Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Programming paradigms

L06: Imperative programming

by Michał Szczepanik

Declarative programming vs imperative programming

- ***Declarative programming*** *is a programming paradigm … that expresses the logic of a computation without describing its control flow.*
- ***Imperative programming*** *is a programming paradigm that uses statements that change a program’s state.*

Imperative programming

The imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform.

Imperative programming focuses on describing how a program operates.

Side effect

In computing, an effect system is a formal system which describes the computational effects of computer programs, such as side effects. An effect system can be used to provide a compile-time check of the possible effects of the program.

In computer science, an operation, function or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, that is to say has an observable effect besides returning a value (the main effect) to the invoker of the operation.

In functional languages like Haskell effects are hidden in monads.

Cell and references

Effects in the programming language must be supported by:

- modifiable data structures;
- imperative control structures;
- input / output operations.

Almost every programming language (even Haskell via monads) provides some form of assignment operation that changes the contents of the mutable cell, which is the simplest modifiable data structure.

Cell and references

The cell is a container with identity and content.

- The identity is a permanent ("name" or "address" of the cell). Data structure representing the cell identity is called a reference.
- The cell is only available by reference.
- The content of the cell can be changed.

The basic operations on references are:

- allocation - allocation of memory to the cell
- assignment (mutation) - change of cell content
- dereferencing - downloading cell content

Polymorphism

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

The most commonly recognised major classes of polymorphism are:

- **Ad hoc polymorphism -** defines a common interface for an arbitrary set of individually specified types.
- **Parametric polymorphism -** when one or more types are not specified by name but by abstract symbols that can represent any type.
- **Subtyping** (also called subtype polymorphism or inclusion polymorphism) - when a name denotes instances of many different classes related by some common superclass.

Stateful objects

It describes the world as a set of objects, some of which have state that changes over the course of time.

An object has a state if its behavior is influenced by its history.

Example: a bank account has a state, because the answer to the question “can I withdraw 100 CHF ?” may vary over the course of the lifetime of the account.

```
class BankAccount {
  private var balance = 0

  def deposit(amount: Int): Int = {
    if (amount > 0) balance = balance + amount
    balance
  }

  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance = balance - amount
      balance
    } else throw new Error("insufficient funds")
}
```

```
val account = new BankAccount // account:
BankAccount = BankAccount
account deposit 50 //
account withdraw 20 // res1: Int = 30
account withdraw 20 // res2: Int = 10
account withdraw 15                 //
java.lang.Error: insufficient funds
```

Applying the same operation to an account twice in a row produces different results! Clearly, accounts are stateful objects.

Take input from a user

```
val a=scala.io.StdIn.readInt()
println("The value of a is "+ a)
```

def readBoolean(): Boolean Reads a Boolean value from an entire line from stdin .

def readByte(): Byte Reads a Byte value from an entire line from stdin .

def readChar(): Char Reads a Char value from an entire line from stdin .

def readDouble(): Double Reads a Double value from an entire line from stdin .

def readFloat(): Float Reads a Float value from an entire line from stdin .

def readInt(): Int Reads an Int value from an entire line from stdin .

def readLine(text: String, args: Any*): String Prints formatted text to stdout and reads a full line from stdin .

def readLine(): String Reads a full line from stdin .

def readLong(): Long Reads a Long value from an entire line from stdin .

def readShort(): Short Reads a Short value from an entire line from stdin .

def readf(format: String): List[Any] Reads in structured input from stdin as specified by the format specifier.

def readf1(format: String): Any Reads in structured input from stdin as specified by the format specifier, returning only the first value extracted, according to the format specification.

def readf2(format: String): (Any, Any) Reads in structured input from stdin as specified by the format specifier, returning only the first two values extracted, according to the format specification.

def readf3(format: String): (Any, Any, Any) Reads in structured input from stdin as specified by the format specifier, returning only the first three values extracted, according to the format specification.

Other way

```
val scanner = new java.util.Scanner(System.in)
println("What is your name")
val name = scanner.nextLine()
```

val vs var

If this value is not significant when defining the variable (this applies only to mutable variables), you can use a wildcard:

```
var x: Int = _
x: Int = 0
```

The Scala compiler initializes the variables:

numeric types with zero,

Boolean values false,

references are null,

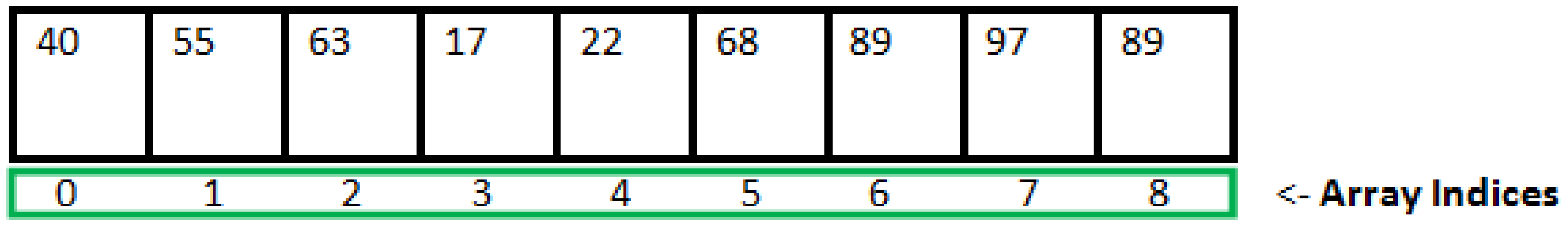
variables of type Char are \u0000 (same as in Java).

var

In Scala, the assignment operation is an Unit expression

```
var z = 0;
z = 5
val v = z = z+2
```

Array



Array Length = 9
First Index = 0
Last Index = 8

Array is a special kind of collection in scala. it is a fixed size data structure that stores elements of the same data type.

```
val numbers = Array(1, 2, 3, 4)
val first = numbers(0)
numbers(3) = 100
val biggerNumbers = numbers.map(_ * 2)
```

While-loops

```
def power(x: Double, exp: Int): Double = {
    var r = 1.0
    var i = exp
    while (i > 0) { r = r * x; i = i - 1 }
    r
}
```

As long as the condition of a *while* statement is true, its body is evaluated.

For-loops

```
for (i <- 1 until 3) { System.out.print(i + " ") }
```

For-loops translate similarly to for-expressions, but using the foreach combinator instead of map and flatMap.

```
def foreach(f: A => Unit): Unit =
    // apply `f` to each element of the collection
```

```
for (i <- 1 until 3; j <- "abc") println(s"$i $j")
```

translate to

```
(1 until 3) foreach (i => "abc" foreach (j =>
println(s"$i $j")))
```

Scala programming

A balanced attitude for Scala programmers

Prefer vals, immutable objects, and methods without side effects. Reach for them first.

Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.

[from book “Programming in Scala”]

Thank you for your attention