cloudera®

Just Enough Scala

201603

cloudera®

Introduction

Chapter 1

Course Chapters

- **Introduction**
- Scala Overview
- Scala Basics
- Working with Data Types
- Grouping Data Together
- Flow Control in Scala
- Using and Creating Libraries
- Conclusion

Chapter Topics

Introduction

- **About this Course**
- About Cloudera
- Course Logistics
- Introductions

Course Objectives

During this course, you will learn

- **What Scala is and how it differs from languages such as Java or Python**
- **Why Scala is a good choice for Spark programming**
- **How to use key language features such as data types, collections, and flow control**
- **How to implement functional programming solutions in Scala**
- **How to work with Scala classes, packages, and libraries**

Introduction

- **This course is meant to to teach you *just enough* Scala programming**
  - Prepares you to understand and use Scala in Cloudera training courses
- **You should already understand basic programming concepts**
  - This course focuses on Scala syntax and idioms
- **No prior experience with analytics, Hadoop, or Cloudera software needed**

Chapter Topics

Introduction

- About this Course
- **About Cloudera**
- Course Logistics
- Introductions

About Cloudera (1)

- **The leader in Apache Hadoop-based software and services**
- **Founded by Hadoop experts from Facebook, Yahoo, Google, and Oracle**
- **Provides support, consulting, training, and certification for Hadoop users**
- **Staff includes committers to virtually all Hadoop projects**
- **Many authors of industry standard books on Apache Hadoop projects**

About Cloudera (2)

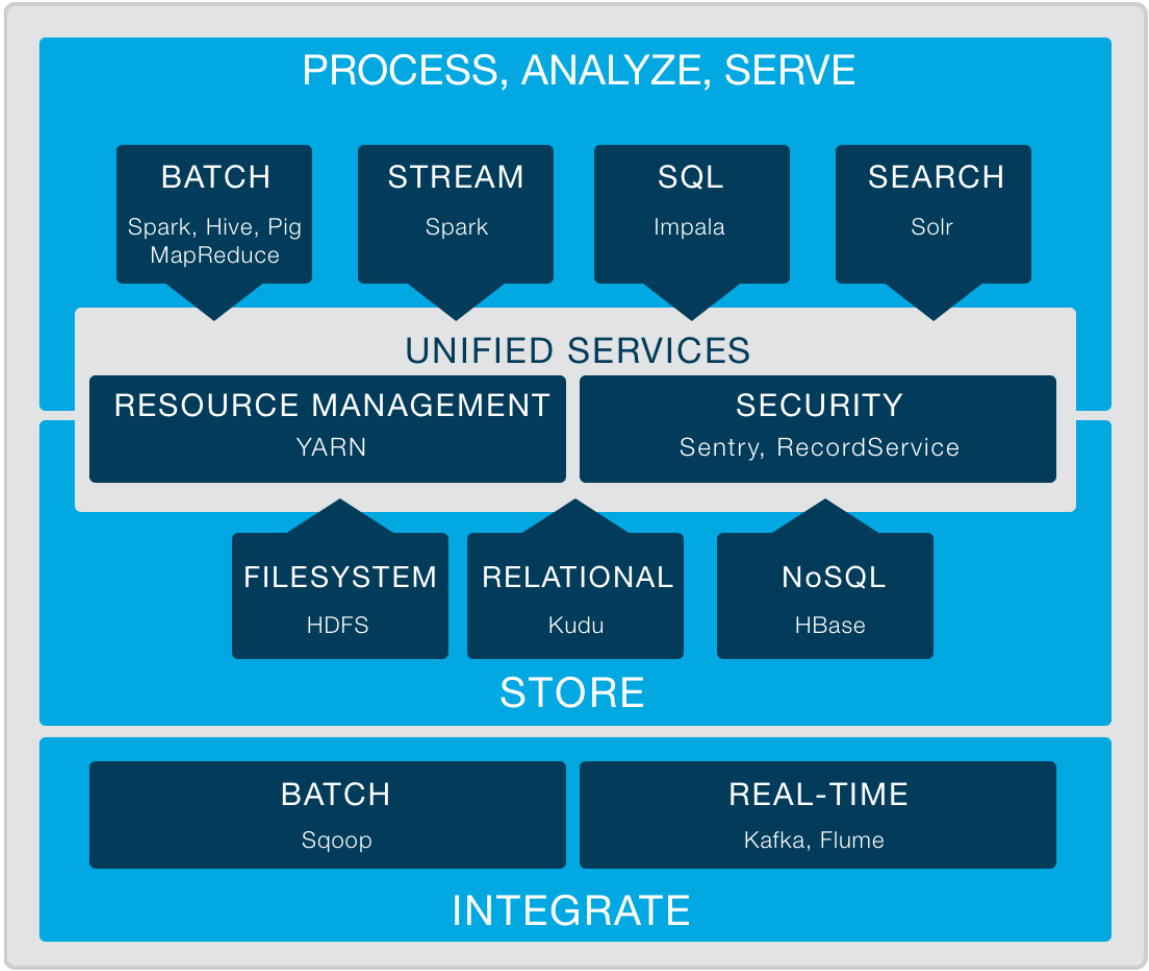
- **Our customers include many key users of Hadoop**
- **We offer several public training courses, such as**
  - *Cloudera Developer Training for Spark and Hadoop*
  - *Data Science at Scale Using Spark and Hadoop*
  - *Cloudera Administrator Training for Apache Hadoop*
  - *Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop*
  - *Designing and Building Big Data Applications*
  - *Cloudera Search Training*
  - *Cloudera Training for Apache HBase*
- **On-site and customized trainings are also available**

About Cloudera (3)

- **In addition to our public training courses, Cloudera offers two levels of certifications**
- **Cloudera Certified Professional (CCP)**
  - The industry's most demanding performance-based certification, CCP evaluates and recognizes a candidate's mastery of the technical skills most sought after by employers
  - CCP Data Engineer
  - CCP Data Scientist
- **Cloudera Certified Associate (CCA)**
  - CCA exams test foundational skills and sets forth the groundwork for a candidate to achieve mastery under the CCP program
  - CCA Spark and Hadoop Developer
  - Cloudera Certified Administrator for Apache Hadoop (CCAH)

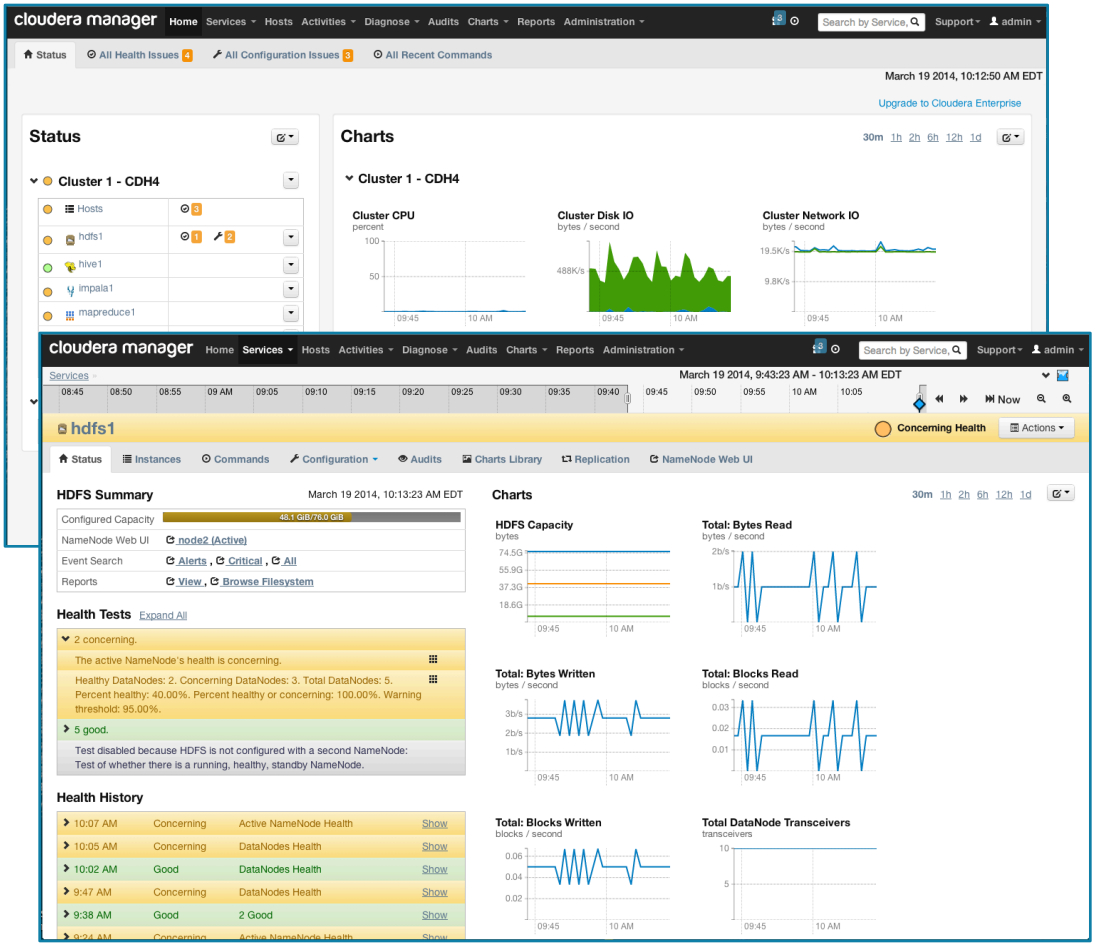
CDH (Cloudera's Distribution including Apache Hadoop)

- 100% open source, enterprise-ready distribution of Hadoop and related projects
- The most complete, tested, and widely deployed distribution of Hadoop
- Integrates all the key Hadoop ecosystem projects
- Available as RPMs and Ubuntu, Debian, or SuSE packages, or as a tarball



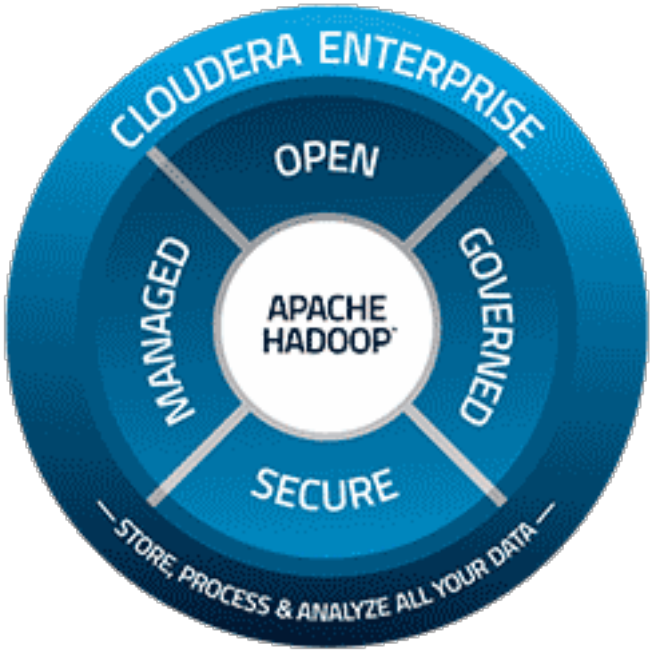
Cloudera Express

- **Cloudera Express**
  - Completely free to download and use
- **The best way to get started with Hadoop**
- **Includes CDH**
- **Includes Cloudera Manager**
  - End-to-end administration for Hadoop
  - Deploy, manage, and monitor your cluster



Cloudera Enterprise

- **Cloudera Enterprise**
  - Subscription product including CDH and Cloudera Manager
- **Includes support**
- **Includes extra Cloudera Manager features**
  - Configuration history and rollbacks
  - Rolling updates
  - LDAP integration
  - SNMP support
  - Automated disaster recovery



- **Extend capabilities with Cloudera Navigator subscription**
  - Event auditing, metadata tagging capabilities, lineage exploration
  - Available in both the Cloudera Enterprise Flex and Data Hub editions

Chapter Topics

Introduction

- About this Course
- About Cloudera
- **Course Logistics**
- Introductions

Logistics

- **Class start and finish times**
- **Lunch**
- **Breaks**
- **Restrooms**
- **Wi-Fi access**
- **Virtual machines**

Your instructor will give you details on how to access the course materials and exercise instructions for the class

Chapter Topics

Introduction

- About this Course
- About Cloudera
- Course Logistics
- **Introductions**

Introductions

- **About your instructor**
- **About you**
  - Where do you work? What do you do there?
  - Which programming languages do you use?
  - How much UNIX or Linux experience do you have?
  - Do you have experience with Hadoop or Spark?
  - What do you expect to gain from this course?

cloudera®

Scala Overview

Chapter 2

Course Chapters

- Introduction
- **Scala Overview**
- Scala Basics
- Working with Data Types
- Grouping Data Together
- Flow Control in Scala
- Using and Creating Libraries
- Conclusion

Scala Overview

In this chapter you will learn

- **Why Scala's design helps to increase developer productivity**
- **How functional programming differs from other programming techniques**
- **How Scala relates to Apache Spark**
- **Why Scala is a good choice for distributed data processing with Spark**

Chapter Topics

Scala Overview

- **Introducing Scala**
- Scala's Role in Distributed Data Processing
- The Motivation for Scala
- Essential Points

What is Scala?

- **Martin Odersky started developing Scala in 2001**
  - Sca (scalable) La (language)
- **Design goals**
  - Eliminate common distributed processing bugs
  - Assign scale decisions to the programming framework
  - Enable the programmer to take control when necessary
- **Strongly influenced by *functional programming* style**
  - Pure functions
  - Immutable data
  - Implicit looping and iteration

Scala and Java

- **Scala is a superset of Java**
  - Scala compiles to Java bytecode
  - Runs on the Java Virtual Machine (JVM)
- **It is interoperable with Java programs and libraries**
  - All Java programs can be ported to Scala
  - Some Scala programs, however, cannot directly be ported to Java
- **Scala is more expressive than Java**
  - Concise code
  - Implicit processing

Features and Benefits of Scala

- **Strong typing in Scala allows many errors to be caught at compile time**
  - Such errors aren't evident until runtime in some languages
- **Integration with popular programming tools**
  - Integrated Development Environments (IDEs)
  - Unit test frameworks
- **Support for multiple programming paradigms, including**
  - Imperative
  - Object-oriented
  - Functional

The Promise of Functional Programming

- **Goals of functional programming languages like Scala, LISP, and Haskell**
  - Fewer programming errors
  - Better modularity
  - Higher-level abstractions
  - Shorter, more concise code
  - Increased developer productivity

Functional Programming Adoption

- **Functional programming has long been popular in academia**
  - It is now also becoming popular in industry
- **Software complexity was the catalyst needed for adoption**
  - Driven by need for multi-core and distributed programming

Software Challenges

- **Triple challenge**
  1. Parallel - How to make use of multicores, GPUs, clusters?
  2. Async - How to deal with asynchronous events?
  3. Distributed - How to deal with delays and failures?
- **Mutable state is a liability in each of the above challenges**
  - Scala encourages the use of immutable variables

Chapter Topics

Scala Overview

- Introducing Scala
- **Scala's Role in Distributed Data Processing**
- The Motivation for Scala
- Essential Points

Programming at Scale

- **Some enterprises still process large amounts of data serially**
  - Divide the data into subsets
  - Process a subset locally
  - Repeat the process until all the data has been processed
  - This painfully slow approach is driven by tools that do not scale
- **What would a system look like that could scale?**
  - Write the program one time and never have to re-engineer it
  - The same program would run
    - On one CPU or on thousands of CPUs
    - Serially on one server or in parallel across a cluster of servers

The Spark Distributed Processing Framework

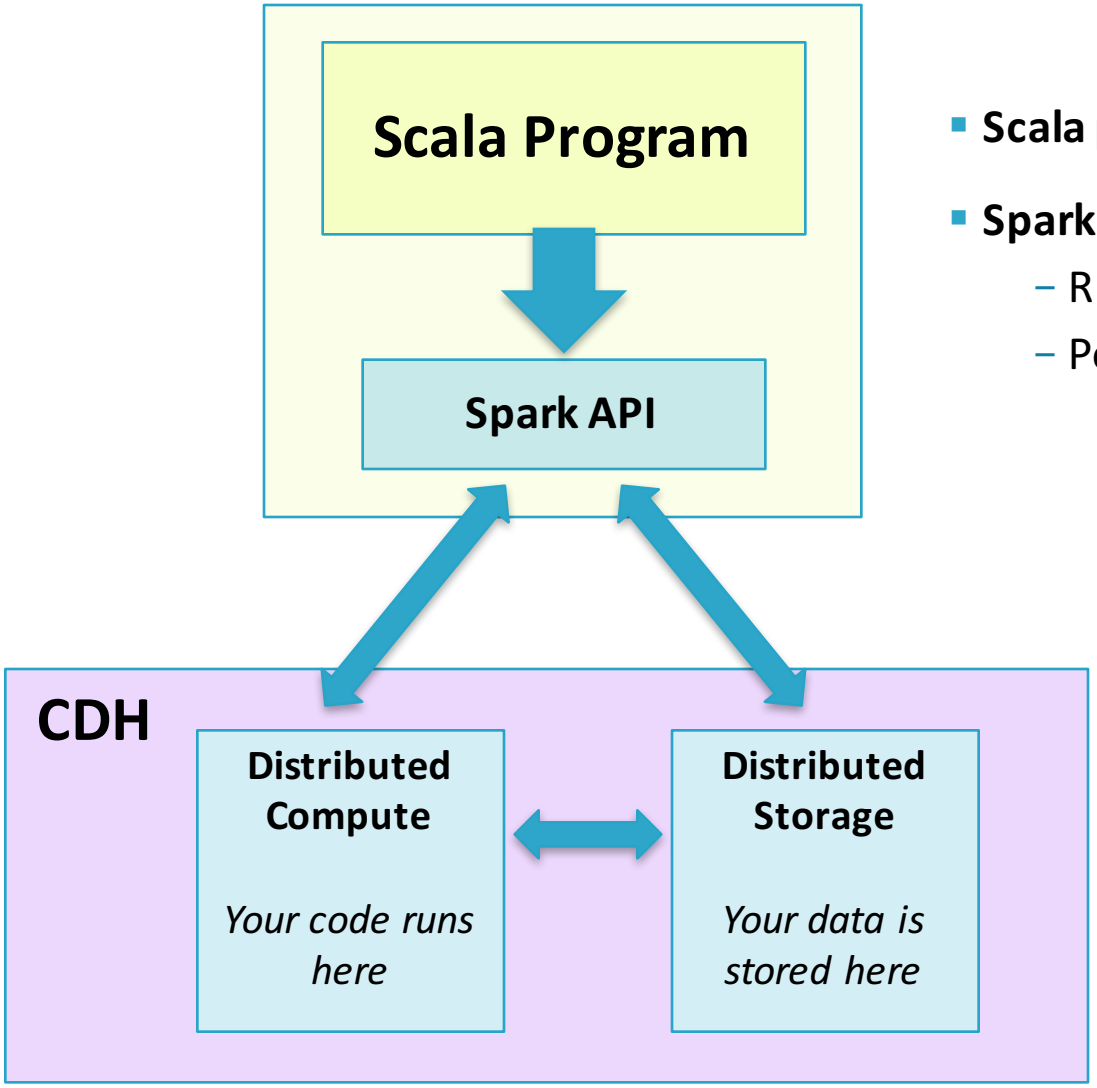
- **Apache Spark is an open source *distributed processing framework***
  - Includes an engine for parallel processing of data across many servers
  - Provides an elegant model for writing concise programs
- **The Spark framework parallelizes the code you write using its APIs**
  - Distributes and executes that code across a group of servers
  - This group of servers is known as a *cluster*
  - Spark scales linearly as more servers are added to the cluster
- **Apache Spark originated at UC Berkeley AMPLab**
  - It was contributed to the Apache Software Foundation

Spark in a Nutshell

- **Programs written in Spark can process huge amounts of data efficiently**
  - The underlying storage system is commonly HDFS
  - The Spark framework is fault tolerant and resilient
- **Spark's in-memory approach to processing provides a performance boost over other distributed frameworks**
- **Spark runs on top of the Java Virtual Machine (JVM)**
  - This makes it possible to use debugging tools built for the Java stack

HDFS: Hadoop Distributed File System

How Scala and Spark Fit Together



- **Scala programs call the Spark API**
- **Spark API talks to the cluster to**
  - Run programs
  - Perform file I/O

Chapter Topics

Scala Overview

- Introducing Scala
- Scala's Role in Distributed Data Processing
- **The Motivation for Scala**
- Essential Points

Overview of Big Data Analysis

- **Successful analysis of Big Data requires many steps**
  - Pre-processing the data to clean and format it (ETL)
  - Profiling data to understand the characteristics of the dataset
  - Iterating over data to achieve the desired results
  - Rebuilding analytical models as needs change
  - Processing all of the data, rather than being limited to a subset
- **Apache Spark helps to support these needs**
  - Scala is widely used for writing Spark applications

ETL: Extract, Transform, and Load

ETL and Pre-Processing Data

- **We can use Scala code in Spark for ETL and other pre-processing tasks**
  - Clean data
  - Format data
  - Fuse columns
  - Convert data

Profiling Data

- **We profile data with Scala and Spark because**
  - Big Data is too large for a human to inspect manually
  - Data can be messy, since it may lack rigid structure
  - It helps us understand the nature of the data in each column
  - It is useful to determine the ranges or length of values in each column

Iterating over Data

- **Spark is efficient for algorithms that iterate over data**
  - Modeling and analysis may require multiple passes for the same data
  - Some algorithms require iteration in order for results to converge
  - Features in Scala support writing iterative programs for Spark

Rebuilding Models

- **Analytical models are used in services that inform real-world decisions**
- **Coding models in Scala for Spark provides flexibility**
  - Models can easily be revised to adapt to changing conditions
  - They may be rebuilt periodically, even in near real-time

Putting Solutions into Production

- **Historically, experiments were performed using a language like R**
  - The resulting solutions were re-written in C++ or Java for production
- **Goal: deploy code used for running experiments directly to production**
  - Spark makes this possible

Why Write Your Spark Application in Scala?

- **Spark is written in Scala, making Scala a top-tier language for Spark**
  - Performance benefits in using Spark's native language
  - New Spark features are typically available for Scala well before Python
- **Using Scala helps you to understand the Spark philosophy**
  - Leads to using the Spark platform more effectively
- **You will learn to write programs in Scala during this course**
  - Another course will teach you to use Scala code to call the Spark APIs

One Framework for Exploration *and* Operation

- **Code for Spark is most often written in Scala or Python**
  - Spark provides APIs for both of these languages
- **Spark supports both exploratory and operational analytics**
- **Use the Spark REPL, or shell, for exploratory analytics**
  - Allows you to run code interactively
- **Compile Spark code for operational analytics**
  - Write production-grade analytics in Spark

REPL: Read-Evaluate-Print Loop

What about Java and Spark?

- **Spark also provides a Java API, but there is no Java-based REPL**
- **Using the same language for exploration and operational deployment is advantageous**
  - This advantage is available with Scala and Python
  - Java only supports operational deployment

Chapter Topics

Scala Overview

- Introducing Scala
- Scala's Role in Distributed Data Processing
- The Motivation for Scala
- **Essential Points**

Essential Points

- **Scala is a scalable language that runs in the Java Virtual Machine (JVM)**
  - Interoperates with Java, but has a much more concise syntax
  - Supports imperative, object-oriented, and functional programming
  - Designed to help developers write better code in less time
- **Can be used for both exploratory and operational analytics**
  - Scala's REPL, or shell, is useful for interactive work
  - You can also compile and package Scala code, as with Java
- **Apache Spark is an open source distributed processing framework**
  - Spark is written in Scala
  - Provides Scala APIs for developing your own Spark programs
  - Allows your program to scale without the need to change your code

cloudera®

Scala Basics

Chapter 3

Course Chapters

- Introduction
- Scala Overview
- **Scala Basics**
- Working with Data Types
- Grouping Data Together
- Flow Control in Scala
- Using and Creating Libraries
- Conclusion

Scala Basics

In this chapter you will learn

- **Why immutable variables are preferred over mutable variables**
- **How to launch the Scala shell**
- **How to compile and execute Scala programs**
- **How to declare and assign variables**
- **How to perform basic I/O operations in Scala**

Chapter Topics

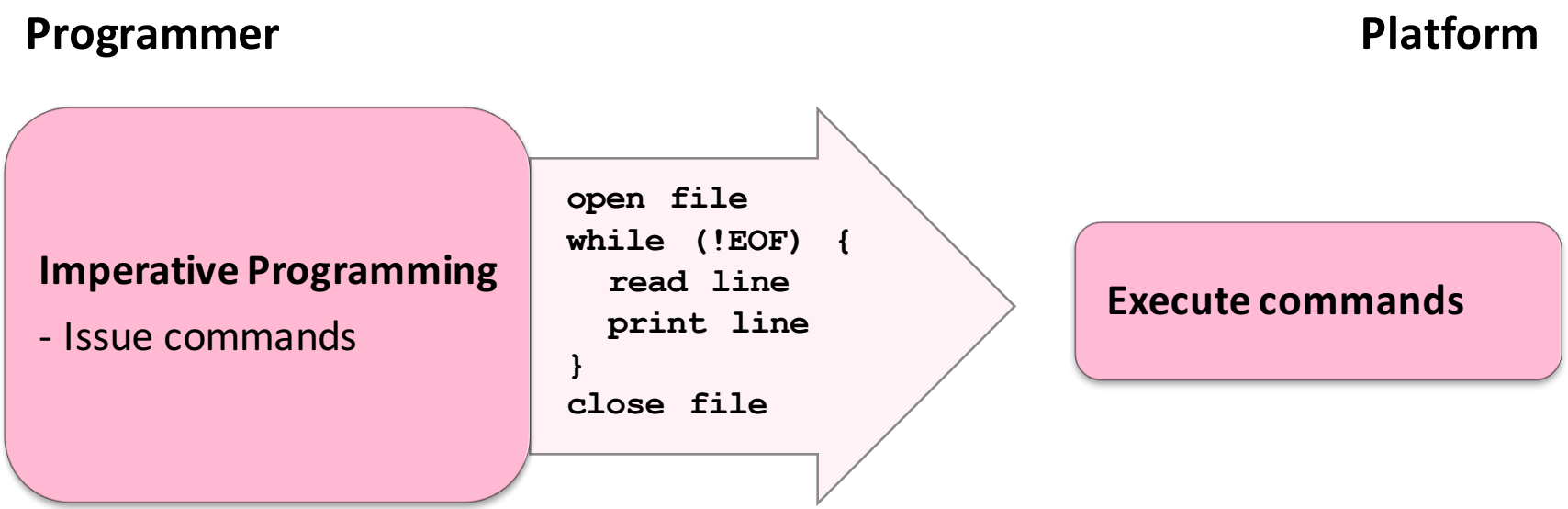
Scala Basics

- **Key Scala Concepts**
- Programming in Scala
- Putting Scala Basics to Work
- Essential Points
- Hands-On Exercise: Working with the VM and Scala Shell
- Hands-On Exercise: Performing Basic Input/Output

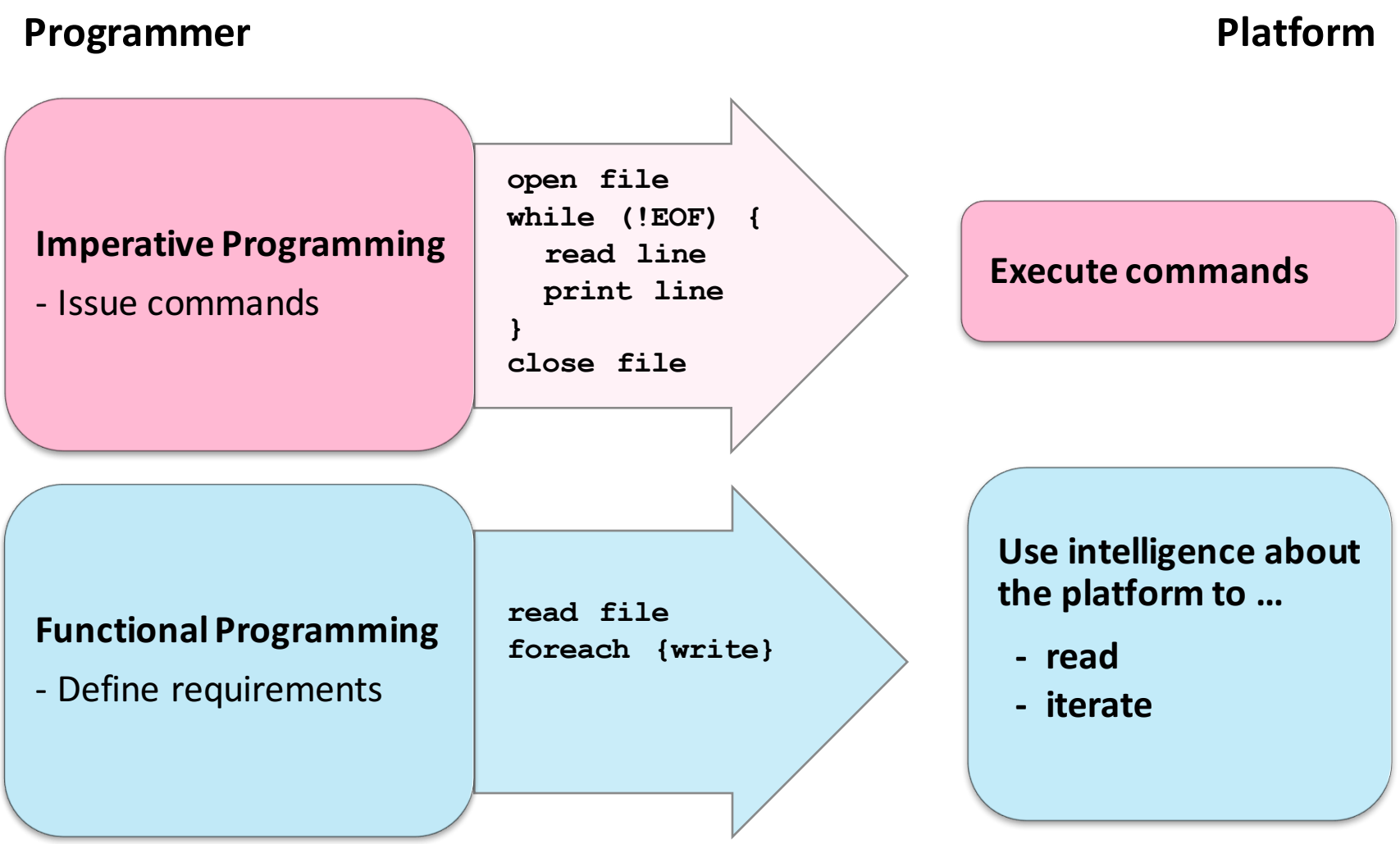
Key Scala Characteristics

- **Scala code is written as code blocks**
  - Scala code blocks are expressions that return a value
  - In Scala, all code-blocks return a value, even if the value is “nothing”
- **Functions are fundamental to Scala**
  - They can take the place of a typical variable
- **Functions can be passed as parameters to other functions**
  - Example: **`function1(function2)`**
    - The call to **`function1`** passes **`function2`** as a parameter
    - **`function1`** is called a *higher-order function*

Why Functional Programming? (1)



Why Functional Programming? (2)



Functional Programming Features

- **Functional programming languages like Scala**
  - Are inspired by math
  - Encourage *pure* functions
  - Discourage functions with *side effects*

What are Pure Functions?

- **A pure function**
  - Consistently computes the same result given the same inputs
  - Returns a result determined only by its input values
  - Has no observable side effects, such as mutation or output of data
  - Provides deterministic results, making it easy to write test cases

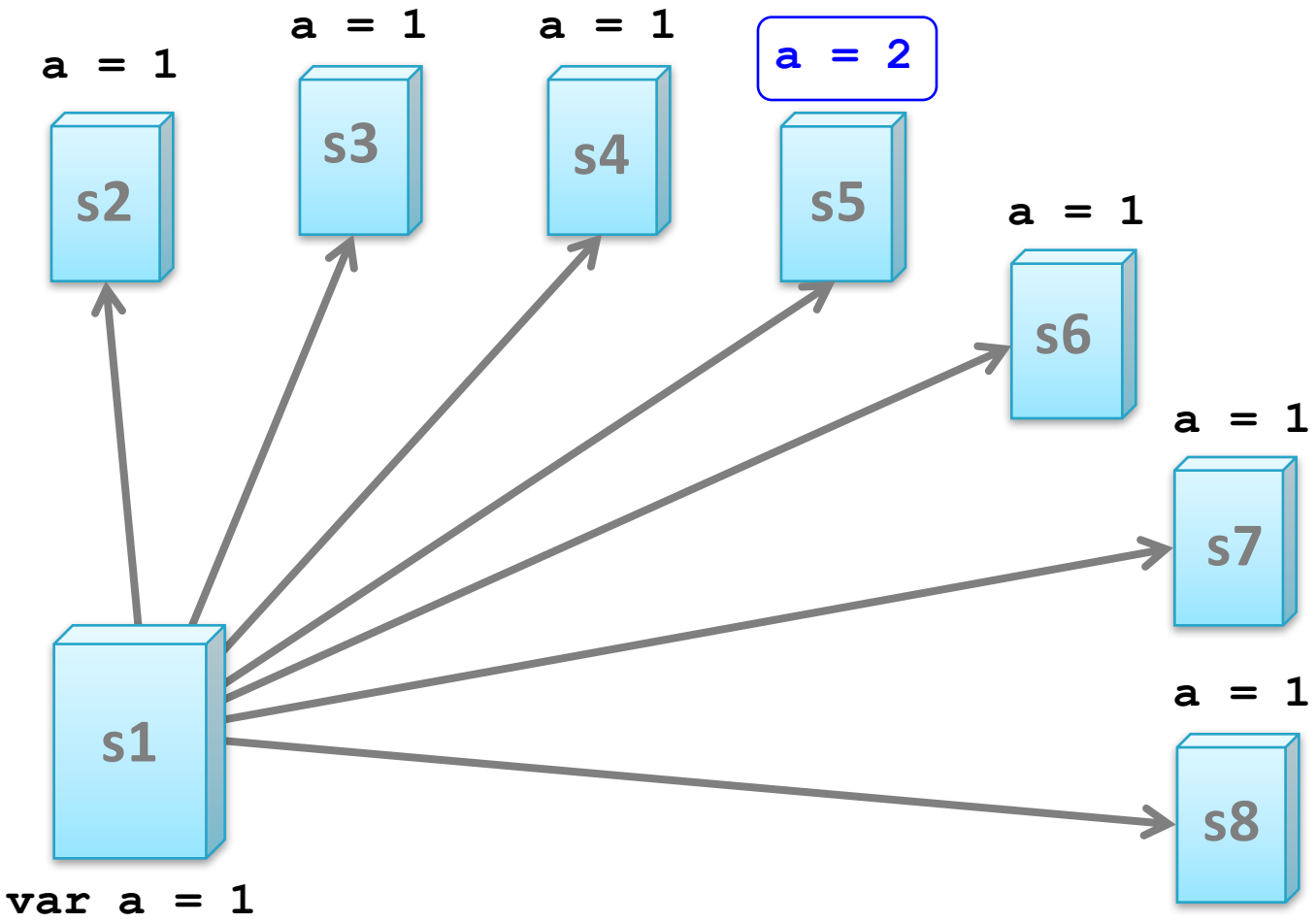
What are Side Effects and What Causes Them?

- **Pure functions do not have side effects**
- **A function causes side effects if it**
  - Modifies state, such as state of a global variable or argument
  - Raises an exception
  - Writes data to a display or file
  - Reads data
  - Calls other functions that cause side effects
- **In the presence of side effects, program behavior and results may depend on order of evaluation**

Differences between Immutable and Mutable Variables

- **Scala variables are declared as either mutable or immutable**
  - **`val`** (value) – immutable
  - **`var`** (variable) – mutable
- **Pure functions are preferred**
  - Use immutable variables whenever possible
  - Scala controls scope to limit changes to mutable data

Distributed Processing and Mutable Variables



Using Implicit Iteration to Enable Scalability

- **How does Scala make iteration over the elements in a collection scalable?**
- **Scala uses a method uncommon in other languages**
  - It creates *implicit variables* to pass data between parts of an expression
  - Eliminates the need to write your own counters and state variables
- **Removes a common source of bugs in distributed systems**
  - These implicit variables cannot be addressed or changed
- **Allows the framework to optimize execution**
  - The framework can determine *how* the iteration is implemented

Imperative Programming vs. Functional Programming

Imperative programming example (pseudocode)

```
file = open("loudacre.log")

while not EOF {
  line = file.readline()
  print line
}

file.close()
```

Scala Characteristics

- **In Scala, flow is implied from the order of chained expressions**
  - Each of these is connected by a period
- **Iteration over a collection is often implicit, as seen with `foreach`**

Functional programming example with Scala

```
Source.fromFile("loudacre.log").foreach(print)
```

Chapter Topics

Scala Basics

- Key Scala Concepts
- **Programming in Scala**
- Putting Scala Basics to Work
- Essential Points
- Hands-On Exercise: Working with the VM and Scala Shell
- Hands-On Exercise: Performing Basic Input/Output

Formatting Conventions of Code Examples

- **Code examples are shown on a blue background**
- **The code you enter is displayed without any prompt in black type**
- **The response is shown with a > prompt and in blue type**

```
val s = "Titanic 4000"
println(s)

> Titanic 4000
```

Use Scala in the Shell or Compile Your Program

- **You can work with Scala in two ways**
  1. Interactively, using the Scala shell
  2. By executing compiled Scala programs, similar to working with compiled Java programs
- **You will learn how to work with both approaches in the slides that follow**

Using the Scala Shell

- **The Scala shell is referred to as a REPL (Read-Evaluate-Print Loop)**
  - When you enter an expression, Scala immediately evaluates it, assigns the value to an implicit variable, and prints it to the console
- **Start the Scala shell from the OS command line by typing: `scala`**

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.7.0_67).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Interacting with the Scala Shell

- **The Scala shell offers *tab completion***
  - Type a partial keyword, class, method, or variable name
  - Hit the [TAB] key to see possible completions
- **Commands used by the Scala shell begin with a colon**

Command	Description
`:help`	Get a list of commands or help on a specific command
`:history`	Show previous commands
`:h? string`	Search the command history for ***string***
`:quit`	Exit the shell
`:sh`	Run a shell command or script in the operating system shell
`:load path`	Load and execute Scala source code from a file at ***path***

Result Variables in the Scala Shell

- **In this example, we define a variable of type integer**
- **Scala creates result variables automatically**
  - **`res0`, `res1`, `res2`, … `resN`**

```
val myInt: Int = 123
myInt: Int = 123

myInt
res0: Int = 123

println(res0)
123
```

Basic Keyboard Input

- **The `scala.io.StdIn.readLine` function reads input from the user**
  - Data typed by the user, shown here in gray, isn't echoed to the terminal

```
var str: String = scala.io.StdIn.readLine("Enter:\n")
> Enter:
Titanic 4000

> str: String = Titanic 4000

print(str)
> Titanic 4000
```

Basic Printing

- **Use the built-in `print()` function**
  - Basic printing of variables: **`print(var1, var2, var3, ...)`**

```
val s = "Sorrento F41L"
print(s)
> s: String = Sorrento F41L
```

Formatted Printing

- **Print formatted string:**
  - ***formatStrName*.format(*var1*, *var2*, *var3*…)**
  - Formatting: **%d** for decimal integer, **%f** for float

```
val formatStr = "Temperature range is %d to %f celsius"

print(formatStr.format(24, 31.24))
> Temperature range is 24 to 31.240000 celsius
```

Basic File I/O

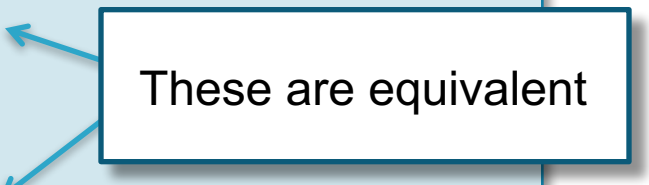
- **Import the `scala.io.Source` library**
- **Use `Source.fromFile`**
  - Returns a variable of type **`scala.io.BufferedSource`**

```
import scala.io.Source

val filename: String="loudacre.log"

val buffer = Source.fromFile(filename)
buffer.foreach(print)

Source.fromFile(filename).foreach(print)
```



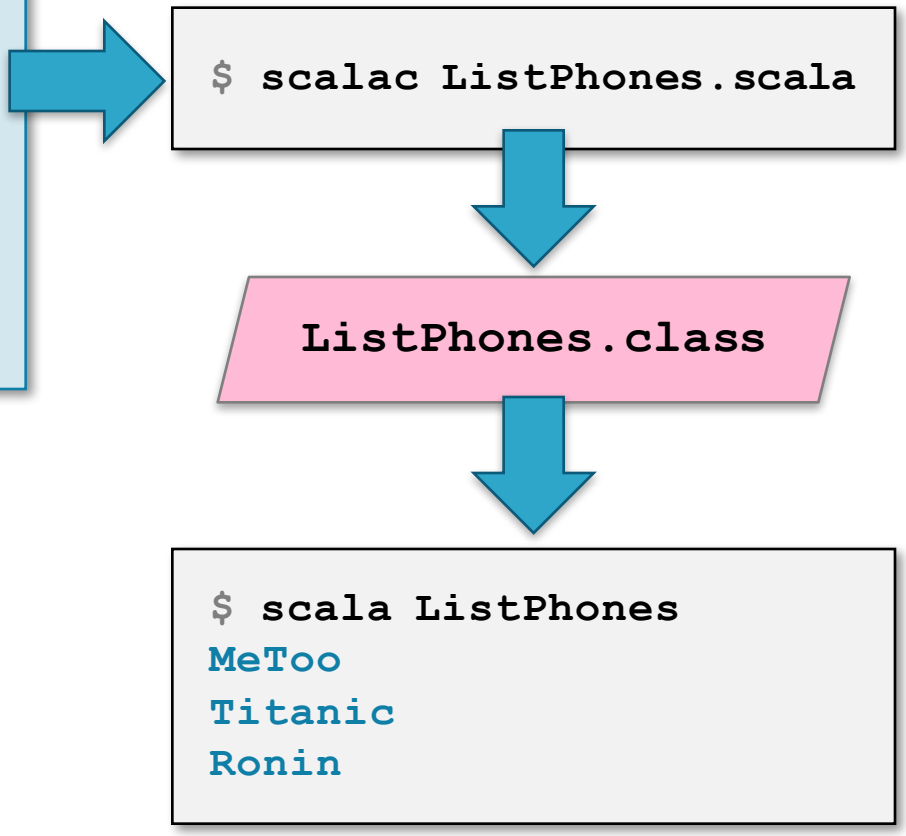
Compiling and Executing Scala Programs

- **Edit code in a file using a text or graphical editor, or an IDE such as Eclipse**
  - The program file must declare an **`object`** with a **`main`** method
  - Scala files have a **`.scala`** extension
- **Compile to a `.class` file with `scalac` command**
- **Execute the class using the `scala` command**

Compiling and Executing Scala Programs: Step by Step

File: `ListPhones.scala`

```
object ListPhones {
  def main(args: Array[String])
  {
    println("MeToo")
    println("Titanic")
    println("Ronin")
  }
}
```



Chapter Topics

Scala Basics

- Key Scala Concepts
- Programming in Scala
- **Putting Scala Basics to Work**
- Essential Points
- Hands-On Exercise: Working with the VM and Scala Shell
- Hands-On Exercise: Performing Basic Input/Output

Accessing Data in Files

- **The data for this course is in a directory that you can easily access using the Linux environment variable, `$SCADATA`**
- **Use `$SCADATA` in your Scala program with the `envOrElse` method in the Scala `Properties` library**
- **In the example below, we set a Scala variable named `datadir` to the path in `$SCADATA`**
  - If **`$SCADATA`** is undefined we set **`datadir`** to the current directory

```
val datadir =
scala.util.Properties.envOrElse("SCADATA", ".")
```

Declaring a Variable

- **We did not specify a type for the variable**
  - Based on the context, Scala considers `datadir` to be a `String`
  - This feature of Scala is known as *type inference*

```
val datadir =
scala.util.Properties.envOrElse("SCADATA", ".")

> datadir: String =
/home/training/training_materials/jes/data
```

Basic I/O with Scala (1)

- Scala comes with many libraries
- Use the `import` command to access the `Source` library

```
import scala.io.Source
```

- Use the `fromFile` method in the `Source` library to get a reference to the Loudacre log file
- Iterate through each line using the `foreach` method
- Output each line of the file using the `print` method

```
Source.fromFile(datadir + "/loudacre.log").foreach(print)
```

Basic I/O with Scala (2)

- The output from the `foreach(print)` command is shown below
- You will work with this data in the course exercises

```
2014-03-15:10:10:20,iFruit 1,6474caf1-7bbf-4594-a526-
9ba8ea82e151,0,15,71,77,0,40,TRUE,enabled,connected,37.9
0310537,-121.5614513

2014-03-15:10:10:20,iFruit 2,5c2d40d8-b1e0-4c2a-b050-
06ca6c590741,1,28,66,67,40,49,TRUE,disabled,connected,34
.12789546,-108.9681595

2014-03-15:10:10:20,iFruit 3,27178d24-3a61-42f7-a784-
e3263f25cc6f,1,30,91,89,41,17,TRUE,enabled,enabled,37.92
489617,-122.2068682
...
```

Chapter Topics

Scala Basics

- Key Scala Concepts
- Programming in Scala
- Putting Scala Basics to Work
- **Essential Points**
- Hands-On Exercise: Working with the VM and Scala Shell
- Hands-On Exercise: Performing Basic Input/Output

Essential Points

- **Scala has higher-order functions**
  - These are functions that can accept functions as parameters
- **Use immutable variables and functional programming as much as possible to reap the benefits of Scala**
- **Scala has many libraries, including `scala.io.Source` which supplies methods for file access**

Bibliography

The following offer more information on topics discussed in this chapter

- **Official Scala Web site**
  - `http://scala-lang.org/`
- ***Programming Scala***
  - `http://tiny.cloudera.com/yeoeg`
- ***Scala in Action***
  - `http://tiny.cloudera.com/inhbd`
- ***Scala for the Impatient***
  - `http://tiny.cloudera.com/vsboh`
- ***Functional Programming Principles in Scala***
  - `https://www.coursera.org/course/progfun`

Chapter Topics

Scala Basics

- Key Scala Concepts
- Programming in Scala
- Putting Scala Basics to Work
- Essential Points
- **Hands-On Exercise: Working with the VM and Scala Shell**
- Hands-On Exercise: Performing Basic Input/Output

Hands-On Exercise: Working with the VM and Scala Shell

- **In this exercise, you will learn to start the course VM and learn to use the Scala shell interactively**
  - Please refer to the Hands-On Exercise Manual for instructions
  - Be sure to read the General Notes section found at the beginning of the Hands-On Exercise Manual before continuing with the exercise

Chapter Topics

Scala Basics

- Key Scala Concepts
- Programming in Scala
- Putting Scala Basics to Work
- Essential Points
- Hands-On Exercise: Working with the VM and Scala Shell
- **Hands-On Exercise: Performing Basic Input/Output**

Exercise Introduction: Loudacre Mobile

- **Data used in this course follows a theme**
- **Loudacre Mobile is a (fictional) wireless carrier**
  - Phones that customers buy from Loudacre send diagnostic information back to the company
  - Loudacre ingests this data into their Hadoop cluster
- **By analyzing this data, the company can**
  - Enable support staff to better help customers
  - Notify technicians of equipment failure
  - Identify where additional cell towers are needed
- **You'll use a sample of this data during hands-on exercises**

Hands-On Exercise: Performing Basic Input/Output

- **In this exercise, you will use the Scala shell to execute commands for performing basic input and output**
  - Please refer to the Hands-On Exercise Manual for instructions

cloudera®

Working with Data Types

Chapter 4

Course Chapters

- Introduction
- Scala Overview
- Scala Basics
- **Working with Data Types**
- Grouping Data Together
- Flow Control in Scala
- Using and Creating Libraries
- Conclusion

Working with Data Types

In this chapter you will learn

- **How to create variables without explicitly declaring types**
- **How to define and work with mutable and immutable variables**
- **How to use built-in methods and functions that will save time in processing numeric variables, booleans, and strings**

Chapter Topics

Working with Data Types

- **Overview of Scala Variables**
- Operating with Numeric Types
- Building Boolean Expressions
- Working with Strings
- Essential Points
- Hands-On Exercise: Exploring Scala Variables and Typing

Properties of Scala Variables: Mutability

- **Variables must be initialized when declared**
- **Variables are either *mutable* or *immutable***
  - Mutable: can reassign a value of the same type
    - Syntax: **`var name: type = value`**
  - Immutable: value cannot be reassigned after initialization
    - Syntax: **`val name: type = value`**

Example: Attempting to reassign to an immutable variable

```
val phoneModel: Int = 3
> phoneModel: Int = 3

phoneModel = 4
> error: reassignment to val
```

Properties of Scala Variables: Type (1)

- **Types may either be explicitly declared or inferred**
  - Scala makes a best guess based on assignment
  - You can also explicitly declare the type

Example: Type inference

```
var phoneModel = 3
> phoneModel: Int = 3
```

Example: Explicit typing

```
var phoneModel: Short = 3
> phoneModel: Short = 3
```

Properties of Scala Variables: Type (2)

- **Variables are statically typed**
  - Scala does not support dynamic typing
  - The type is established on first use and never reassigned
  - Using the same variable name with a different type will cause an error

Example: Attempt to reassign type

```
var phoneModel = 3
> phoneModel: Int = 3

phoneModel = "iFruit 9000"
> error: type mismatch;
   found   : String("iFruit 9000")
   required: Int
```

Redefining Variables

- **Although you cannot reassign an immutable variable, or assign a new type to any defined variable, you can redefine a variable**

Example: Redefining a variable

```
var phoneModel = 3
> phoneModel: Int = 3

phoneModel = "iFruit 9000"
> error: type mismatch;
   found   : String("iFruit 9000")
   required: Int

var phoneModel = "iFruit 9000"
> phoneModel: String = iFruit 9000
```

Some Important Types

- **The table below shows examples of common data types in Scala**

Type	Description	Example
`Byte`	8-bit signed integer	`3`
`Short`	16-bit signed integer	`32`
`Int`	32-bit signed integer	`327`
`Long`	64-bit signed integer	`32754L`
`Double`	8-byte floating point	`3.1415`
`Float`	4-byte floating point	`3.1415F`
`Char`	Single character	`'c'` (single quotes)
`String`	Sequence of characters	`"iFruit"` (double quotes)
`Boolean`	Either `true` or `false`	`true` (case sensitive)

Special `Unit` Type

- **`Unit`**
  - When a function passes back "nothing" in Scala, it passes back **`Unit`**
  - This is equivalent to the **`void`** return type in a Java method
  - There is only one **`Unit`** in Scala; it is non-instantiable

```
val myreturn = println("Hello, world")
> myreturn: Unit = ()
```

Special **`Any`** Type and Explicit Casting of Type

- **`Any`**
  - Used when Scala cannot determine which specific type to use
  - Can be cast to a specific type using the method **`asInstanceOf[`*`type`*`]`**

```
val myreturn = if (true) "hi"
> myreturn: Any = hi

val mystring = myreturn.asInstanceOf[String]
> mystring: String = hi
```

Interrogating Variables with `getClass`

- **Use `getClass` to determine the class of an object**
- **The code below also shows that `getClass` is an example of an *arity-0* function**
  - **The *arity* of a function is the number of parameters it accepts**

```
val PhoneStyle: Char = 'd'
> PhoneStyle: Char = d

PhoneStyle.getClass
> Class[Char] = char

PhoneStyle.getClass()
> Class[Char] = char
```

Arity-0 methods that are purely functional (cause no side effects), can be called without empty parentheses

Calling with empty parentheses will also work

Chapter Topics

Working with Data Types

- Overview of Scala Variables
- **Operating with Numeric Types**
- Building Boolean Expressions
- Working with Strings
- Essential Points
- Hands-On Exercise: Exploring Scala Variables and Typing

Numeric Variables and Arithmetic

- **Scala does not have operators, per se**
  - It uses *operator notation* to invoke methods
- **In most cases, Scala determines operator precedence based on the first character of the methods used in operator notation**
  - For example, the ***** character has higher precedence than **+**

```
val ab = 1 + 2 * 3
> ab: Int = 7
```

- **Scala supports assignment operators**
  - **`ab += bc`** is equivalent to **`ab = ab + bc`**

Operator Precedence (1)

- **The table shows precedence in decreasing order**
  - Precedence is based on the first character of a method
- **Characters on the same line have equal precedence**

Operator Precedence
(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

Operator Precedence (2)

- **Exception to the precedence rule**
  - Concerns assignment operators, which end in an equals character
  - If an operator ends with = and the operator is not one of the comparison operators, then the precedence of the operator is the same as that of the simple assignment operator, =
- **Example:**
  - `x *= y + 1` is equivalent to

```
x *= (y + 1)
```

Operator Precedence
(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

Scala's `math` Library

- **Scala provides no standard operator for exponentiation**
  - Use the **`math`** library

```
math.pow(3, 2)
> Double = 9.0
```

Scala Division

- **Scala integer division**

```
val quotientInt = 7 / 5
> quotientInt: Int = 1
```

- **Getting the remainder of integer division**

```
val remainder = 7 % 5
> remainder: Int = 2
```

- **Scala float division**

```
val quotientFloat = 7.0 / 5.0
> quotientFloat: Double = 1.4
```

Implicit Numeric Conversion

- **Scala automatically performs type conversions for numeric operations when operands are of different types**

```
val tempCelsius = 35
> tempCelsius: Int = 35

val tempFahrenheit = 9.0 / 5.0 * tempCelsius + 32
> tempFahrenheit: Double = 95.0
```

Explicit Numeric Conversion

- **Some Scala types provide methods for converting between types**

Example: Casting a **`Double`** to an **`Int`**

```
val fahrInt: Int = tempFahrenheit.toInt
> fahrInt: Int = 95
```

Example: Casting an **`Int`** to a **`Double`**

```
val iLat = 331913
> iLat: Int = 331913

val dLat = iLat.toDouble * 1000
> dLat: Double = 3.31913E8
```

Chapter Topics

Working with Data Types

- Overview of Scala Variables
- Operating with Numeric Types
- **Building Boolean Expressions**
- Working with Strings
- Essential Points
- Hands-On Exercise: Exploring Scala Variables and Typing

Using `Booleans` to Control Program Flow

- **Boolean variables are used to control program flow**
  - Such as branching, conditional execution, and looping
- **Boolean variables can be set to `true` or `false`**
  - Lower case **`true`** and **`false`** only

```
val gpsStatus: Boolean = false
val gpsStatus = true
```

- *Creates a Boolean variable*

```
val gpsStatus = "true"
```

- *Creates a string of characters* ***t-r-u-e***

```
val gpsStatus = True
<Error>
```

- ***True*** *and* ***TRUE*** *are not Boolean literals*

Performing Relational, Logical, and Bitwise Operations

- **You can mix numeric types**
  - For example, the result of `1 == 1.0` is `true`

Relational Operators	**<**	**>**	**<=**	**>=**	**==**	**!=**
Logical Operators	**&&** (and)	**||** (or)				
Bitwise Operators	**&** (and)	**|** (or)	**^** (xor)			

Chapter Topics

Working with Data Types

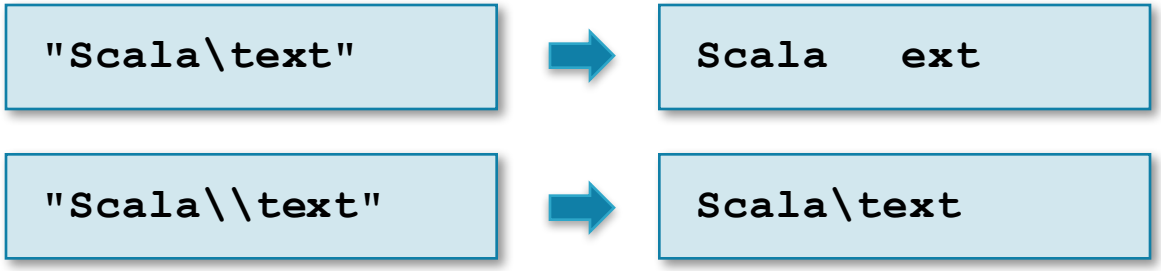
- Overview of Scala Variables
- Operating with Numeric Types
- Building Boolean Expressions
- **Working with Strings**
- Essential Points
- Hands-On Exercise: Exploring Scala Variables and Typing

Strings with Special Characters

- **In Scala, string literals are enclosed in double quotes**
- **Escape special character literals using backslash (\)**

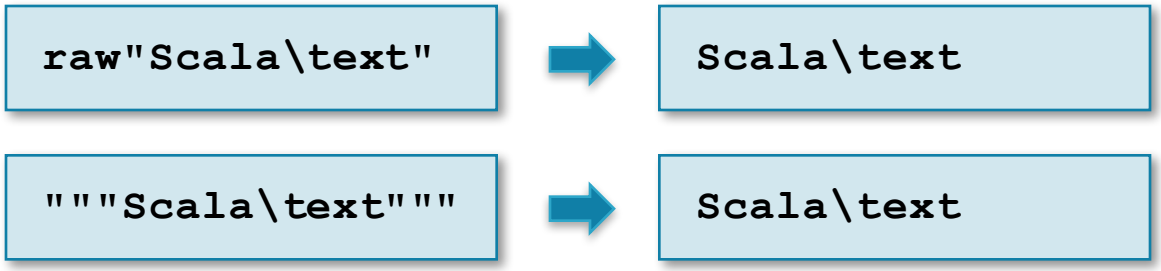
Escape Sequence	Corresponding Character Literal
`\t`	Tab
`\n`	Newline
`\b`	Backspace
`\r`	Carriage return
`\\`	Backslash

- **Examples:**



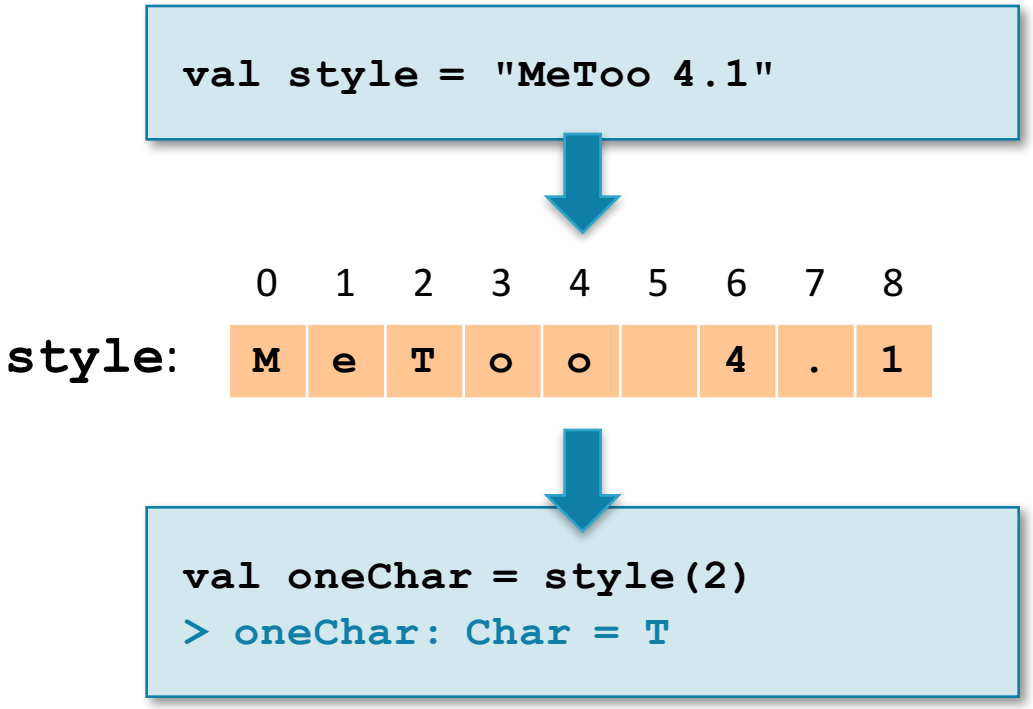
Controlling String Interpretation

- **Use `raw` to prevent Scala from interpreting a backslash as an escape**
- **Alternately, you can use three double quotes to do this**
- **Examples:**



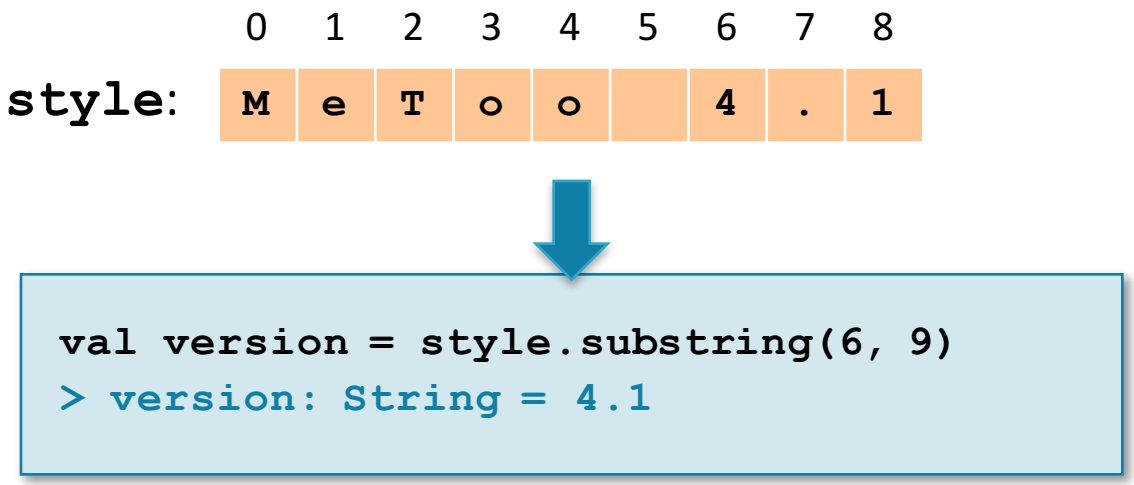
Getting a Character from a String with Indexing

- **`String(offset)`**
  - Returns the character at the specified offset
  - The offset is zero-based



Getting a Slice of a String Using `substring`

- **Use `String.substring(from, until)` to extract a substring**



Time-Saving Methods for Processing Strings

- **There are many methods currently defined for the `String` type**
- **Some examples are shown below**

```
val s = "bananas, apples, and oranges"

s.sorted
> String = "   ,,aaaaaabdeeglnnnnopprsss"

s.toUpperCase
> String = BANANAS, APPLES, AND ORANGES

s.toArray
> Array[Char] = Array(b, a, n, a, n, a, s, ,,  , a, p, p,
l, e, s, ,,  , a, n, d,  , o, r, a, n, g, e, s)
```

Splitting Strings

- **Scala provides multiple ways to split strings**

```
val s = "bananas, apples, and oranges"

s.splitAt(4)
> (String, String) = (bana,nas, apples, and oranges)

s.split(',')
> Array[String] = Array(bananas, " apples", " and oranges")
```

Comparing Strings

- **In Scala, you test equality with the == method**

```
val s1 = "bananas"
val s2 = "oranges"
val s3 = "bananas"

s1 == s2
> Boolean = false

s1 == s3
> Boolean = true
```

Chaining Methods

- In Scala, methods can be *chained*
- Example: the output from the `sorted` method are passed as input to the `toUpperCase` method

```
val device: String = "titanic 2300"
> titanic 2300

device.toUpperCase
> "TITANIC 2300"

device.sorted
> "0023aciintt"

device.toUpperCase.sorted
> "0023ACIINTT"
```

Listing Methods in the Shell

- **Enter a literal, value, or variable followed by a period, then press `TAB` to see available methods**

```
val phoneName = "Ronin"
> phoneName: String = Ronin

phoneName.[TAB]
+                 endsWith              replace
asInstanceOf      equalsIgnoreCase      replaceAll
charAt            getBytes              replaceFirst
chars             getChars              split
 …
```

Substituting Variables in Output

- **Let's define some variables to use:**

```
val phoneName = "Titanic"
val phoneTemp = 35
```

- **Precede a string with *s* to substitute the value of the named variable**

```
println(s"Name: $phoneName")
> Name: Titanic

println(s"Name: $phoneName", s" Temp: $phoneTemp")
> (Name: Titanic, Temp: 35)
```

Formatting Output Using Format Strings

- **Given:**

```
val phoneTemp = 46
```

- **Use `f` to format the string using C language-style format strings**

```
println(f"Temp: $phoneTemp%f")
> Temp: 46.000000

println(f"Temp: $phoneTemp%.2f")
> Temp: 46.00

println(f"Temp: $phoneTemp%h as hex")
> Temp: 2e as hex
```

Format Strings

- `%c` - character
- `%s` - string
- `%d` - decimal
- `%e` - exponential
- `%f` - floating point
- `%i` - integer
- `%o` - octal
- `%h` - hexadecimal

Chapter Topics

Working with Data Types

- Overview of Scala Variables
- Operating with Numeric Types
- Building Boolean Expressions
- Working with Strings
- **Essential Points**
- Hands-On Exercises: Exploring Scala Variables and Typing

Essential Points

- **Explicit definition of variables is required in Scala**
  - Variables are either mutable (**`var`**) or immutable (**`val`**)
  - Immutable variables are preferred whenever mutability is not required
- **Scala supports inferred typing but types can be specified explicitly**
- **The `getClass` method identifies a variable's type**
- **Boolean values are `true` and `false`**

Bibliography

The following offer more information on topics discussed in this chapter

- ***Scala: The Static Language that Feels Dynamic***
  - **`http://tiny.cloudera.com/qodsz`**
- **Information on the Scala type `Unit`, which is similar to `void` in Java**
  - **`http://tiny.cloudera.com/mbgri`**
- **Information on method invocation in Scala**
  - **`http://tiny.cloudera.com/otezp`**
- **Information on string operations in Scala**
  - **`http://tiny.cloudera.com/ipkqn`**
- **Information on string interpolation in Scala**
  - **`http://tiny.cloudera.com/xxxbt`**

Chapter Topics

Working with Data Types

- Overview of Scala Variables
- Operating with Numeric Types
- Building Boolean Expressions
- Working with Strings
- Essential Points
- **Hands-On Exercise: Exploring Scala Variables and Typing**

Hands-On Exercise: Exploring Scala Variables and Typing

- **In this exercise, you will interactively explore variables in the Scala shell and write a program to extract fields from a single data record**
  - Please refer to the Hands-On Exercise Manual for instructions

cloudera®

Grouping Data Together

Chapter 5

Course Chapters

- Introduction
- Scala Overview
- Scala Basics
- Working with Data Types
- **Grouping Data Together**
- Flow Control in Scala
- Using and Creating Libraries
- Conclusion

Grouping Data Together

In this chapter you will learn

- **What are the different properties of each collection type**
- **Which factors to consider when selecting a collection type**
- **How to create, manage, interrogate, and update collections**

Chapter Topics

Grouping Data Together

- **Storing Elements of Different Types**
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- Common Collection Type Conversions
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Using Scala to Store a Record of Data

- **A `Tuple` in Scala consists of a fixed number of individual values that can be treated as a single entity**
- **Common uses of `Tuples`**
  - For returning more than one value from a function
  - Key-value pairs
  - Frequently used with and returned by iteration methods such as **`map`**
- **`Tuples` are more restrictive and less flexible than Collections**
  - The number of values in a tuple cannot be changed after it is initialized
  - Tuples consist of between two (minimum) and 22 (maximum) values

`Tuple2`: For Storing Records Having Two Elements

- **`Tuple2`, also called a *pair,* can be declared with several syntaxes**

Example: Explicit Declaration

```
val myTup2 = Tuple2(4, "iFruit")
> myTup2: (Int, String) = (4,iFruit)

myTup2.getClass
> Class[_ <: (Int, String)] = class scala.Tuple2
```

Example: Alternate Syntax

```
val myTup2 = 4 -> "iFruit"
> myTup2: (Int, String) = (4,iFruit)

val myTup2 = (4, "iFruit")
> myTup2: (Int, String) = (4,iFruit)
```

The **->** syntax only works with **`Tuple2`**

Special Methods for `Tuple2`

- **Elements of a `Tuple`**
  - Can be of different types
  - Are accessed using the `._1, ._2` syntax

```
val myTup2 = 4->"iFruit"
> myTup2: (Int, String) = (4,iFruit)

myTup2._1
> Int = 4

myTup2._2
> String = iFruit

myTup2.swap
> (String, Int) = (iFruit,4)
```

What About Storing Larger Records?

- **Tuples with more than two elements are stored in `TupleN`**
  - The maximum value for ***N*** is **22**
- **The same syntax applies as for `Tuple2`**

```
val myTup5 = (4, "MeToo", "1.0", 37.5, 41.3)

myTup5.getClass
> Class[_ <: (Int, String, String, Double, Double)] = class
scala.Tuple5

println(myTup5._3 + " / " + myTup5._5)
> 1.0 / 41.3
```

Learning the Class and Size of a Tuple

- **Use method `productPrefix` to obtain the tuple's class name as a string**
- **Use method `productArity` to obtain the tuple size as an integer**

```
val myTup6 = ("Plato", "Kant", "Voltaire", "Descartes",
"deBeauvoir", "Camus")

myTup6.productPrefix
> String = Tuple6

myTup6.productArity
> Int = 6
```

Powerful `partition` Method Converts `String` to `Tuple`

- **Use the `toString` method to convert the tuple to a string**
- **Use the `partition` method to convert a string to a tuple**
  - The first element in the tuple contains data that satisfies the condition
  - The second element contains data that failed to satisfy the condition

```
val myStr = myTup6.toString
> myStr: String =
  (Plato,Kant,Voltaire,Descartes,deBeauvoir,Camus)

val newTup = myStr.partition(_.isUpper)
> (String, String) =
   (PKVDBC, (lato,ant,oltaire,escartes,deeauvoir,amus))

newTup.getClass
> Class[_ <: (String, String)] = class scala.Tuple2

val sortedLastNameInitials = newTup._1.sorted
> String = BCDKPV
```

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- **Overview of Scala Collection Types**
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- Common Collection Type Conversions
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Storing Fixed- and Variable-Sized Data

- **In Scala there are a large number of collection classes available**
  - Classes are optimized for use in particular circumstances
  - They may be optimized for head/tail access or for fast update
- **Collection classes vary in the methods they support**
  - Immutable Collection classes are defined in package **`scala.collection.immutable`**
  - Mutable Collection classes are defined in package **`scala.collection.mutable`**

Traversable: Using foreach to Scale Out

- **Declare an object of type Traversable to use the very important foreach method which facilitates parallel and distributed processing**
  - Performs a specified action on all members of the collection
- **Scala will apply the function you supply to foreach to each element**
  - Allows the platform to parallelize processing and improve performance

```
val modelTrav = Traversable("MeToo", "Ronin", "iFruit")

modelTrav.foreach(println)
> MeToo
> Ronin
> iFruit
```

The **Traversable foreach** method receives a function as a parameter; for example **println**, which will be called once for each element in the collection.

`Iterable`: Managing Memory Wisely

- **`Iterable` adds the ability to iterate through each element, one at a time**
  - Data is resident in memory only as it is used

```
val models = Iterable("MeToo", "Ronin", "iFruit")
> models: Iterable[String] = List(MeToo, Ronin, iFruit)

val modelIter = models.iterator
> ModelIter: Iterator[String] = non-empty iterator

modelIter.next
> String = MeToo

modelIter.next
> String = Ronin

modelIter.next
> String = iFruit
```

The **`iterator`** method returns an **`Iterator`** object, which provides a way to traverse each element in sequence, *one time*

Seq: Using an Index to Access Specific Elements

- **Seq adds the ability to access each element at a fixed offset (index)**
- **First element is at index 0**
- **Seq(*n*) returns the value of the element at offset *n***

```
val mySeq = Seq("MeToo", "Ronin", "iFruit")
> mySeq: Seq[String] = List(MeToo, Ronin, iFruit)

mySeq(1)
> String = Ronin
```

`Set`: Storing Data with Automatic De-duplication

- **`Set` removes duplicates**
- **Does not change ordering**
- **`Set(`*`value`*`)` returns `true` or `false`**

```
val mySet = Set("MeToo", "Ronin", "iFruit")
> mySet: scala.collection.immutable.Set[String] =
  Set(MeToo, Ronin, iFruit)

mySet("Banana")
> Boolean = false
```

Map: Storing Key-Value Pairs

- **Map stores (key → value) pairs**

```
val wifiStatus = Map(
"disabled"  -> "Wifi off",
"enabled"   -> "Wifi on but disconnected",
"connected" -> "Wifi on and connected")

wifiStatus("enabled")
> String = Wifi on but disconnected
```

Collection Variable Declaration

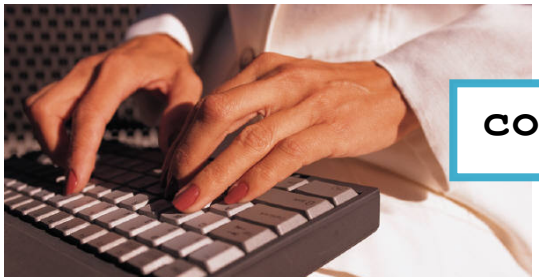
- **We have covered the different collection types, but the elements of a collection also have a type**
- **Element type may be specified explicitly or inferred**

```
val myMap: Map[Int,String] = Map(1 -> "a", 2 -> "b")

val myMap = Map(1 -> "a", 2 -> "b")
```

Scalability in Collection Processing

- **Scala collections include methods for processing all items in a collection without returning each item to the calling program**
- **By processing all items and only returning the result, Scala can optimize the program for distributed processing**



```
collection.foreach()
```

Platform

How to
- ***iterate***
- ***process data***

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- **Creating a Collection of Unique Elements**
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- Common Collection Type Conversions
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Storing Unique Values with Lookup Convenience

- **A `Set` is an Iterable that contains no duplicate elements**

```
val mySet = Set("Titanic", "Sorrento", "Ronin",
"Titanic", "Sorrento", "Ronin")
> mySet: scala.collection.immutable.Set[String] =
Set(Titanic, Sorrento, Ronin)

mySet.size
> Int = 3

mySet("Ronin")
> Boolean = true
```

Dropping Elements from a `Set`

- **`drop` removes the first *`n`* elements**

```
val mySet = Set("Titanic", "Sorrento", "Ronin")

val myset2 = mySet.drop(1)
> myset2: scala.collection.immutable.Set[String] =
    Set(Sorrento, Ronin)

mySet
> mySet: scala.collection.immutable.Set[String] =
   Set(Titanic, Sorrento, Ronin)
```

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- **Fast Access to Head of Collection**
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- Common Collection Type Conversions
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Storing and Processing Data of Varying Types

- **A `List` is a finite immutable sequence**
  - Very commonly used in Scala programming
  - Accessing the first element and adding an element to the front of the list are constant-time operations
- **A `List` literal can be constructed using `::` (cons operator) and `Nil`**

```
val newList = "a" :: "b" :: "c" :: Nil
> newList: List[String] = List(a, b, c)
```

Accessing `List` Elements

- **Create a list using the `List` keyword**
  - An alternative to using the cons operator and `Nil`
- **Elements of a `List` can be accessed using an index**

```
val models = List("Titanic", "Sorrento", "Ronin")
> models: List[String] = List(Titanic, Sorrento, Ronin)

models(1)
> String = Sorrento
```

Flexible Element Types

- **`Lists` can contain a single data type or type `Any`**

```
val randomlist = List("iFruit", 3, "Ronin", 5.2)

> randomlist: List[Any] = List(iFruit, 3, Ronin, 5.2)
```

- **`Lists` can contain Collection and `Tuple` elements as well as simple types**

```
val devices = List(("Sorrento", 10), ("Sorrento", 20),
("iFruit", 30))

> devices: List[(String, Int)] = List((Sorrento,10),
(Sorrento,20), (iFruit,30))
```

Convenient `List` Methods

```
val myList: List[Int] = List(1, 5, 7, 1, 3, 2)
> myList: List[Int] = List(1, 5, 7, 1, 3, 2)

myList.sum
> Int = 19

myList.max
> Int = 7

myList.take(3)
> List[Int] = List(1, 5, 7)

myList.sorted
> List[Int] = List(1, 1, 2, 3, 5, 7)

myList.reverse
> List[Int] = List(2, 3, 1, 7, 5, 1)
```

Produce the Union or Intersection of Lists

```
val myListA = List("iFruit", "Sorrento", "Ronin")
val myListB = List("iFruit", "MeToo",    "Ronin")

val myListC = myListA.union(myListB)
> myListC: List[String] = List(iFruit, Sorrento, Ronin,
iFruit, MeToo, Ronin)

val myListD = myListA ++ myListB
> myListD: List[String] = List(iFruit, Sorrento, Ronin,
iFruit, MeToo, Ronin)

myListC == myListD
> Boolean = true

val myListC = myListA.intersect(myListB)
> myListC: List[String] = List(iFruit, Ronin)
```

Appending Values to a List

- **Operations using the lists leave the original lists unchanged**

```
myListA ++ myListB

myListA
> res14: List[String] = List(iFruit, Sorrento, Ronin)

myListB
> res15: List[String] = List(iFruit, MeToo, Ronin)
```

- **Use :+ to append to a list**

```
val myListE = myListA :+ "xPhone"
> myListE: List[String] = List(iFruit, Sorrento, Ronin,
xPhone)
```

Lists that Can Be Modified

- A `ListBuffer` is the mutable form of a `List`
- A `ListBuffer` provides constant time prepend and append operations

```
val listBuf =
scala.collection.mutable.ListBuffer.empty[Int]

listBuf += 17
listBuf += 29
listBuf += 45
> listBuf.type = ListBuffer(17, 29, 45)

listBuf -= 17
> listBuf.type = ListBuffer(29, 45)
```

Mutable, but not Reassignable

- **`ListBuffer` is mutable with respect to its elements, however, attempts to reassign the pointer address are not allowed if it was declared with `val`**

```
import scala.collection.mutable.ListBuffer

val listBuf2 = ListBuffer("abc")

listBuf2 += "def"
> listBuf2.type = ListBuffer(abc, def)

listBuf = listBuf2
> error: reassignment to val listBuf = listBuf2
```

Mutable *and* Reassignable

- **Use `var` to create a mutable and reassignable `ListBuffer`**

```
var listBufVar = ListBuffer("one")
listBufVar += "banana"
> listBufVar.type = ListBuffer(one, banana)

listBuf2
> scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def)

listBufVar = listBuf2

listBufVar
>scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def)
```

Warning for Reassignable Collection Variables

- **Review this example carefully**
  - What is happening when **`listBuf2`** is modified?

```
listBuf2 += "xyz"

listBuf2
> scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def, xyz)

listBufVar
> scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def, xyz)
```

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- **Fast Access to Arbitrary Elements**
- Fast Access with a Key
- Common Collection Type Conversions
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Storing Data of a Known Size

- **An `Array` is mutable but not resizable**
  - Created with a fixed number of elements
    - You cannot change the number of elements in the array
    - You *can* update the value of an existing element
  - Array elements can be of a single type or **`Any`**

```
val devs = Array("iFruit", "MeToo", "Ronin")
> devs: Array[String] = Array(iFruit, MeToo, Ronin)

devs(2) = "Titanic"

devs
> Array[String] = Array(iFruit, MeToo, Titanic)
```

Updating Array Elements

- **Arrays are fixed in both size and type**

```
val devices: Array[String] = new Array[String](4)
devices.update(0, "Sorrento")
devices
> Array[String] = Array(Sorrento, null, null, null)

devices(0) = "Titanic"
devices
> Array[String] = Array(Titanic, null, null, null)

devices(1) = 256
> error: type mismatch; found: Int(256) required: String

devices.length
> Int = 4
```

`Vector`: Random Access, Flexible Size

- **`Vector`, `Array`, and `List` all inherit from the `Seq` type**
  - `List` belongs to to the `LinearSeq` branch of `Seq`
  - `Vector`, `Array`, and `String` belong to the `IndexedSeq` branch
- **A `Vector` is more efficient for random access than a `List`**
  - Allow access to any element in effectively constant time
  - Strikes a good balance between random selection and update speed

`Vector`: Random Access, Flexible Size

- **`Vector` is immutable, modifications are not made in place**

```
val vec = Vector(1, 18, 6)
> scala.collection.immutable.Vector[Int] = Vector(1, 18, 6)

vec.updated(1, 30)
> scala.collection.immutable.Vector[Int] = Vector(1, 30, 6)
```

- **Unlike `Array`, a `Vector` has flexible size**

```
var vec = Vector(1, 6, 21)
> scala.collection.immutable.Vector[Int] = Vector(1, 6, 21)

vec = vec :+ 5
> Vector(1, 6, 21, 5)

vec = 77 +: vec
> Vector(77, 1, 6, 21, 5)
```

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- **Fast Access with a Key**
- Common Collection Type Conversions
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Storing and Processing Key-Value Pairs

- **A `Map` is a collection of key-value pairs**
  - Immutable by default – values are not modified in place
- **Declare a map variable using either of these techniques**
  - `Map((key1, value1), (key2, value2))`
  - `Map(key1 -> value1, key2 -> value2)`
- **Keys and values**
  - Keys are unique and may only appear once; values are not unique

Common Uses for Maps

- **Commonly used for in-memory tables requiring fast access**
- **Used to associate names with values**
  - Single record buffer of data
  - Parameters required for calling an API

Example: Phone Status `Map`

```
val phoneStatus = Map(
("DTS"       -> "2014-03-15:10:10:31"),
("Brand"     -> "Titanic"),
("Model"     -> "4000"),
("UID"       -> "1882b564-c7e0-4315-aa24-228c0155ee1b"),
("DevTemp"   -> 58),
("AmbTemp"   -> 36),
("Battery"   -> 39),
("Signal"    -> 31),
("CPU"       -> 15),
("Memory"    -> 0),
("GPS"       -> true ),
("Bluetooth" -> "enabled"),
("WiFi"      -> "enabled"),
("Latitude"  -> 40.69206648),
("Longitude" -> -119.4216429))
```

- The values are associated with keys that are easily understood string names.
- For example, to determine if the WiFi is turned on, access **`phoneStatus("WiFi")`**

Map Access Methods (1)

```
phoneStatus.contains("DTS")
> Boolean = true

phoneStatus.keys
> Iterable[String] = Set(AmbTemp, GPS, Memory,
  Battery, Latitude, Signal, Longitude, DevTemp,
  Model, WiFi, UID, CPU, DTS, Brand, Bluetooth)

phoneStatus.values
> Iterable[Any] = MapLike(36, true, 0, 39,
  40.69206648, 31, -119.4216429, 58, 4000, enabled,
  1882b564-c7e0-4315-aa24-228c0155ee1b, 15,
  2014-03-15:10:10:31, Titanic, enabled)
```

Map Access Methods (2)

- Use `get` or `getOrElse` to avoid an exception for non-existent keys

```
phoneStatus("DTS")
> Any = 2014-03-15:10:10:31

phoneStatus("key_does_not_exist")
> java.util.NoSuchElementException: key not found:
key_does_not_exist ...

phoneStatus.get("key_does_not_exist")
> Option[Any] = None

phoneStatus.get("DTS")
> Option[Any] = Some(2014-03-15:10:10:31)

phoneStatus.getOrElse("key_does_not_exist", "No Key")
> Any = No Key
```

Mutable Map

- We can not change `Wireless` to `disabled`

```
phoneStatus("Wireless") = "disabled"
> error: value update is not a member of
  scala.collection.immutable.Map[String,String]
```

- Changing a value requires explicitly creating a mutable map

```
val mutRec = scala.collection.mutable.Map(("Brand" ->
"Titanic"), ("Model" -> "4000"), ("Wireless" -> "enabled"))
> scala.collection.mutable.Map[String,String] =
  Map(Wireless -> enabled, Model -> 4000, Brand -> Titanic)

mutRec("Wireless") = "disabled"
mutRec
> scala.collection.mutable.Map[String,String] =
 Map(Wireless -> disabled, Model -> 4000, Brand -> Titanic)
```

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- **Common Collection Type Conversions**
- Essential Points
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Converting Between Collection Types

- **Scala provides several methods for converting between collection types**

```
val myList = List("Titanic", "F01L", "enabled", 32)

val myArray = myList.toArray
> myArray: Array[Any] = Array(Titanic, F01L, enabled, 32)

val myIterable = myList.toIterable
> myIterable: Iterable[Any] = List(Titanic, F01L, enabled,
  32)

val myList2 = myIterable.toList
> myList2: List[Any] = List(Titanic, F01L, enabled, 32)

val myList3 = myArray.toList
> myList3: List[Any] = List(Titanic, F01L, enabled, 32)
```

Converting a **Tuple** to a **List**

```
val myTup = (4, "MeToo", "1.0", 37.5, 41.3, "Enabled")
> myTup: (Int, String, String, Double, Double, String) =
  (4,MeToo,1.0,37.5,41.3,Enabled)

myTup.getClass
> Class[_ <: (Int, String, String, Double, Double, String)]
  = class scala.Tuple6

val myList = myTup.productIterator.toList
> myList: List[Any] = List(4, MeToo, 1.0, 37.5, 41.3,
  Enabled)
```

String Conversions

- **Strings in Scala are treated as collections similar to `Arrays`**
- **Strings can be converted to other `Collection` types**

```
val myStr = "A Banana"
myStr(2)
> Char = B

myStr.toArray
> Array[Char] = Array(A,  , B, a, n, a, n, a)

myStr.toList
> List[Char] = List(A,  , B, a, n, a, n, a)

myStr.toSet
> scala.collection.immutable.Set[Char] = Set(n, A, a,  , B)
```

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- Common Collection Type Conversions
- **Essential Points**
- Hands-On Exercise: Exploring Tuples, Lists, and Maps

Essential Points (1)

- **`Tuple`**
  - Fixed size: **`Tuple2`**, **`Tuple3`**, …, **`Tuple22`**
  - Not part of the collection library
  - Created at compile time, which restricts their flexibility
- **`List`**
  - Flexible size
  - Elements are immutable, so they cannot be changed by assignment
  - Fast addition and removal at head
  - Slow access to arbitrary indexes
- **`ListBuffer`**
  - Flexible size
  - Elements are mutable
  - Constant time append and prepend operations

Essential Points (2)

- **`Array`**
  - Created with a fixed number of elements and not resizable
  - Fast access to arbitrary indexes
- **`Map`**
  - For working with key-value pairs
    To create a mutable **`Map`**, import **`scala.collection.mutable`** explicitly and declare the **`Map`** as **`mutable.Map`**

Bibliography

The following offer more information on topics discussed in this chapter

- **Information on Scala sequences**
  - **`http://tiny.cloudera.com/kaaws`**
- **Information on Scala sets**
  - **`http://tiny.cloudera.com/ljfwq`**
- **Information on Scala maps**
  - **`http://tiny.cloudera.com/lsmpq`**

Chapter Topics

Grouping Data Together

- Storing Elements of Different Types
- Overview of Scala Collection Types
- Creating a Collection of Unique Elements
- Fast Access to Head of Collection
- Fast Access to Arbitrary Elements
- Fast Access with a Key
- Common Collection Type Conversions
- Essential Points
- **Hands-On Exercise: Exploring Tuples, Lists, and Maps**

Hands-On Exercise: Exploring Tuples, Lists, and Maps

- **In this exercise, you will use tuples, lists, and maps, and convert between different collection types**
  - Please refer to the Hands-On Exercise Manual for instructions

cloudera®

Flow Control in Scala

Chapter 6

Course Chapters

- Introduction
- Scala Overview
- Scala Basics
- Working with Data Types
- Grouping Data Together
- **Flow Control in Scala**
- Using and Creating Libraries
- Conclusion

Flow Control in Scala

In this chapter you will learn

- **How to loop and perform repetitive operations**
- **What are some common techniques for iterating over items in a collection**
- **How to use pattern matching**
- **Why partial functions are useful and how to develop them**

Chapter Topics

Flow Control in Scala

- **Looping**
- Using Iterators
- Writing Functions
- Passing Functions as Arguments
- Collection Iteration Methods
- Pattern Matching
- Processing Data with Partial Functions
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

Controlling Program Flow

- **Functional programming flow control is different from imperative programming flow control and object oriented flow control**
  - Scala supports all three
- **Imperative: program *explicitly* operates on data**
- **Object-oriented: program *explicitly* invokes a method**
- **Functional: program *implies* what needs to be done**
  - Framework figures out how to satisfy requirements

Looping through Data

- **`while`**
  - Loop construct that tests for an exit condition
- **`for`**
  - Loop for a range of values

Looping that Limits Distributed Processing

- **`while` loops are typical of imperative programming**
- **You can do this in Scala but it is not best practice. *Do you know why?***

```
val sorrentoPhones = List("F00L", "F01L", "F10L", "F11L",
"F20L", "F21L", "F22L", "F23L", "F24L")

var i = 0
while (i < sorrentoPhones.length){
  println(sorrentoPhones(i))
  i = i + 1
}
```

`for` Loops with Ranges

- **The `<-` syntax is called an enumerator *generator***
  - You must adjust the number of iterations when using **`to`**
  - Use **`until`** to avoid this extra math to adjust for length
  - The **`by`** keyword allows you to increment by a custom value

```
for (i <- 0 to sorrentoPhones.length - 1) {
  println(sorrentoPhones(i))
}

for (i <- 0 until sorrentoPhones.length) {
  println(sorrentoPhones(i))
}

for (i <- 0 until sorrentoPhones.length by 2) {
  println(sorrentoPhones(i))
}
```

Sometimes Counting is Necessary

```
for (i <- 0 until sorrentoPhones.length) {
  println(i.toString + ": " + sorrentoPhones(i))
}
> 0: F00L
> 1: F01L
> 2: F10L
> 3: F11L
> 4: F20L
> 5: F21L
> 6: F22L
> 7: F23L
> 8: F24L
```

- In this example, the index is required by the application, but in many cases it is not
- Eliminating the local counting variable would remove a common source of scalability issues

`for` Iteration Over a Collection

- **This is the preferred form of explicit iteration in Scala**
- **Note: no counting variable**
  - No bounds issues, no mutability issue to limit scalability
- **The generator already knows to process each item in the collection**

```
for (model <- sorrentoPhones) {
  print(model + " ")
}

> F00L F01L F10L F11L F20L F21L F22L F23L F24L
```

`for` with Multiple Generators

- Generators within the `for()` must be separated by semicolons (`;`)
- They are treated as if they were nested `for` loops, left to right

```
val phonebrands = List("iFruit", "MeToo")
val newmodels = List("Z1", "Z-Pro")

for (brand <- phonebrands; model <- newmodels) {
   println(brand + " " + model)
}

iFruit Z1
iFruit Z-Pro
MeToo Z1
MeToo Z-Pro
```

Selecting a Subset of the Collection

- `if` is used to discard items that do not match
- However, this loop is generating *each item* and then only printing those items that match the criteria

```
val sorrentoPhones = List("F00L", "F01L", "F10L", "F11L",
"F20L", "F21L", "F22L", "F23L", "F24L")

for (model <- sorrentoPhones) {
  if (model.contains("2")) print(model + " ")
}

> F20L F21L F22L F23L F24L
```

Scala has a better option…

Advantage of Using `for` Filters

- **This example moves the `if` condition inside the `for` loop**
  - This is called a generator *filter*
- **Scala will only generate items that match the filter criteria**

```
val sorrentoPhones = List("F00L", "F01L", "F10L", "F11L",
"F20L", "F21L", "F22L", "F23L", "F24L")

for (model <- sorrentoPhones; if( model.contains("2"))) {
  print(model + " ")
}

> F20L F21L F22L F23L F24L
```

Collecting Data into a new Collection

- **`yield` returns a new collection of items rather than processing each item one at a time**

```
val phonebrands = List("iFruit", "MeToo")
val newmodels = List("Z1", "Z-Pro")

val newlist =
  for (brand <- phonebrands; model <- newmodels)
    yield brand + " " + model

> newlist: List[String] = List(iFruit Z1, iFruit Z-Pro,
  MeToo Z1, MeToo Z-Pro)
```

Chapter Topics

Flow Control in Scala

- Looping
- **Using Iterators**
- Writing Functions
- Passing Functions as Arguments
- Collection Iteration Methods
- Pattern Matching
- Processing Data with Partial Functions
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

Iterating Over Elements in a Collection

- **`Iterators` provide a way of iterating over elements in a collection**
- **`Iterators` can refer to distributed elements**
- **Iterators are scalable, making them ideal for Big Data applications**

Using `next` to Iterate

- **Create an `Iterator` from a collection using `toIterator`**
  - For a tuple use **`productIterator`**
- **The `Iterator` is used one time – use is "destructive"**

```
val phones = Array("iFruit", "MeToo")

val iter = phones.toIterator
> iter: Iterator[String] = non-empty iterator

iter.next
> String = iFruit

iter.next
> String = MeToo

iter.next
> java.util.NoSuchElementException: next on empty iterator
```

Using Iterators in a `while` Loop

- **This example shows the preferred use of `while` in Scala**
  - There are no counting variables or I/O dependencies

```
val titanicPhones = List("1000", "2000", "3000", "Bananas")

val iter = titanicPhones.toIterator

print(iter.next)
> 1000

print(iter.next)
> 2000

while (iter.hasNext) {
  print(iter.next + " ")
}
> 3000 Bananas
```

Other `Iterator` Methods

- **The table below describes several key methods for working with iterators**

Method	Description
`size`	The remaining number of elements
`isEmpty`	`true` if there are remaining elements
`exists(`*`element`*`)`	`true` if the element exists in the list
`take(`*`n`*`)`	Returns a new `Iterator` with just the next *`n`* elements
`filter(boolean-`*`expression`*`)`	Returns a new `Iterator` with elements for which the expression is `true`
`foreach(`*`function`*`)`	Execute *`function`* for each element provided by the iterator

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- **Writing Functions**
- Passing Functions as Arguments
- Collection Iteration Methods
- Pattern Matching
- Processing Data with Partial Functions
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

Introducing `def`

- **Variable types and values are evaluated immediately upon assignment**
- **Contrast with the function definition where only the type is evaluated**
  - The value will be evaluated when the function is called

```
val myConstant = 10

var myVariable = 24

def myFunction = myConstant + myVariable
> myFunction: Int

myFunction
> Int = 34
```

Defining a Function with `def`

```
val myConstant = 10
var myVariable = 24

def myFunction = myConstant + myVariable
> myFunction: Int

myVariable = 9
myFunction
> Int = 19

myVariable = 20
myFunction
> Int = 30

val myConstant = 3
myFunction
> Int = 30
```

myFunction evaluates to a different result when **myVariable** is reassigned to **20** because the value is passed in by reference

However, when **myConstant** is reassigned to **3**, there is no change to the result returned by **myFunction** because **myConstant** was passed by value, not by reference

Using an Expression Block with `def`

- **The multi-line function definition uses curly braces**
- **All functions return something**
  - If there is no explicit return type, Scala returns `Unit`
- **Parentheses are only required if the function accepts parameters**

```
def listPhones {
  println("MeToo")
  println("Titanic")
  println("iFruit")
}
> listPhones: Unit

listPhones
> MeToo
> Titanic
> iFruit
```

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- Writing Functions
- **Passing Functions as Arguments**
- Collection Iteration Methods
- Pattern Matching
- Processing Data with Partial Functions
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

Function Example with Parameter and Return Value

```
def CtoF(celsius: Double) = {
  (celsius * 9 / 5) + 32
}
> CtoF: (celsius: Double)Double

CtoF(34.0)
> Double = 93.2

def CtoF(celsius: Double) =
  (celsius * 9 / 5 ) + 32

def CtoF(celsius: Double) =
  (celsius * 9 / 5 ) + 32 : Double
```

- Use = to define a function with a return value
- No **return** keyword
- The evaluation of the final expression is returned

For simple expressions, the curly braces are not needed

Return type may be explicit or inferred

Passing a Function as a Parameter (1)

- **`convertList` is called a higher-order function because it takes another function as a parameter**
- **`convert` is the name of the parameter that accepts a function**
  - **`convert`** specifies the type for the input parameter to the left of the **=>** transformation symbol
  - It specifies the return type to the right of **=>**

```
def CtoF(celsius: Double) = (celsius * 9 / 5) + 32

def convertList(myList:List[Double],
                convert:(Double) => Double) {
  for(n <- myList)
    println(n,convert(n))
}
> convertList: (myList: List[Double],
convert: Double => Double)Unit
```

Passing a Function as a Parameter (2)

```
def CtoF(celsius: Double) = (celsius * 9 / 5) + 32

def convertList(myList:List[Double],
                convert:(Double) => Double) {
  for(n <- myList)
    println(n,convert(n))
}
> convertList: (myList: List[Double],
convert: Double => Double)Unit

val phoneCelsius = List(34.0, 23.5, 12.2)

convertList(phoneCelsius, CtoF)
> (34.0,93.2)
> (23.5,74.3)
> (12.2,53.96)
```

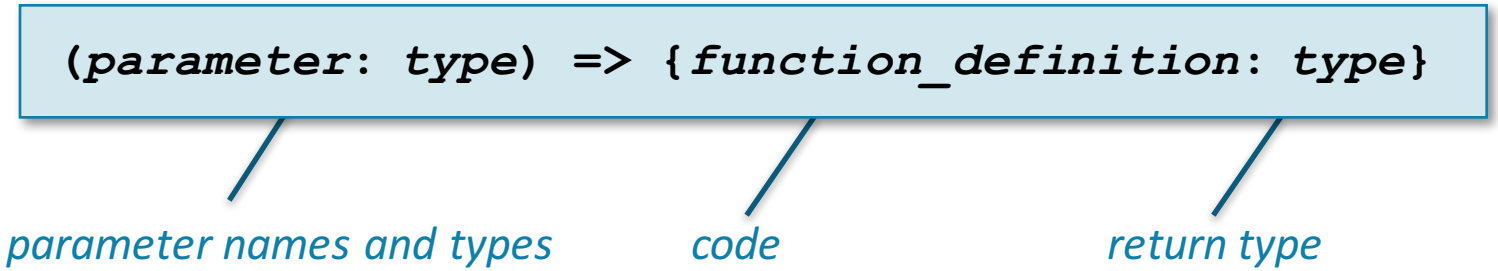
In this case, **CtoF** is the function passed into the **convert** parameter

Anonymous Functions

- **Anonymous functions are an alternate syntax for defining functions**
  - They do not require a function name or label
  - Also referred to as *lambda functions*
- **Anonymous functions in source code are called function literals**
  - Often used when a function will be called only once

Define an Anonymous Function

- **An anonymous function is a way to define a function *inline*,**
  - In other words, it is embedded in other code



Using an Anonymous Function

- **The example below shows an anonymous function for converting temperature from Celsius to Fahrenheit**

```
def convertList(myList:List[Double],
convert:(Double)=>Double) {
  for (n <- myList)
    println(n, convert(n))
}

val phoneCelsius = List(34.0, 23.5, 12.2)

convertList(phoneCelsius, cc => (cc * 9 / 5) + 32)
> (34.0,93.2)
> (23.5,74.3)
> (12.2,53.96)
```

A function literal can be used in the call to a higher-order function as an anonymous function.

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- Writing Functions
- Passing Functions as Arguments
- **Collection Iteration Methods**
- Pattern Matching
- Processing Data with Partial Functions
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

Higher-Order Collection Methods

- **Commonly used collection methods include**
  - `foreach`
  - `map`
  - `filter`
- **Using these methods help your program to scale**
  - They delegate control over iteration to the framework

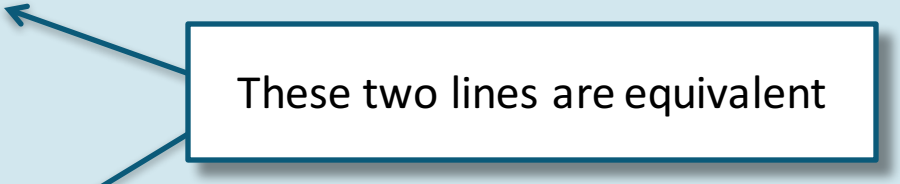
`foreach` Method

- **`List` inherits the `foreach` method**
- **The _ (underscore) is a placeholder variable**
  - It is a reference to the current element being operated on by **`foreach`**

```
val phones = List("MeToo", "Titanic", "Ronin")

phones.foreach(println(_))
> MeToo
> Titanic
> Ronin

phones.foreach(println)
> MeToo
> Titanic
> Ronin
```



Placeholder _

- **Using _ may create ambiguity that prevents Scala from inferring the type**
  - This is illustrated in the first example below
- **In these cases, the type must either be specified or made more inferable**
  - The second example hints to Scala that the list contains **`String`**s

```
val phones = List("MeToo", "Titanic", "Ronin")

phones.foreach(println(_).toUpperCase)
> <console>:12: error: missing parameter type for expanded
function ((x$1) => x$1.toUpperCase)
       phones.foreach(println(_.toUpperCase))

phones.foreach(println(_).toString.toUpperCase)
> MeToo
> Titanic
> Ronin
```

Using the `map` Method to Apply a Function to Each Element

```
def CtoF(celsius: Double) = celsius * 9 / 5 + 32

val phoneCelsius = List(34.0, 23.5, 12.2)

phoneCelsius.map(c => CtoF(c))
> List[Double] = List(93.2, 74.3, 53.96)
```

Passing a named function

```
phoneCelsius.map(CtoF(_))
> List[Double] = List(93.2, 74.3, 53.96)
```

Using a placeholder parameter

```
phoneCelsius.map(c => c * 9 / 5 + 32)
> List[Double] = List(93.2, 74.3, 53.96)
```

Passing an anonymous function (function literal)

```
phoneCelsius.map(_ * 9 / 5 + 32)
> List[Double] = List(93.2, 74.3, 53.96)
```

Passing an expression with a placeholder parameter

Filtering Numeric Values

- In this example, the underscore placeholder refers to a numeric
- Create the filter condition using relational operators
- In the example, there is an implicit conversion of the integer literal to a floating point value

```
val phoneCelsius = List(34.0, 23.5, 12.2)

phoneCelsius.filter(val1 => val1 < 23)
> List[Double] = List(12.2

phoneCelsius.filter(_ < 23)
> List[Double] = List(12.2)
```

`filter` Method

- **Since the placeholder in this case refers to a `String`, we can call string methods like `startsWith` and `length` on the placeholder**

```
val phones = List("1000", "2000", "2500", "Bananas")

phones.filter(_.startsWith("2"))
> List[String] = List(2000, 2500)

phones.filter(_.length > 4 )
> List[String] = List(Bananas)
```

Providing an Operator as an Argument Using `sortWith`

- **`sortWith` uses the passed in operator to compare the two elements**
  - The first underscore refers to the first parameter, the second one refers to the second parameter

```
val phoneCelsius = List(34.0, 23.5, 12.2)

phoneCelsius.sortWith((val1, val2) => val1 < val2)
> List[Double] = List(12.2, 23.5, 34.0)

phoneCelsius.sortWith(_ < _)
> List[Double] = List(12.2, 23.5, 34.0)

phoneCelsius.sortWith(_ > _)
> List[Double] = List(34.0, 23.5, 12.2)
```

Chaining Collection Methods

```
var myList: List[Int] = List(1, 5, 7, 3, 2, 1)

myList.map(_ + 10)
> List[Int] = List(11, 15, 17, 13, 12, 11)

myList.filter(_ > 4)
> List[Int] = List(5, 7)

myList.map(_ + 1).filter(_ > 4)
> List[Int] = List(6, 8)
```

```
titanicPhones.filter(_.endsWith("00")).sortWith(_ > _)
> List[String] = List(2500, 2000, 1000)
```

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- Writing Functions
- Passing Functions as Arguments
- Collection Iteration Methods
- **Pattern Matching**
- Processing Data with Partial Functions
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

Comparing Literals Using `match..case`

- **`case` can match any literal of any type**

```
val phoneWireless = "enabled"
var msg = "Radio state Unknown"

phoneWireless match {
  case "enabled"    => msg = "Radio is On"
  case "disabled"   => msg = "Radio is Off"
  case "connected"  => msg = "Radio On, Protocol Up"
}

println(msg)
> Radio is On
```

Handling Non-Matching Literals

- **A `match` can implicitly return a value**
  - **`msg`** is assigned the result of the **`match`**…**`case`**

```
val phoneWireless = "happy"
var msg = "unknown"

val msg = phoneWireless match {
   case "enabled"    => "Radio is on";
   case "disabled"   => "Radio is off";
   case "connected"  => "Radio on, protocol up";
   case default      => "Radio state unknown"
}

println(msg)
> Radio state unknown
```

`match` … `case` with Mixed Types (1)

- This array has a mix of types, use `match…case` to process each type
- Do you expect `'F'` to be reported as a `Char`?

```
val mixedArr = Array("11", 12, "thirteen", 14.0, 'F', null)

for (elem <- mixedArr) {
  elem match {
    case elem:String => println("String:   " + elem)
    case elem:Int    => println("Integer:  " + elem)
    case elem:Double => println("Float:    " + elem)
    case elem:AnyRef => println("Unknown:  " + elem)
    case elem:Char   => println("Char:     " + elem)
    case null        => println("Found null")
  }
}
```

`match` … `case` with Mixed Types (2)

- **`'F'` is reported as "Unknown"**

```
String:   11
Integer:  12
String:   thirteen
Float:    14.0
Unknown:  F
Found null
```

- **The ordering of `case` statements within a `match` is significant**
  - The first **`case`** that matches is executed
- **Reorder the `case` statements to get the intended result**
  - In this case, **`elem:Char`** must precede **`elem:AnyRef`**

Managing Response to Illegal Input Data

- An `Option` is a special type with a value of `Some(n)` or `None`
- An `Option` can be used to "wrap" a function that would potentially throw an error if it produced an illegal value
- If the value is good, then it is returned wrapped in `Some`
- `Option` can be used in a `match…case` by the caller

Using `getOrElse` to Control Program Flow

- **`Some(x)` contains the value, where `x` is the returned value**
- **`Some` and `None` can be explicitly set, as illustrated**
- **`getOrElse`**
  - Returns the wrapped value if **`Some`**, otherwise it performs the action

```
val superPhone = Some("Model 6")
> superPhone: Some[String] = Some(Model 6)

superPhone.getOrElse("Not found")
> String = Model 6

val superPhone = None
> superPhone: None.type = None

superPhone.getOrElse("Not found")
> String = Not found
```

Using `Option` in a Function

- **This example shows a common use of `Option` in functions**
  - The function returns a value encapsulated in a **`Some`** / **`None`** (**`Option`**) so that the caller can take appropriate action

```
def str2Double(in: String): Option[Double] = {
  try {
    Some(in.toDouble)
  } catch {
    case e: NumberFormatException => None
  }
}

str2Double("35.2")
> Option[Double] = Some(35.2)

str2Double("Warm")
> Option[Double] = None
```

Option with match and case

- **Process Some(x) inputs**
  - In this example, we use typed pattern matching

```
def convert2Float(x: Option[Any]) = x match {
  case Some(d: Double) => d.toFloat
  case Some(i: Int)    => i.toFloat
  case Some(f: Float)  => f
  case Some(_: Any)    => println("Invalid data provided.")
  case None => println("No data provided.")
}

convert2Float(Some(25.0))
> AnyVal = 25.0

convert2Float(Some(25F))
> AnyVal = 25.0

convert2Float(Some(25))
> AnyVal = 25.0
```

Option with match and case

- Example to process None inputs and Any inputs

```
def convert2Float(x: Option[Any]) = x match {
  ...
  case Some(_: Any) => println("Invalid data provided.")
  case None => println("No data provided.")
}

convert2Float(Some("twenty-five"))
> Invalid data provided.
  AnyVal = ()

convert2Float(None)
> No data provided.
  AnyVal = ()
```

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- Writing Functions
- Passing Functions as Arguments
- Collection Iteration Methods
- Pattern Matching
- **Processing Data with Partial Functions**
- Essential Points
- Hands-On Exercise: Iterating through Data Efficiently

What are Partial Functions?

- **A *partial function* is used when an answer should be returned only for a subset of possible input values**
  - Defines the (partial) data it can handle
  - Can be queried to determine whether a given value can be handled
- **Simple examples where partial functions can be useful**
  - Division by zero
  - Square root of a negative number

Why Partial Functions?

- **Take divide by zero as an example**

```
val div = (x: Int) => 24 / x
```

- **Providing a zero for `x` will cause an arithmetic exception**
  - Partial functions can offer a way to avoid such an exception

Implementing a Partial Function

- **Must be declared as a `PartialFunction`**
- **`PartialFunction` defines two methods that you must implement**
  - **`apply`** performs the actual processing for your method
  - **`isDefinedAt`** evaluates whether the supplied input is valid

```
val div = new PartialFunction[Int, Int] {
  def apply(x: Int) = 24 / x
  def isDefinedAt(x: Int) = x != 0
}
```

Partial Functions vs. Exception Handlers

- **Partial functions allow a caller to test input before using it in a parameter**

```
val div = new PartialFunction[Int, Int] {
  def apply(x: Int) = 24 / x
  def isDefinedAt(x: Int) = x != 0
}
```

```
div.isDefinedAt(0)
> Boolean = false

div.isDefinedAt(2)
> Boolean = true

if (div.isDefinedAt(2)) div(2)
> AnyVal = 12
```

Partial Functions with Pattern Match

- **When a partial function includes one or more `case` statements, the `apply` and `isDefinedAt` methods are generated automatically**

```
val getThirdItem: PartialFunction[List[Int], Int] = {
  case x :: y :: z :: _ => z
}

getThirdItem.isDefinedAt(List(25))
> Boolean = false

getThirdItem.isDefinedAt(List(25, 35, 45, 85))
> Boolean = true

getThirdItem(List(25, 35, 45, 85))
> Int = 45
```

Partial Functions Summary

- **Use complete functions whenever possible**
- **A partial function may compile fine, but you may experience runtime errors for unhandled values**
- **Partial functions are useful when you are certain that**
  - An unhandled value will never be supplied
  - Values are always checked with **`isDefinedAt`** before an explicit or implicit call to the **`apply`** method

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- Writing Functions
- Passing Functions as Arguments
- Collection Iteration Methods
- Pattern Matching
- Processing Data with Partial Functions
- **Essential Points**
- Hands-On Exercise: Iterating through Data Efficiently

Essential Points

- **Scala supports imperative programming and functional programming**
- **Scala provides iterative methods for scalability**
- **Scala supports higher-order functions**
- **If possible, use Collection methods rather than imperative programming**
- **Pattern matching behaves differently from "switch" in other languages**

Bibliography

The following offer more information on topics discussed in this chapter

- **Information on Scala partial functions**
  - **`http://tiny.cloudera.com/ntryr`**
  - **`http://tiny.cloudera.com/klzaw`**

Chapter Topics

Flow Control in Scala

- Looping
- Using Iterators
- Writing Functions
- Passing Functions as Arguments
- Collection Iteration Methods
- Pattern Matching
- Processing Data with Partial Functions
- Essential Points
- **Hands-On Exercise: Iterating through Data Efficiently**

Hands-On Exercise: Iterating through Data Efficiently

- **In this exercise, you will practice using for loops, iterators, anonymous functions, and filtering**
  - Please refer to the Hands-On Exercise Manual for instructions

cloudera®

Using and Creating Libraries

Chapter 7

Course Chapters

- Introduction
- Scala Overview
- Scala Basics
- Working with Data Types
- Grouping Data Together
- Flow Control in Scala
- **Using and Creating Libraries**
- Conclusion

Using and Creating Libraries

In this chapter you will learn

- **Which types of libraries are available in Scala**
- **How to use Scala libraries**
- **How to create your own Scala packages to enable program reuse**

Chapter Topics

Using and Creating Libraries

- **Using Classes and Objects**
- Creating and Using Packages
- Importing Part of a Package
- Essential Points
- Hands-On Exercise: Working with Libraries in Scala

Working with Scala Libraries

- **Code is organized by *classes***
  - Object-oriented code organization
  - Contains data elements and methods
  - Establishes local scope within the class
  - Enables instantiation of objects
- **Classes are grouped together into *packages***
  - Separate namespaces to avoid collision of classes
- **Packages are defined in Scala source files**
  - Multiple packages can be contained in a single file
- **Use the `import` keyword to import libraries to use in your code**

Defining and Instantiating Classes

```
class Device(name: String) {
   val phoneName = name
   def display = s"Phone is $phoneName"
}
> defined class Device

val a = new Device("Sorrento")
> a: Device = Device@266b5a30

a.display
> String = Phone is Sorrento
```

Defining Member Variables in a Constructor

```
class Device(val phoneName: String) {
   def display = s"Phone is $phoneName"
}
> defined class Device

val a = new Device("Ronin")
> a: Device = Device@266b5a30

a.display
> String = Phone is Ronin
```

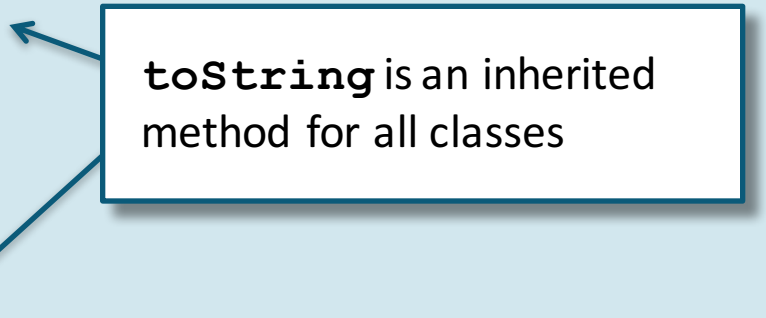
Overriding Inherited Methods (1)

```
class Device(val phoneName: String) {
   def display = s"Phone is $phoneName"
}
> defined class Device

val a = new Device("Ronin")
> a: Device = Device@266b5a30

a.display
> String = Phone is Ronin

a.toString
> String = Device@266b5a30
```



Overriding Inherited Methods (2)

```
class Device(val phoneName: String) {
   def display = s"Phone is $phoneName"
   override def toString = s"$phoneName"
}
> defined class Device

val a = new Device("Titanic")
> a: Device = Titanic

a.toString
> String = Titanic
```

Singleton Objects

- **Singleton objects are created without an explicit class**
- **Scala generates a class automatically, the object is a *companion* of the class**

File: `TestDevice.scala`

```
object TestDevice {
   def main(args: Array[String]) {
      val a = new Device("iFruit 3000")
      println(a.display)
   }
}
```

```
$ scalac TestDevice.scala
```

Class: `TestDevice.class`
Companion Object: `TestDevice$.class`

Chapter Topics

Using and Creating Libraries

- Using Classes and Objects
- **Creating and Using Packages**
- Importing Part of a Package
- Essential Points
- Hands-On Exercise: Working with Libraries in Scala

Packages in Scala

- **Packages are used to group classes into a library**
- **Package naming follows Java conventions**
  - Begin with the reverse domain name of the organization
  - Additional names can be appended, following a dot
  - Example: `com.loudacre.libraries` becomes `com/loudacre/libraries`
- **Default packages imported during compilation**
  - `java.lang`
  - `scala`
  - `scala.Predef`

Defining a Package

File: `Device.scala`

```
package com.loudacre.phonelib

class Device(val phoneName: String) {
   def display = s"Phone is $phoneName"
   override def toString = s"$phoneName"
}
```

Compilation

```
$ scalac Device.scala
```

Class file will be produced at:
`com/loudacre/phonelib/Device.class`

Importing a Class from a Package

```
package com.cloudera.training

import com.loudacre.phonelib.Device

object TestDevice {
   def main(args: Array[String]) {
      val a = new Device("iFruit 3000")
      println(a.display)
   }
}
```

Executing an Object in a Package

- **Start Scala with the full package reference of the object**
- **The `main` method will be called automatically**

```
$ scala com.cloudera.training.TestDevice
> Phone is iFruit 3000
```

Chapter Topics

Using and Creating Libraries

- Using Classes and Objects
- Creating and Using Packages
- **Importing Part of a Package**
- Essential Points
- Hands-On Exercise: Working with Libraries in Scala

Importing All Classes

- **`import pack1._`**
  - Imports all classes from **`pack1`**
  - Equivalent to the Java code **`import pack1.*`**
- **`import pack1._, pack2._`**
  - Imports all classes from **`pack1`** and from **`pack2`**

Import Subset of Classes

- **`import pack1.Class1`**
  - Imports only **`Class1`** from **`pack1`**
- **`import pack1.{Class1, Class3}`**
  - Imports only **`Class1`** and **`Class3`** from **`pack1`**
- **`import pack1.Class1._`**
  - Imports all members of **`Class1`** *including implicit definitions*
  - No equivalent in Java
- **`import pack1.{Class1 => MyClass}`**
  - Imports **`Class1`** from pack1 and renames it **`MyClass`**
  - This avoids collision with an existing function named **`Class1`**

Chapter Topics

Using and Creating Libraries

- Using Classes and Objects
- Creating and Using Packages
- Importing Part of a Package
- **Essential Points**
- Hands-On Exercise: Working with Libraries in Scala

Essential Points

- **Libraries in Scala are implemented as packages**
  - Similar to Java
- **Packages consist of a set of related classes and objects**

Chapter Topics

Using and Creating Libraries

- Using Classes and Objects
- Creating and Using Packages
- Importing Part of a Package
- Essential Points
- **Hands-On Exercise: Working with Libraries in Scala**

Hands-On Exercise: Working with Libraries in Scala

- **In this exercise, you will create and use a library for parsing log data**
  - Please refer to the Hands-On Exercise Manual for instructions

cloudera®

Conclusion

Chapter 8

Course Chapters

- Introduction
- Scala Overview
- Scala Basics
- Working with Data Types
- Grouping Data Together
- Flow Control in Scala
- Using and Creating Libraries
- **Conclusion**

Conclusion

During this course, you have learned

- **What Scala is and how it differs from languages such as Java or Python**
- **Why Scala is a good choice for Spark programming**
- **How to use key language features such as data types, collections, and flow control**
- **How to implement functional programming solutions in Scala**
- **How to work with Scala classes, packages, and libraries**

Which Course to Take Next?

Cloudera offers a range of training courses for you and your team

- **For developers**
  - *Cloudera Developer Training for Spark and Hadoop*
  - *Designing and Building Big Data Applications*
  - *Cloudera Training for Apache HBase*
- **For system administrators**
  - *Cloudera Administrator Training for Apache Hadoop*
- **For data analysts and data scientists**
  - *Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop*
  - *Data Science at Scale Using Spark and Hadoop*
- **For architects, managers, CIOs, and CTOs**
  - *Cloudera Essentials for Apache Hadoop*