variables: Examples

- var weight = 75
- var weight: Int = 75
- weight is a public variable of type Int and its initial value is 75

6

Variables: Examples

- private var weight = 75
- private var weight: Int = 75
- weight is a private variable of type Int and its initial value is 75

7

Variables: Examples

- var price = 10.5
- var price: Double = 10.5
- price is a public variable of type Double and its initial value is 10.5

8

Variables: Examples

- var condition = true
- var condition: Boolean = true
- condition is a public variable of type Boolean and its initial value is true

9

Variables: Examples

- var firstCharacter = 'a'
- var firstCharacter: Char = 'a'
- firstCharacter is a public variable of type Char and its initial value is 'a'

10

Immutable Variables (or Values)

- [private] val name[: type] = value
- val is the keyword of Scala that is used to define immutable variables (i.e., constants) also called **values**
- Immutable variables must be initialized during their definition
- The type of immutable variables is implicitly inferred from the initialization value if the type is not specified
- Immutable variables are public if the keyword private is not specified

11

Immutable Variables: Examples

- val sentence = "Hello world"
  - Implicit inference of the date type of sentence
- val sentence: String = "Hello world"
  - Explicit definition of the date type of sentence
- sentence is a public immutable variable of type String and its initial value is "Hello world"

12

Standard operators

- Mathematical operators
  - +, -, *, /, %
- Boolean operators
  - |, &, !

13

Standard operators: Examples

- var num = 10
- var den = 3
- var ris = 10/3
- var ris = 10%3
- var text = “Hello World”
- text = text + “ Paolo”

14

Strings

- Scala's String is built on Java's String
- Scala's String has also some additional features
  - Multiline literals
  - String interpolation

15

Strings: Examples

- Use of double quotes and special characters escaped with backslash

```
val hello = "Hello There"  /* Hello There */
val signature = "With regards, \nYour friend"
/*
With Regards,
Your friend
*/
```

16

Strings: Examples

- String concatenation (or string addition)

val hello = “Hello, “+ “World” /* Hello, World */

- String comparison

val matched = (hello == “Hello, World”) /* true */

  - Unlike Java, in Scala the equals operator (==) compares the contents of the Strings

17

Multiline Strings

- A multiline String can be created using triple-quotes
  - Multiline Strings are literal, and do not recognize the use of backslashes, i.e., they do not recognize special characters

val greeting = """She suggested reformatting the file

by replacing tabs (\t) with newlines (\n);

"Why do that?", he asked. """

18

String interpolation

- Building a string based on other variables/values can be do with string addition
- E.g.,

```
val approx = 355/113f
/* approx: Float = 3.141593 */
println("Pi, using 355/113, is about " + approx + "." )
/* Pi, using 355/113, is about 3.141593. */
```

19

String interpolation

- String interpolation is another way to combine variables/values inside a string
- The Scala notation for is an "s" prefix added before the first double quote of the string
- The dollar sign operator $ (with optional braces) can be used to insert references to variables/values

20

String interpolation: Example

```
val approx = 355/113f
/* approx: Float = 3.141593 */
println(s"Pi, using 355/113, is about ${approx}." )
/* Pi, using 355/113, is about 3.141593. */
```

21

Strings: Other particular operations

- Repeat the same sequence of characters multiple times

```
val repeatHi="Hi "*5
println(repeatHi)
val str1="Paolo "
val num=10
val rep=str1*num
println(rep)
```

22

Regular expressions

- As many other languages, Scala supports regular expressions
- A regular expression is a string of characters and punctuation that represents a search pattern
  - The format is the same used by the Java class java.util.regex.Pattern

23

Use of regular expressions: Example

- matches
  - It is used to check if the content of String matches the provided regular expression

```
var sentence= "Test matching operation"
var res: Boolean =sentence.matches("Test .*")
println(res)
```

24

Use of regular expressions: Example

- replaceAll
  - Replaces all matches of the regular expression with the specified replacement text

```
var sentence= "milk, tea, muck"

var res: String = sentence.replaceAll("m[^ ]+k",
"coffee")

println(res)
```

25

Use of regular expressions: Example

- replaceFirst
  - Replaces the first match of the regular expression with the specified replacement text

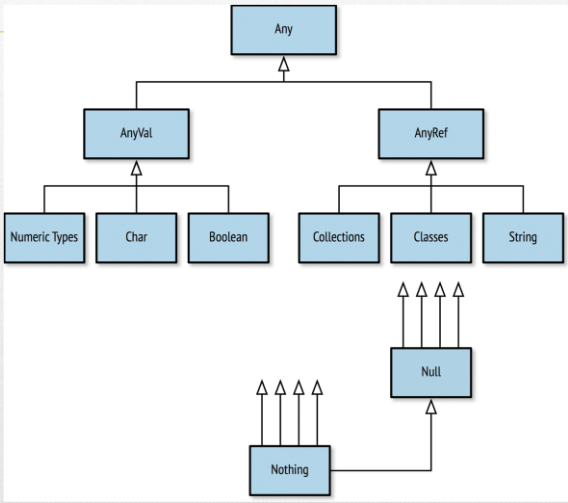
```
var sentence= "milk, tea, muck"

var res: String = sentence.replaceFirst("m[^ ]+k",
"coffee")

println(res)
```

26

The Scala data type hierarchy



27

Common operations

- The following operations are available on all types
- asInstanceOf[<type>]
  - Converts the value to a value of the desired type
  - Causes an error if the value is not compatible with the new type.
  - E.g., 5.asInstanceOf[Long]

28

Common operations

- getClass
  - Returns the type (i.e., the class) of a value
  - E.g., (7.0 / 5).getClass

29

Common operations

- isInstanceOf
  - Returns true if the value has the given type
  - E.g., (5.0).isInstanceOf[Float]

30

Common operations

- to<type>
  - Conversion functions to convert a value to a compatible
  - E.g., 20.toByte
  - 47.toFloat

31

Common operations

- toString
  - Renders the value to a String
  - E.g., (3.0 / 4.0).toString

32

Cast: Data type conversion

- asInstanceOf[<type>] and the to* methods can be used to convert data from one data type to another
  - Obviously only if the content of the "input" variable/value is compatible with the "output" data type
- The to* methods are preferred

33

Cast: Examples

```
var longAge: Long = 40
var intAge: Int = longAge.toInt
print(longAge+"---"+intAge)
------------------------------------------------------------------
var stringAge: String = "40"
var intAge=stringAge.toInt
print(stringAge+"---"+intAge)
```

34

Arrays, Lists, Maps, Tuples

Basic operations

Collections and complex data types in Scala

- Array
- List
- Map
- Tuple

Arrays

- Scala provides the Array data type
- Scala supports
  - Homogeneous arrays
    - All the elements of the array are associated with the same data type
  - Heterogeneous arrays
    - The elements of the array belong to different data types

37

Homogeneous arrays: Example

- Definition of an array of integers containing the values 1, 2, 3

```
val numbers = Array(1, 2, 3)
```

- numbers is of type Array[Int]

38

Homogeneous arrays: Example

- Print on the console the value of the first element

```
val numbers = Arrays(1, 2, 3)
println(numbers(0))
```

39

Heterogeneous arrays: Example

- Definition of an array containing integers and strings

```
val mix = Array(1, "Hello", "World", 10)
```

- mix is of type Array[Any]

40

Collections: List

- Scala provides the List data type
- Scala supports
  - Homogeneous lists
    - All the elements of the list are associated with the same data type
  - Heterogeneous lists
    - The elements of the list belong to different data types

41

Homogeneous lists: Example

- Definition of a list of integers containing the values 1, 2, 3

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

or

```
val numbers = 1 :: 2 :: 3 :: Nil
```

- numbers is of type List[Int]

42

Homogeneous lists: Example

- Print on the console the value of the first element

```
val numbers = List(1, 2, 3)
println(numbers(0))
```


Heterogeneous lists: Example

- Definition of a list containing integers and strings

```
val mix = List(1, "Hello", "World", 10)
```

or

```
val mix = 1 :: "Hello" :: "World" :: 10 :: Nil
```

- mix is of type List[Any]

Concatenation of lists

- Concatenate lists
  - val res = list1 ::: list2
  - Or
  - val res = List.concat(list1, list2)
    - List.concat() receives as arguments two or more lists

Collections: Map

- Scala provides the Map data type
- Map is used to maintain the mapping between keys and values
  - Each key is associated with only one value

Maps

- Add a new pair key -> value
  - mapvariable += newkey -> newvalue
- Retrieve the value associated with a key
  - mapvariable(key)
  - or
  - mapvariable.get(key).get
- Default value for missing keys
  - mapvariable.get(key).getOrElse(default value)

47

Maps: Example

- Definition of a map variable that maps integers (key) to strings (values)

```
/* Define the Map and insert the first key -> value pair */
val mapper = Map(1 -> "Hello")
/* Add a new pair */
mapper += 2 -> "World"
println(mapper(1))
```

48

Maps: Example

- Definition of a map variable that maps integers (key) to strings (values)

```
/* Define an empty Map of type Map[Int, String] */
val mapper: Map[Int, String] = Map()
/* Add two new pairs */
mapper += 1 -> "Hello"
mapper += 2 -> "World"
println(mapper(1))
```

49

Maps: Example #2

- Definition of a map variable that maps string (key) to strings (values)

```
val stateCapitals = Map( "Alabama" -> "Montgomery",
"Alaska" -> "Juneau")
println("Alabama: " + stateCapitals("Alabama")
.getOrElse("Unknown"))
println("Italy: " +
stateCapitals.get("Italy").getOrElse("Unknown"))
```

50

Tuples

- Scala has a specific data type that is used to represent tuples
  - Tuples are groups of N items
  - Elements are unrelated to each other. The data types can be different
  - They are useful to return a set of values from a method without defining a new class od structure
- Pay attention that the items of tuples are immutable

51

Tuples

- Definition of tuples
  - var name=(comma separated list of items)
- Retrieval of the N-th item
  - name._N

52

Tuples: Examples

- Scala:

val tuple: Tuple2[Int, String] = (1, “apple”)

val quadruple =

(2, “orange”, 0.5d, false)

- Java:

Pair<Integer, String> tuple = new Pair<Integer, String>(1, “apple”)

Quadruples: No equivalent in Java

53

Tuples: Examples

```
/*  Define a tuple with three items */
val profile=(“Paolo”, “Garza”, 40)
println(“Name:” + profile._1)
println(“Surname:” + profile._2)
println(“Age:” + profile._3)
```

54

Generics in Scala

- Analogously to Java, also Scala supports generics

55

Generics: Examples

- Scala:

List[String]

List[Int]

Map[Int, String]

…

- Java:

List<String>

List<Integer>

Map<Int, String>

56

Expressions

Expressions

- Expression
  - A unit of code that returns a value
  - One or more lines of code can be considered an expression if they are collected together using curly braces
    - This is known as expression block

Expressions: Example

- "hello"
  - Is a very simple expression
- "hel"+"lo"
  - Is another very simple expression
- As we already did, expressions can be used to assign values to variables and immutable variables (values)
- val message="hello"

59

Expression blocks

- An expression block is a sequence of one or more lines of code
- An expression has its own scope, and may contain values and variables local to the expression block
- The last expression in the block is the return value for the entire block

60

Expression blocks: Example

- Example with expressions containing only one line of code

val x = 5 * 20;

val amount = x + 10

- Example with an expression block with multiple lines of code

val amount = {val x = 5 * 20; x + 10 }

- The value of amount is the same in both cases

61

Expression blocks: Example

- Example with expressions containing only one line of code

val x = 5 * 20;

val amount = x + 10

- Example with an expression block with multiple lines of code x is visible only inside the expression block

val amount = {val x = 5 * 20; x + 10 }

- The value of amount is the same in both cases

62

Expression blocks: Example

- Example with expressions containing only one line of code

```
val x = 5 * 20;
val amount = x + 10
```

- Example with an expression block with multiple lines of code

This is the returned value (expression)

```
val amount = {val x = 5 * 20; x + 10 }
```

- The value of amount is the same in both cases

63

Expression blocks: Example

- Example with an expression block with multiple lines of code

```
val amount = {val x = 5 * 20; x + 10 }
```

- Is equivalent to

```
val amount = {val x = 5 * 20
             x + 10 }
```

64

Expression blocks: Example

- Example with an expression block with multiple lines of code

val amount = {val x = 5 * 20; x + 10 }

- Is equivalent to

val amount = {val x = 5 * 20

x + 10 }

This is the returned value (expression)

65

Expression blocks: Example

- Example of a three-deep nested expression block

val res = { val a = 1; { val b = a * 2; { val c = b + 4; c } } }

66

Expression blocks: Example

- Example of a three-deep nested expression block

```
val res = { val a = 1; { val b = a * 2; { val c = b + 4; c }
} }
```

This is the returned value (expression)

67

Conditional expressions

if-then-else

- if-then-else in Scala is analogous to those of the Java, C, C++ languages
- However, it is also an expression
- The if expression evaluates a Boolean expression
  - If the result of the Boolean expression is equal to true a block of code is executed
  - Otherwise the block of code associated with the else part of the statement is executed

69

if-then-else

- Scala:

```
if (test) {
/* code */
} else {
...
}
```

- Java:

```
if (test) {
...
} else {
...
}
```

70

if-then-else

- Scala:

```
if (test) {
...
} else if (test2) {
        ...
    } else {
     ...
    }
```

- Java:

```
if (test) {
...
} else if (test2) {
        ...
    } else {
     ...
    }
```

71

if-then-else as an expression

- if-then-else in Scala is also an expression
- If the Boolean condition of the if-then-else is true then last expression of the if expression block is the returned value
- Otherwise, the last expression of the else block is the returned value

72

if-then-else as an expression: Example

```
val x=10
val y=20
val max = {if (x>y) x else y}
println(max)
```

73

if-then-else as an expression: Example#2

```
val x=10
val y=20
val max = { if (x>y) {
                println(x +">"+ y)
                x }
            else {
                println(x +"<="+ y)
                y }
          }
println(max)
```

74

Match expressions

Match expressions

- Match expressions are like the "switch" statements in Java and C++
  - A single input item is evaluated and the first pattern that is "matched" is executed and its value returned

Match expressions

- Like C's and Java's "switch" statements, Scala's match expressions support a default or wildcard "catch-all" pattern
- Unlike them, only zero or one patterns can match

77

Match expressions

- The traditional "switch" statement is limited to matching by value
- Scala's match expressions are flexible and also enable matching such diverse items as types, regular expressions, numeric ranges, and data structure contents
- Moreover, match expressions are expressions
  - Hence, they can return values

78

Match expressions: Syntax

```
<expression> match {
    case <pattern match> => <expression>
    [case...]
}
```

79

Match expressions: Example

```
val test: Char ='a'
test match {
   case 'a' => { println("Code associated with a") }
   case 'b' => { println("Code associated with b") }
}
```

80

Match expressions: Example #2

```
val test: Char ='A'
test match {
    case 'a'|'A' => { println("Code associated with a or A") }
    case 'b'|'B'|'c'|'C'| => { println("Code associated with b, B, c, or C") }
}
```

81

Match expressions: Returned values

- Match expressions are expressions
- Hence, they return values if the last expression of the executed code is an expressions

82

Match expressions: Example #3

```
val max = x > y match {
            case true => x
            case false => y
          }
```


Match expressions: Example #3

```
val max = x > y match {
            case true => x
            case false => y
          }
```

The match expression is evaluated on this Boolean test

Match expressions: Example #3

```
val max = x > y match {
            case true => x
            case false => y
          }
```

If x>y is true then the value of x is returned

Match expressions: Example #3

```
val max = x > y match {
            case true => x
            case false => y
          }
```

If x>y is false then the value of y is returned

Match expressions: Example #4

```
val day = "MON"
val kind = day match {
                case "MON" | "TUE" | "WED" |
                     "THU" | "FRI" => "weekday"
                case "SAT" | "SUN" => "weekend"
            }
println(kind)
```

87

Match expressions: "Default" value

- There are two kinds of wildcard patterns you can use in a match expression
  - Value binding and
  - Wildcard (aka "underscore") operators

88

Match expressions: Value binding

- With **value binding** the input to a match expression is bound to a local value (immutable variable)
  - The local value can then be used in the body of the case block.
- Because the pattern contains the name of the value to be bound there is no actual pattern to match against
  - Thus value binding is a wildcard pattern because it will match any input value

89

Match expressions: Value binding example

- The following example sets status (integer) to
  - 200 if the message is "Ok"
  - -1 otherwise

90

Match expressions: Value binding example

```
val message = "Ok"
val status = message match {
                    case "Ok" => { println("matched Ok“
                                    200
                    case other => { println(other+" matches nothing")
                                    -1
                                  }
            }
println(status)
```

91

Match expressions: Value binding example

```
val message = "Ok"
val status = message match {
                    case "Ok" => { println("matched Ok“
                                    200
                    case other => { println(other+" matches nothing")
                                    -1
                                  }
            }
println(status)
```

other is set to the value of message if the previous cases are not matches

92

Match expressions: Wildcard operator

- The **wildcard** cannot be accessed on the right side of the arrow, unlike with value binding.
  - If you need to access the value of the wildcard in the case block, consider using a value binding, or just accessing the input to the match expression (if available)

93

Match expressions: Value binding example

- The following example sets status (integer) to
  - 200 if the message is "Ok"
  - -1 otherwise

94

Match expressions: Value binding example

```
val message = "Ok"
val status = message match {
                    case "Ok" => { println("matched Ok“
                                   200
                    case _ =>      { println("matches nothing")
                                   -1
                                  }
            }
println(status)
```


Match expressions: Value binding example

```
val message = "Ok"
val status = message match {
                    case "Ok" => { println("matched Ok“
                                   200
                    case _ =>      { println("matches nothing")
                                   -1
                                  }
            }
println(status)
```

Wildcard operator

Match expression: Pattern guards

- A **pattern guard** adds an if expression to a value-binding pattern
  - It allows mixing conditional logic into match expressions
- When a pattern guard is used the pattern will only be matched when the if expression returns true

97

Match expression: Pattern guard example

```
val response: String = null
response match {
    case s if (s != null) => println("Received "+s)
    case s => println("Error! Received a null response")
}
```

98

Match expression: Pattern guard example

```
val response: String = null
response match {
    case s if (s != null) => println("Received "+s)
    case s => println("Error! Received a null response")
}
```

Value binding. The value of response is assigned to s

99

Match expression: Pattern guard example

```
val response: String = null
response match {
    case s if (s != null) => println("Received "+s)
    case s => println("Error! Received a null response")
}
```

Pattern guard. If the condition is true the code of this case is executed

100

Match expression: Pattern guard example

```
val response: String = null
response match {
    case s if (s != null) => println("Received "+s)
    case s => println("Error! Received a null response")
}
```

Pattern guard. If the condition is false the next cases are considered

Match expression: Pattern guard example

The following code is equivalent to the previous one

```
val response: String = null
response match {
    case s if (s != null) => println("Received "+s)
    case _ => println("Error! Received a null response")
}
```

Match expression: Pattern guard example

The following code is equivalent to the previous one

```
val response: String = null
response match {
    case s if (s != null) => println("Received "+s)
    case _ => println("Error! Received a null response")
}
```

Here the wildcard is sufficient

Match expression: Matching types

- In Scala you can specify a matching also on the type of the input
- Java and C++ do not support this type of test

Match expression: Matching type example

```
val list = List("a", 1, 'c', 34.5)
list(0) match {
        case v: String => println("This is a string")
        case v: Int => println("This is an integer")
        case v: Char => println("This is a char")
        case v => println("This is another type")
}
```


Loops

for loop

- The for loop in Scala is different with respect to those of C, C++
- The for loop in Scala always iterates over an input collection
  - For each element **e** of the input collection, the block of code associated with the for loop is executed
- The for loop in Scala is a "functional for loop"

for loop: Example 1

- Scala:

```
for (i <- 0 to 3) {
   /* code */
}
```

- The expression "0 to 3" defines a collection of integers containing the values 0, 1, 2, 3

- Java:

```
for (int i = 0; i < 4; i++) {
 /* code */
}
```

for loop: Example 2

- Scala:

```
for (s <- args) {
    println(s)
}
```

- **args** is the collection of arguments of the application. Each element is a string

- Java:

```
for (String s : args) {
    System.out.println(s);
}
```

109

for loop: yield option

- A for loop can return a collection
- At each iteration, the last expression is the returned value
- Syntax
- for (<identifier> <- <iterator>) yield {<expression>}
- for (x <- 1 to 7) yield { s"Day $x:" }

110

for loop: yield option example

- Returns (“Day 1”, “Day 2”, .. , “Day 7”,)

```
val res=for (x <- 1 to 7) yield { "Day "+x }
```

- Returns (“Even”, “Odd”, “Even”, .., “Even”,)

```
val res=for (x <- 1 to 7) yield { if (x%2==0) {"Odd"}
else {“Even"} }
```

111

while loop

- The while loop in Scala is analogous to those of the Java, C, C++ languages.
- The while loop executes the code of block associated with it as long as the evaluated condition is true

112

while loop

- Scala:

```
while (expression==true)
{
    ...
}
```

- Java:

```
while (expression==true)
{
    ...
}
```

113

do-while loop

- The do-while loop in Scala is analogous to those of the Java, C, C++ languages.
- The do-while loop executes the code of block associated with it as long as the evaluated condition is true
  - The code is always executed at least one time

114

do-while loop

- Scala:

```
do {
    ...
}while (expression==true)
```

- Java:

```
do {
    ...
}while (expression==true)
```

115

Console: Basic operations

- Scala:

```
Console.println("Hello")
    or simply
println("Hello")

var i:Int = 10
println("Value of i:"+ i)
```

- Java:

```
System.out.println("Hello");

Integer i=10;
System.out.println("Value of i:"+ i)
```

116

Console: Basic operations

- Scala:

```
val line = Console.readLine()
```

or simply

```
val line = readLine()
```

- Java:

```
BufferedReader r = new
BufferedReader(new
InputStreamRead(System.in)
String line = r.readLine();
```

117

Console: Read operations

- The read* methods of the Console Object are used to read data from the console in Scala
  - Console.readBoolean
  - Console.readChar
  - Console.readDouble
  - Console.readInt
  - Console.readLine
  - …

118

Read operations: Example

```
Console.println("Insert your name")
var name=Console.readLine()
Console.println("Hi "+name)
Console.println("How old are you?")
var age=Console.readInt()
Console.println("Do you like Scala?")
var like=Console.readBoolean()
Console.println("Name: "+name + " Age:" + age + "Like Scala: "+like )
```

119

Console: readf* methods

- The methods readf, readf1, readf2, and readf3 can be used to read multiple values at the same time
- Console.readf(String)
  - Returns a list of values of type Any (List[Any])
- Console.readf1(String)
  - Returns one value of type Any
- Console.readf2(String)
  - Returns two values of type Any (Any,Any)

120

Console: readf* methods

- The parameter of the readf* methods is a string that specify the type of the expected input values
- E.g.
  - "{0} {1,number}" means that the expected values are a string and then a number
  - "{0,number} {1,number} {2}" means that the expected values are two numbers and a string
- An parse exception is generated if the input values are not consistent with the expected data types

121

Console: readf* methods

- The main problem of readf, readf1, readf2, and readf3 is the type of the returned values
  - All the returned values are of type Any
  - Values must be casted to the correct data type

122

Console: readf* methods: Example

```
Console.println("Insert your name and age")
val (a,b)=Console.readf2("{0} {1,number}")
var name=a.toString
var age=b.toString.toInt
Console.println("Name: "+name + " Age:" + age)
```

123

Console: readf* methods: Example

```
Console.println("Insert your name and age")
val values: List[Any]=Console.readf("{0} {1,number}")
var name=values(0).toString
var age=values(1).toString.toInt
Console.println("Name: "+name + " Age:" + age)
```

124

Console: Read operations

- Since version 2 of Scala the Console.read* methods are deprecated. However, the "equivalent" methods scala.io.StdIn.read* are provided
  - scala.io.StdIn.readBoolean
  - scala.io.StdIn.readChar
  - scala.io.StdIn.readDouble
  - scala.io.StdIn.readInt
  - scala.io.StdIn.readLine
  - …

125

Read operations based on the split method

- Another approach is based on the split method of String
  - split("splitting character") returns and Array of Strings
- Also in this case a manual cast is needed

126

Read operations based on the split method: Example

```
Console.println("Insert your name and age")
val input=Console.readLine()
val vals: Array[String]=input.split(" ")
var name=vals(0).toString
var age=vals(1).toString.toInt
Console.println("Name: "+name + " Age:" + age)
```

127

Read operations based on Java Scanner

- Scala can also use the Java java.util.Scanner class to read data from the console
- However, if the input does not match what you expect and error/exception will be thrown

128

Read operations based on Java Scanner: Example

```
val scanner=new java.util.Scanner(System.in)
print("How old are you?")
val age=scanner.nextInt()
Console.println("You are "+age+"years old")
```