Just Enough Scala: Instructor Guide

CONFIDENTIAL

This guide is confidential, and contains Cloudera proprietary information. It must not be made available to anyone other than Cloudera instructors and approved Partner instructors.

Version	Release Date	Description
201511	11/03/2015	Initial Release

NOTE: This release of the course does not include Instructor Guide back matter (annotated exercise instructions).

Suggested Course Timings

Day 1 **[Total classroom time: 7 hours, 20 minutes]**

Arrivals and registration [15 minutes total]

1. **Introduction** [20 minutes total]
   - 20 minutes lecture

2. **Scala Basics** [1 hour, 15 minutes total]
   - 45 minutes lecture
   - 30 minutes exercise

3. **Variables** [1 hour, 15 minutes total]
   - 45 minutes lecture
   - 30 minutes exercise

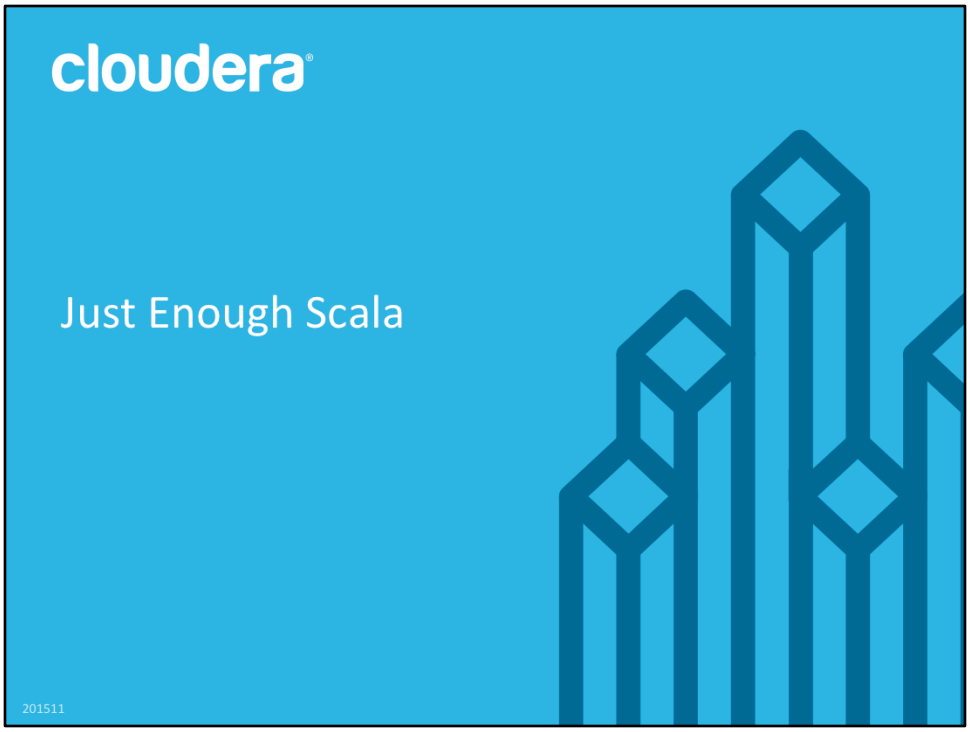
4. **Collections** [1 hours, 35 minutes total]
   - 75 minutes lecture
   - 20 minutes exercise

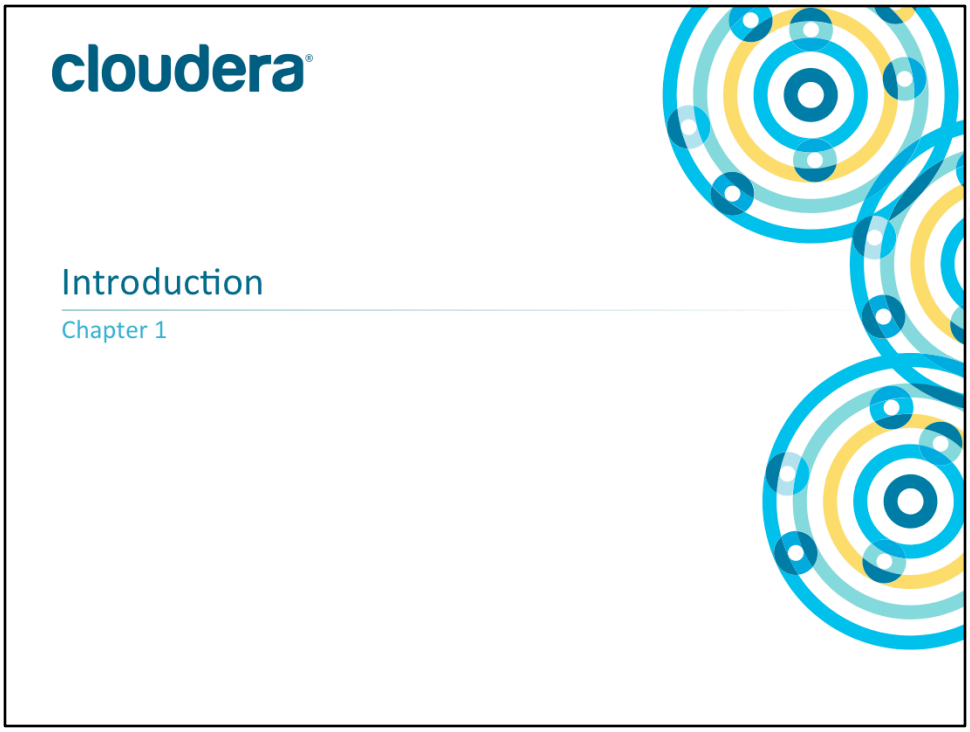
5. **Flow Control** [1 hours, 15 minutes total]
   - 60 minutes lecture
   - 45 minutes exercise

6. **Libraries** [1 hours, 10 minutes total]
   - 25 minutes lecture
   - 45 minutes exercise

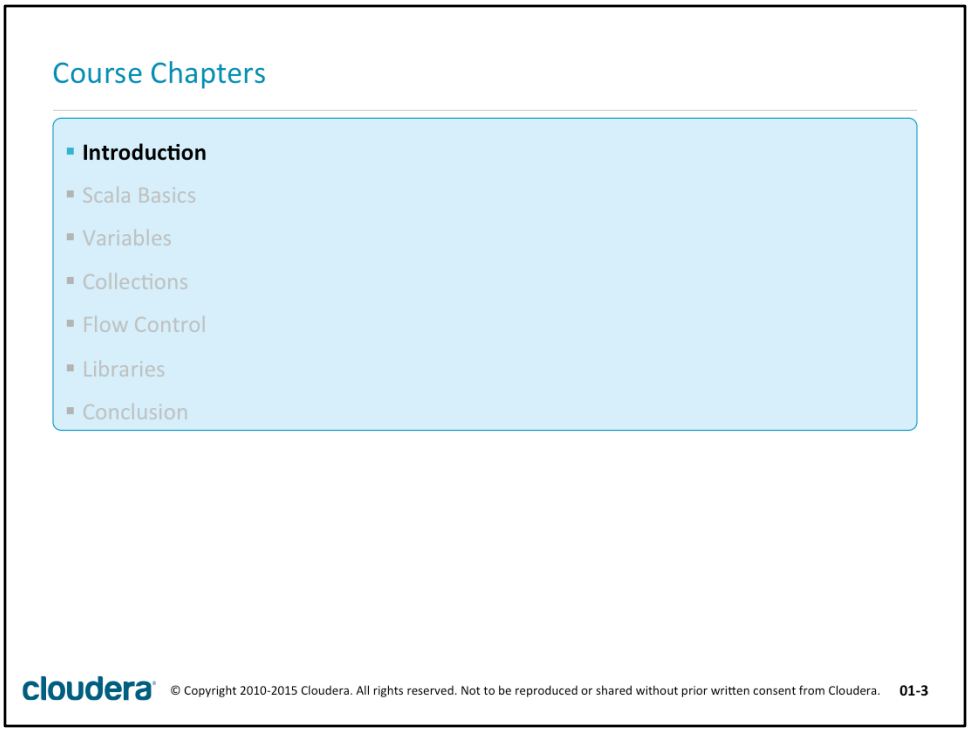
7. **Conclusion** [5 minutes total]
   - 5 minutes lecture

Final questions and post-course survey [15 minutes total]

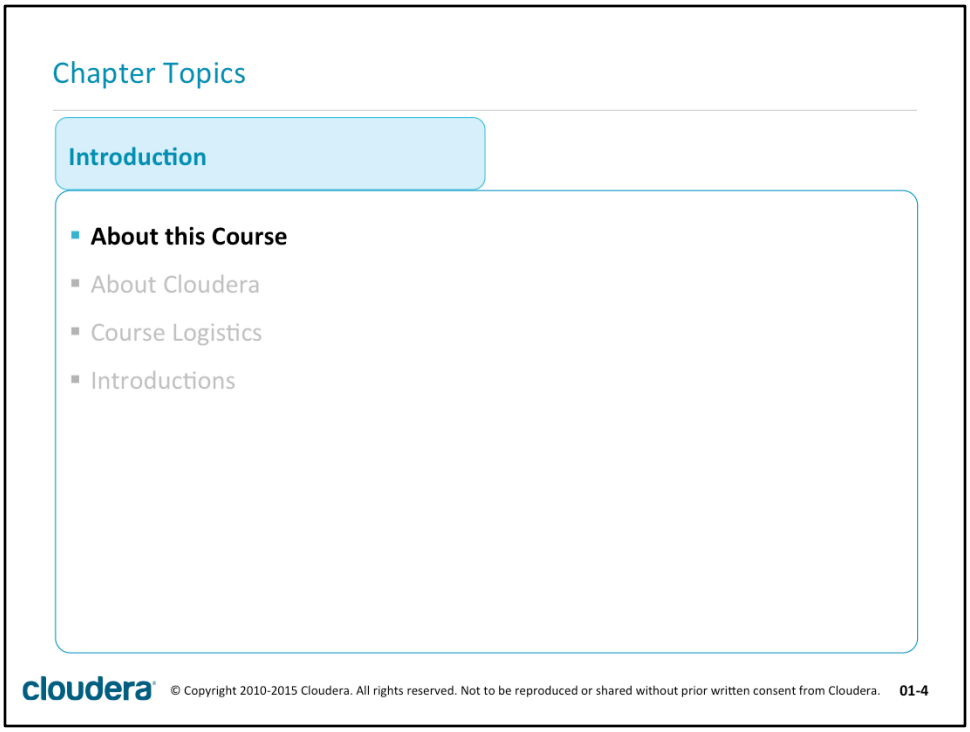




Goal: This chapter is intended to inform students what to expect from the course and for the instructor to learn about the students' level of expertise as well as how they plan to apply what they'll learn.



This slide lists all the chapters in the course, with the current chapter highlighted to provide some context for where we are. A similar slide will appear as the second slide for each subsequent chapter, with *that* chapter's name highlighted rather than *this* one.



This slide shows the main topics covered in the current chapter. The upcoming topic is highlighted to illustrate what is about to be covered.

Course Objectives

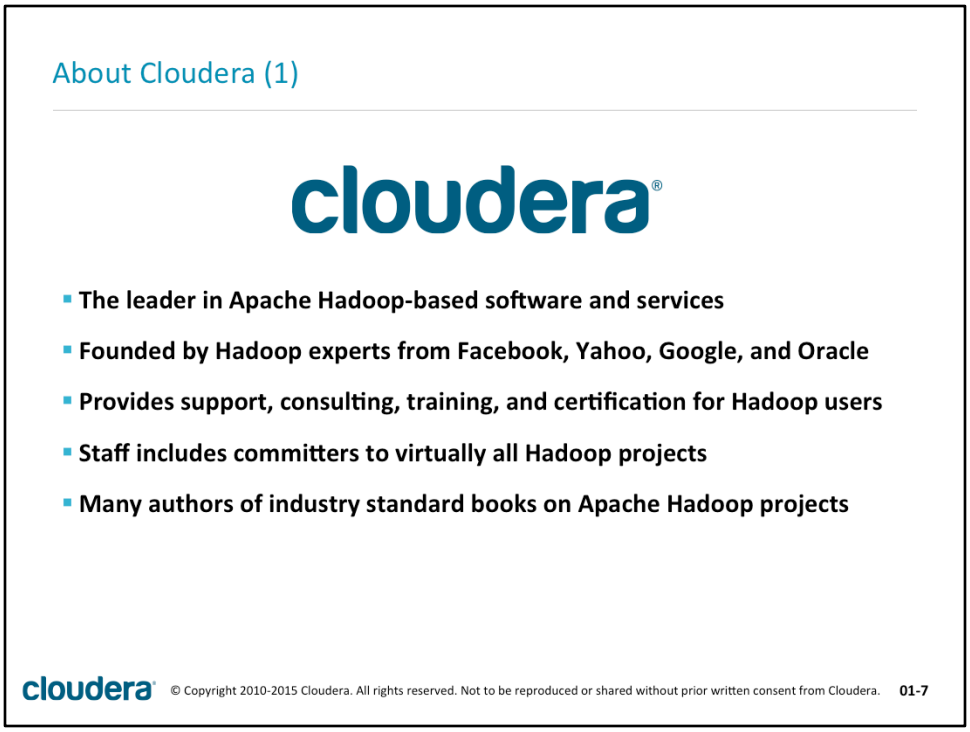
During this course, you will learn

- **What Scala is and how it differs from other languages such as Java or Python**
- **Key Scala concepts such as data types, collections and program flow control**
- **How to implement both imperative and functional programming solutions in Scala**
- **How to work with Scala classes, packages and APIs**

Chapter Topics

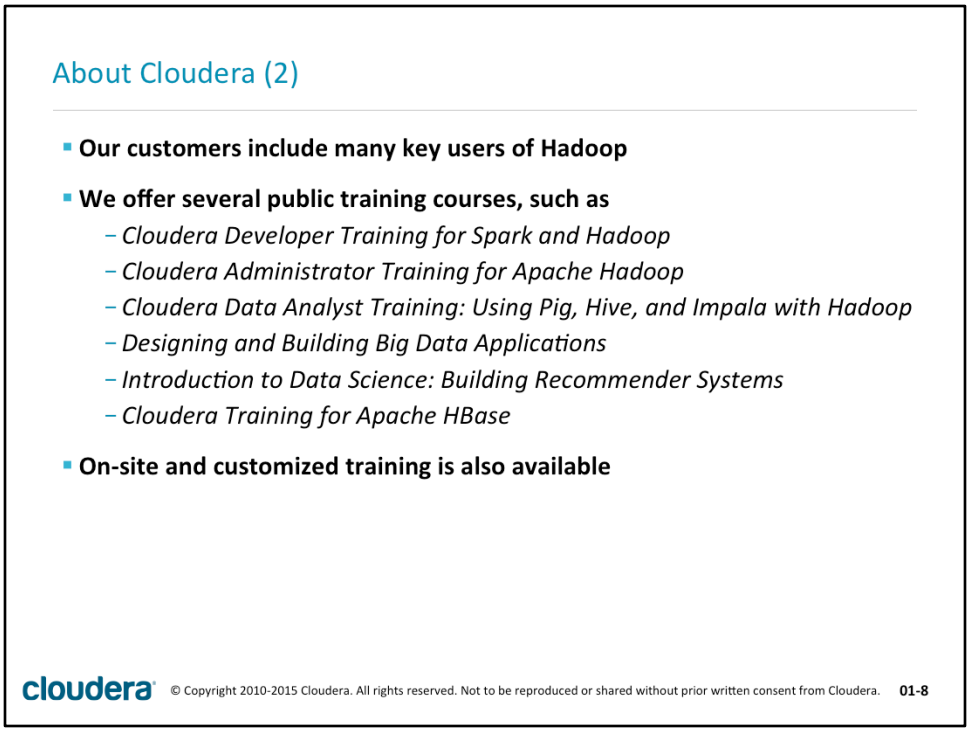
Introduction

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

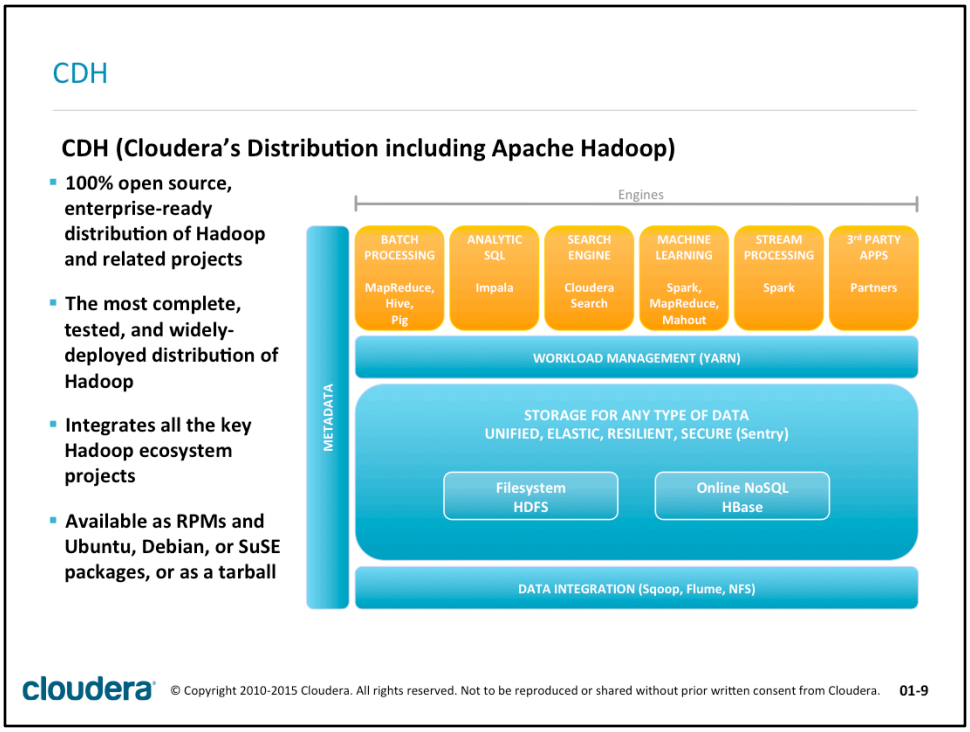


Cloudera was founded in 2008. Our staff also includes the ASF chairperson and creator of Hadoop, Doug Cutting, as well as many involved in the project management committee (PMC) of various Hadoop-related projects. The person who literally wrote the book on Hadoop, Tom White, works for Cloudera, as does the person who wrote the book on HBase, Lars George.

There are many Cloudera employees who have written or co-authored books on Hadoop-related topics, and you can find an up-to-date list here [`http://www.cloudera.com/content/cloudera/en/developers/home/hadoop-ecosystem-books.html`]. Instructors are encouraged to point out that many of these books are available at a substantial discount to students in our classes [`http://shop.oreilly.com/category/deals/cloudera.do`].



You can see a list of customers that we can reference on our Web site [`http://cloudera.com/content/cloudera/en/our-customers.html`]. Note that Cloudera also has many customers who do not wish to be referenced, and it is essential that we honor this. The only exception to this important rule is that you may reference something that was intentionally made available to the public in which Cloudera or that customer has disclosed that they are a Cloudera customer. For example, it is permissible to mention an article in a reputable trade publication in which Cloudera's CEO mentions a specific customer or the keynote address that the customer's CTO gave at the Strata conference talking about the benefits they've experienced as a Cloudera customer.

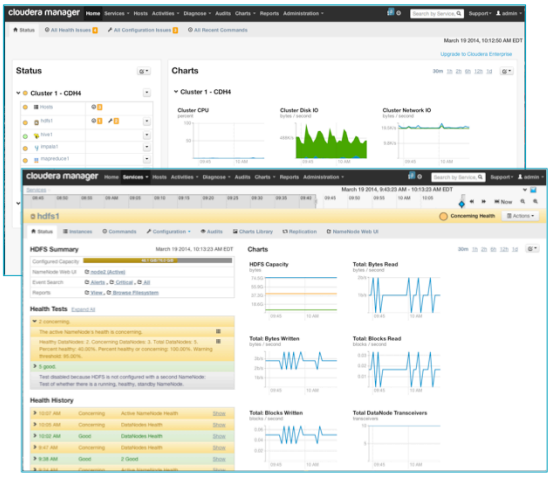


You can think of CDH as analogous to what RedHat does with Linux: although you could download the "vanilla" kernel from kernel.org, in practice, nobody really does this. If you're in this class, it's probably because you're thinking about using Hadoop in production and for that you'll want something that's undergone greater testing and is known to work at scale in real production systems. That's what CDH is: a distribution which includes Apache Hadoop and all the complementary tools you'll be learning about in the next few days, all tested to ensure the different products work well together and with patches that help make it even more useful and reliable. And all of this is completely open source, available under the Apache license from our Web site.

==========

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



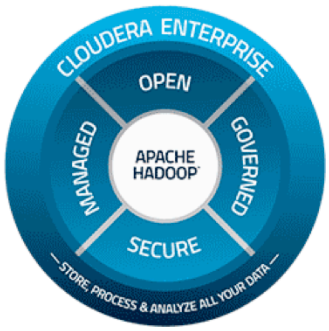
 01-10

Main point: Cloudera Express is free, and adds Cloudera-specific features on top of CDH, in particular Cloudera Manager (CM).

Note in case anyone asks: the free version of Cloudera Manager (i.e., the product now known as Cloudera Express) once had a 50-node limit, but no longer does.
==========

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

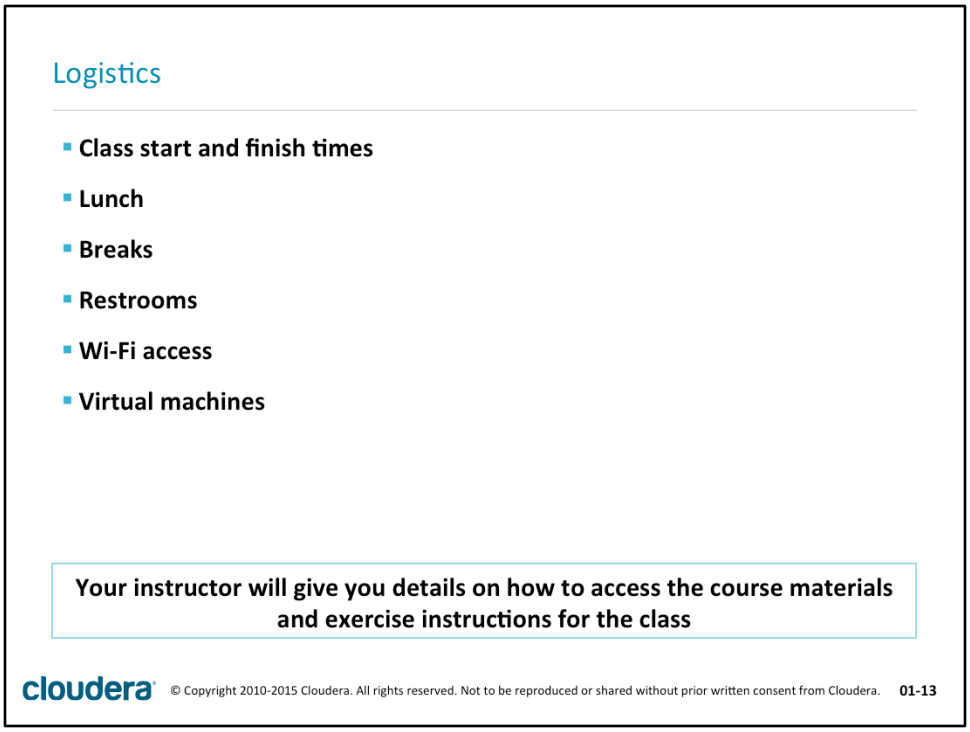


LDAP = Lightweight Directory Access Protocol
SNMP = Simple Network Management Protocol
==========

Chapter Topics

Introduction

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



"Virtual machines" is a cue for the instructor to briefly explain how hands-on exercises will be performed in this class; for example, whether that is through virtual machines running locally or in the cloud. This is also a good time to verify that the virtual machines are already running, and to start them if they are not.

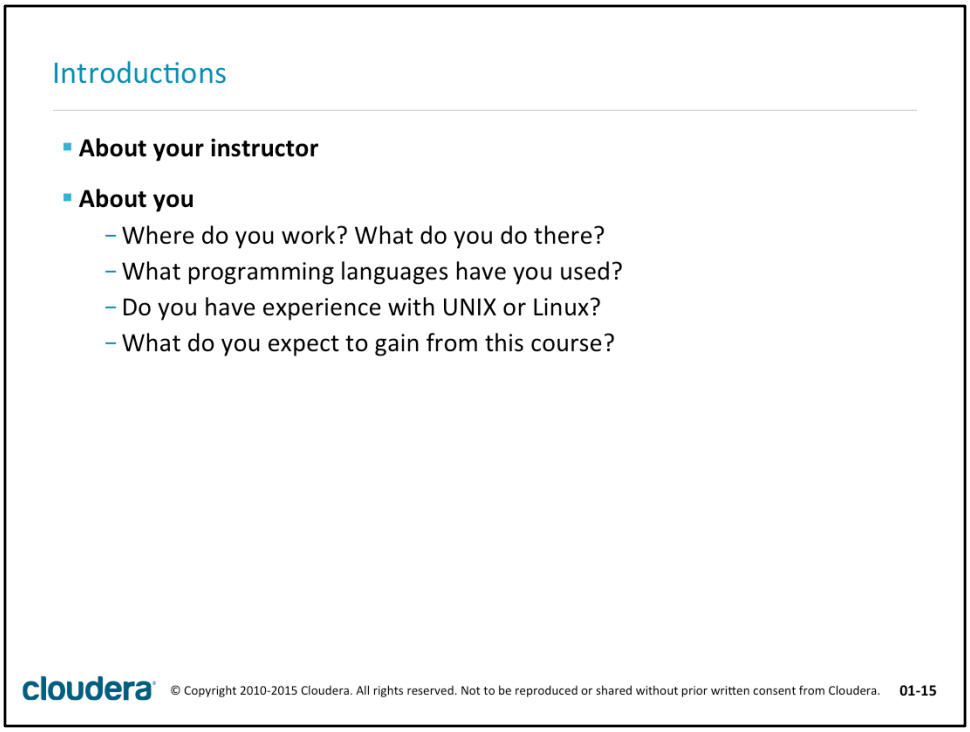
The registration process for students is:

1. Visit [`http://training.cloudera.com/`]
2. Register as a new user (ensure that you give an e-mail address which you can check from here, since the system will send a confirmation e-mail to you)
3. Confirm registration by clicking on the link in the e-mail
4. Log in (if the system has not already logged you in)
5. Enter the course ID and enrollment key (which the instructor will have been sent the week before the class starts)
6. Once this is done, the student will be taken to the course page, from which they can download the slides and exercise instructions. Emphasize that they must, at the very least, download the exercise instructions. Also, unless this is an onsite course they should not download the VM – it's already on the classroom machines, and trying to download it will just swamp the training center's bandwidth.

Chapter Topics

Introduction

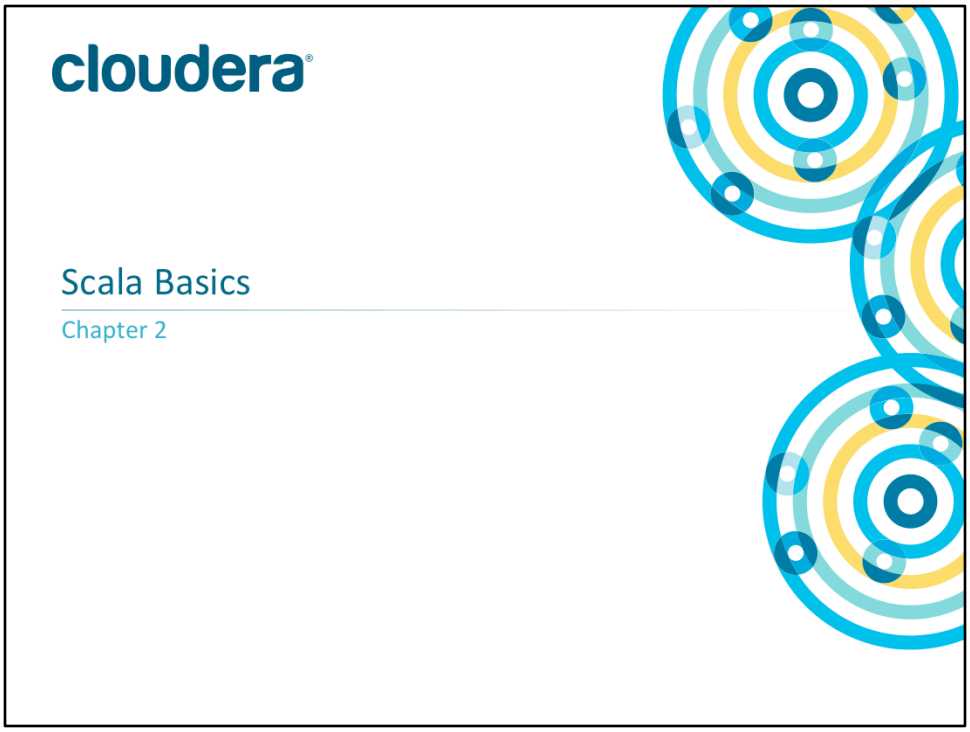
- About this Course
- About Cloudera
- Course Logistics
- **Introductions**



Establish your credibility and enthusiasm here. You'll likely want to mention your experience as an instructor, plus any relevant experience as a developer, system administrator, DBA or business analyst. If you can relate this to the audience (because you're from the area or have worked in the same industry), all the better.

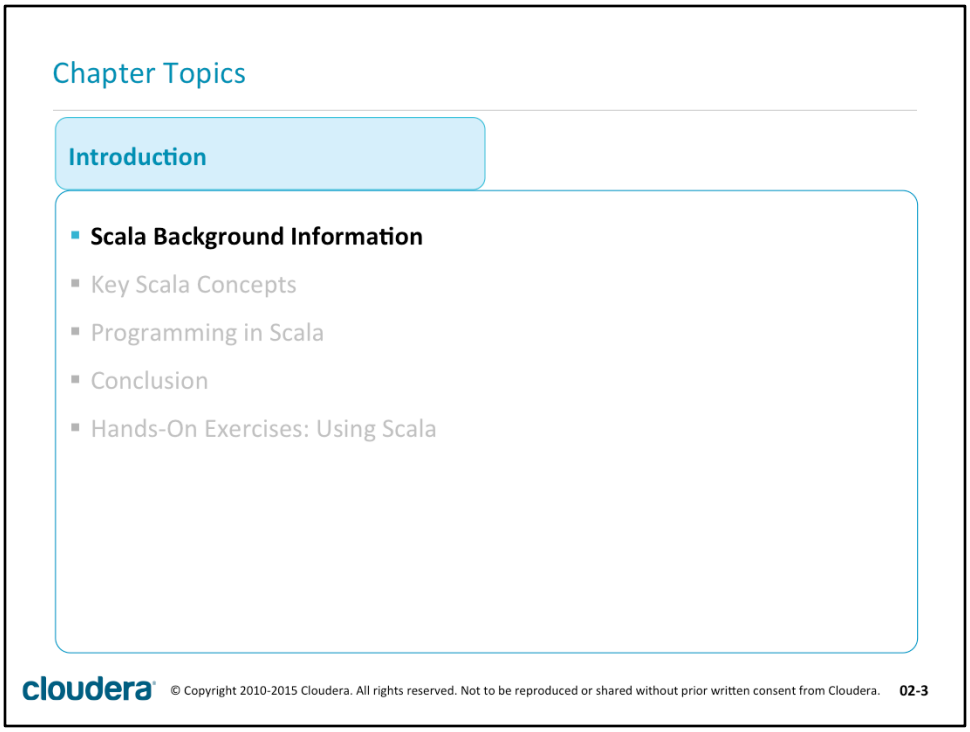
This is also an opportunity to get to know the students, so you can tailor your explanations and analogies to the experience that they already have. It's a good idea to draw out a grid corresponding to the seat layout and write students' names down as they introduce themselves, allowing you to remember someone's name a few days later based on where they're sitting.

The outline for all our courses are available online [`http://cloudera.com/content/cloudera/en/training/courses.html`], so you should be familiar with them and will know whether a student's expectations from the course are reasonable.

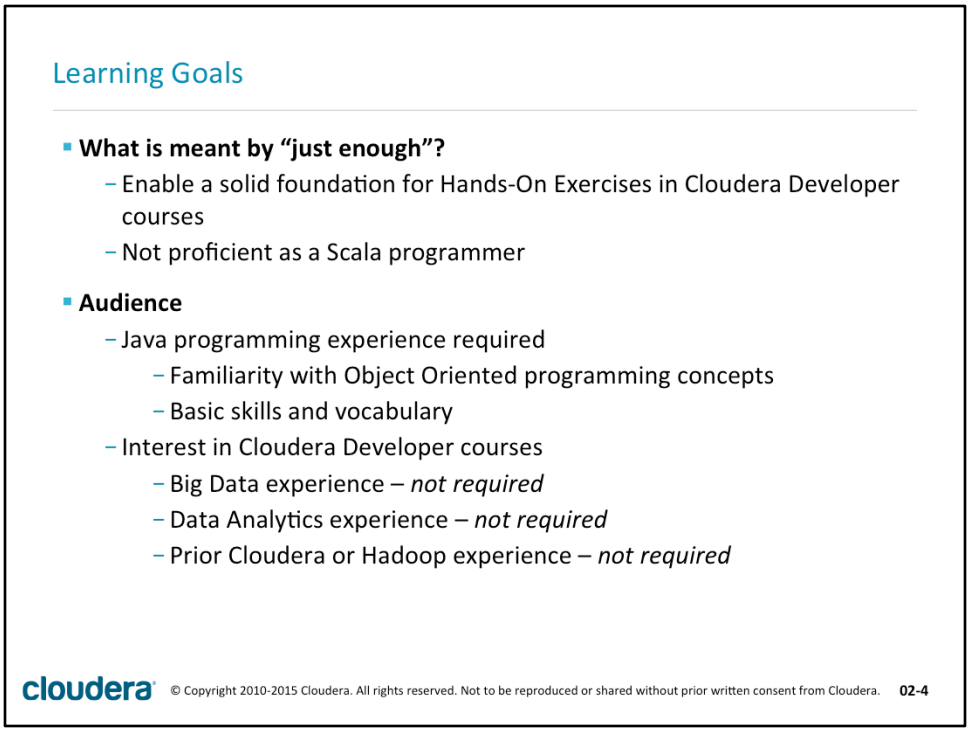


Course Chapters

- Introduction
- **Scala Basics**
- Variables
- Collections
- Flow Control
- Libraries
- Conclusion



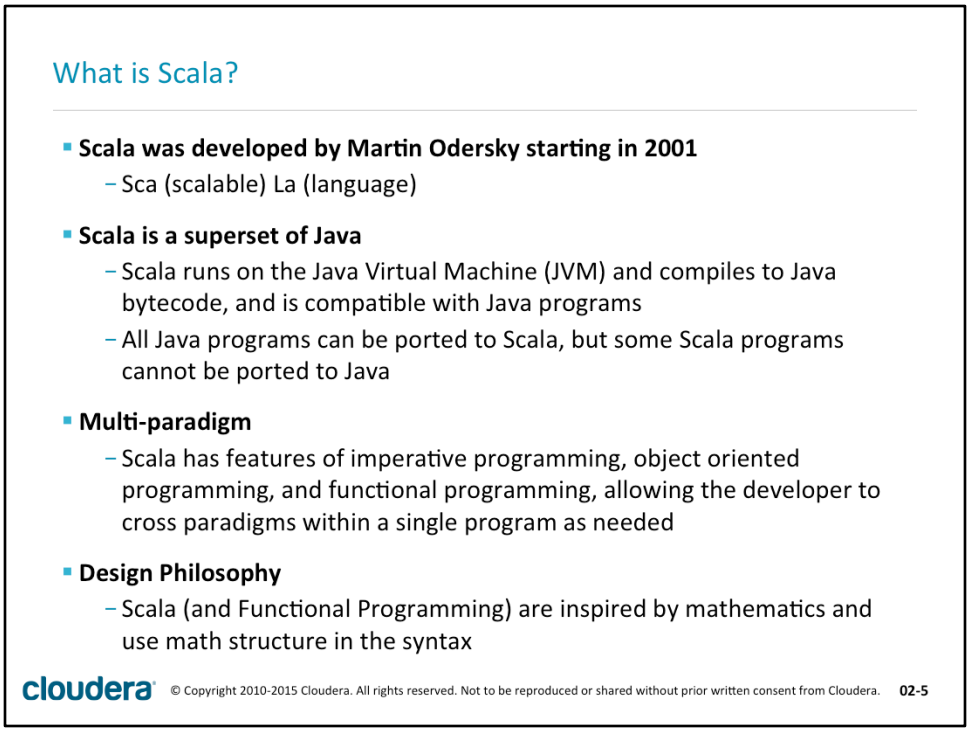
Let's discuss a little about Scala, and why it has become a popular language in the Big Data era.



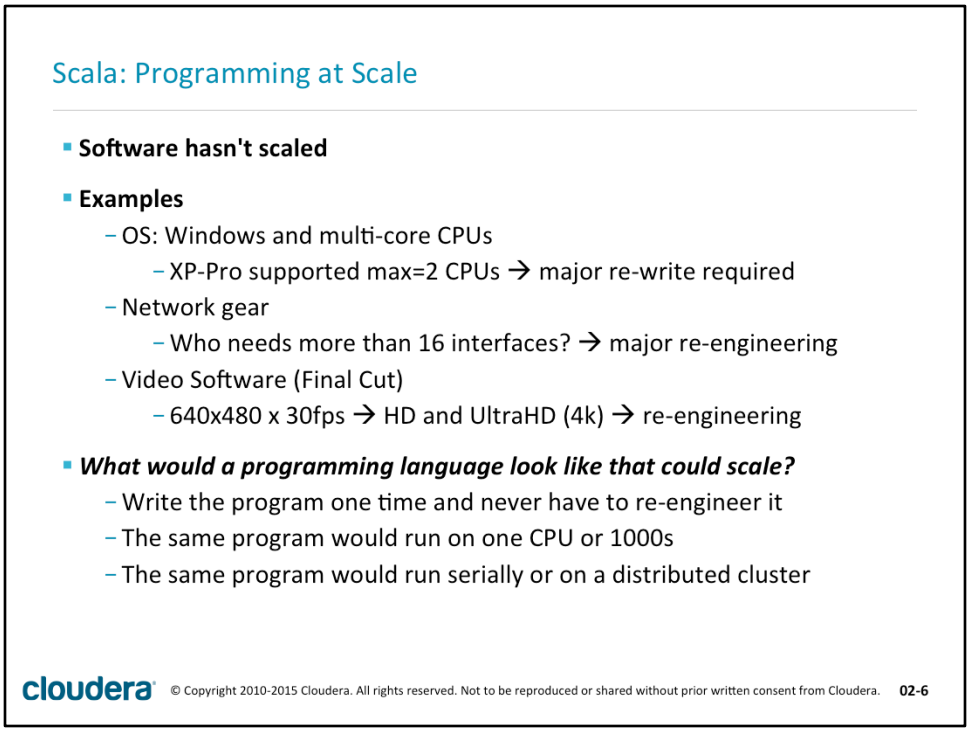
In this class we are very specifically preparing you for our Cloudera classes as rapidly as possible. So… we aren't trying to cram everything into this crash course. In fact, it is not a goal for us to make you into an expert Scala Programmer, but rather to teach you 'Just Enough' Scala to not have that become a distraction when you are trying to learn to develop with Hadoop in our later classes.

Note that knowledge of Java is a *pre-requisite* for this course. Scala shares many elements with Java, so this course will highlight the parts of Scala that 1) differ from Java, and 2) are required in Cloudera courses.

What this means in practice is that we will be focusing more on Scala using an API/ framework than in developing one. For instance, we will not be covering "traits" much (Scala's version of Java's interfaces, allowing multiple inheritance) because these are most useful when developing a framework than when using one.



Java programs can be implemented or ported into Scala. However, the reverse is not true. A program that uses Scala's additional features cannot be directly implemented in Java



Countess Lovelace is said to have posed the question to Charles Babbage: "What if there were a language which, when a question is properly stated in that language, that statement IS the answer to the question?"
And that was the beginning of computer programming languages.

Developers learn after having been through multiple rounds of re-engineering of software and systems that software doesn't scale. The solution for one ATM or a hundred ATMs is not a solution that scales globally. Underlying simplifying assumptions that enabled software to be developed rapidly (i.e. Agile method) reveal their weaknesses at scale. There is an old saying in the Silicon Valley that "after you have survived the test of failure, you have to survive the test of success", because success – scale – is what breaks everything and reveals the tacit assumptions that were incorrect.

With Scala, we ask the question "What if there were a language which, once a program is properly written in that language, is automatically scalable and never has to be re-engineered?"

Properties of a Scalable Language

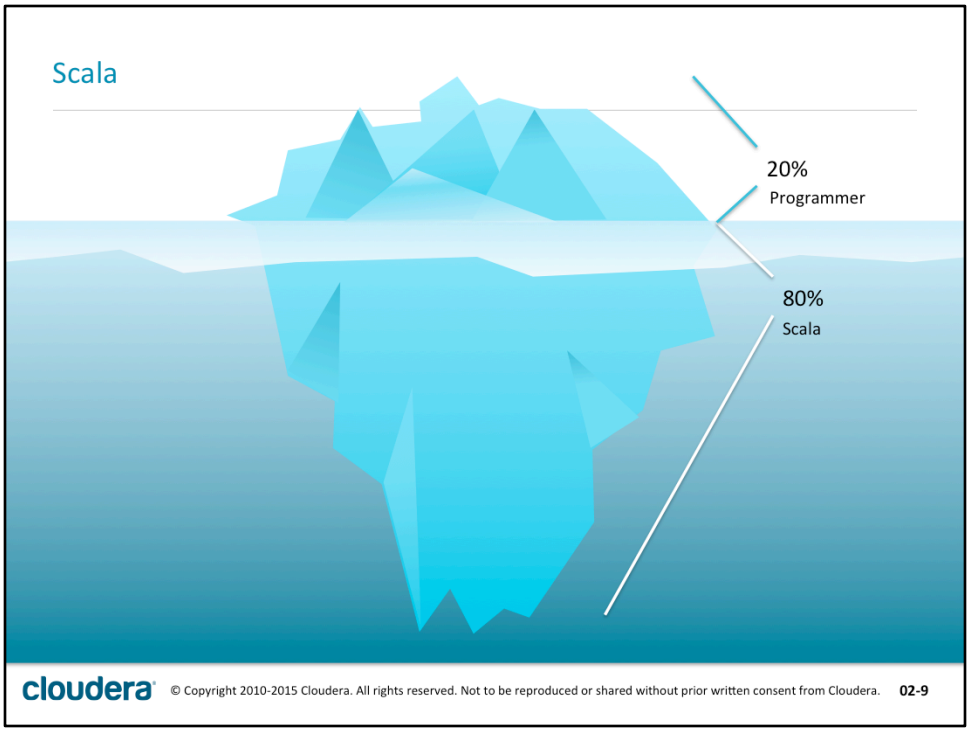
- **Design Goals**
  - Eliminate, hide, and prevent the source of distributed processing bugs
  - Scale = implementation and optimization decisions
  - Assign scale decisions to the programming framework
  - Enable the programmer to take control when required by the application – to exchange control for scalability
- **Methods**
  - Pure functions
  - Immutable data
  - Implicit looping and iteration over collections

cloudera

We will explore the methods in the coming slides.

Scala and Java

- **All Java types are available**
  - But in most cases you will want to use the Scala equivalents to gain the distributed/scalable benefits
  - Using native Java types might make your program scale poorly
- **Scala is more expressive**
  - Concise / dense
    - 10 lines of code in Java might be equivalent to 2 lines in Scala
  - Implicit
    - Scala implicitly passes data between parts of an expression, hiding it, so you do not have to manage it



It is estimated that 1 line of Scala can replace 10 lines of Java.
Scala is like an iceberg. 80% of what it does is below the surface – concealed from the programmer.
While that makes it a very sophisticated and powerful language, it also makes it challenging to learn and understand

Scala Documentation and Resources

- **Official Scala website: `scala-lang.org`**
  - Documentation such as API, glossary, style guide, language specification and FAQs
  - Download Scala binaries, source code, documentation, or tools
- **Scala books**
  - *Programming in Scala* by Martin Odersky, Lex Spoon, and Bill Venners (published by Artima)
  - Programming Scala (published by O'Reilly)
  - *Scala in Depth* and *Scala in Action* (published by Manning)

cloudera

Instructors: you may wish to bring up the scala-lang.org site for students, especially if they do not have internet access in the classroom. If internet access is available, suggest they visit the site.

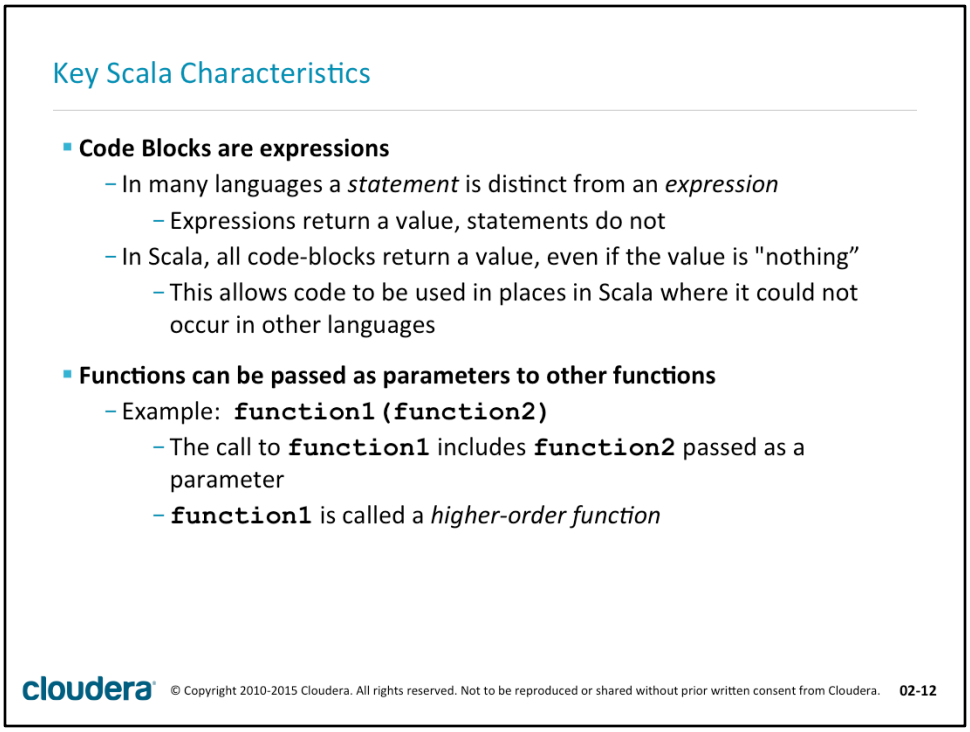
Students are *encouraged* to use the API and other documentation throughout the course!

The first two books in the list are available for reading online for free.
A more complete list of Books is here: http://www.scala-lang.org/documentation/books.html (including non-English books; international instructors may wish to point students to this page)

Chapter Topics

Introduction

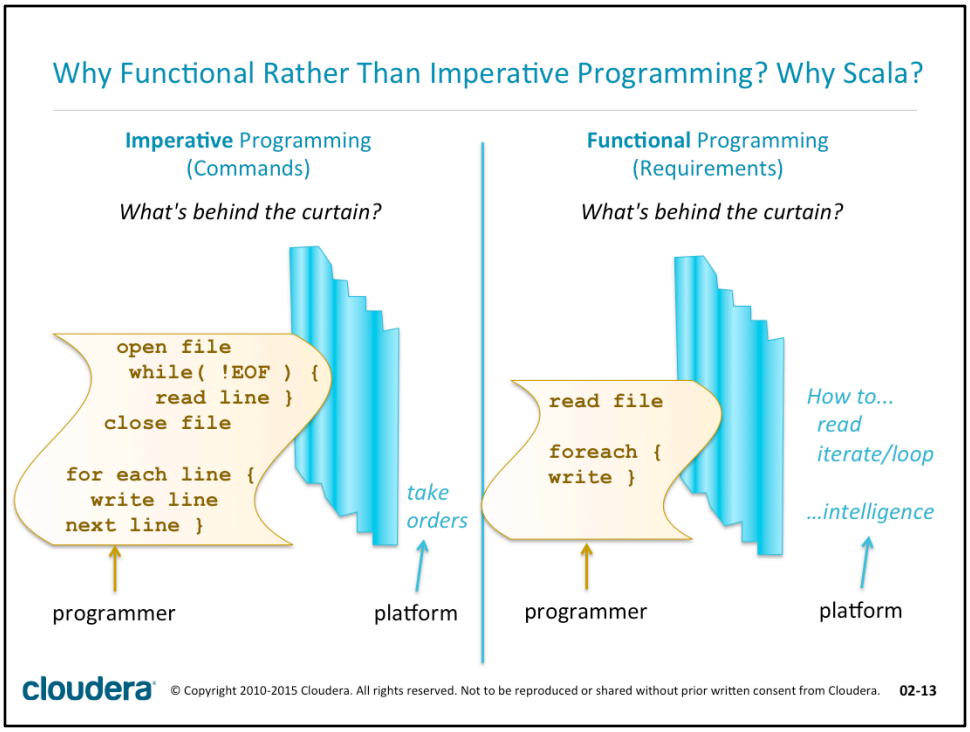
- Scala Background Information
- **Key Scala Concepts**
- Programming in Scala
- Conclusion
- Hands-On Exercises: Using Scala



Instructor:

The second bullet telegraphs "functions receiving other functions as parameters", and highlights the common jargon that can be confusing to the new learner. I would guess that about 80% of the available Scala resources try to explain this feature by ***nominalizing*** – they call the quality of the language either support for "higher-order functions" or they say that "functions are first class citizens", and then they FAIL to explain it further. It is as if the author has given up on actually explaining the foundational concept, and so resorts to "name calling", as if the learner is supposed to understand from the name all the subtleties of the concept that have gone unexplained. Higher-order functions are defined in a branch of mathematics called the "untyped lambda calculus". Higher-order functions are only supported as of this writing (July, 2015) in: Scala, Python, F#, Haskell, Javascript, Perl, Gosu, Ruby, Matlab, Clojure, and Scheme. Java programmers, the target audience for this class, will not have exposure to the concept. Python programmers can easily avoid higher-order functions, as it is not a core construct of the language. In Scala and Haskell, higher-order functions are essential to the way proper programs are written in the language, and avoiding them creates poorly designed code in those languages.

You will sometimes hear about scala the phrase “first class citizens”. Computer languages that support higher-order functions are said to treat functions as *first class citizens*, meaning that functions have properties which, in other languages, are only available to variables.



Scala is built on Java. What makes ScaLa Scalable? What does that really mean? And why do we have to learn to program differently or use a different language?

The illustration above shows the difference between Imperative Programming and Functional (sometimes called declarative) Programming. Imperative Programming (on the left) is the design method implemented in most common computer languages at this time. In Imperative Programming, the idea is that the programmer issues commands that tells the computer exactly how to accomplish the result. A recipe is a good analogy. The recipe provides explicit detailed instructions. And if the recipe is correct and is followed correctly it reliably and repeatedly produces the expected results. This leaves very little flexibility for the platform. The platform is basically an order-taker and resource scheduler. So it is not possible for the platform to apply much intelligence. For example, if the platform had 1000 CPUs available could it scale up to effectively use those CPUs? Probably not. Because the platform doesn't have the domain of control or permission to do that. So the programmer would have to re-write the program to scale it up to effectively use additional CPUs.

On the right side of the illustration is Declarative Programming. One form of this is Functional Programming, which is implemented in Scala. In a functional design, the idea is that the programmer declares what is required, but doesn't get into the detail of exactly how the result is to be accomplished. Instead, the platform is responsible for implementing the requirements specified by the programmer. That means the platform has the "wiggle room" to apply intelligence and to employ different methods to produce the results. So long as the results are as expected – meet the requirements – the platform is able to adapt the implementation of the program.

There are several other languages that offer FP. Scala seamlessly allows the programmer to move between FP, IP, and OO (Object Oriented) paradigms. So it's very useful for production applications, where the programmer can take control with imperatives, and then release control to the platform for scale.

Pure Functions Versus Side Effects

- **Scala design is inspired by math**

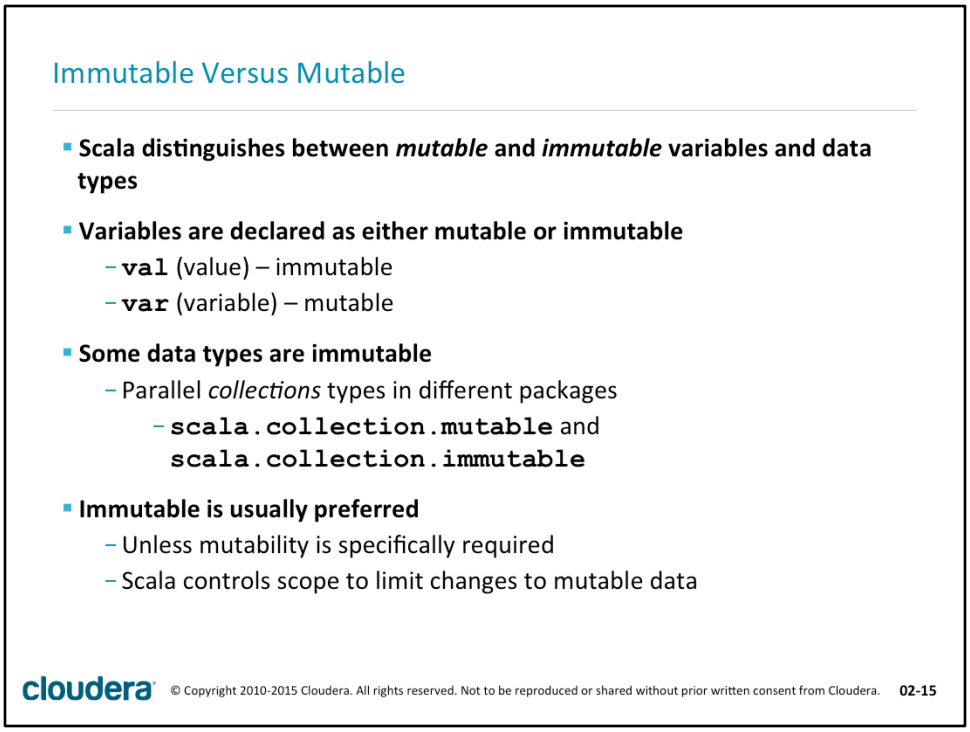
$$y = f(x)$$

A mathematically *pure* function
- For every value of x there is one and only one value of y
- The value of y for any x cannot change over time
- f(x) depends on no outside values or variables
- f(x) effects no outside values or variables

Side effects are required for computing
- Input requires outside data
- Output requires generating data
- Real-world values may change over time
- External dependencies may be a practical necessity

cloudera

The idea of functional programming isn't to *eliminate* side effects, but to *isolate* them from pure functions, minimizing the window of vulnerability for distributed computing errors



Immutable variables or data cannot be changed once set.
Mutable variables or data can be changed -- possibly without the data creating having awareness of the change
Mutable data is a major cause of problems in distributed computing.
The combination of immutable data and pure functions that do not have side effects results in programs that can be implemented by a scalable framework in different ways (depending on resources available) and still produce the same or equivalent results.
Sometimes you need to share data, so mutable is available
Mutable data is not prevented, but discouraged – just as pure functions are not required but suggested

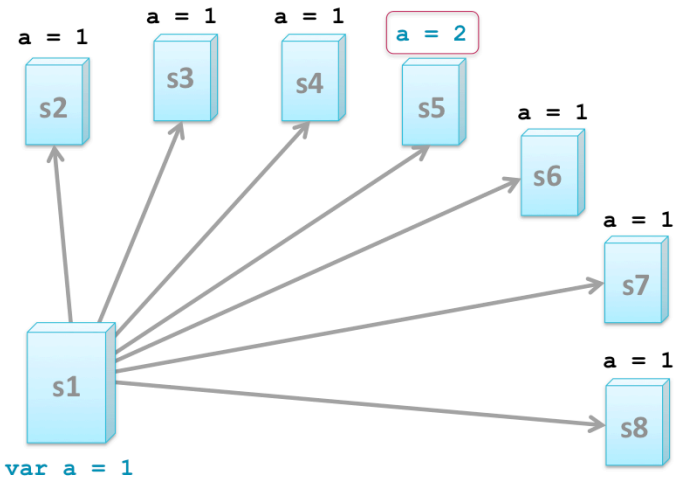
Instructor Value-Add

Dealing with distributed application design is not new. For example, Networking is exactly a distributed problem in Computer Science. And every routing algorithm is an attempt to deal with the potential issues that can arise in distributed processing. It's a complicated area of Computer Science. Fortunately, Scala provides some basic structural safety features to constrain the developer to write scalable code that is ready for distributed systems.

It's a bit strange for the person new to distributed systems programming, because language idioms that the programmer has relied on in a uniprocessing context are no longer acceptable, and it won't be immediately obvious why this reliable technique is now off limits, and new techniques must be learned.

The next slide digs into this concept.

Distributed Processing and Mutable Variables



 02-16

This illustration is to help explain why Mutable Variables are the source of many problems in Distributed Processing.

1. There are eight servers in the cluster.
2. On the lower left we have s1, the originator of variable 'a', that sets it's value to '1'.
3. The variable is shared with the other servers in the cluster, so all have the value of a=1.
4. Server s5, in the upper right, changes the value of a to '2'. Now we have a system integrity problem. Overall, for the system as a whole, is a=1 or a=2 true?
5. Which system is the authority for "truth"? How do the other servers learn the truth? And what happens if they are processing based on old data?

Let's say that any system that updates the value is assumed to do so with good reason, so the most recent value is the "real" value, and the other systems in the cluster are behind. s5 needs to have some kind of coordination message causing the other servers to halt processing until all the servers in the cluster have updated their local value of a to a=2. If you have each server block until all 8 servers report they are updated, you preserve temporal integrity and parallelism of processing, but the point of parallel processing is lost because the overall system is now no longer truly parallel, but instead it's single processing within the lock step defined by synchronizing around the value of a. It's fake parallel processing. If you allow each server to resume as soon as it's value for a is updated, then you reduce blocking at the expense of losing temporal parallelism. Worse, what happens if another server decides that "a=3" before the entire cluster is updated to "a=2" ? This is called contention, and the process of resolving it is called convergence. It leads to a class of race conditions (time-based bugs) that are really, really, really hard to troubleshoot. You can design safety into such a system, but for each degree of convergence resolved, there is another degree n+1, which is not solved and can produce an even harder bug to resolve.

One solution, assume that if a=1 is set as an immutable variable, what Scala calls a "value", then no system can ever change it. s5 can't change the value of a, so the problem is removed. The programmer will be prevented from writing code that modifies a, and will have to use some other approach to accomplish their purposes that does not create distributed problems. A second solution is to break the dependency between the servers in the cluster. So initially, every server gets the value of a=1, and their local copy of a is a separate variable from the original and from one another. They can modify it as much as they want, with the caveat that the change in value is local and will NEVER be communicated back to the creator of the variable or to any of the other servers in the cluster. And this is the concept behind Scala's implementation of a mutable variable. So when the programmer codes "var" instead of "val" they are taking responsibility to have the discipline to never write code with distributed dependencies.

You can think of coding "var" instead of "val" as "seatbelts off". You've got to drive carefully because the consequences to scalability are serious. Therefore, Scala does everything possible to encourage using "val", but doesn't prevent "removing the seatbelts" if that seems necessary.

Implicit Iteration

- **How does Scala make iteration over the elements of a collection (such as a list) scalable?**
- **Scala uses a method uncommon in other languages**
  - It creates *implicit variables* to pass data between parts of an expression
  - Eliminates the need for *counters* and *state variables* to keep track of iterable state
- **Removes the main source of distributed bugs**
  - Counters and state variables cannot be addressed/changed because they are hidden/implicit
- **Puts optimization in the execution framework**
  - Scala or the distributed framework used by Scala determines *how* the iteration is implemented

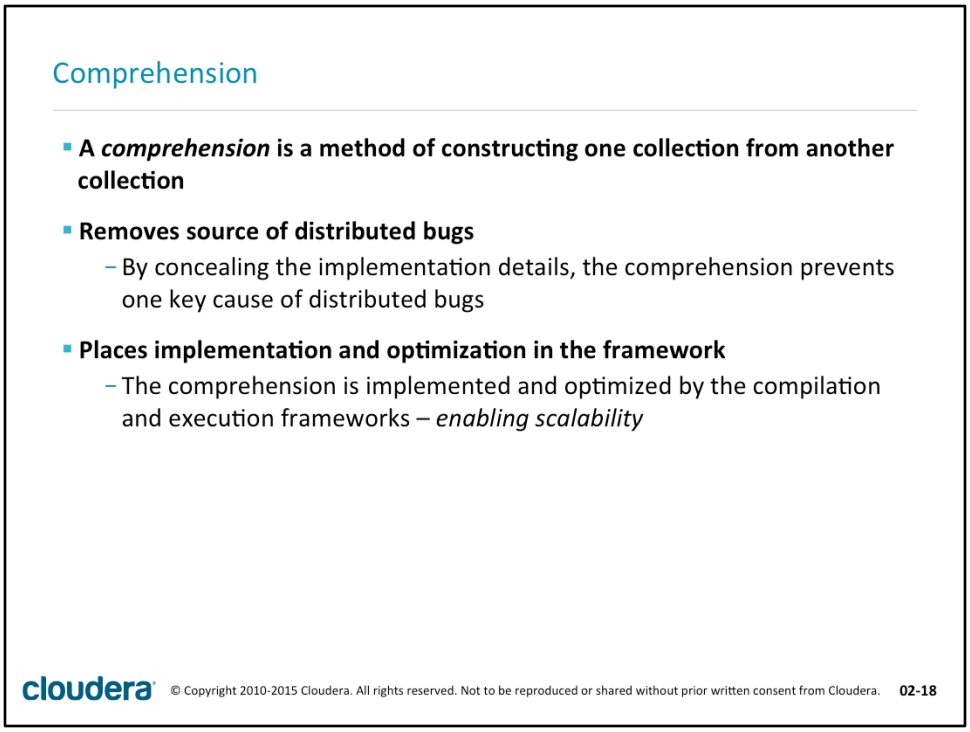
cloudera 02-17

Scala's use of implicit variables in iteration is one of the novel aspects of the language – that makes Scala hard to read and hard to debug at first

Instructor Value Add: This works well as a chalk-talk, drawing on a whiteboard:

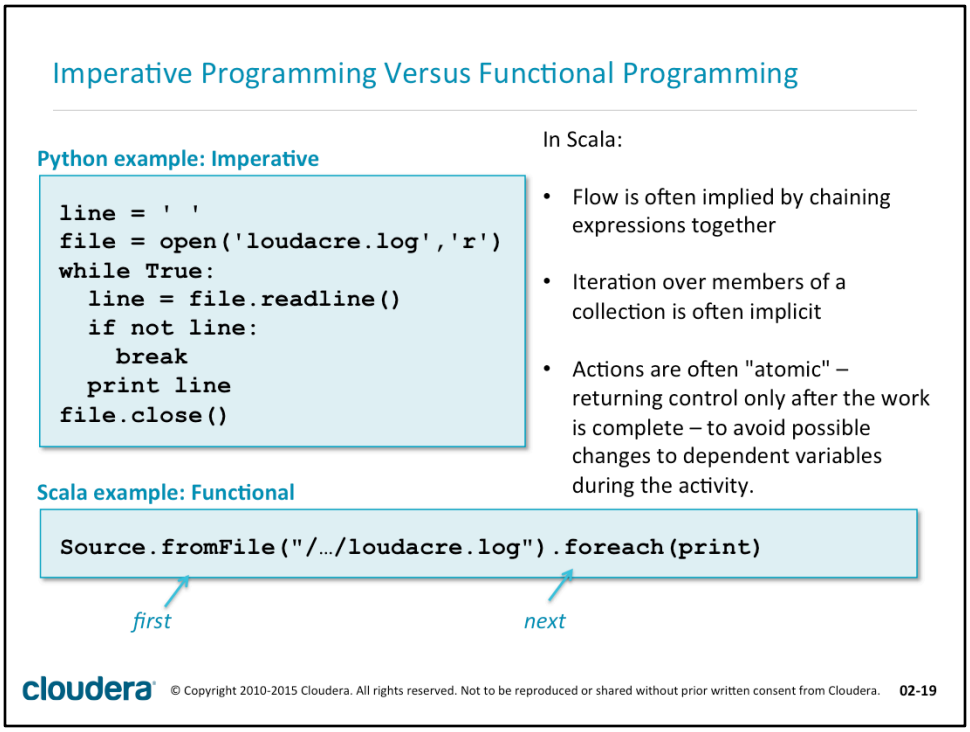
Show a collection of work to be performed. You can illustrate this as a 16-square block made of 4x4 squares. Number them 1-16.
Next, draw four servers in the cluster that will perform the work.
Now, place a single counter, 'a', for the address of the next block of work to be performed.
Each server has to block on 'a' until it is available. Then they take a lock on 'a' so no one else can get to it.
They take the next block of work and increment 'a' by 1.
Explain how since the servers are all blocking on one another's access of 'a' that
really not a scalable solution. If there were 1000 servers, the overall system would spend most of its time blocked on itself and not processing. Consequently, the more servers in the system, the more the system approaches uniprocessor performance – the worse it gets.
Now explain an alternative where the work is divided by row, and each server is given its own row of four work items.
Each server has its own copy of 'a', which it uses to track its own work.
This is a scalable solution. What happened?
By dividing the work up in advance, before processing actually began, the dependency between parts of the distributed system was eliminated.
This is a simple example of the change in thinking that is necessary for scalable programming.
And running through this example will help the features of Scala make sense.

As much as possible, we want to hand Scala something that it can optimize without scaling issues. So a lot of the time we are not trying to design the program for scalability, but instead trying to avoid the pitfalls of non-scalability so we don't hamstring Scala or the distributed infrastructure its using to scale up processing.



A comprehension has the same effect in many cases as iterating or looping over the elements of a collection while generating a new collection

Note the two themes here – [1] preventing the source of distributed bugs through concealing code, and [2] placing implementation and optimization in the hands of the framework.
The repetition is not accidental. This is a key aspect of Scala's design. And we will be elaborating on these themes in the coming slides, then illustrating with code examples.



This example shows a typical way to print out each line of a file. Students may not know Python, which is okay: present it as pseudo code to show a typical *imperative* way to do a task. The Scala code is a typical *functional* way to do the same thing.

It is not important at this point to cover the details of the syntax, this will all be covered later. Here are the important points to make about the Scala code:

1. Scala is a much more efficient (1 line vs. several)
2. Method calls are chained together, each one operating on the return value of the previous one.
3. we are passing a function (print) to the foreach method -- we discussed earlier that a key concept of Scala is the ability to pass a function to a function. This example shows why that's useful…the iteration is done within the foreach method, rather than having to be coded explicitly as in the imperative example.

Scala programming is very different from procedural / imperative programming languages. Scala relies a lot on inheriting functions into derived types of variables, and then calling methods on those variables that handle all of the elements of a collection. For example, in the code above, the object that buffers the data being read is of the type: `scala.io.BufferedSource` This object type handles all of the iteration involved in reading lines of text from a file (using .fromFile) and it also handles all iteration involved in output as in .foreach{ }.

Because Scala is a superset of Java, you could write Imperative code. However, your program would not automatically scale up when run on a distributed platform such as Spark. Therefore, we are going to focus on writing functional code as is used in Cloudera's classes.

Chapter Topics

Introduction

- Scala Background Information
- Key Scala Concepts
- **Programming in Scala**
- Conclusion
- Hands-On Exercises: Using Scala

Programming in Scala

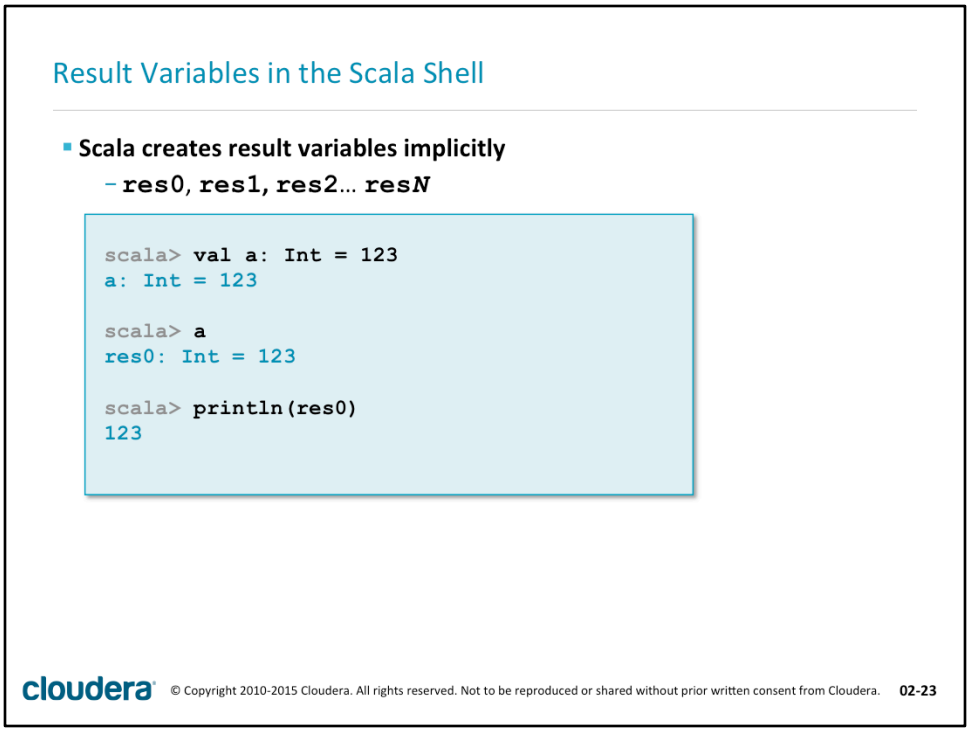
- **You can work with Scala in two ways**
  - Interactively using the Scala shell
  - Writing, compiling and executing Scala programs in a file

Scala Shell

- **Start the Scala Shell from the OS command line by typing: `scala`**
  - 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
    - REPL is useful for interactive code exploration
  - Provides command completion
  - Does not interface with the host file system, no path completion

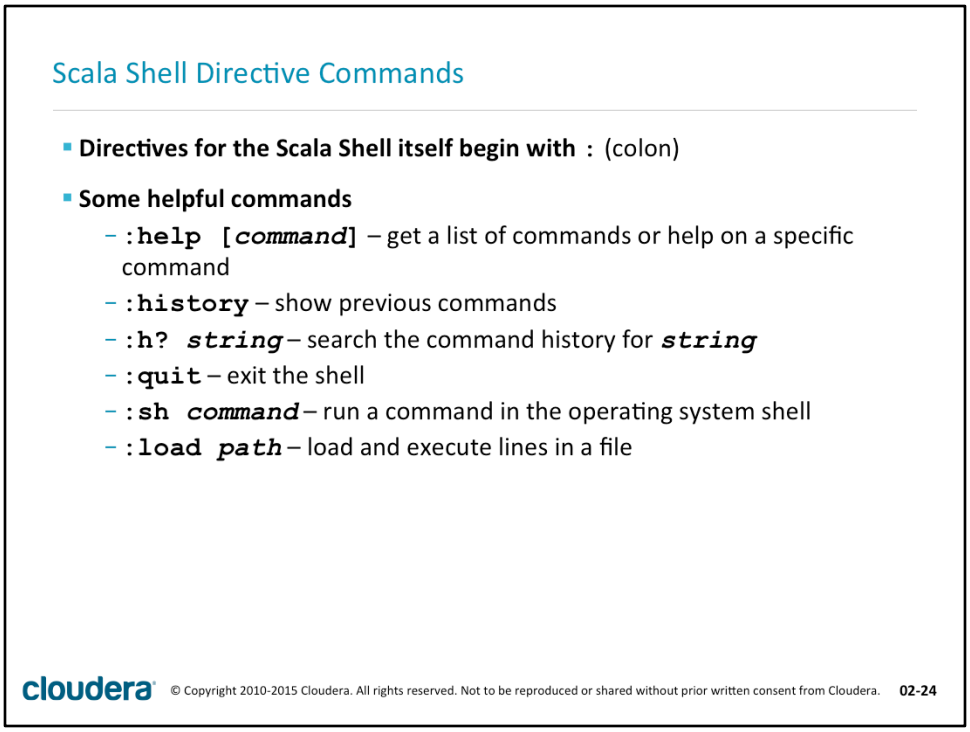
```
Welcome to Scala version 2.11.5 (Java HotSpot(TM) 64-Bit Server
VM, Java 1.7.0_51).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```


http://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop



Note that Scala's REPL numbers implicit variables. So res0, above, is the first implicit variable. These are real variable names. So, for example, you could address 'a' by using the name res0 in the shell.

This code example shows the scala> prompt shell users will see. From here on in the class, though, we will not show the scala> prompt because the code examples are equally applicable to shell or to a standalone program. As will be covered shortly, throughout the rest of the class, Scala output is preceded with >; code is shown without a prompt.



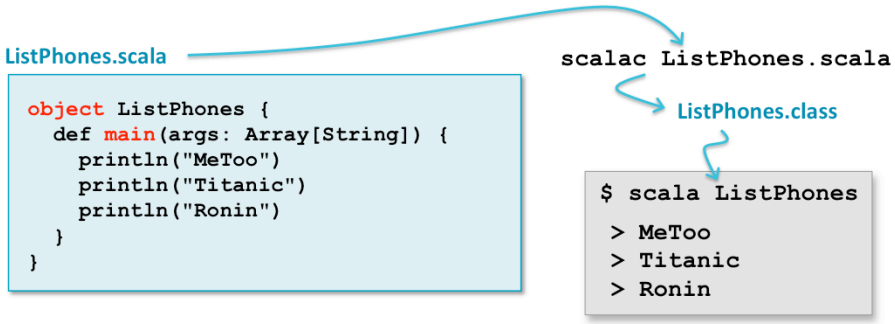
This is just a few of the directives available in the Scala shell.

Note that [command] in brackets means it is optional.

Command in italics is required.

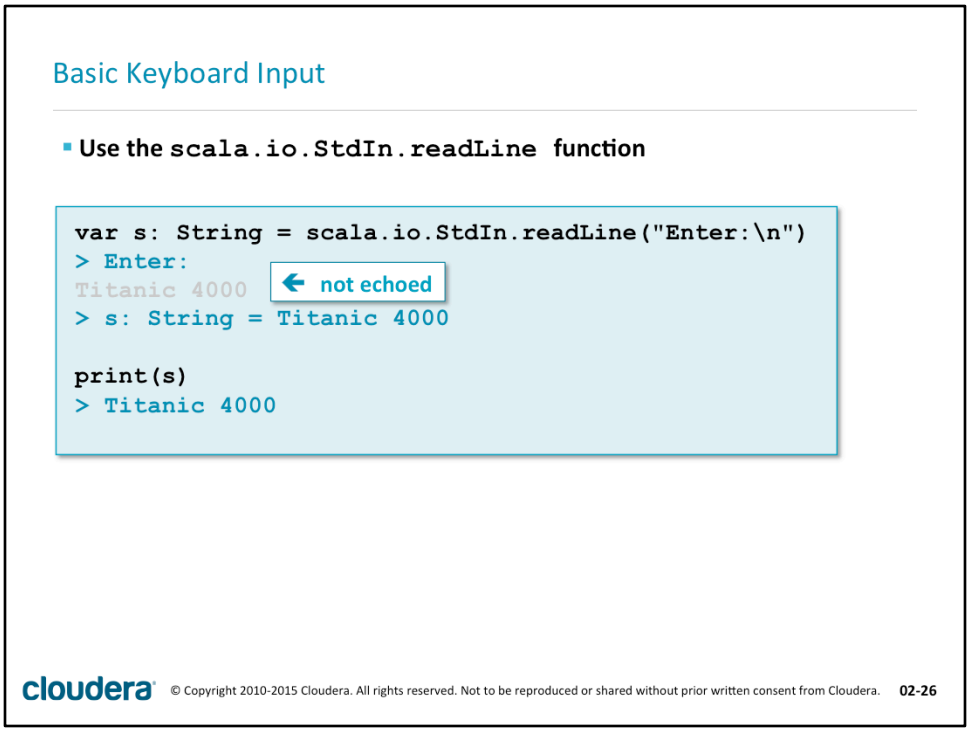
Scala Compilation

- **Edit code in a file using a text or graphical editor**
  - Main program file must contain an **`object`** and **`main`**
  - Convention: Scala files are **`*.scala`**
- **Compile to `.class` file with `scalac` *`program`*`.scala`**
- **Execute class file with `scala` *`ObjectName`***



 02-25

Sample source code is located in training_materials/jes/examples



In this example the user enters Titanic 4000, which is shown in grey but doesn't actually echo back when using the shell or running a program using readLine.

Basic Printing

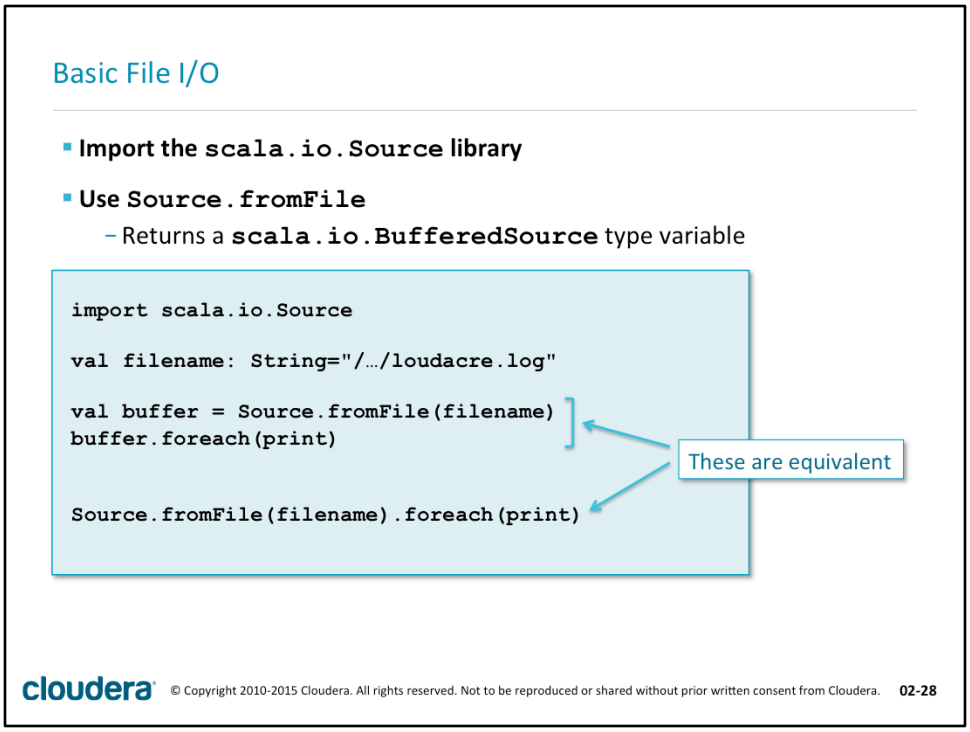
- **Use the built-in `print()` function**
  - Basic printing of variables: **`print(`*`var1, var2, var3…`*`)`**
  - Print formatted string:
    - ***`formatstring`*`.format(`*`var1, var2, var3…`*`)`**
    - Formatting: **`%d`**=integer, **`%f`**= float, **`%s`**=string

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

val formatStr = "Device temperature %d to %f celsius"
print(formatStr.format(24, 31.24))
> Device temperature 24 to 31.240000 celsius
```


another way to format strings is using string iterpolation, which will be covered later in the Variables chapter.

the print (and its sibling println) work similarly to the versions in Java, but they are locally scoped, so you do not need to reference them as System.out.println as you do in Java.

The difference between print and println, as in Java, is that println appends a newline and print does not.



The file read is atomic, a single call that returns after the entire file has been read. This prevents mutation from occurring during the file read.

We haven't covered this syntax yet, which shows both chaining a series of methods, each operating on the output of the other, and implicit looping using the collection's foreach method. Don't cover in detail, it will be covered much more later.

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
```

```
months={1:'JAN',2:'FEB',3:'MAR',4:'APR',5:'MAY',
        6:'JUN',7:'JUL',8:'AUG',9:'SEP',10:'OCT',
        11:'NOV',12:'DEC'}
```

This code should all appear on a single line. It has been formatted for readability.

Chapter Topics

Introduction

- Scala Background Information
- Key Scala Concepts
- Programming in Scala
- **Conclusion**
- Hands-On Exercises: Using Scala

Essential Points

- **Scala = Scalable Language**
  - Pure functions versus side effects
  - Immutable versus mutable
  - Implicit looping / comprehensions over collections
  - Higher-order functions, passing a function as a parameter
  - Method chaining
- **Just Enough…**
  - This course is designed to teach "just enough" Scala to support Cloudera's Developer courses

Chapter Topics

Introduction

- Scala Background Information
- Key Scala Concepts
- Programming in Scala
- Conclusion
- **Hands-On Exercises: Using Scala**

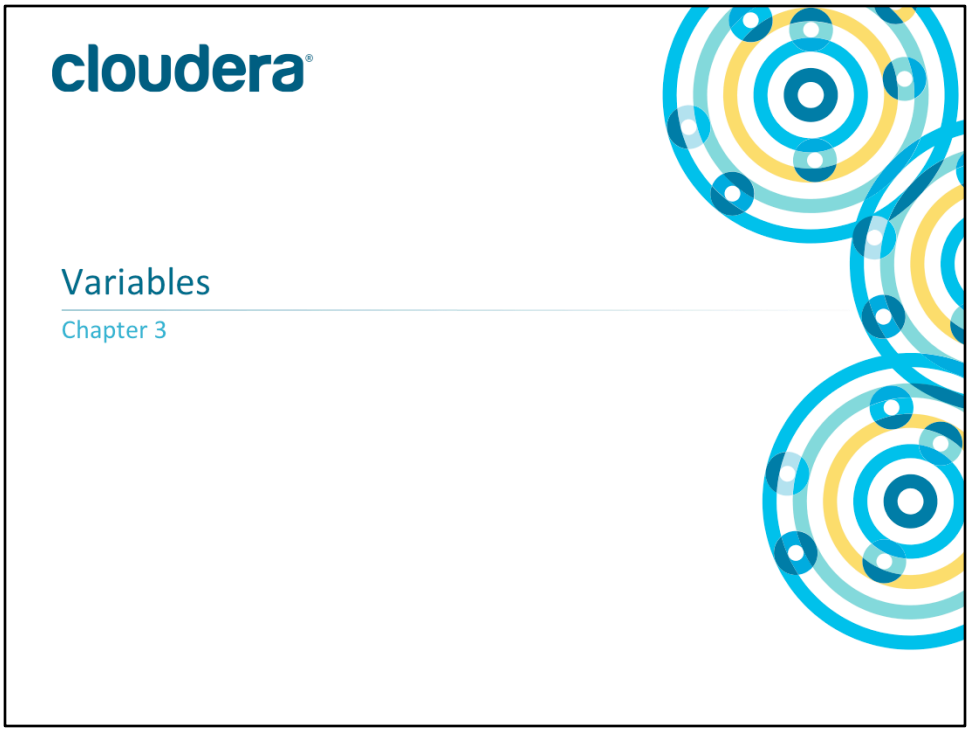
Exercise Introduction: Loudacre Mobile

- **Loudacre is a (fictional) mobile phone carrier**
  - Hands-On Exercises use example Loudacre device log data from mobile phones
- **About the log data**
  - Every time a phone has a error requiring a soft or warm restart, the phone sends device status information to the central system where it is collected for later analysis

Instructor: Review the next several slides on basic programming before going to the first preparatory exercise.

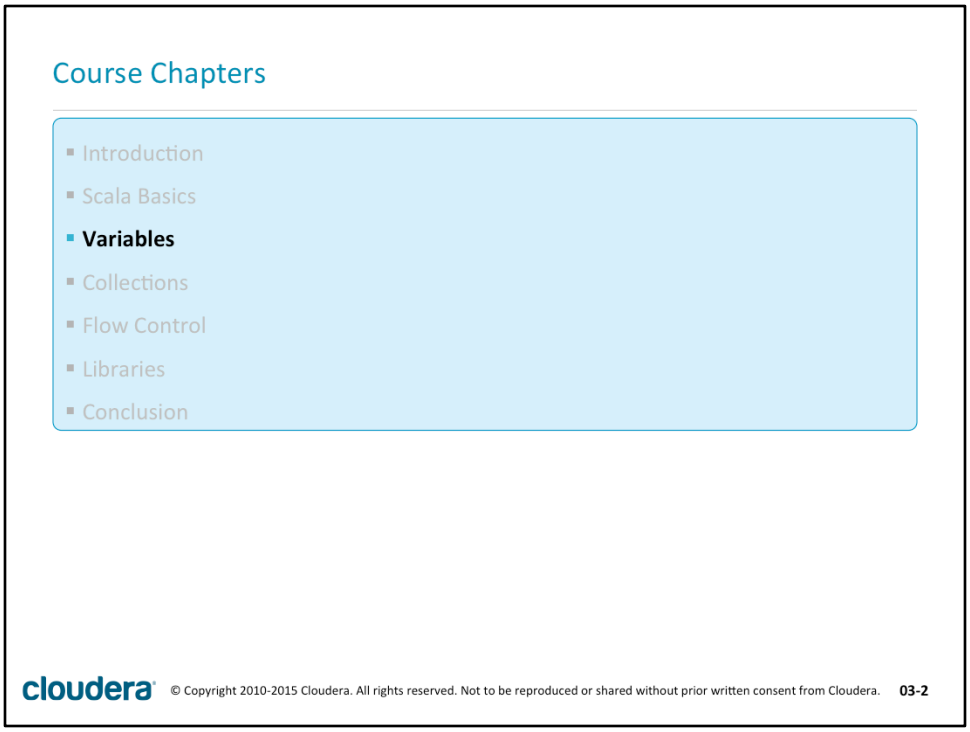
Hands-On Exercise: Using Scala

- **In this exercise, you will**
  - Perform some simple Scala commands using the Scala Shell
  - Write, compile and run a simple Scala program
  - Explore the `loudacre.log` data file
- **Please refer to the Hands-On Exercise Manual for instructions**



In this chapter we will explore basic variables.

The data structures supported in a language have a profound influence on the general approach to problem-solving and the way in which programs are written.

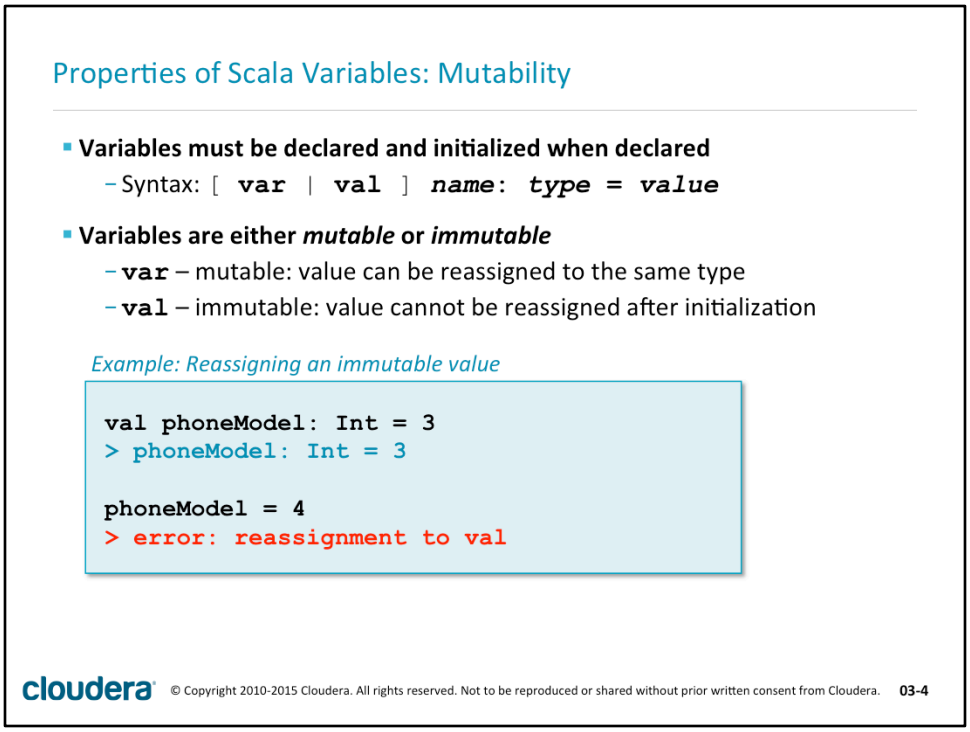


In this chapter we look at Scala's support of simple variables. And in the next chapter we will look at collections.

Chapter Topics

Variables

- **Scala Variables**
- Numeric Types
- Boolean
- String
- Conclusion
- Hands-On Exercises: Variables

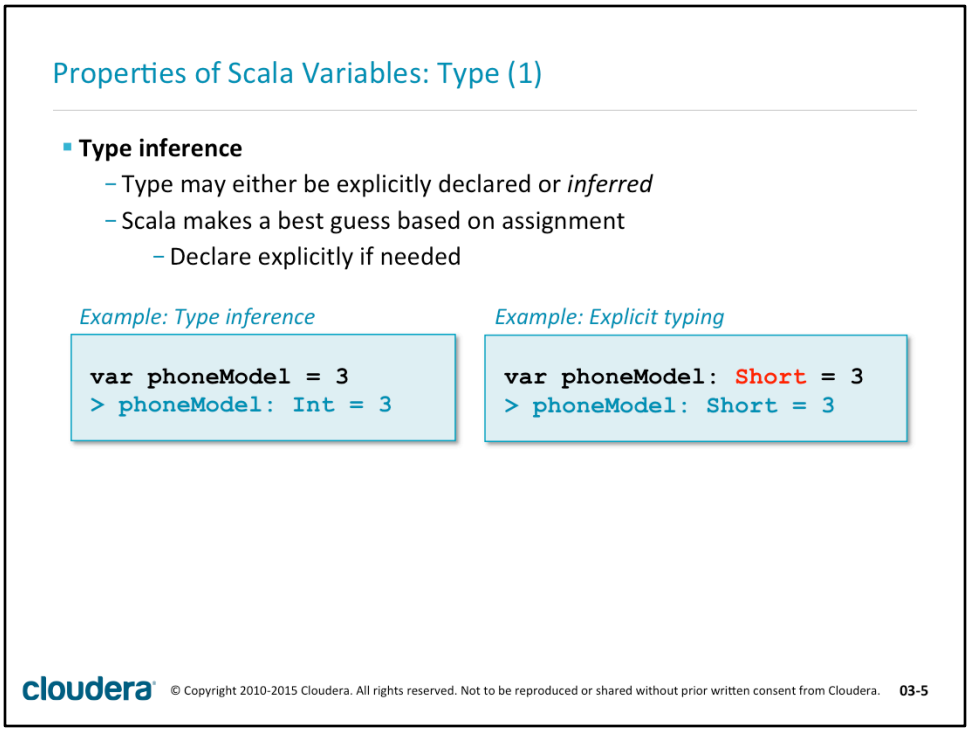


Remember that Scala's use of mutable and immutable variables was introduced in Chapter 1. The main purpose of distinguishing these types of values is to facilitate support for a distributed framework such as Spark.

- **`var`** → variable
  - The variable is *mutable*
  - Can be re-assigned at any time to the same type

- **`val`** → value
  - The variable is *immutable*
  - Must be assigned a value during definition
  - Can't be assigned a different value after initialization

In general, always use val unless there is some specific reason the program will have to modify the variable once set.



Sometimes Scala can determine the type of a variable by the value assigned to it.

If the value being assigned is known (for example, it is assigned from another variable, or the return value of a function with a given type), that value will be used.

If the value assigned is a literal (as shown here), Scala will attempt to make a best guess based on the format of the literal. So an integer like 3 as shown here is assumed to be an Int.

However, sometimes Scala will be unable to determine a type, or the type it infers is not what you need it to be, in which case you can declare explicitly. In this example, we want phoneModel to be a Short (meaning a two byte integer, rather than a four byte integer)

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
    - Attempts to use 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
```


Note that although, as demonstrated on this slide, you cannot reassign an immutable variable, you can *redefine* any variable. So although this is not valid:

val x = 1
x = 2

This is valid:
val x = 1
val x = 2

Redefining Variables

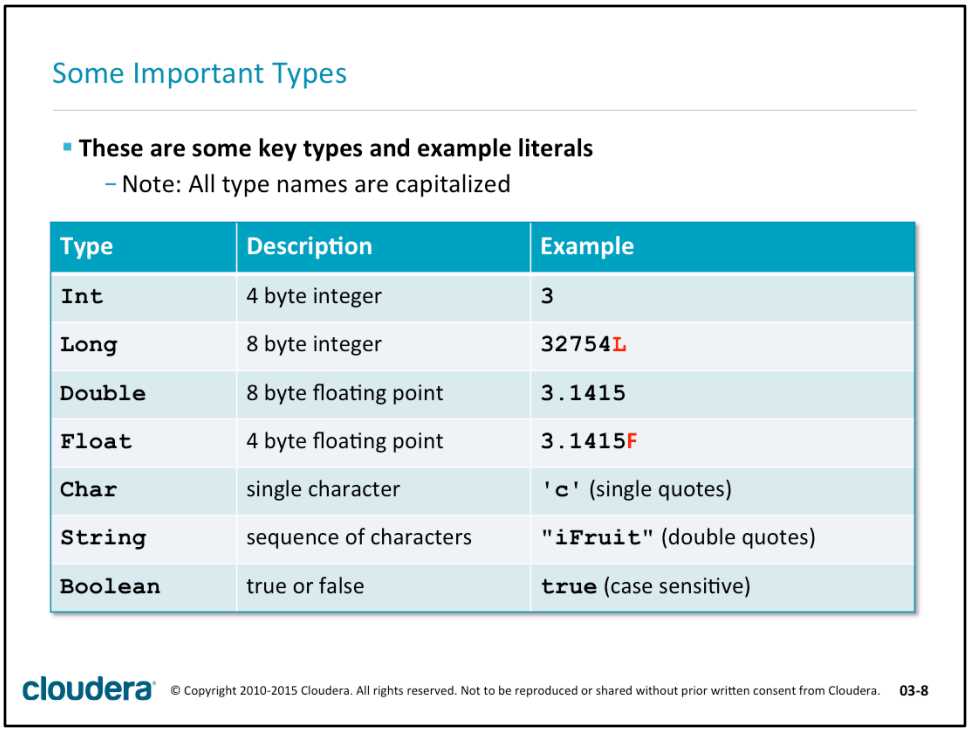
- **Although you cannot reassign an immutable variable, or assign a new type to any defined variable, you can redefine a variable:**

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
```


Note also that although, as demonstrated on the previous slides, you cannot reassign an immutable variable, you can *redefine* any variable. So although this is not valid:
val x = 1
x = 2

This *is* valid:
val x = 1
val x = 2



Note that Scala differs from Java in that it does not have primitive types (like `int`) and corresponding wrapper classes (like `Int`). All types are classes and therefore provide various methods for manipulating and converting the data and so on. (Note the capitalization; as with Java, the convention is that classes begin with an upper case letter, whereas variables, keywords, functions, methods and so on start with lower case.

Boolean must be true or false not True or False. (Opposite of Python!)

Earlier in the course it was mentioned that for backwards compatibility, Java types are supported.
The Types shown in this slide are the native Scala types.
In general, favor the native Scala types over legacy Java equivalents because the Scala types were implemented with scalability in mind.

Why might you need Java types? When you are calling Java APIs or methods that require those types. You can use them by referencing their full wrapper class names.
E.g. `val javaInt: java.lang.Integer = 3`

That that the literals for Long and Float, upper case letters are shown in the example (L and F) but lower case can also be used. We used upper case on the slide to avoid

Special `Unit` Type

- **Unit**
  - When a function passes back 'nothing' in Scala, it passes back **`Unit`**
  - There is only one **`Unit`** in Scala, it is non-instantiable

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

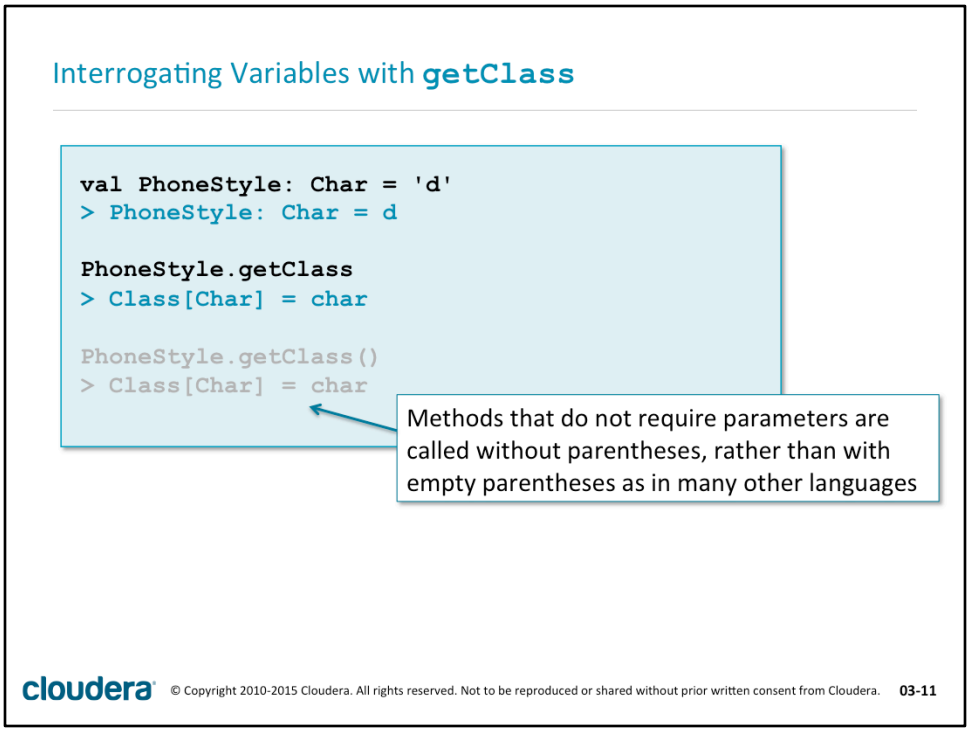

Remember that every statement in Scala is an expression that returns a value. So this example shows a variable set to the return value of a println statement. But what does println return? Nothing useful, so it returns a generic Unit object.

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"
> res2: Any = hi

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

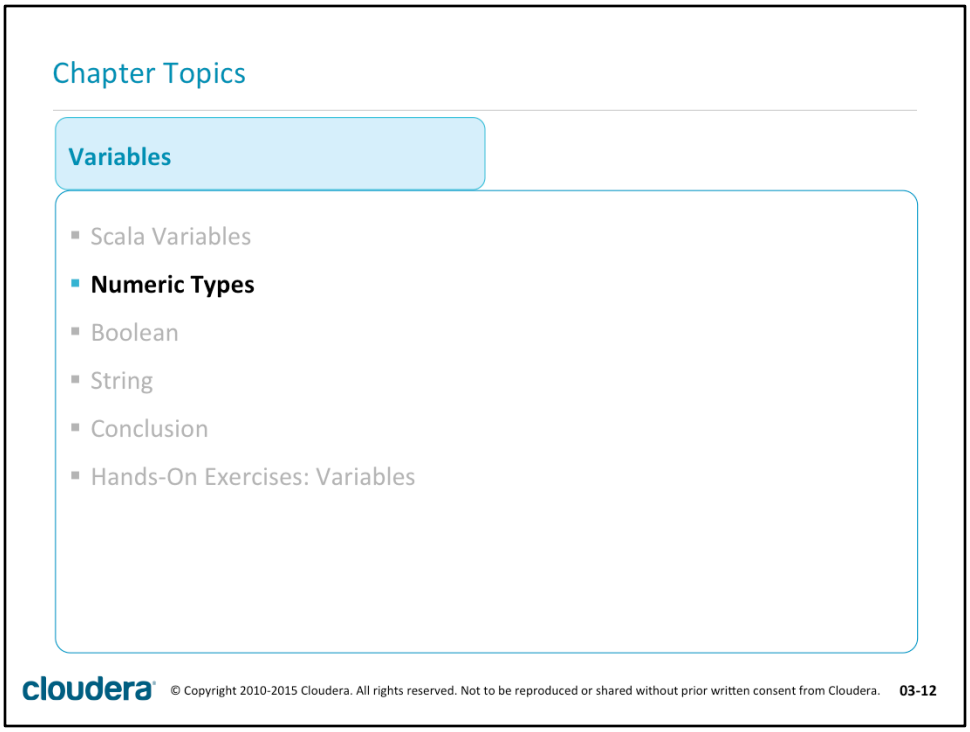

`Any` is particularly useful with collection types, because it allows a collection to hold objects and values of any type. in order to use method of those object or treat them as a specific type, you need to explicitly cast the object to the correct type.



A handy way to verify what kind of object you are working with.

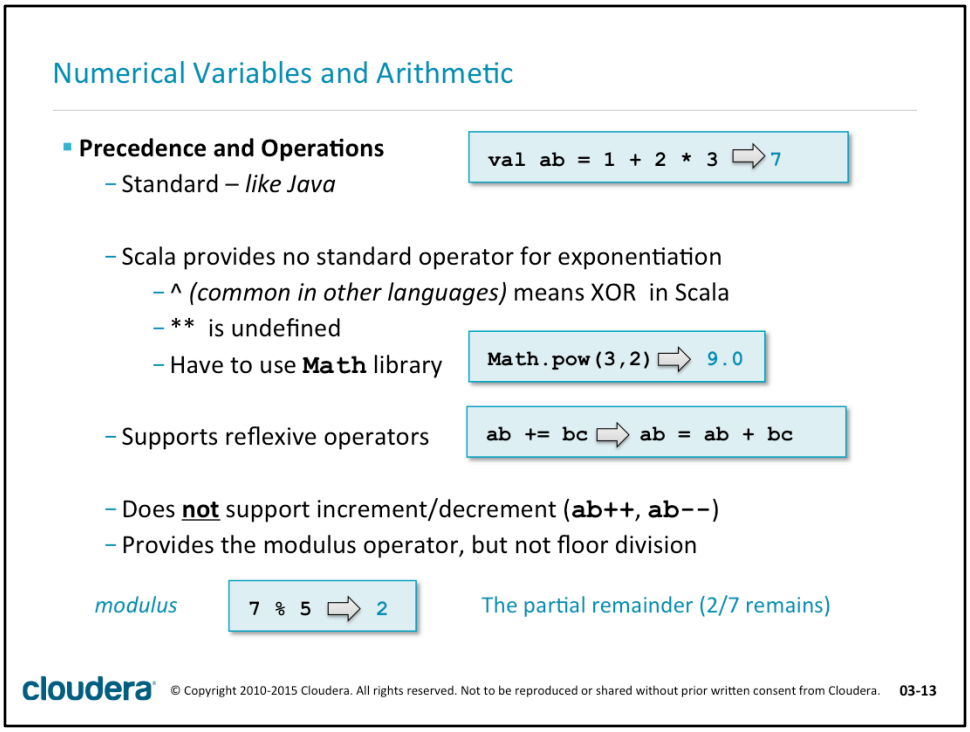
This is a good point to identify that methods that require no parameters can be called with empty parenthesis as in other languages, or can be called without parenthesis at all, which is the preferred "clean" style. An example showing the method called with empty parens is here for contrast. It also shows that in most cases coding empty parens does not throw an error even though it's poor coding style.

Instructor Value-Add
It IS possible for Scala to throw an error when a method is called with empty parens. Functions can be declared with a parameter list, including an empty one, or without a parameter list altogether. If a function is declared without a parameter list, it can NEVER be called with empty parenthesis, and coding empty parenthesis will cause Scala to throw an error. If a function is declared with a parameter list, but no parameters are required, then it has the behavior shown in this example.



Most Scala variable function similarly to their Java equivalents.

Some notable exceptions are discussed in the next few sections.



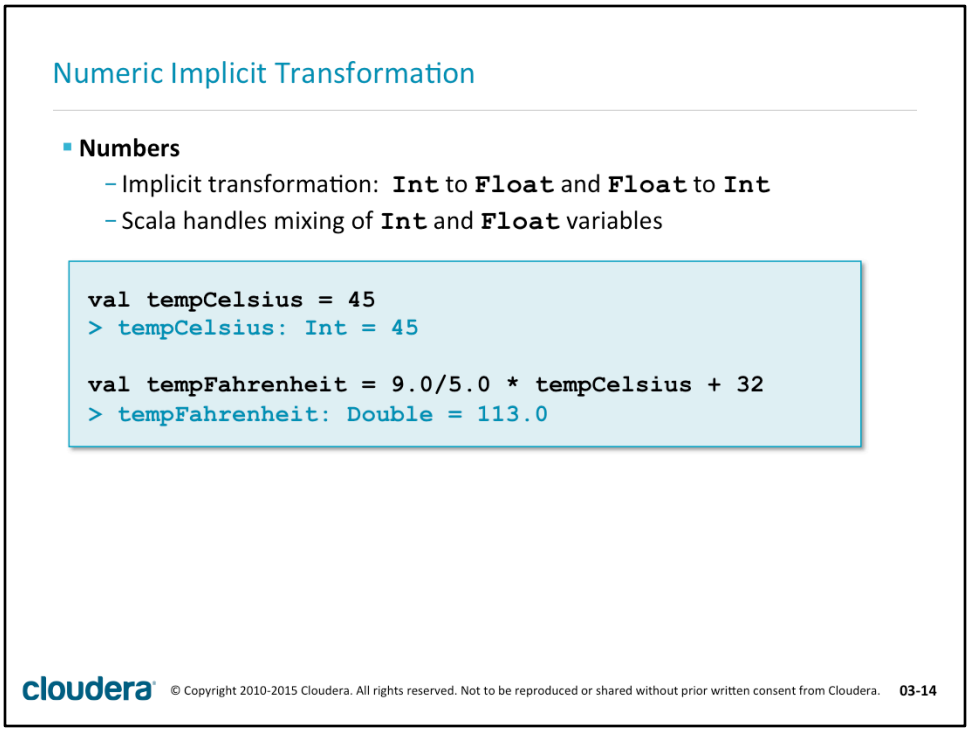
Pretty much as you'd expect…

There are a few differences from other languages.

[*] Scala doesn't provide an operator for exponentiation.
** isn't defined. ^ mean XOR; use Math.pow(x,y) instead.

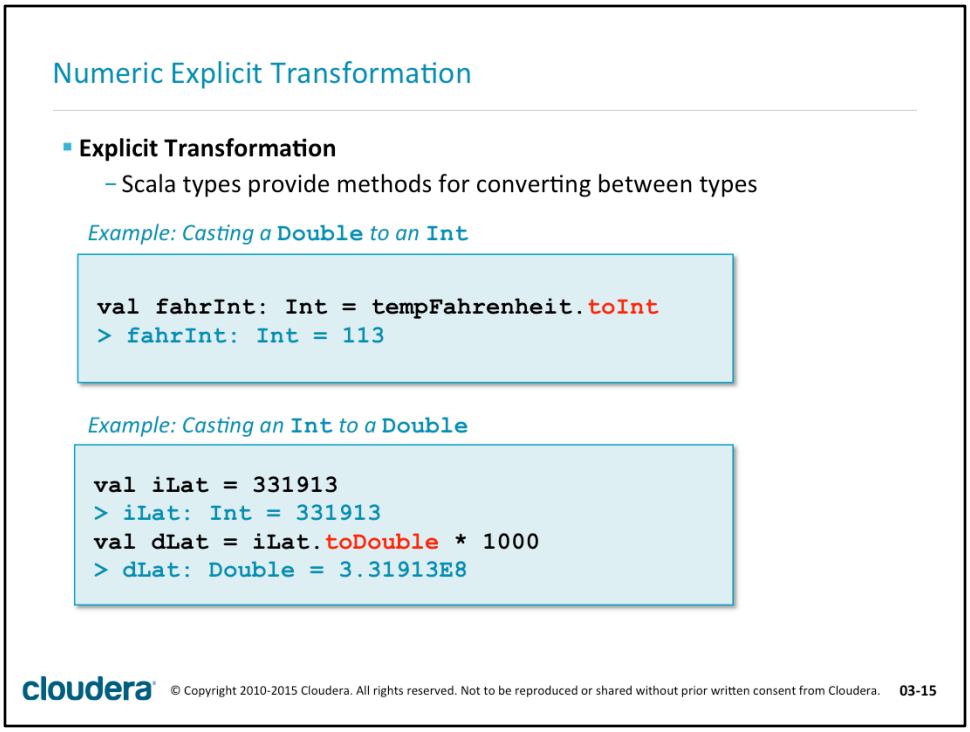
[*] Also, for programmers accustomed to increment and decrement (x++ and x--), Scala doesn't support this. Instead, use the reflexive x += 1

[*] Scala does not provide a floor division operator
Instructor value-add.
In Python 7 // 5 would provide “floor division”. INT(7 / 5) = 1, so 7 // 5 would equal 1
In Scala, ‘//’ means “the rest of the line is a comment”. So… 7 // 5 would equal 7, because the rest of the line would be ignored!
In Java, you would used Math.floor( ) or integer division.



Scala figures out if there is a Double involved and makes the resulting variable assignment into a Double

tempCelsius → the mobile device temperature in the Loudacre data. Overheating may be one reason that certain models restart.
If the GPS is enabled, the device reports Latitude and Longitude.



The numeric datatypes in scala provide conversion methods as shown in these examples.

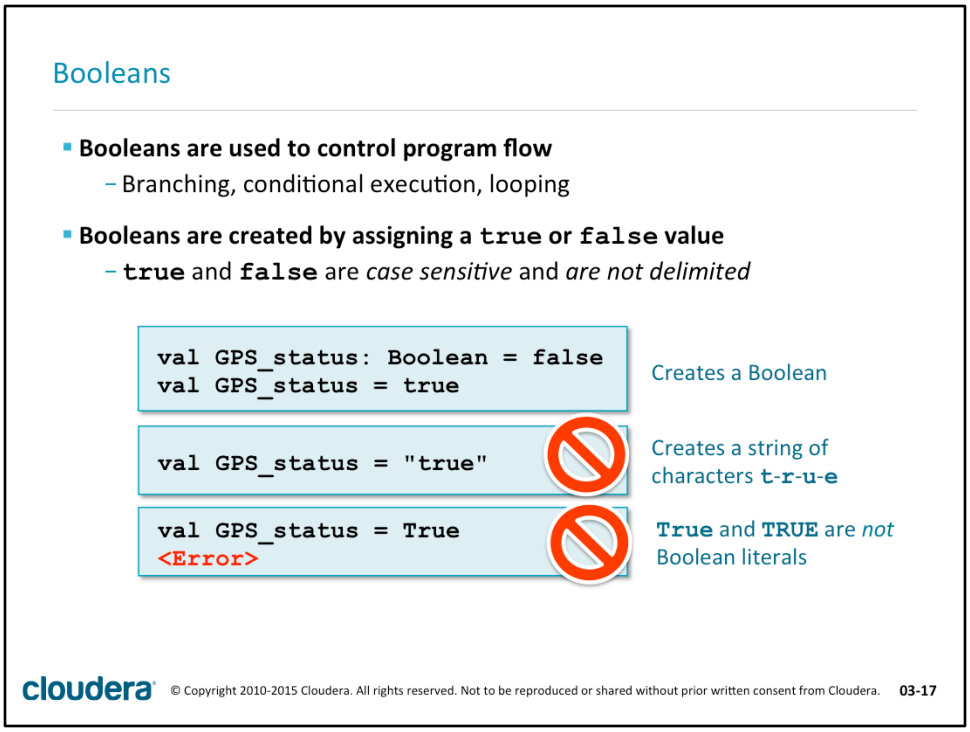
There is no equivalent in Scala to the Java `cast` function. casting is handled through implicit or explicit conversions as shown here.

Reminder: In Scala, methods that do not require parameters are called without parenthesis, rather than with empty parenthesis as in many other languages.

Chapter Topics

Variables

- Python Variables
- Numeric Types
- **Boolean**
- String
- Conclusion
- Hands-On Exercises: Variables



Main point here are that Boolean values in Python are True and False (with initial caps) and that 'true' and 'TRUE' are not boolean values, but undefined variable names in Python.

GPS_status is a boolean value set in the Loudacre data, indicating whether GPS is enabled (True) or disabled (False) on the mobile device.

Instructor: Students coming from other languages may be habituated to pre-processor directives in which textual variations have been set to expand to the logical value – meaning that even if the language is case sensitive, the programmer may be habituated that it doesn't make a difference.

Booleans are used for Flow Control in Python. We have a whole chapter on that later in the class. So we are just mentioning it here for foreshadowing.

Boolean Comparatives

- **Inequalities**
  - **<, >, <=, >=**
    - Java syntax **=>** or **=<** in invalid
    - Note that **=>** has a different meaning
- **Equalities**
  - **==, !=**
    - Java syntax **<>** is not valid
  - You can mix numerical types
    - `1 == 1.0` returns **`true`**
- **Logic**
  - **`Boolean`: `&&`** (and), **||** (or)
  - **`Boolean`** or **`Int`: `&`** (and), **|** (or), **^** (xor)

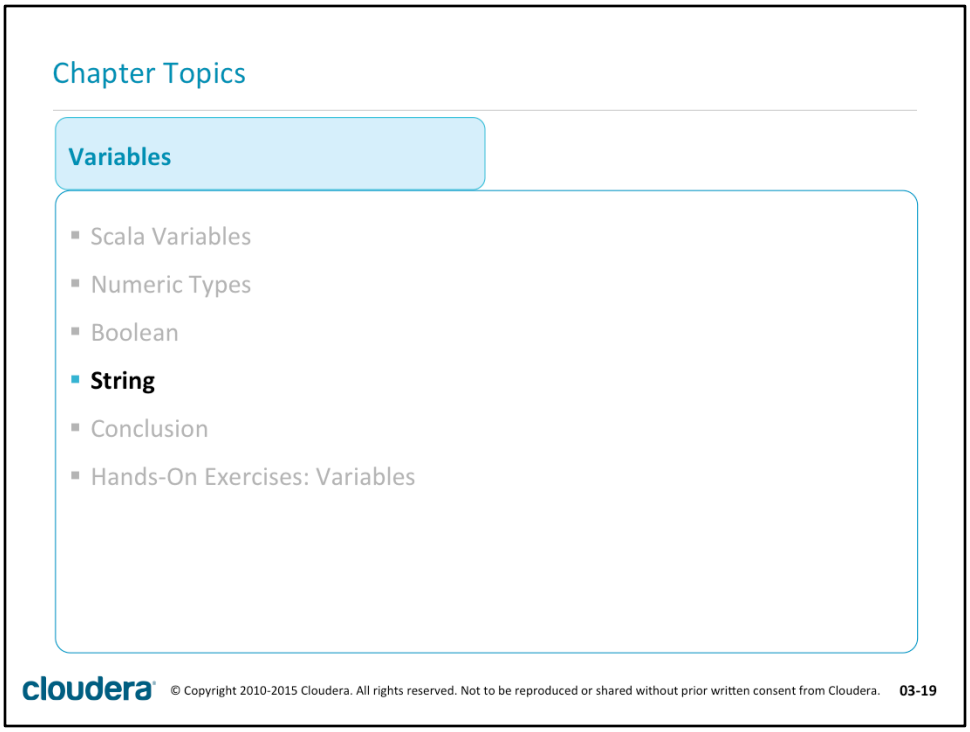
cloudera 03-18

In Scala, && and || terminate evaluation when the result is found, they don't continue processing after.

This is an example of forethought in the design and implementation of Scala to plan for scalability. In other languages, the designer might think "what's the difference"? In Scala, the designers realized that the extra work would become a significant user of resources if the code is scaled up.

Note - In Python you can use id(a) to get the object ID for a and compare it to id(b) to see if the two variables point to the same object in memory.

Scala has no equivalent to this.



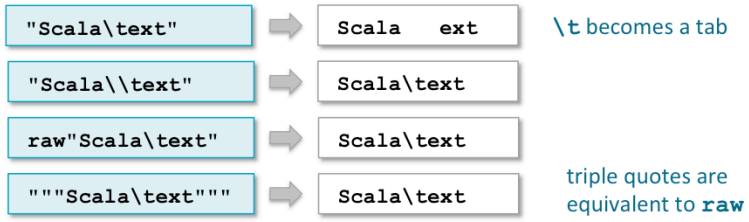
In Scala, technically a String is the Java String type: java.lang.String. So:

```
scala> var myString: String = "Hello, World"
myString: String = Hello, World

scala> myString.getClass
res0: Class[_ <: String] = class java.lang.String
```

But, in use, Scala implicitly converts String types to Scala type StringOps. This is a wrapper type that provides all the same methods Java strings have plus many additional methods not available to Java Strings.
http://www.scala-lang.org/api/2.10.2/index.html#scala.collection.immutable.StringOps
The additional available methods are mostly specialized and not of wide interest, and so are not covered in this course. But encourage students to visit the StringOps API docs to see the full list.

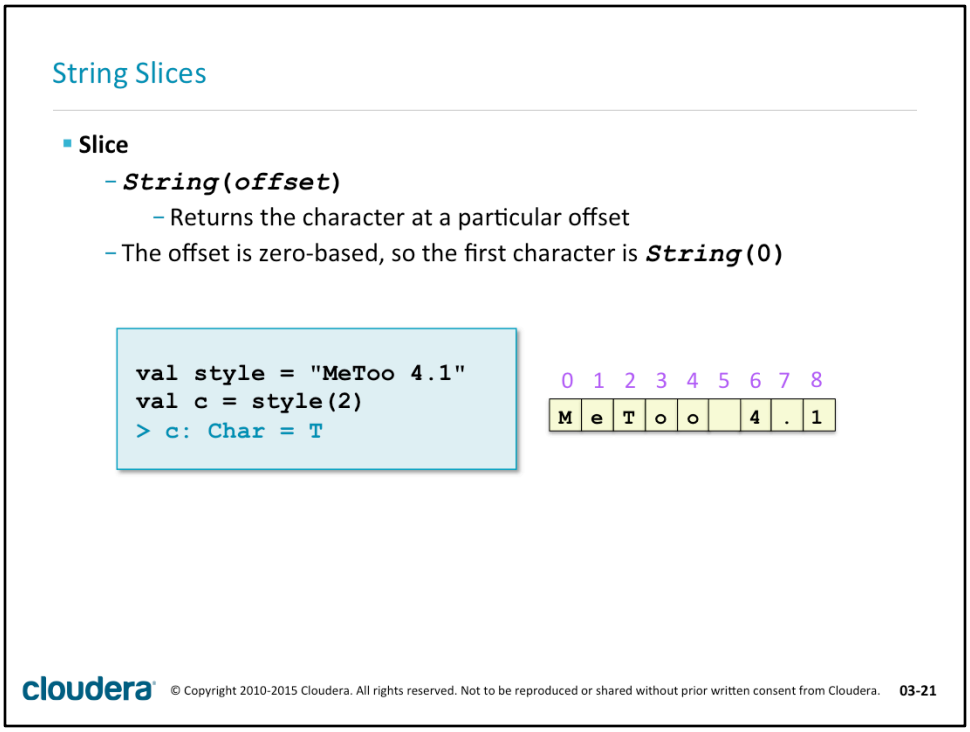
String Composition

- **Composition**
  - Use the escape sequence (backslash: \) for literal characters
  - \t = tab, \n = newline, \b = backspace, \r = carriage return
  - Use double-escape (\\) to indicate a single backslash
  - Scala provides "raw" mode
  - Examples



 03-20

Raw mode can be handy if you are passing a string to something that needs to accept the raw string as input, and you don't want Scala attempting to interpret the string and modifying it.



[*] In some languages name(x), is called an element of an array, a subscripted element, or a 'slice'. And name(x.y) is called a substring, a 'slice range', or just a range.

Unlike in Java, a String is considered a collection, much like an array. (the wrapper StringOps type is in the collections package). Therefore characters in a string can be referenced much as array elements can be referenced, as shown here.

(The other standard Java string manipulation methods such as substring are also available.)

String Methods

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

```
val s = "efdabc"
> s: String = efdabc

s.sorted
> String = abcdef

s.toUpperCase
> String = EFDABC

s.splitAt(4)
> (String, String) = (efda,bc)

s.toArray
> Array[Char] = Array(e, f, d, a, b, c)
```

cloudera 03-22

Instructor
Show them the current API for String Operations
http://www.scala-lang.org/api/2.11.5/index.html#scala.collection.immutable.StringOps

Instructor value-add: At the time of this writing, many = 203 methods!

Chaining Example

- In Scala, String methods can be *chained*.
- Example: the results of the `sorted` method are passed to the `toUpperCase` method

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

device.toUpperCase
> "TITANIC 2300"

device.sorted
> "0023aciintt"

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

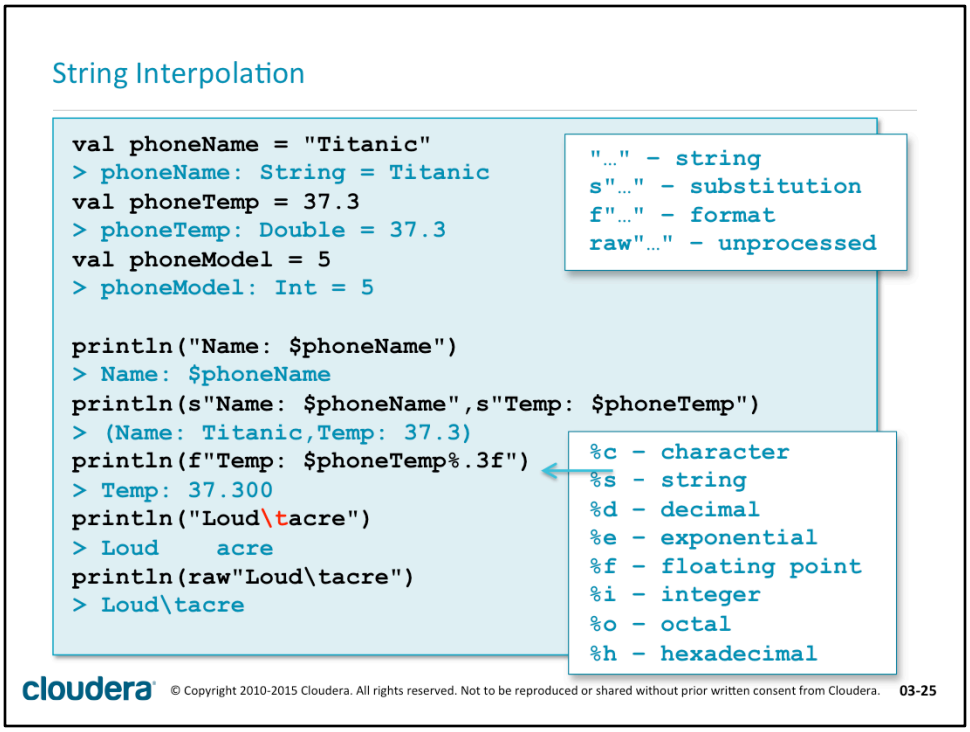

Instructor
We already mentioned method chaining earlier. This slide focuses on chaining string methods to get complex effects.

You can endlessly chain methods in Scala. The methods are processed from right-to-left and the results of each method become the source for the subsequent method.

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"
PhoneModel: String = Ronin

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

http://docs.scala-lang.org/overviews/core/string-interpolation.html: “Starting in Scala 2.10.0, Scala offers a new mechanism to create strings from your data: String Interpolation. String Interpolation allows users to embed variable references directly in *processed* string literals.”

Standard string interpolation " … " just prints the string literally.
The 's' prefix s" … " enables variable substitution. A dollar sign before the variable name causes it to be replaced with a string representation of the value.
The 'f' prefix f" … " enables substitution with formatting. A format suffix offers more control over the string representation.
The 'raw' prefix raw" … " disables substitution. In the example shown, '\t' is literally interpreted as a tab character. But raw enables it to be printed as '\t'

Not shown here, but string interpolators can also take arbitrary expressions such as println(s"1 + 1 = ${1 + 1}")

Chapter Topics

Variables

- Scala Variables
- Numeric Types
- Boolean
- String
- **Conclusion**
- Hands-On Exercises: Variables

Essential Points

- **Explicit definition of variables is required in Scala**
- **Variables are either mutable (`var`) or immutable (`var`)**
  - Immutable is usually the default
- **Scala supports inferred typing but can be explicitly typed if necessary**
- **Most Scala types work similarly to their Java equivalents**
- **Scala will throw an error if you try to use data types incorrectly**
- ***`name`*`.getClass` tells you a variable's type**
- **Boolean values are `true` and `false`**

cloudera

Instructor: Just reviewing key points of the overview before diving into the details with each variable type.

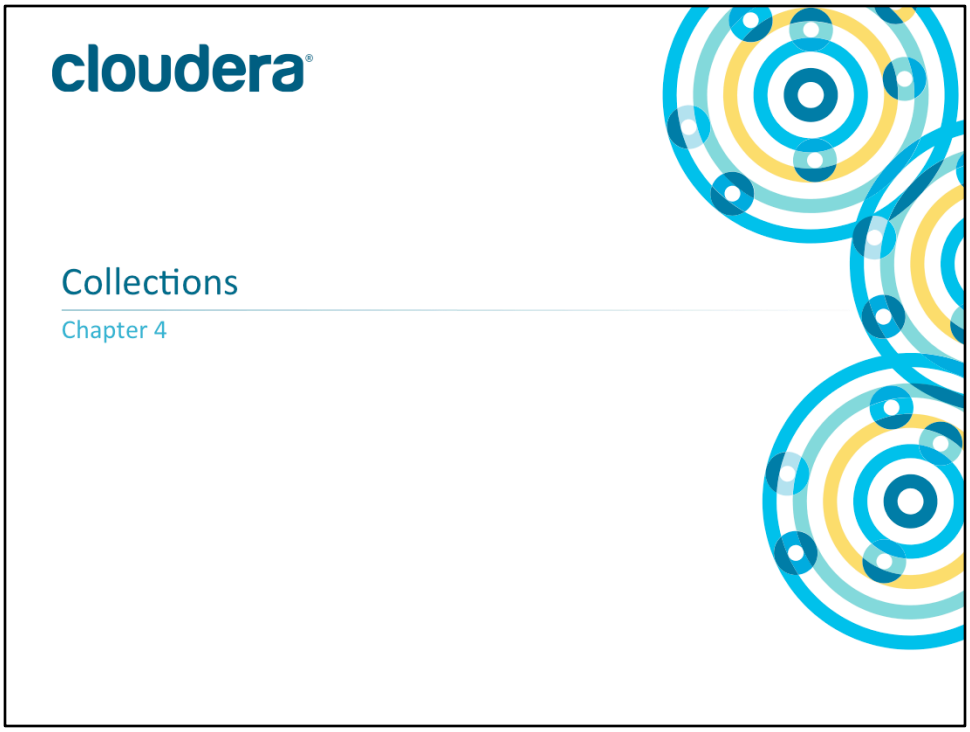
Chapter Topics

Variables

- Scala Variables
- Numeric Types
- Boolean
- String
- Conclusion
- **Hands-On Exercises: Variables**

Hands-On Exercise: Variables

- **In this Hands-On Exercise you will**
  - Interactively explore variables in the Scala shell
    - Numerical variables
    - Operators and reflexive operators
    - Advanced math functions using **`scala.Math`**
    - Booleans and comparators
  - Write a program to extract fields from a single data record
    - String methods **`substring`**, **`indexOf`**, and **`contains`**
    - Booleans
- **Please refer to the Hands-On Exercise Manual for instructions**

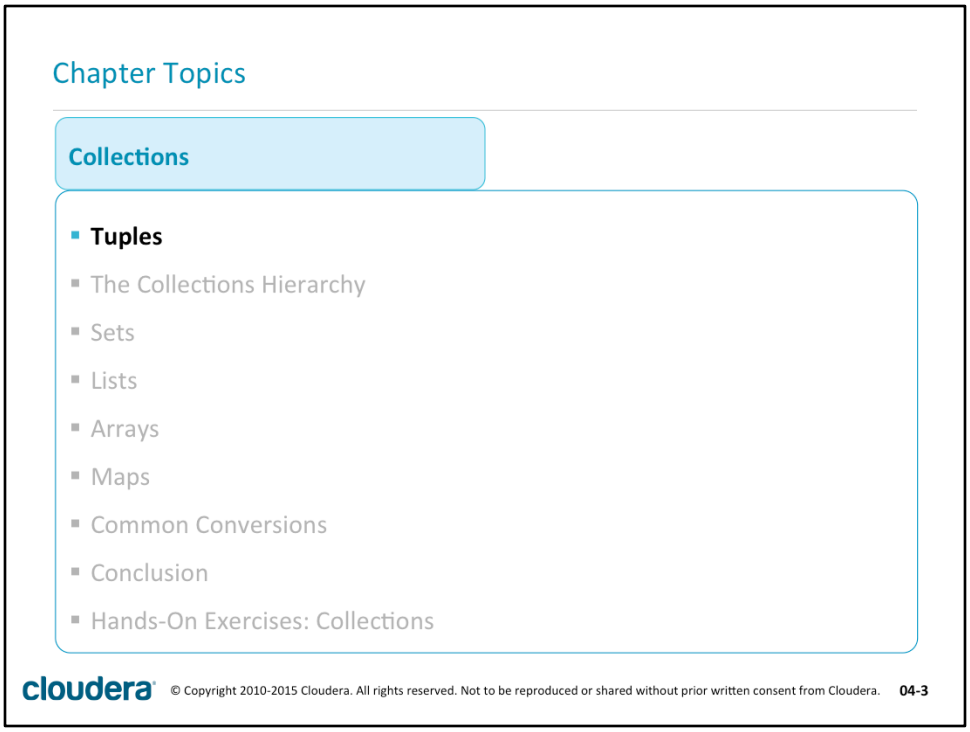


In this chapter we will explore compound Data Structures called 'Collections'.

The Data Structures supported in a language have a profound influence on the general approach to problem-solving and the way in which programs are written.

Course Chapters

- Introduction
- Scala Basics
- Variables
- **Collections**
- Flow Control
- Libraries
- Conclusion



Tuples are a method of grouping variables together. They are similar to Collections, but not part of the Collections Hierarchy in Scala.
So we are going to cover them first.

Tuples

- **A Tuple is a group 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`
  - For sending multiple values in a single message between concurrent processes
  - For buffering a data record / related data of varying types
- **Limitations of Tuples**
  - The number of values in a Tuple cannot be changed after it is initialized
  - Tuples consist of between two (min) and twenty-two (max) values

cloudera

Strictly speaking, a Tuple is *not* a Collection. A Tuple is a *compile time* entity whereas a Collection is a *runtime* entity. Therefore Tuples are more restrictive and less flexible than Collections.

Tuple2

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

Example: Explicit Declaration

```
val myTup2A = Tuple2(4,"iFruit")
> myTup2A: (Int, String) = (4,iFruit)

myTup2A.getClass
> Class[_ <: (Int, String)] = class scala.Tuple2
```

Example: Alternate Syntax

```
val myTup2B = 4 -> "iFruit"
> myTup2B: (Int, String) = (4,iFruit)

val myTup2C = (4,"iFruit")
> myTup2C: (Int, String) = (4,iFruit)
```

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

`Tuple2` Methods

- **Elements of a Tuple**
  - Can be of different types
  - Are accessed using the `_.1, _.2` syntax

```
val myTup2B =  4->"iFruit"
> myTup2B: (Int, String) = (4,iFruit)

myTup2B._1
> Int = 4

myTup2B._2
> String = iFruit

myTup2B.swap
> (String, Int) = (iFruit,4)
```

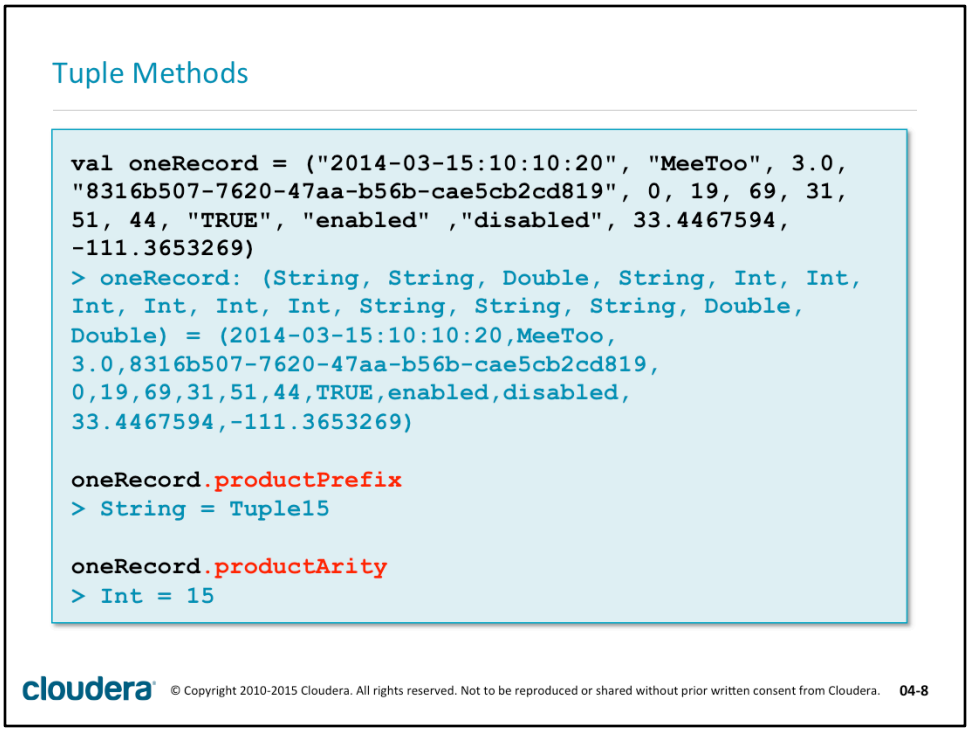
`swap` only works with `Tuple2`

Tuples >2

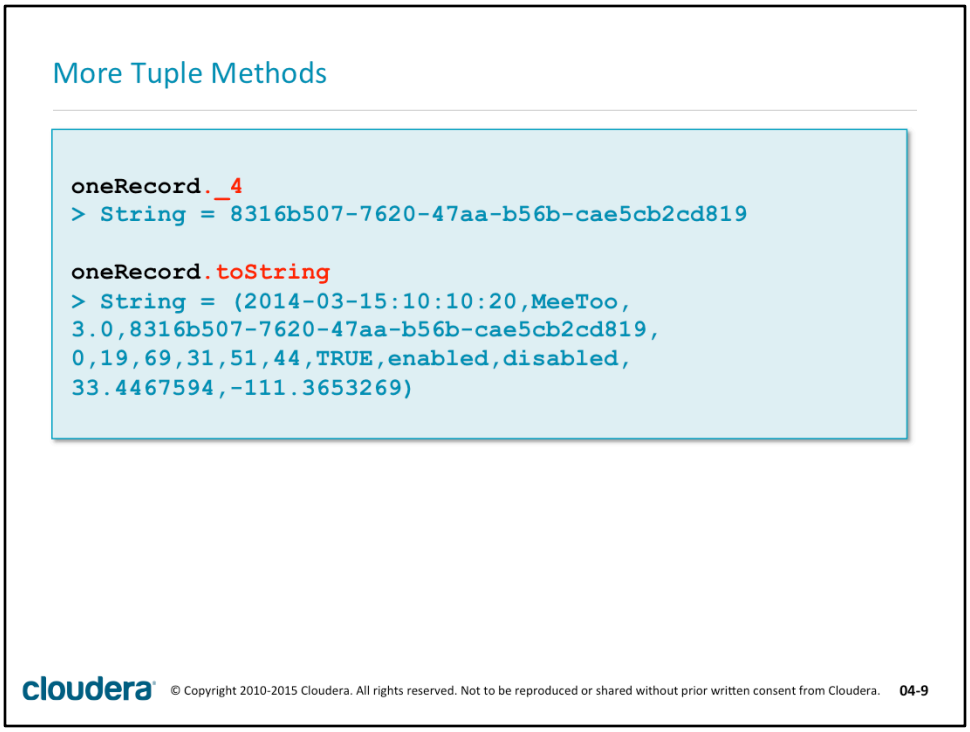
- **Tuples with more than 2 elements: `Tuple`*N***
- **The same syntax applies as for Tuple2**

```
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

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

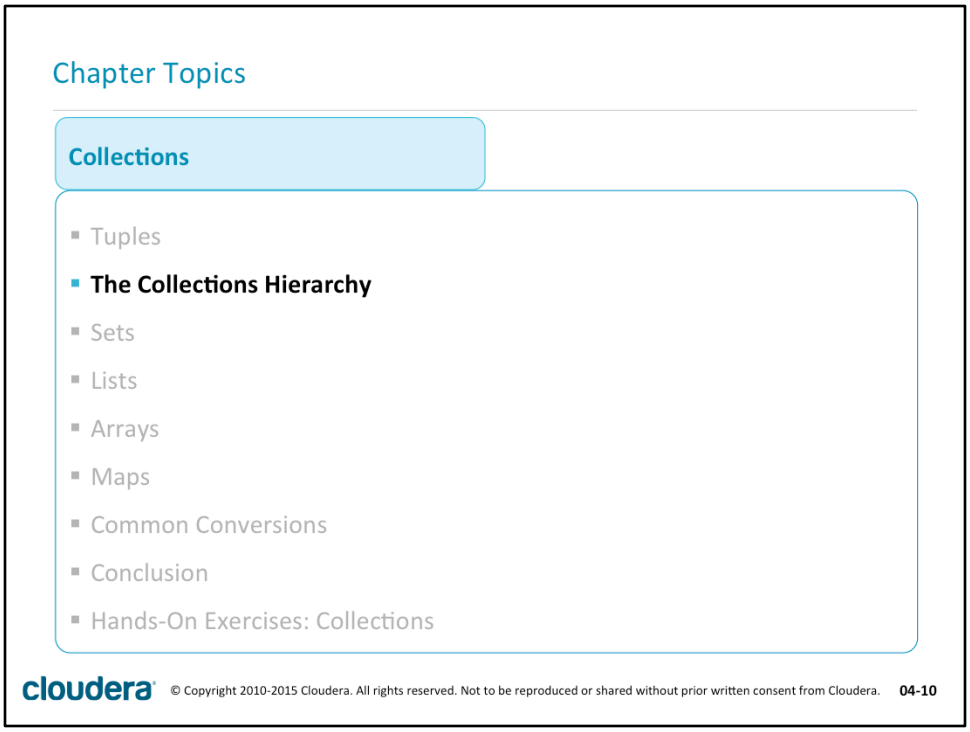

Demonstrates definition of a Tuple with more than 2 elements.
Then elements are accessed using the ._# syntax



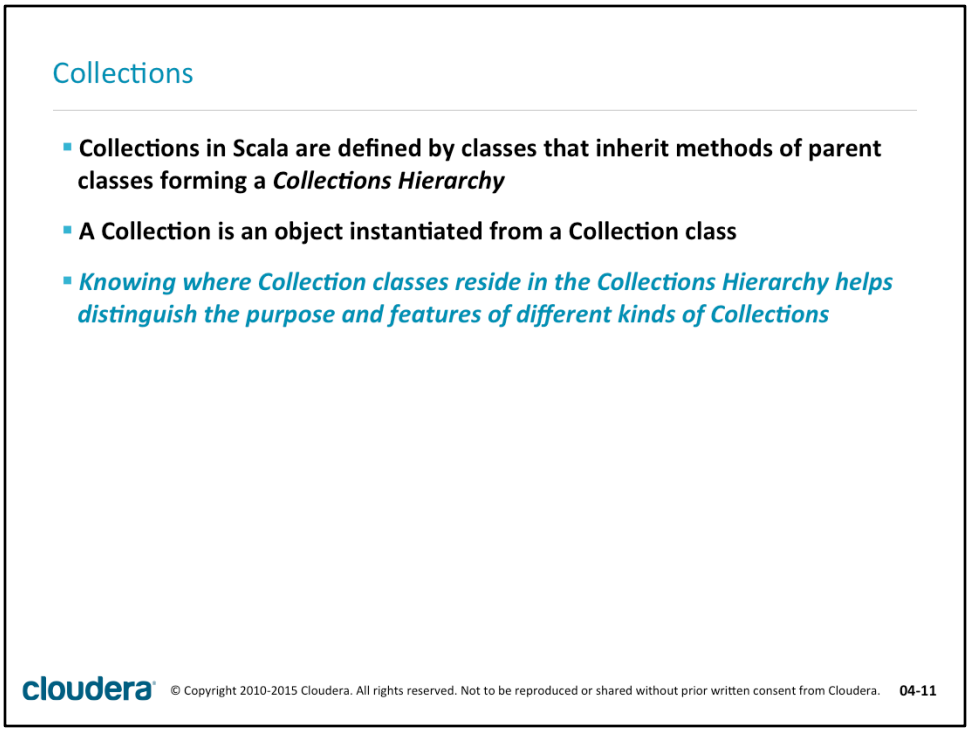
.productPrefix gives the Tuple number as a String.
.productArity gives the number of elements as an Integer.



.toString converts all values to strings and concatenates them into a single string.



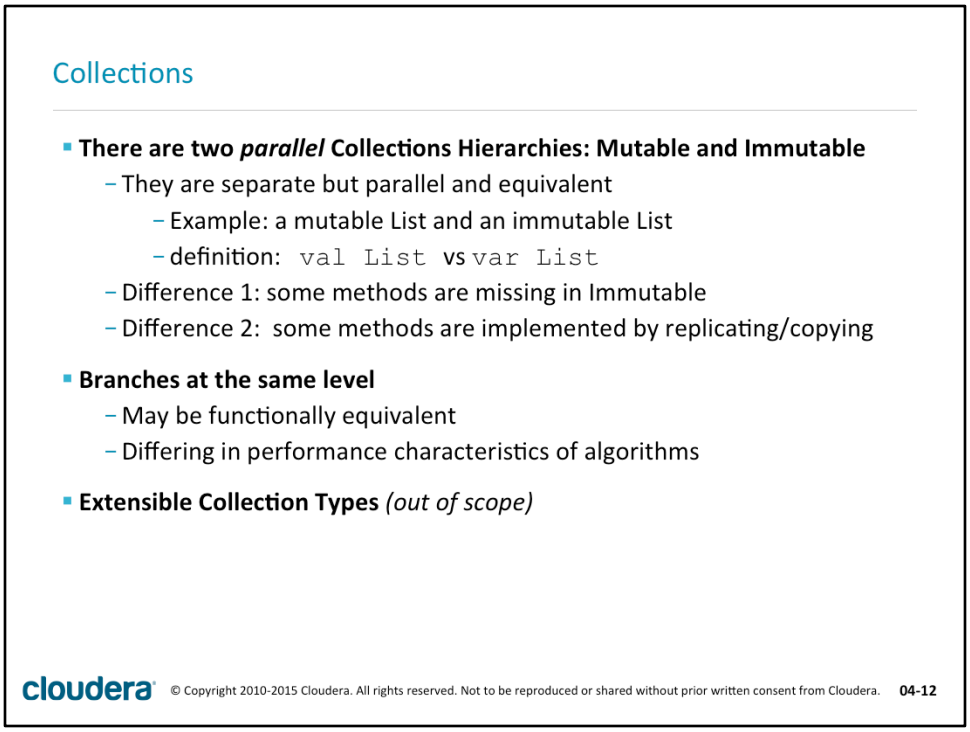
Instructor
We are going to talk about how the Collections are dervied from one another and from root Abstract types.
That helps understand the relationship between Collections and sets expectations about what kind of methods will be available.



A Trait is like a collection of methods without the central data object that the methods are to operate on.
Traits avoid the "diamond problem", where a single class inherits from two mixins that in turn each inherit from the same base class. If the base class has method a', and the two intermediary classes both overload a with a'' and a''', respectively, then how does a mixin class containing both intermediaries decide whether to use a', a'', or a''' versions of the method? Traits overcome this issue by providing neutral methods that are written to handle any kind of object, as they have no data component themselves.

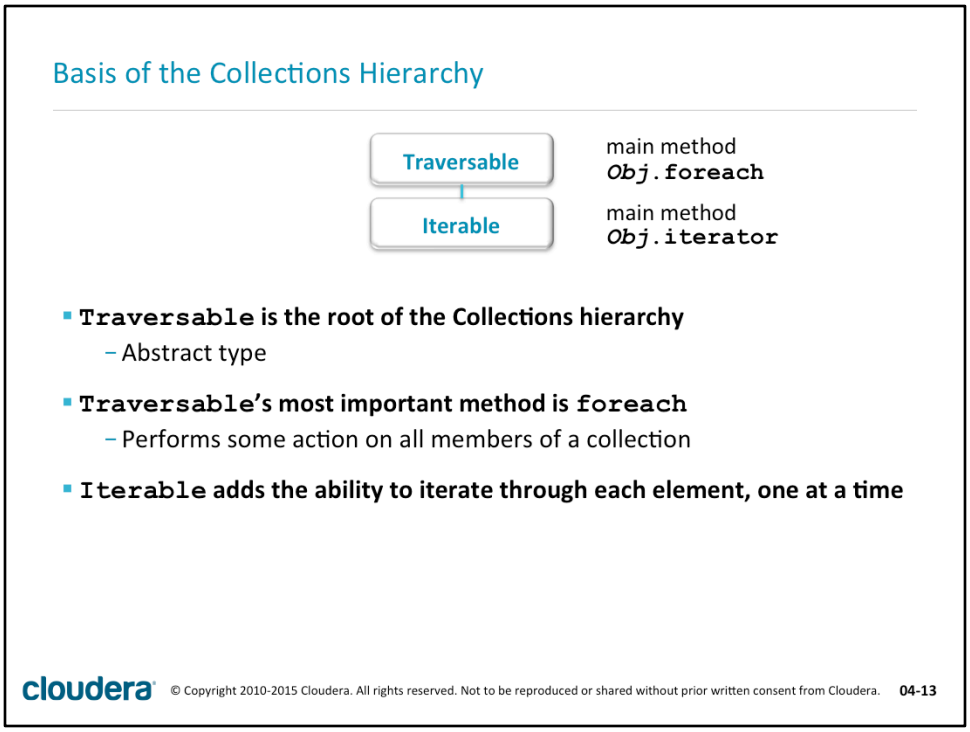
Instructor Value Add (optional) chalktalk

Traits – Multiple Inheritance Problems – also called in Computer Science the "Diamond Problem"
Draw a diamond. Label the top "A", the left vertex "B1", the right vertex "B2", and the bottom "C".
"A" creates Abstract Method .do
"B1" inherits ".do" as is from "A"
"B2" inherits ".do" from "A" and overwrites it with it's own version of .do, which we'll call .do#2
Now C uses multiple inheritance and gets features from both B1 and B2. But when it comes to method .do, does it get the original .do from B1/A or does it get .do#2 ?
A Trait is a class that contains Abstract Methods, but essentially has no constructor – no 'data' component of the class. It only operates on data belonging to the class it is "mixed into". Traits are always mixed in rather than inherited, so the problem is avoided. Any class that mixes in the Trait will have the original method, and any class that wants to have something different must define it locally.



Extensible
Example is "Enumerate", which is a separate class. You can extend an existing collection type, such as a list, and add the enumerate methods as traits (a mixin) to create a new enumerated list type.

Creating an extended collection using Traits is common in advanced Scala, but not used in our development classes, so it out of scope.
In 2.8 the Scala Collections hierarchy was restructured.



They key point here is how Scala collection design prioritizes concealing iteration from the programmer and assigning it to the framework.
The designers have broken down iteration, which is a single atomic concept in most languages, into three separate traits, and baked them into the hierarchy in a particular order.
Because they all inherit from Traversable, it is possible to tell the framework to perform some action on all the elements of a collection without the program itself iterating.
And that's true for ALL collections because of the design of the hierarchy.

At the top of the collection hierarchy is trait Traversable, and its *only* abstract method is foreach

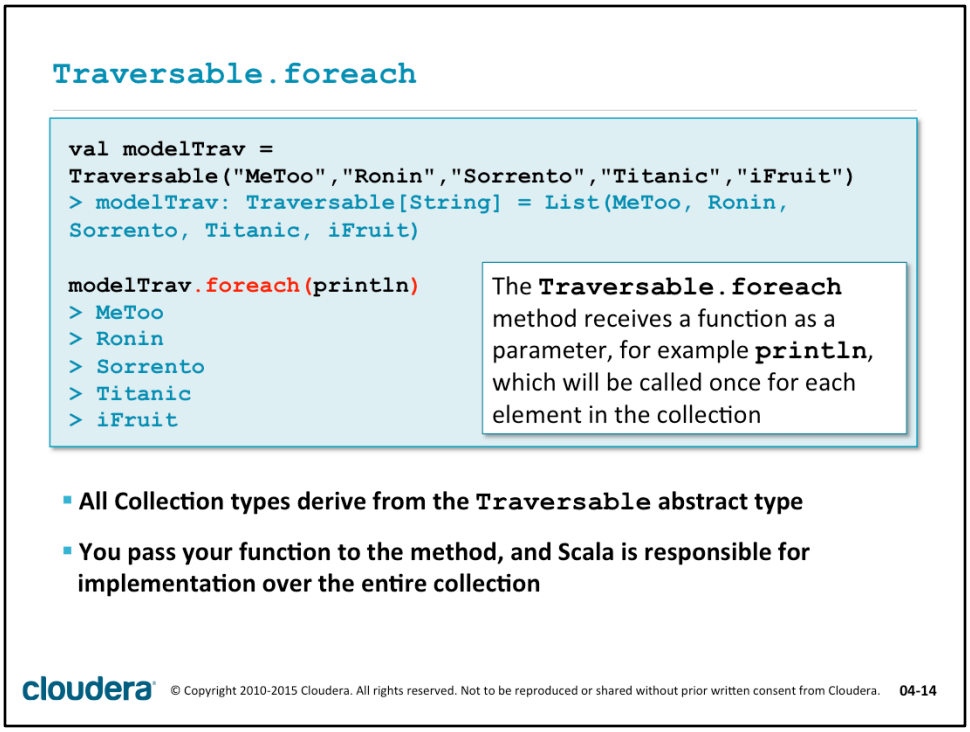
Conversions and interrogation methods … many
http://docs.scala-lang.org/overviews/collections/trait-traversable.html

The next step in the hierarchy is trait Iterable

Iterable provides methods to return each element of a collection
OBJ.iterator

yields each element in the same order as it would have been traversed by foreach

http://docs.scala-lang.org/overviews/collections/trait-iterable.html



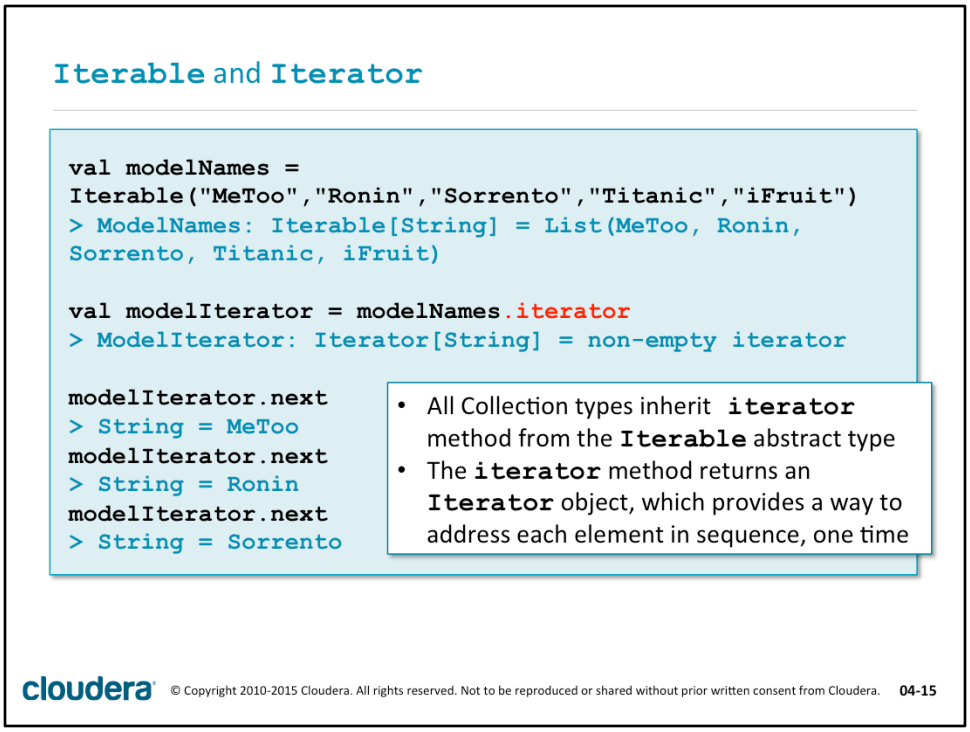
Traversable is covered first for contrast and clarity.

Note that the basic Traversable method is "foreach( )".

Foreach accepts a function parameter. In this example we are passing the builtin function with no parameters, which means that println is simple called repeatedly with the current element in the traversal as the parameter. This is a shortcut syntax. The full syntax and much more detail on how to pass functions as parameters to higher order functions like foreach is covered in the Flow Control chapter. For now, this should be sufficient to make the point.

Instructor Value Add:
[1] The foreach( ) method is overwritten in some collections that inherit trait Traversable, to provide better performance.
[2] While you are guaranteed that all elements of the collection will be processed, you are not guaranteed of an order of processing.
It's easy to imagine, for example, the work being allocated amongst four processors, and each working through its stack, produces results in parallel.
In this case, every fourth item in the original collection would be processed first, followed by the next, and so forth.

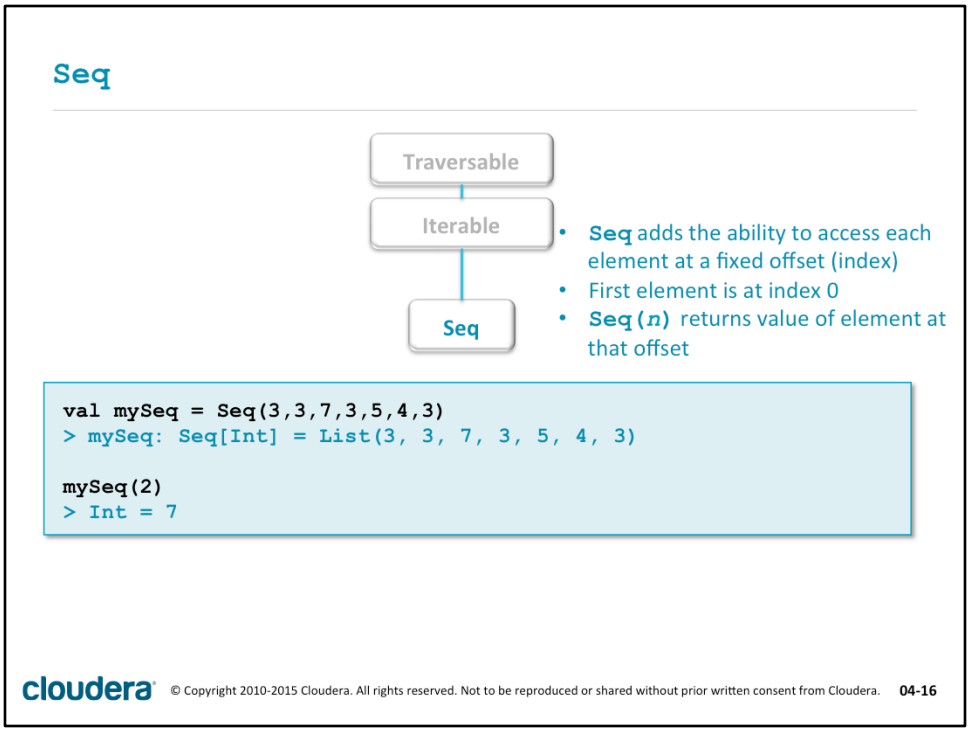
Note: In the original version, I created a List and cast it to an Abstract type Traversable, to show the pure illustration of a Traversable without additional methods available. However, this might have been confusing and not necessary, since Traversable type is rarely used in practice. So instead, List was determined to be a simpler way to illustrate Traversable even though it is far down the Collections Hierarchy.



All collection items are Iterables, and therefore have the iterator method.
The iterator method returns an Iterator for the collection.
Iterators are covered in detail in the chapter on Flow Control

An Iterable collection is capable of providing an iterator, which returns each element from the collection in the order it would have been processed by trait Traversable.
An Iterator is a destructive method -- it can only be used once. This keeps it safe for distributed processing.

Instructor Value Add
The reason we are covering Iterable is because the Spark API frequently returns an Iterable.
Most likely, "behind the curtain", Spark derives its own internal types from Iterable. And when these internal types are passed back to the Scala program they are returned as type Iterable. The purpose of using type Iterable and an Iterator rather than a List or an Array has to do with when the data from the collection is manifest in memory. With Spark and Big Data it's easy to manifest data that's too large for available memory. An iterator in Scala, interfaces in Spark, in such a way that it only manifests data in memory as it is used rather than staging it in memory first, as would occur with an Array or a List. The data that an Iterator refers to may be in multiple JVMs – it can be a distributed data set.



A sequence is an iterable with a fixed length and whose elements can be accessed as index offsets from 0

http://docs.scala-lang.org/overviews/collections/seqs.html
Subtypes LinearSeq (fast head and tail operations) and IndexedSeq (fast apply, length, and update operations)

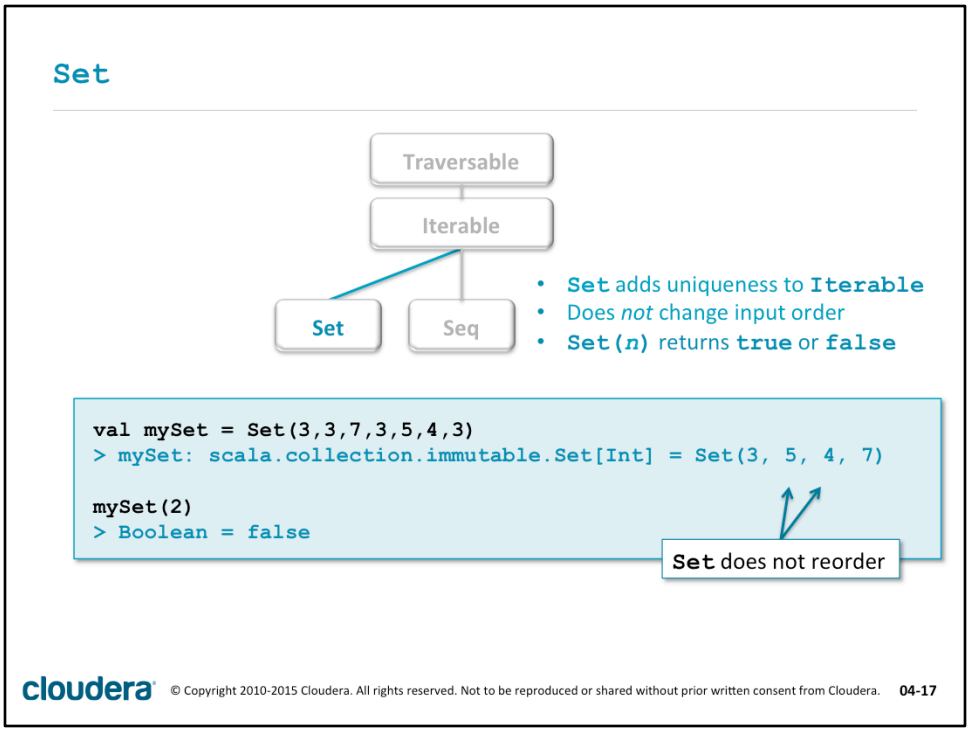
The key point here is what happens with syntax x(n). On the next several slides you'll see that exactly the same syntax x(n) returns completely different data types and results.
x(2) = 7

Note that the first element is x(0), not x(1), which is in keeping with Scala's mathematical roots.

The Seq collection type may seem like an insignificant variation on a List. In many languages, a List is the first collection containing elements that can be addressed at a fixed offset by an index. In Scala, the Seq collection is indexed, but is not full-featured like a List. This is another example in Scala where the philosophy and mathematical concepts are baked into the design. List will come up later in the slides and in the hierarchy.

Some types, such as Seq, are most useful when used as part of the mixins to create an extended data type (again, out of scope for this class).

Instructor: Compare Seq on this slide with Set on the next slide. The subscript syntax x(n) performs as expected in Seq. The same syntax does not perform as expected on the next slide in Set.



Sets don't work like they do in other languages. The two key differences are described below.
Seq: x(2) = 7
Set: x(2) = false

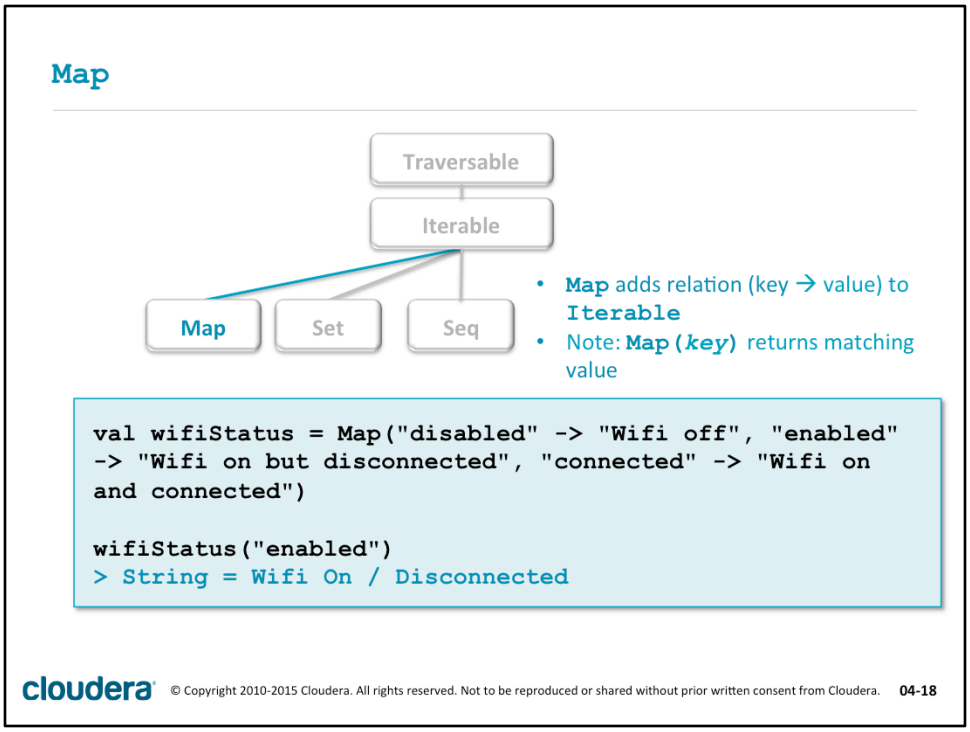
[1] In other languages, testing for membership is separate from accessing an element at an offset using subscript n(x) notation. MySet(2) would return the number 4 in other languages.

[2] In other languages, the Set collection enforces order. No matter what order the elements were entered, they would have resulted in Set(3,4,5,7). You should point out that while Scala's Set enforces uniqueness among the elements, it does not alter the input order.

Thankfully, Map performs as expected creating a relation between two elements.

Note the difference between the example on the previous slide and this one. MySeq(2) returns the element at position 2 in the collection. This is the behavior the learner will expect both because they'll assume Set inherits from Seq, and also because that's the way Set collections work in many other languages.

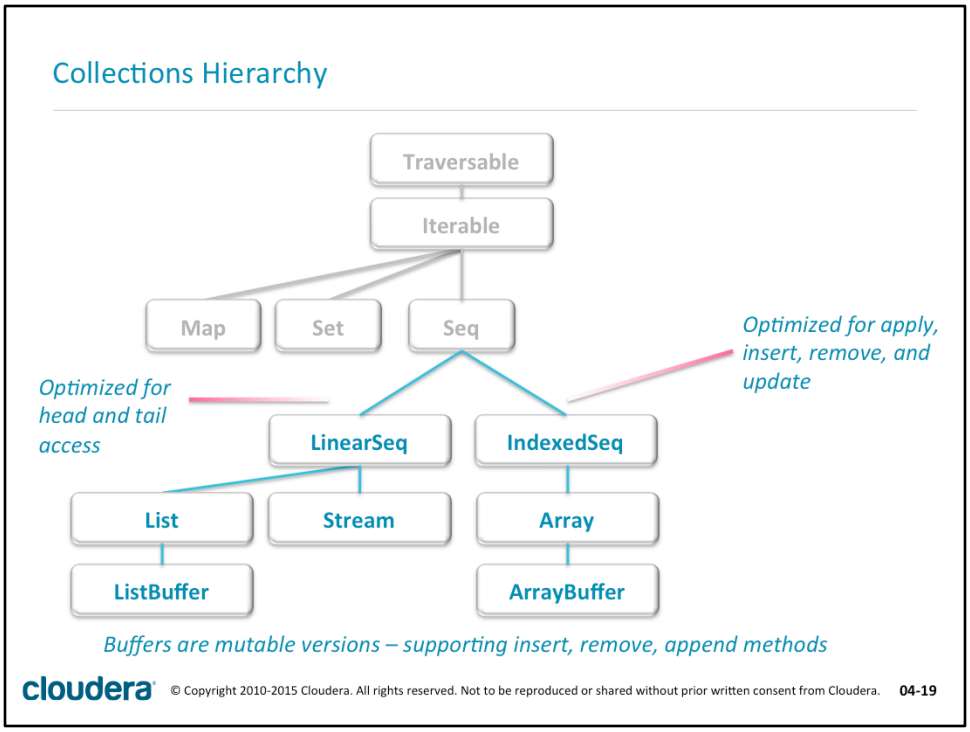
MySet(2) tests to see whether the Int 2 is a member of the set, and if so, returns the boolean value 'true', or in the example in the slide, 'false', since 2 is not a member of the set.



Map creates a relation between two elements.
Note that exactly the same syntax x(n) as in Set and Seq returns something else in this data type.

Notice also how Map and Set and Seq are all inheritors of Iterable, but are not related to each other. They are alternatives at the same level in the hierarchy.

Another key to understanding Scala collections is that data types in the hierarchy may exist for theoretical or design purposes to isolate traits – and corresponding methods – but may not be commonly used for data in programs, because more functionality is usually required by a program. Map and Set are useful collection types. Seq is really a stepping stone to other data types, and is not commonly used, as we will see next.



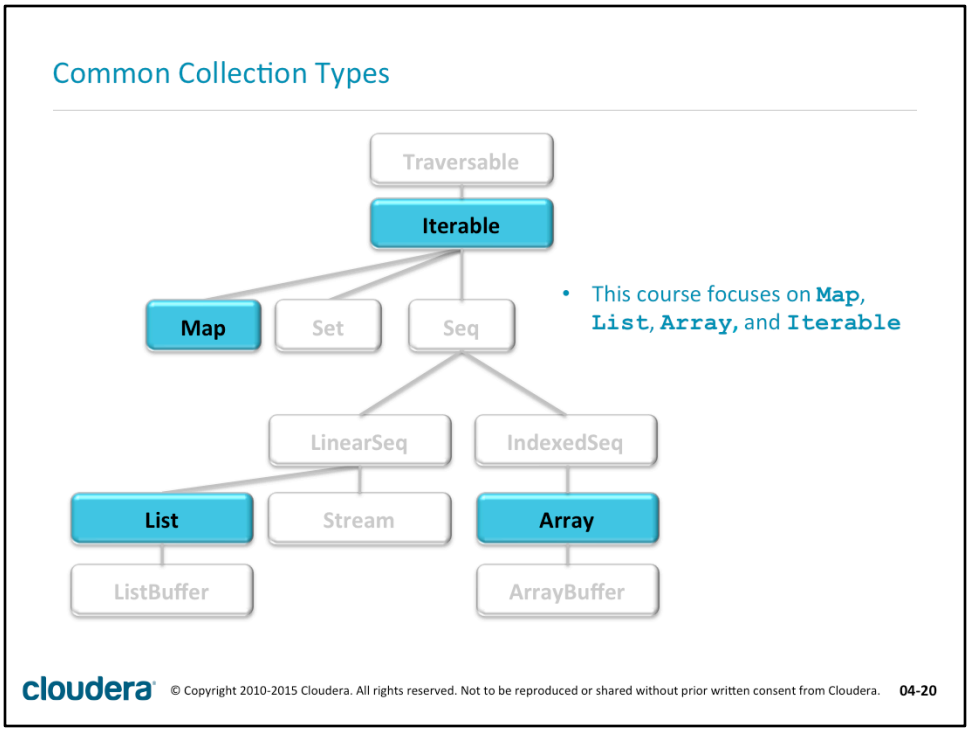
The key points on this slide are that beneath Seq, there are two branches. One is optimized for head and tail access, (aka 'queues') and the other is optimized for insert, remove, and update, access. The buffer versions of a list and an array support insert/remove/update even if the performance isn't optimized for that activity.

The main point you want to make is that from here down, the hierarchy is designed to implement collection access methods with different performance characteristics in mind. If you dig into the literature, you can find the mathematical descriptions of the performance for each method for each collection type. And that will reveal which methods are inherited and which are overwritten to provide a different performance characteristic with identical data manipulation results. The performance differences are out of scope for this class. However, if you were designing a big data API, you'd want to select data type based on these differences. In practice, this work has already been done for us within the Big Data API we will be using on subsequent Cloudera classes, such as the Spark API.

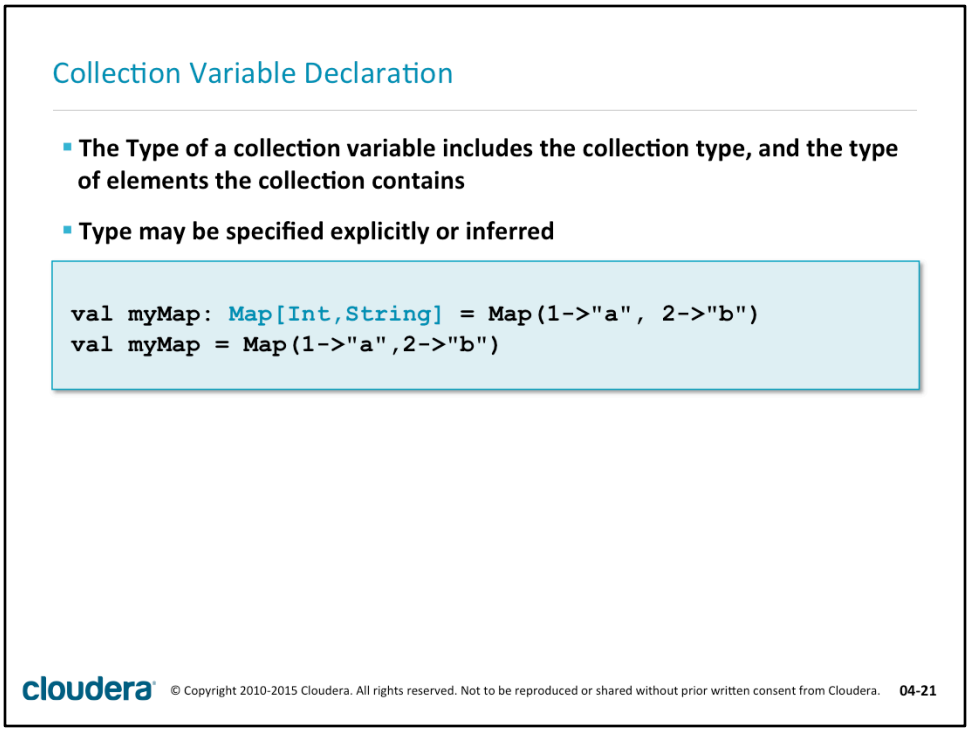
A stream is a *lazy* list

Non-finite sequence, only elements requested are processed at request
Otherwise, similar performance to lists for equivalent methods

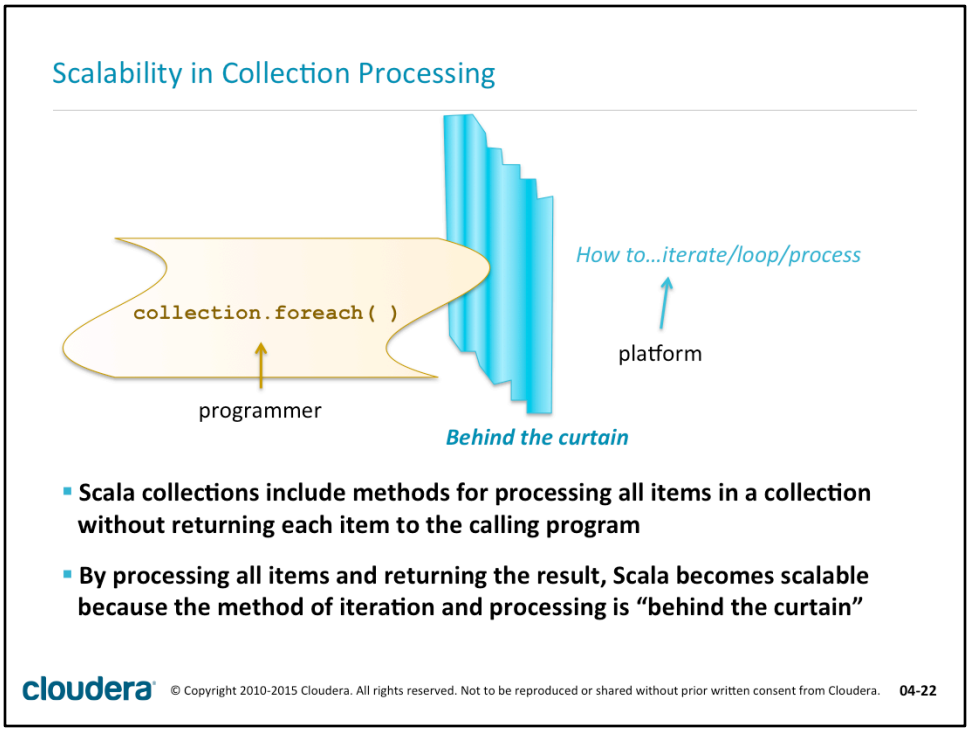
Getting into specifics of the performance is beyond the scope of this class. But it is important to understand that this is integral to the design philosophy of the hierarchy, in order to understand why there are trivial distinctions between collection types that otherwise would appear to perform the same data manipulations.



The most common collection types are identified. We will concentrate on these types for the remainder of the class.
In subsequent Cloudera classes, you will often be programming using an API. The API has defined the collection types that each function accepts and returns (probably based on performance or data manipulation characteristics). In the context of Spark, you don't usually choose which type to use; rather, you are working with an API that is defined to return or accept certain types. The types in this list are the most common types used in Spark.



Note that al

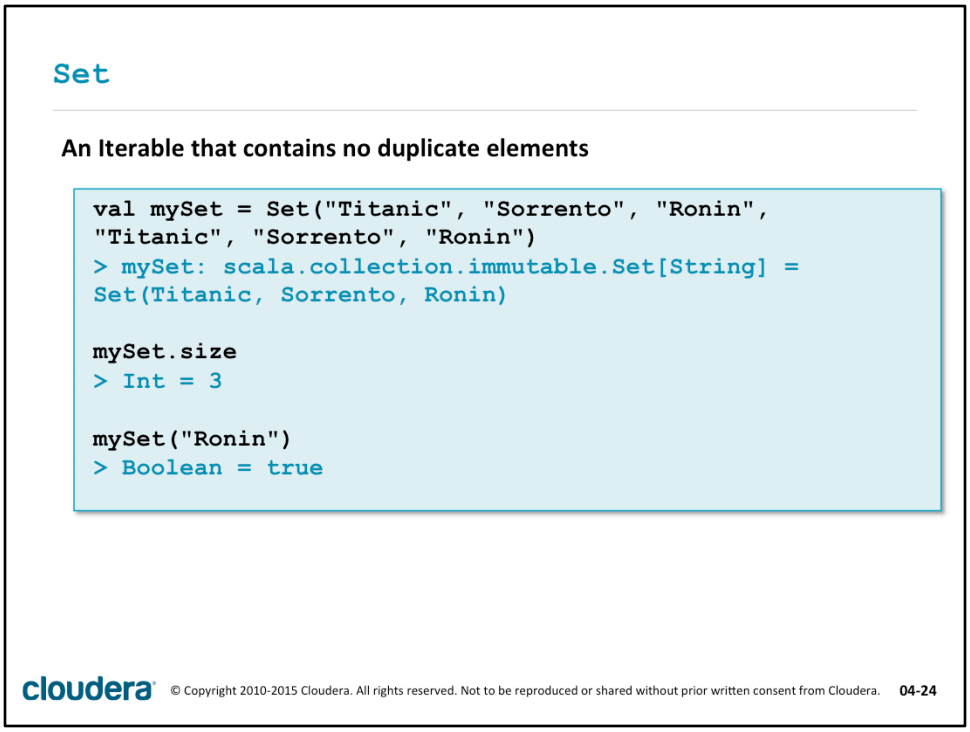


If we are not iterating through each item in our code… and if Scala doesn't have to report back to us after processing each item… but can process all items and only report back the result – then Scala is scalable, because the method of iteration and processing is "behind the curtain".

Chapter Topics

Collections

- Tuples
- The Collections Hierarchy
- **Sets**
- Lists
- Arrays
- Maps
- Common Conversions
- Conclusion
- Hands-On Exercises: Collections

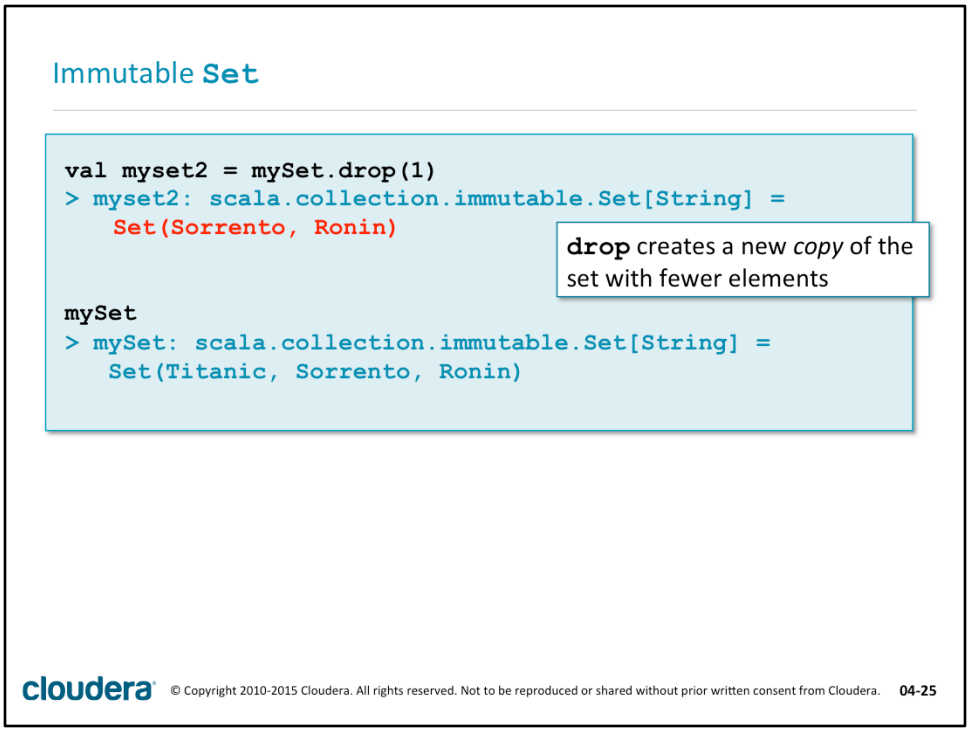


Adding a literal in parenthesis following the variable causes it to test whether the literal is a member of the set, and returns a Boolean.

The basic Set syntax is similar to what we would see in other languages, with a few differences.

First, note that the items in the resulting set are unique – Scala has removed the duplicates – but they are not ordered. Many other languages would have returned an ordered set, "Ronin, Sorrento, Titanic". And since this is a val, the order is cannot be changed after the Set is created, except by generating a new Set.

Second, note that following the Set with parenthesis and a literal causes it to test for membership. In other languages this syntax might result in concatenation of the literal as a new member. So it's important to point out the unique way Scala handles some syntax.



drop(n) drops the first n elements.
remove(n) drops the nth element.

`Set` is an example of a class that exists in two different packages:
scala.collection.immutable.Set and
scala.collection.mutable.Set

Both provide mostly the same set of methods and operators, but they work a little differently. On the Mutable version, methods and operators that modify the set such as drop, ++, --, remove, and update modify the set *in place.* On the immutable version, those same methods and operators return a new copy of the collection, with the modifications applied, leaving the original collection untouched.

This class focuses on immutable types, because those are most common, especially in use in APIs and frameworks like Spark.

Note that the mutability of a collection is not related to whether the variable that points to it is a var or val.

Also note that when inferential typing is used, Scala will always default to the immutable version. You must type explicitly to use the mutable collection type. (This is because immutability is preferred unless there's a specific reason mutability is required, because it is more scalable and performant.)

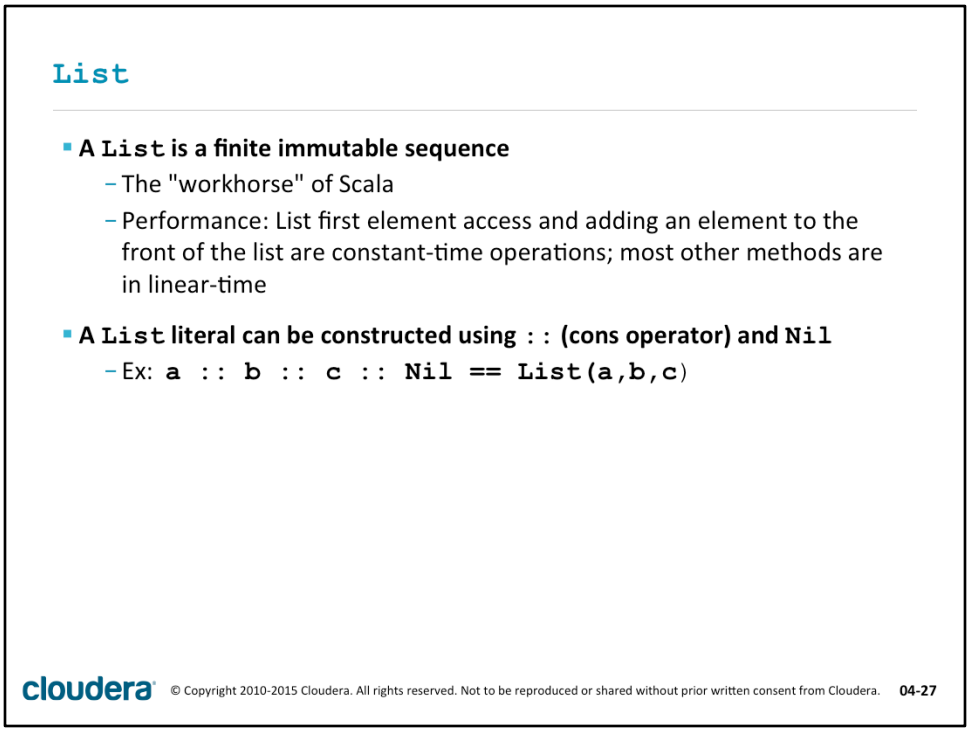
Note that if we had an Array, we could not 'drop' an element, as that is not a supported method. An Array has a fixed number of elements. A Set does not. So even though this is a val, we were able to resize the collection.

Note how these subtle differences define what can be done with alternative collection types.

Chapter Topics

Collections

- Tuples
- The Collections Hierarchy
- Set
- **Lists**
- Arrays
- Maps
- Common Conversions
- Conclusion
- Hands-On Exercises: Collections



Note that unlike Set, there's only one type of list: scala.collection.immutable.List

The corresponding mutable type is scala.collection.mutable.MutableList

Again, we will focus on the immutable version as that's most common for our purposes.

Accessing `List` Elements

- **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
```

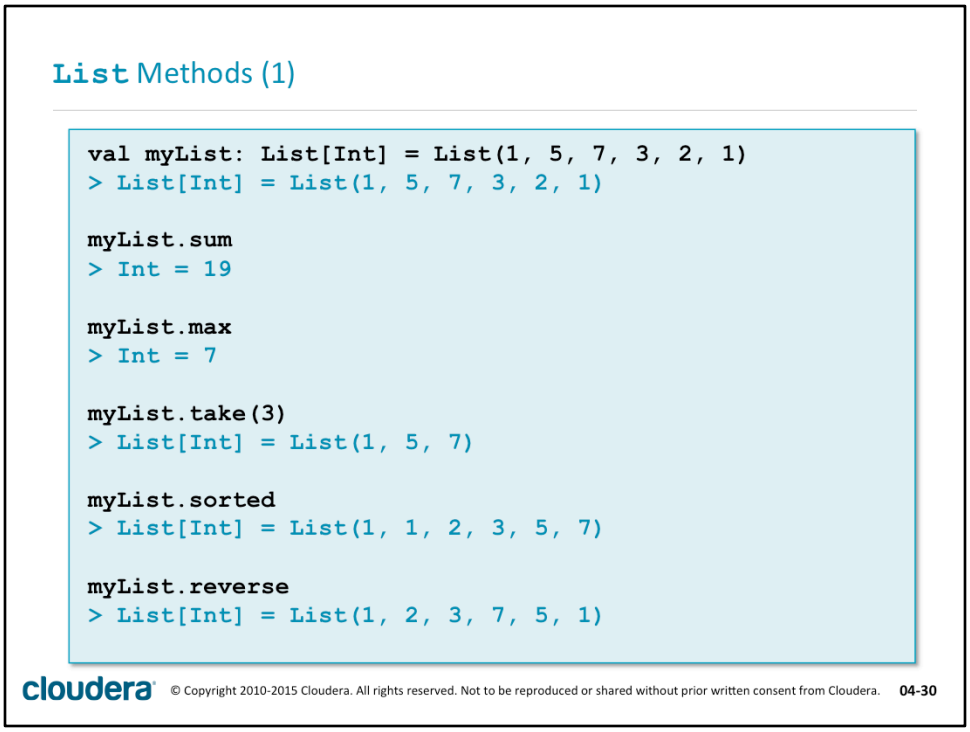

`List` Element Types

- `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","F01L"),
("Sorrento","F41L"), ("iFruit","3"), ("iFruit","3a"))
> devices: List[(String, String)] = List((Sorrento,F01L),
(Sorrento,F41L), (iFruit,3), (iFruit,3a))
```

This page shows a few example methods of List, that operate on a single list.

`List` Methods (2)

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

val myListB = List("iFruit","MeToo","Ronin")
> myListB: List[String] = List(iFruit, MeToo, Ronin)

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

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


This page shows an example of List operations that work on two lists.

Creating Complex `Lists` using `zip`

- **The `List.zip` method pairs items from two lists together into `Tuple2`**
  - **`unzip`** separates **`Tuple2`** elements into individual Lists

```
val myListA = List(1,2,3,4,5)
> myListA: List[Int] = List(1, 2, 3, 4, 5)
val myListB = List("iFruit","Sorrento","Ronin","Titanic",
"MeToo")
> myListB: List[String] = List(iFruit, Sorrento, Ronin,
Titanic, MeToo)

val myListC = myListA.zip(myListB)
> myListC: List[(Int, String)] = List((1,iFruit),
(2,Sorrento), (3,Ronin), (4,Titanic), (5,MeToo))

myListC(1)._2
> String = Sorrento
```

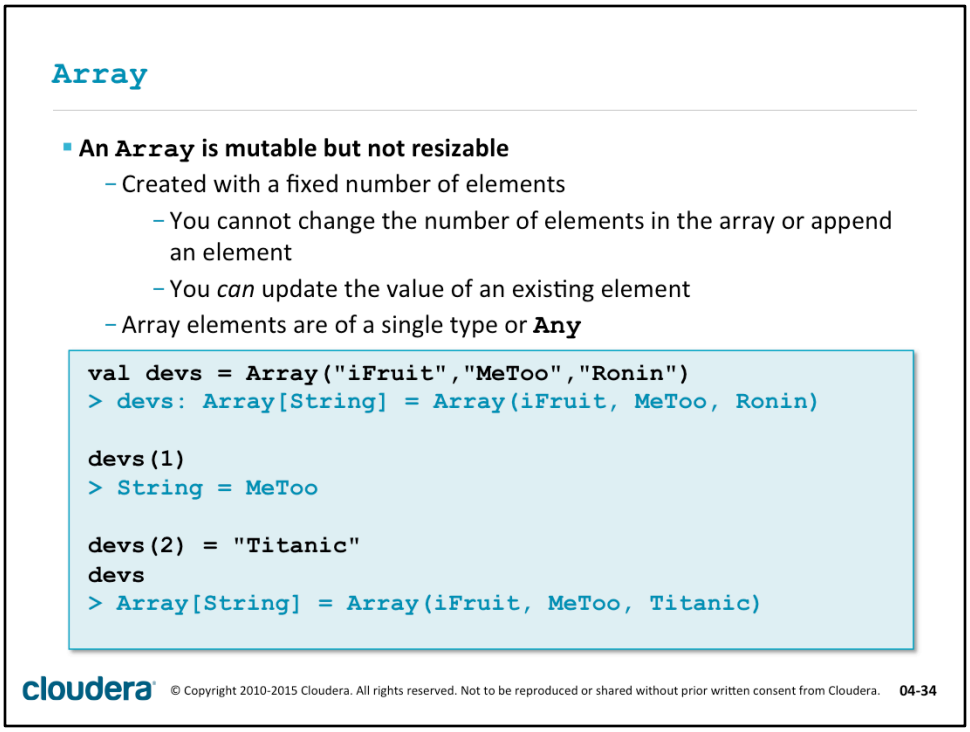

This produces a list of Tuple2s.
The last line shows using both the List subscript syntax and the ._# Tuple element syntax together.

Note that if the lists are of unequal length, the resulting list will have the number of items of the *smaller* of the two Lists. The items at the end of the larger list will be ignored.

Chapter Topics

Collections

- Tuples
- The Collections Hierarchy
- Iterable
- Lists
- **Arrays**
- Maps
- Common Conversions
- Conclusion
- Hands-On Exercises: Collections



The **main distinction** between an Array and a List is that an Array is optimized for random access of elements, whereas a List is optimized for head/tail access.

Important note: in this example, we are using an *immutable* variable to hold a *mutable* collection. This can be confusing. A collection is mutable if the elements or size can be changed. A variable is mutable if it can be changed to have a new value… for instance, to point to a different Array.

More Array Examples

```
val devices: Array[String] =  new Array[String](5)
> devices: Array[String] = Array(null, null, null, null,
null)

devices
> Array[String] = Array(null, null, null, null, null)

devices.update(0,"Sorrento")
devices
> Array[String] = Array(Sorrento, null, null, null, null)

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

devices.length
> Int = 5
```


Why would an API require a List versus an Array ? The two collection types seem to have nearly identical data manipulation features for common activities.

This is probably a good time to remind learners that “under the covers” the two have different performance characteristics. And the Array collection comes from the “optimized for insert/remove/update” and the List collection comes from the “optimized for head/tail access”.

Without the performance distinction, the choice of Array versus List can seem arbitrary and annoying. Especially true if the programmer has experience in a language like Python where the data types are commonly determined mainly by data manipulation requirements, and performance characteristics is often an afterthought.

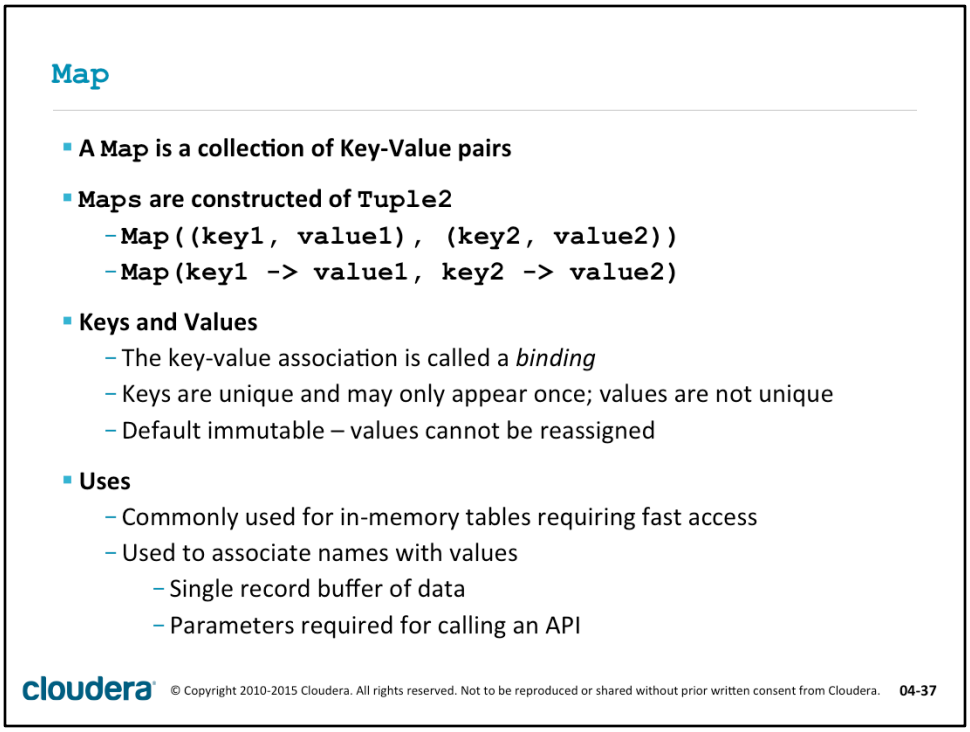
The Scala compiler converts devices(0) = “Titanic” syntax into devices.update(0,”Titanic”) syntax, which is the form that is actually compiled. So there is no performance difference between the two alternative syntaxes. There are often multiple valid ways in Scala to say the same thing, and the simplest syntax is usually preferred and most common. So you will usually see “devices(0) =“

Scala is designed for it’s data types to be extensible. Extending the Scala language, including data type extension, is beyond the scope of this class. The .update method takes any number of arguments. This enables a data type to be extended with additional parameters, only the last of which is treated as a value. That is WHY the .update syntax exists.

Chapter Topics

Collections

- Tuples
- The Collections Hierarchy
- Iterable
- Lists
- Arrays
- **Maps**
- Common Conversions
- Conclusion
- Hands-On Exercises: Collections



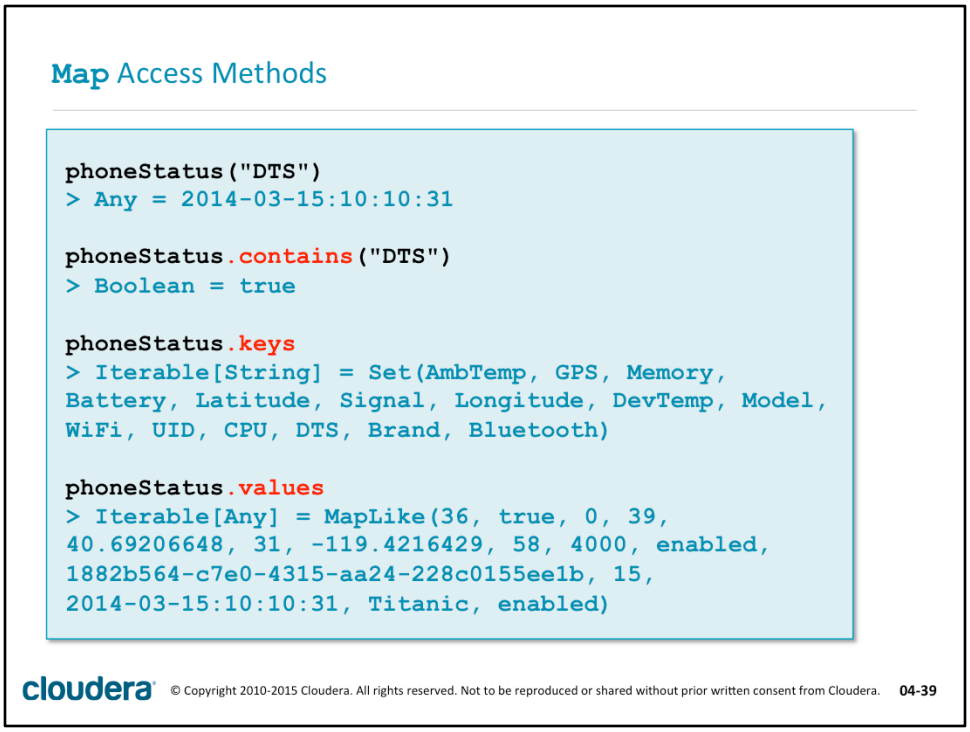
Both the immutable and mutable Collection hierarchies have a Map type. If you don't specify the type, the immutable version will be used.

Example: Phone Record **Map**

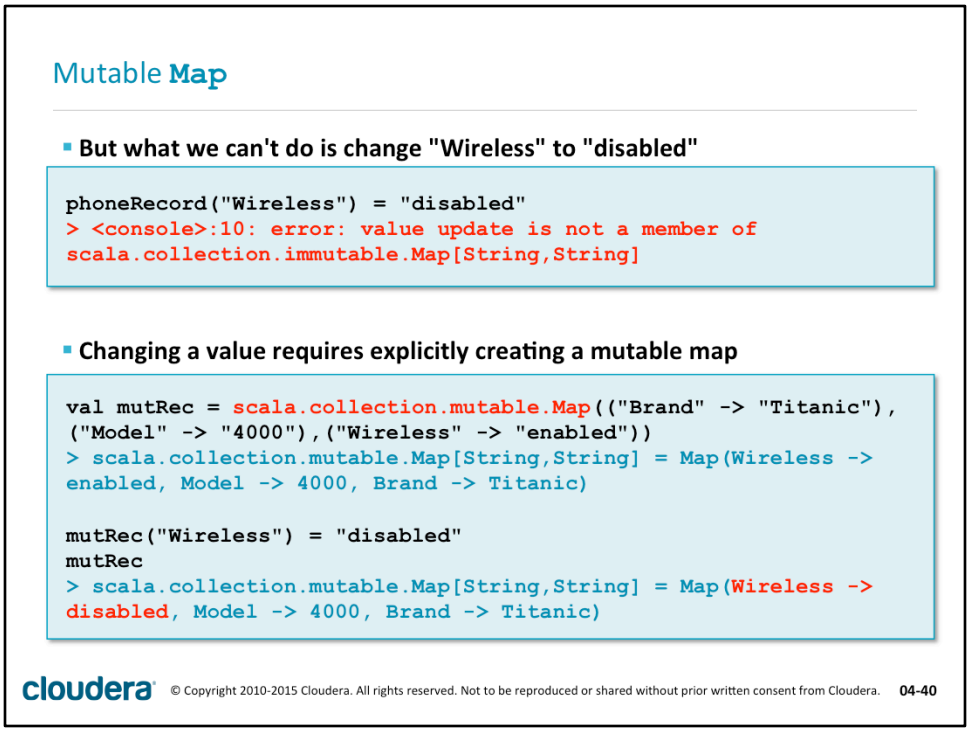
```
val phoneStatus = Map(
("DTS" -> "2014-03-15:10:10:31"),
("Brand" -> "Titanic"),
("Model" -> "4000"),
("UID" -> "1882b564-c7e0-4315-aa24-228c0155ee1b"),
("DevTemp" -> 58),
("AmbTemp" -> 36),
("Battery" -> 39),
("Signal" -> 31),
("CPU" -> 15),
("Memory" -> 0),
("GPS" -> true ),
("Bluetooth" -> "enabled"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

In this example, a Map is used to buffer the values provided when a phone reported an error to LoudAcre.
The values are associated with easily read and understood key strings such as "WiFi".
In the program, the code might access phoneStatus("WiFi").



map-name("key") returns the associated value.
.contains test whether the key is a member of the map, and returns a boolean.
.keys and .values return the list of all keys or all values, respectively.



Seems reasonable. We want to set "Wireless" to "disabled". But we can't do that with the default immutable map. It generates an error.

In this case explicitly creating a mutable map enables updating values.

As noted before, the variable mutRec is immutable, but the Map object it refers to in this example IS mutable.

Chapter Topics

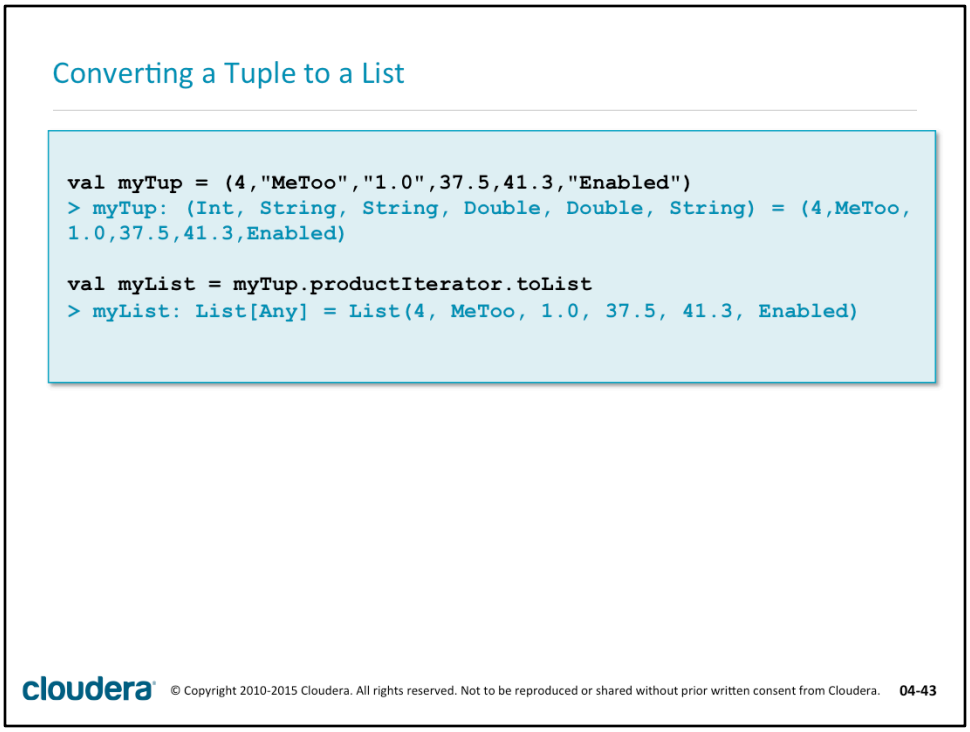
Collections

- Tuples
- The Collections Hierarchy
- Iterable
- Lists
- Arrays
- Maps
- **Common Conversions**
- Conclusion
- Hands-On Exercises: Collections

Converting Between Collection Types

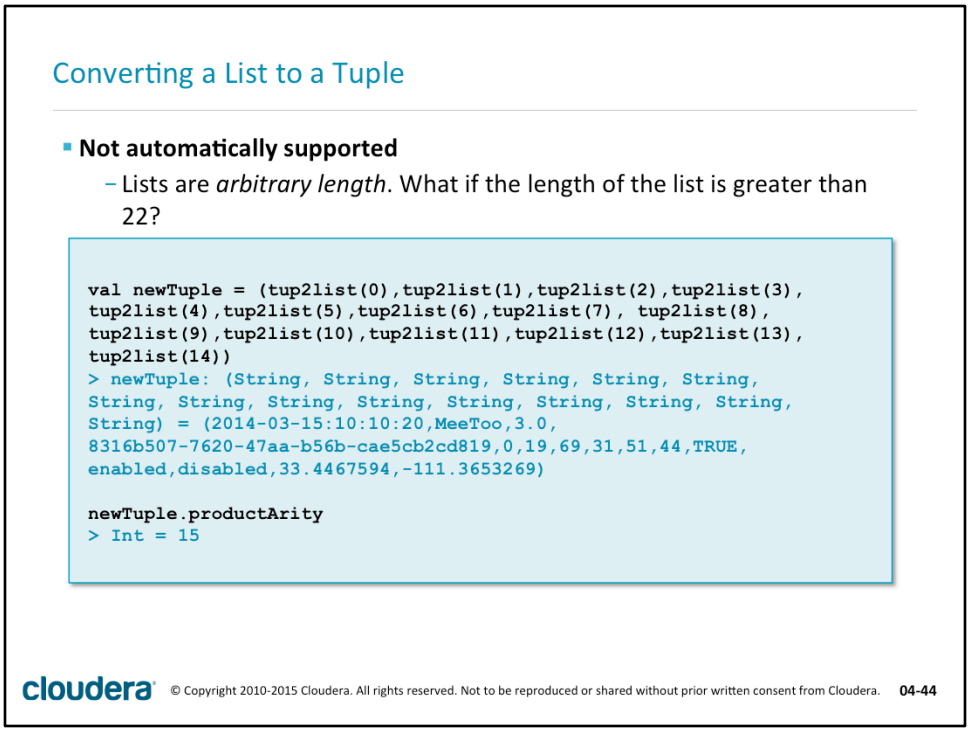
- Easily convert between `Array`, `List` and `Iterable` with
  - `toArray`
  - `toList`
  - `toIterable`

```
val myList = List("Titanic","F01L","enabled",32)
> myList: List[Any] = 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)
```

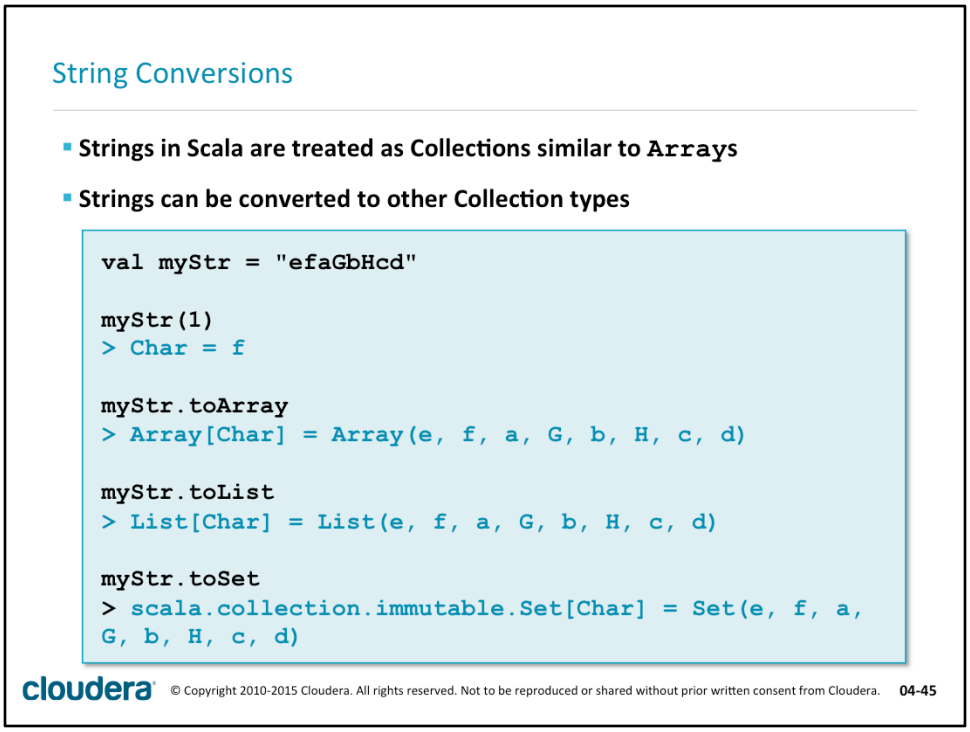



Instructor
Why does the list build by tacking the new element onto the head rather than appending to the tail as expected?

Appending to the end of a linked list is expensive. Each append is an O(N) operation, where N is the original size of the list.
So appending N elements, each to the end of an empty list is an O(N^2) operation.
Therefore, it's less expensive to build the list from the head and then reverse the elements in it. For this reason, the '+=' operator was deprecated.



You can't do this in a type-safe way. In Scala, lists are *arbitrary-length* sequences of elements of some type. As far as the type system knows, x could be a list of arbitrary length.
In contrast, the arity of a tuple must be known at compile time. It would violate the safety guarantees of the type system to allow assigning x to a tuple type.



We covered Strings in the last chapter but now that we have also covered collections, we can mention that Strings are treated as Collections in Scala.

This is probably too much detail for students unless they ask:
Technically, String is Scala is the Java String type: java.lang.String. but Scala implicitly operates on Strings using the wrapper type scala.collection.immutable.StringOps. StringOps is a sibling class to Array, so in effect, Strings work much like arrays and have similar methods and syntax. That is why the syntax String(n) gets the nth offset character in a string.

Chapter Topics

Collections

- Tuples
- The Collections Hierarchy
- Iterable
- Lists
- Arrays
- Maps
- Conversions
- **Conclusion**
- Hands-On Exercises: Collections

Essential Points

- **Tuples**
  - Fixed size: **`Tuple2`**, **`Tuple3`**, …, **`Tuple`*`N`***
  - Not part of the Collections hierarchy but similar
- **Scala Collections hierarchy**
  - Starts with abstract classes Traversable and Iterable
  - Parallel mutable and immutable types
- **List**
  - The "workhorse" of Scala
- **Array**
  - Created with a fixed number of elements and not resizable
- **Map**
  - For working with key-value pairs

Chapter Topics

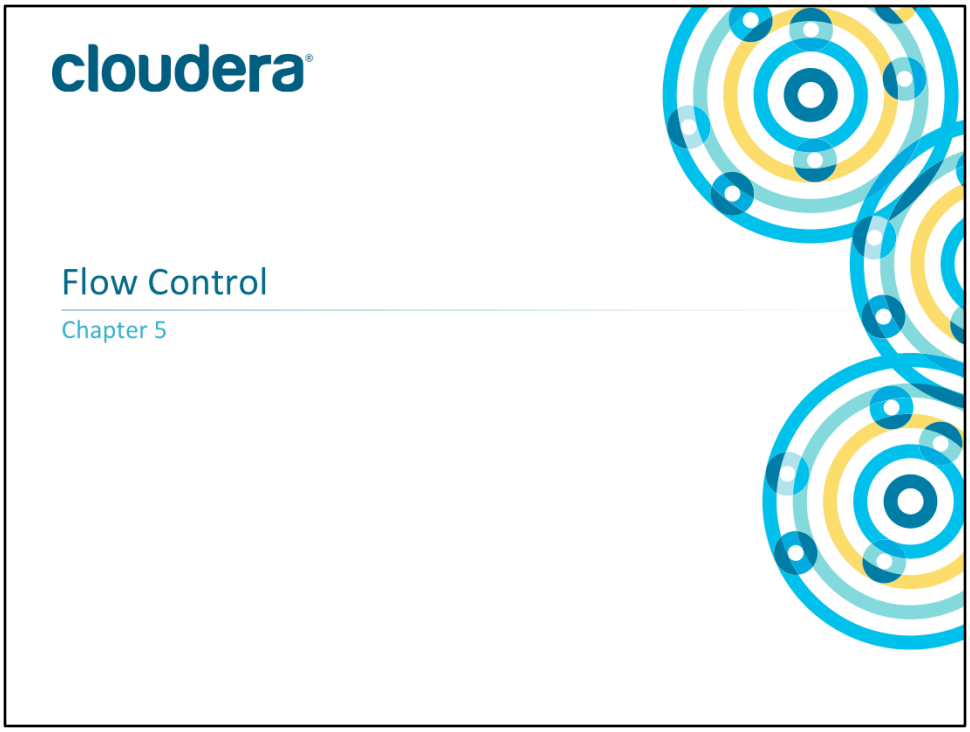
Collections

- Tuples
- The Collections Hierarchy
- Iterable
- Lists
- Arrays
- Maps
- Conversions
- Conclusion

Hands-On Exercises: Collections

Hands-On Exercise: Collections

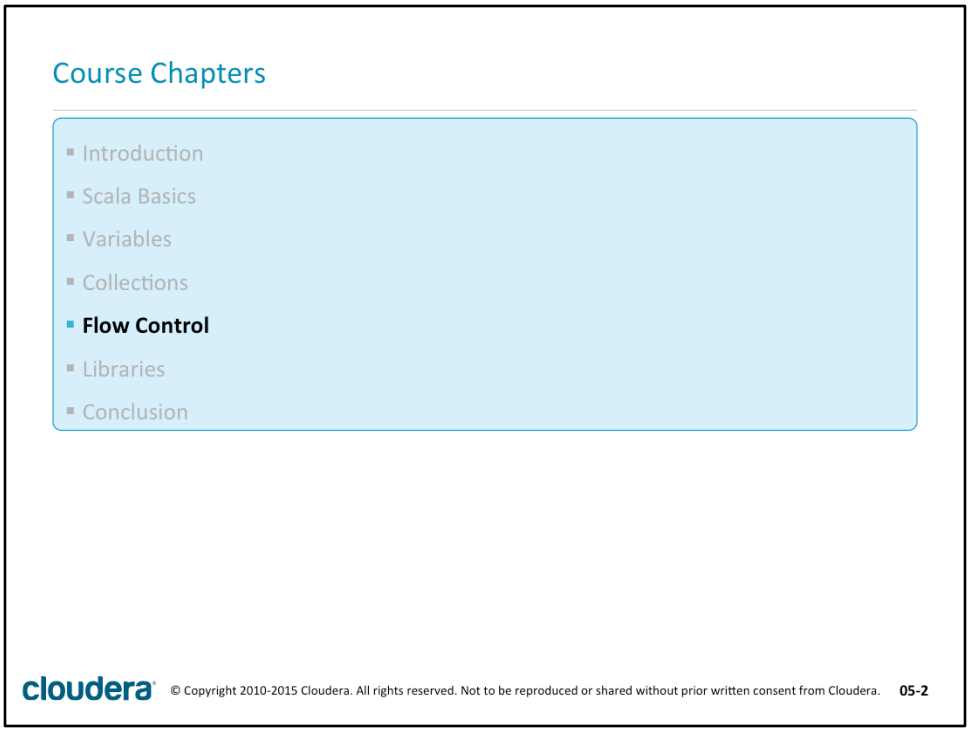
- **In this Hands-On Exercise, you will**
  - Explore Tuples and Collection types
  - Convert between different Collection types
- **Please refer to the Hands-On Exercise Manual for instructions**



The story of this chapter is as follows.

(1) It starts with conventional Flow Control as you would see in any language.
Sometimes you need to use conventional flow techniques in your application, even if it isn't scalable.

(2) Then the chapter builds into Functional Programming flow control, where the goal is to define a collection and call a high-level method on the collection.



[*] Flow control – how control is passed from one section of code to another.

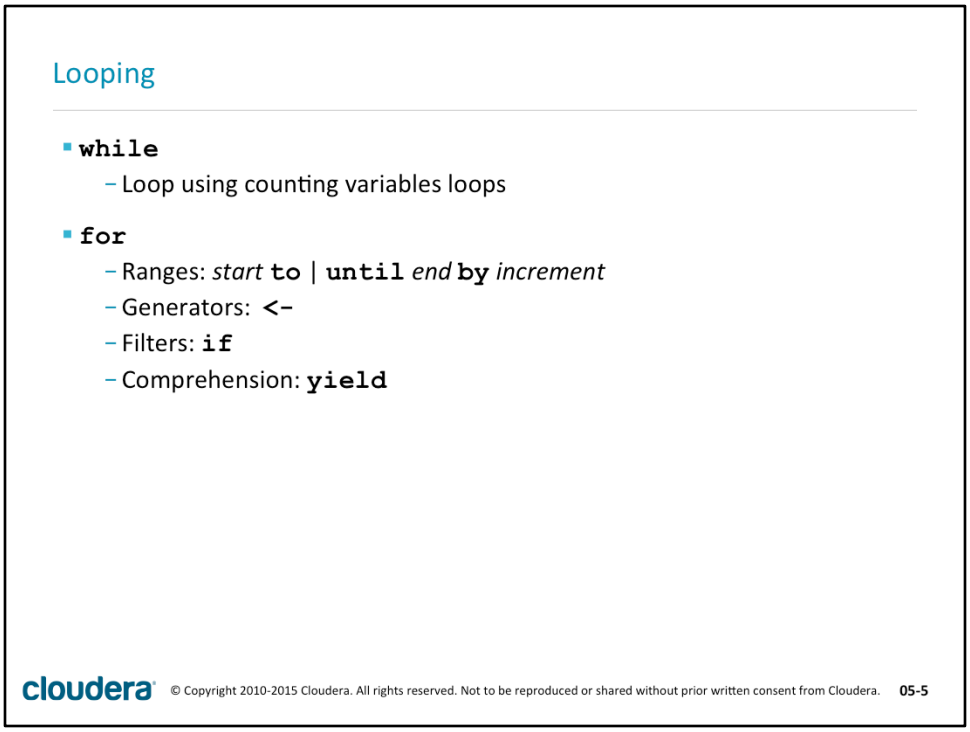
Chapter Topics

Flow Control

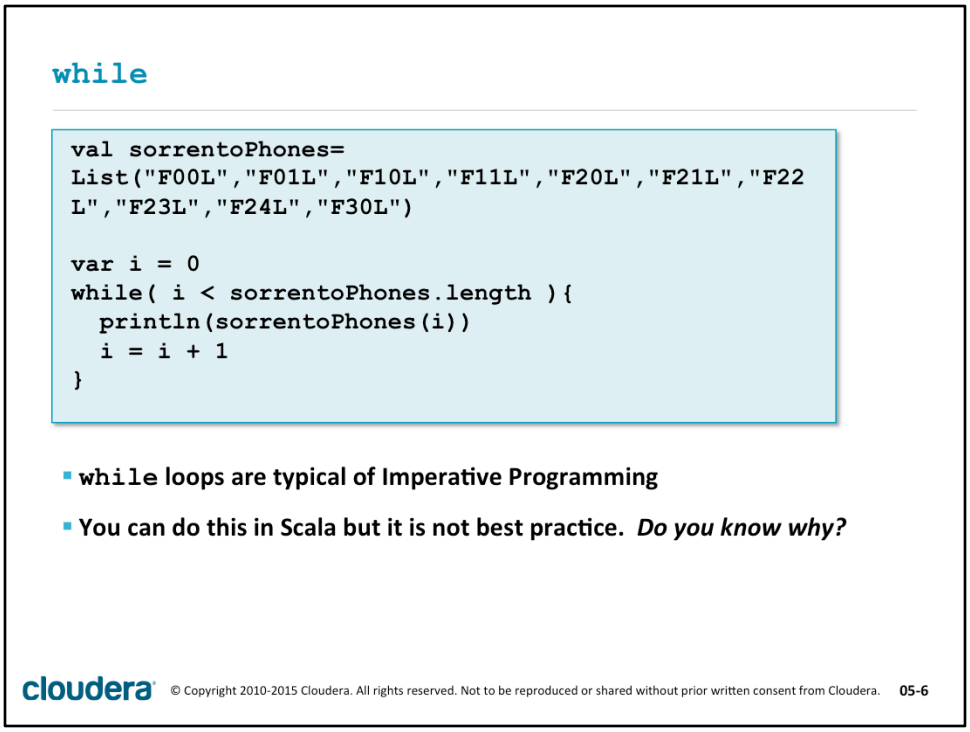
- **Looping**
- Iterators
- Functions
- Passing Functions
- Collection Iteration Methods
- Pattern Matching
- Conclusion
- Hands-On Exercises: Flow Control

Flow Control

- **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: Programs sends message to data to drive a method**
  - Data operates on itself
- **Functional: Program *implies* what needs to be done**
  - Framework figures out what needs to be done and does it



While loops require a testable changing value to determine when the loop should terminate; therefore, they require variables –vars– which are contradictory to Scala's scalability design principles.
Therefore, you will likely see much less use of while loops in Scala code.



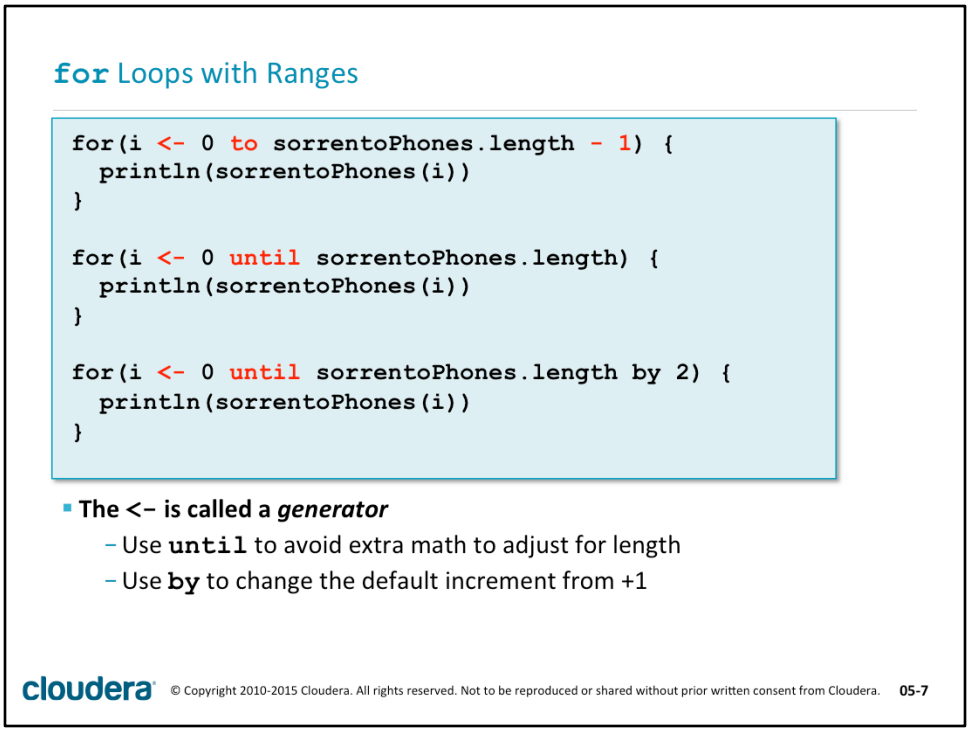
"**Do you know why?**" This code is going to look perfectly fine to Java programmers. And we are going to use it to impress the learners with the fact that Scala programming is a different skill and in some cases conflicts with the good practices they've learned in Java.

We are using "while" here to introduce "for". And we explain why counting variables are a bad idea for scalability.
We will come back to "while" in its proper use within Scala (with iterators) following the discussion on "for".

Because a while loop has to have some condition to test that must change. That means either I/O or, as in the code example here, a *counting variable*.
In Functional Programing, I/O is a "side effect" and therefore should be isolated away from pure functional code. And variables don't scale well to distributed frameworks, where parts of the algorithm on separate computers can get out of synch if the variable is modified by one part when another part is not expecting it.

So although while is supported and exists for backward compatibility and to support imperative programming, it's use is discouraged. And you won't see it used with counting variables often in Scala code.

Another, subtler problem about the use of an index to access the members of a collection, is that in Scala, some collections are not guaranteed to be processed in order.
In the hierarchy, we have Traversable, Iterable, and Seq (sequence). Only Seq and it's inheritors are guaranteed to have consistent objects at indexed offsets.
Traversable will process all items in the collection. But since it is allowed to process the items in parallel in multiple threads or on multiple computers, the order is not guaranteed. Traversable might divide up 20 items into groups of 5 and process them in four threads, so that item(1), item(5), item(10) and item(15) are processed first in time. Iterable guarantees that the collection can be processed in order. But the .foreach( ) method of iterable guarantees only that it will provide the program with each item in order as it would have been processed by Traversable. So it's not until we get to Seq that the index method above is guaranteed to work as expected.

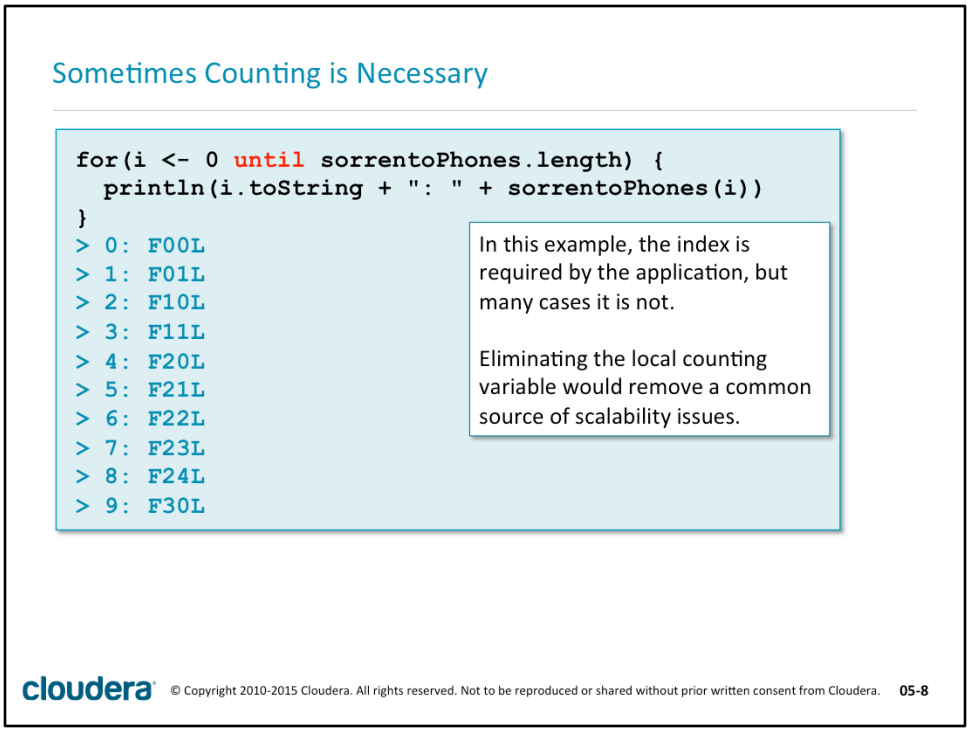


Why count and use a local counting variable if we can avoid it? One less thing to go wrong.

The for() construct in Scala is the center of item-at-a-time iteration. It doesn't work like for loops in other languages.
Over the next several slides we are going to build up a solid foundation on how to use the Scala for() construct.

Along the way we will dismiss counting variables.

To avoid a bounds error, 1 had to be subtracted from the length. In the second example **`until`** avoids the extra math which is error prone



Shows that if the output requires counting, in this case a numbered list, then "for" or "while" with a local counting variable is correct.
However, in other cases, what's desired is scalability. And getting rid of the local counting variable is critical to making the code able to be distributed.

`for` Iteration Over a Collection

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

> F00L F01L F10L F11L F20L F21L F22L F23L F24L F30L
```

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

This is the preferred form of *explicit* iteration in Scala.

As we will see, there are many more types of implicit iteration available.

`for` with Multiple Generators

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

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

> iFruit Z1, iFruit Z-Pro, iFruit Elite-Z, MeToo
Z1, MeToo Z-Pro, MeToo Elite-Z, Ronin Z1, Ronin Z-
Pro, Ronin Elite-Z,
```

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

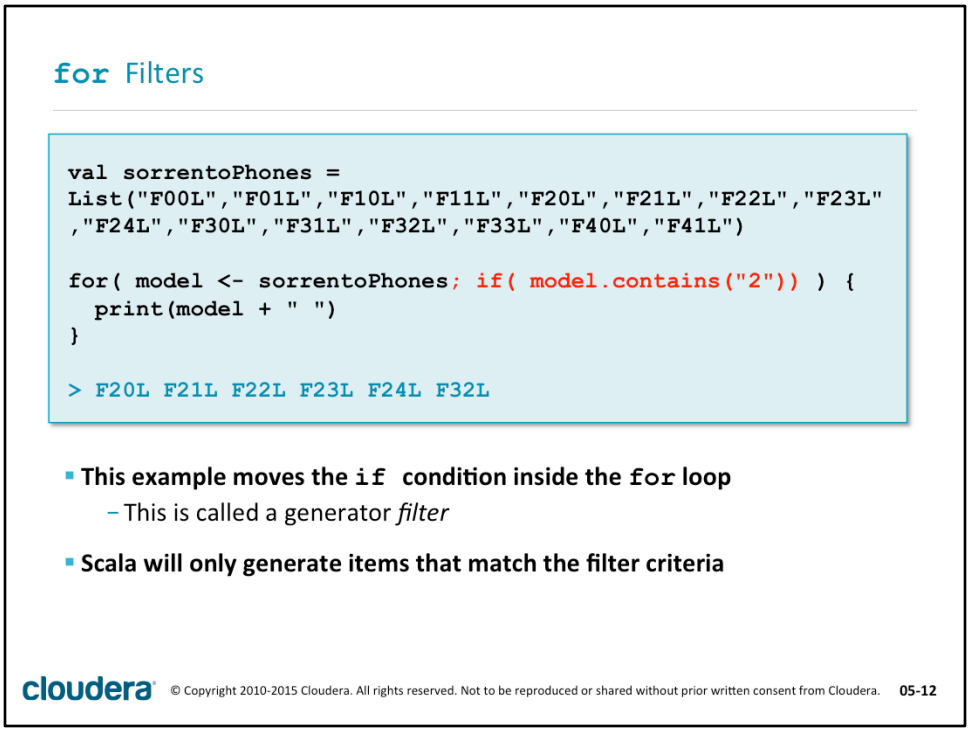
Product Marketing wants to see how the new model names would look when matched up with different brands of phone.
This example shows multiple generators within a single for() loop.

Subset of the Collection

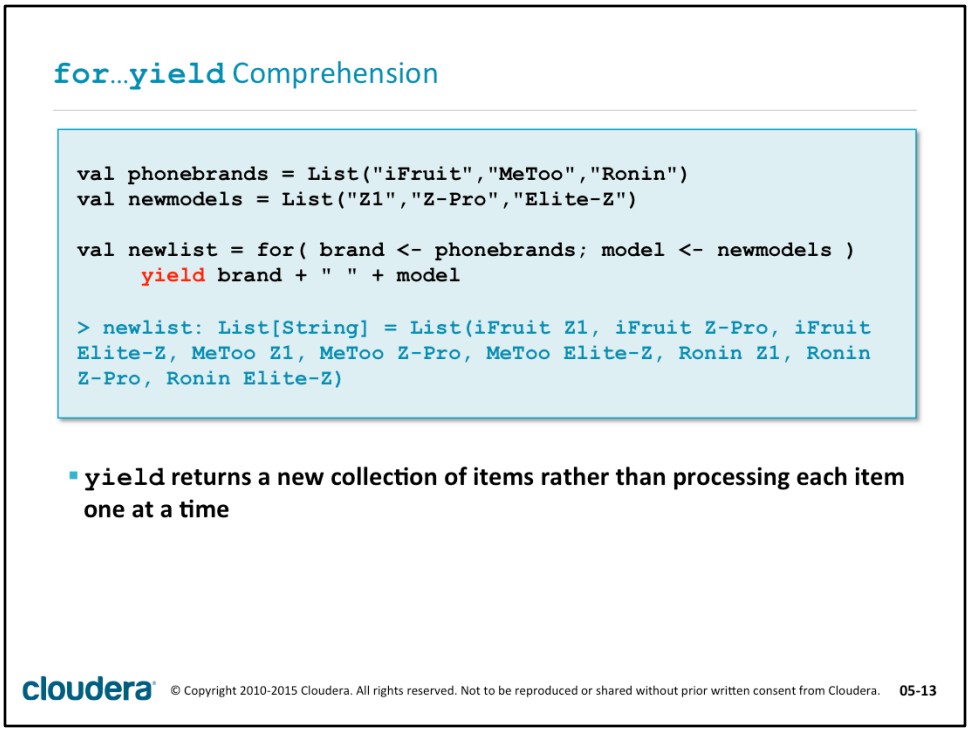
```
val sorrentoPhones =
List("F00L","F01L","F10L","F11L","F20L","F21L","F22L","
F23L","F24L","F30L","F31L","F32L","F33L","F40L","F41L")

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

> F20L F21L F22L F23L F24L F32L
```

- `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

Scala has a better option…



It's a hallmark of Big Data methods to detect conditions where you would be generating data and then discarding it, and then to remove the unnecessary processing by filtering out the data before it is processed by not generating it in the first place.



A "comprehension" is a Functional Programming concept: it is a syntactic construct for generating a list (or set, or other collection).

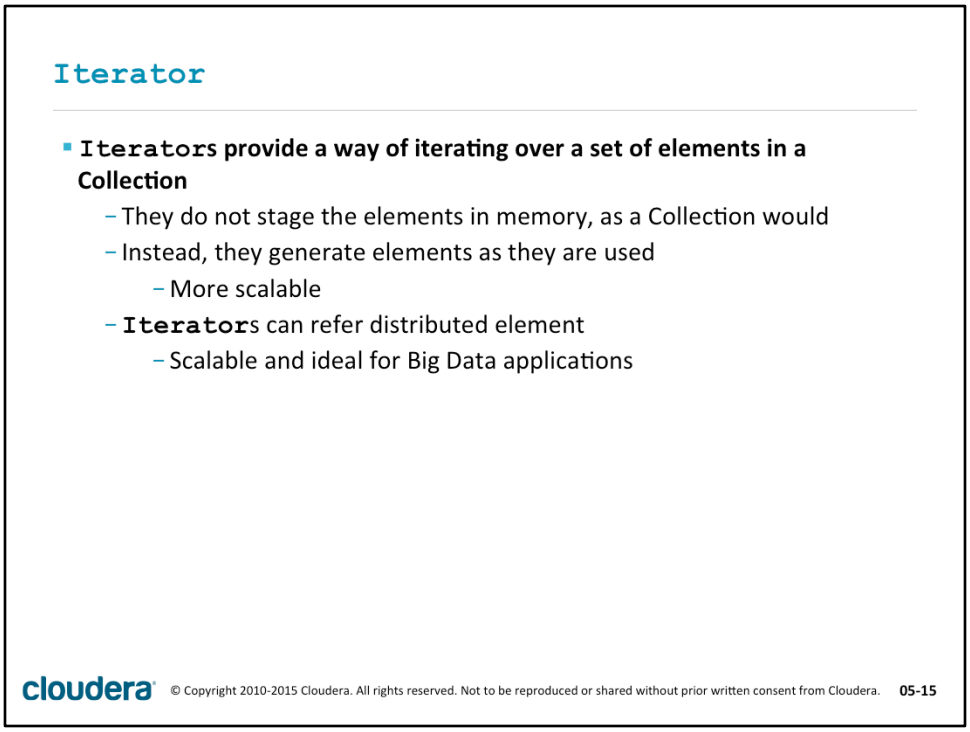
The yield keyword saves the results of each cycle through the loop. Scala infers the collection type that should be used to accumulate the results.

Note that the yield in Python works differently than in Scala. In Python, "yield" returns a generator (similar to Scala's iterator). In Scala, it returns a materialized collection (which could be used to create an iterator, as covered shortly.)

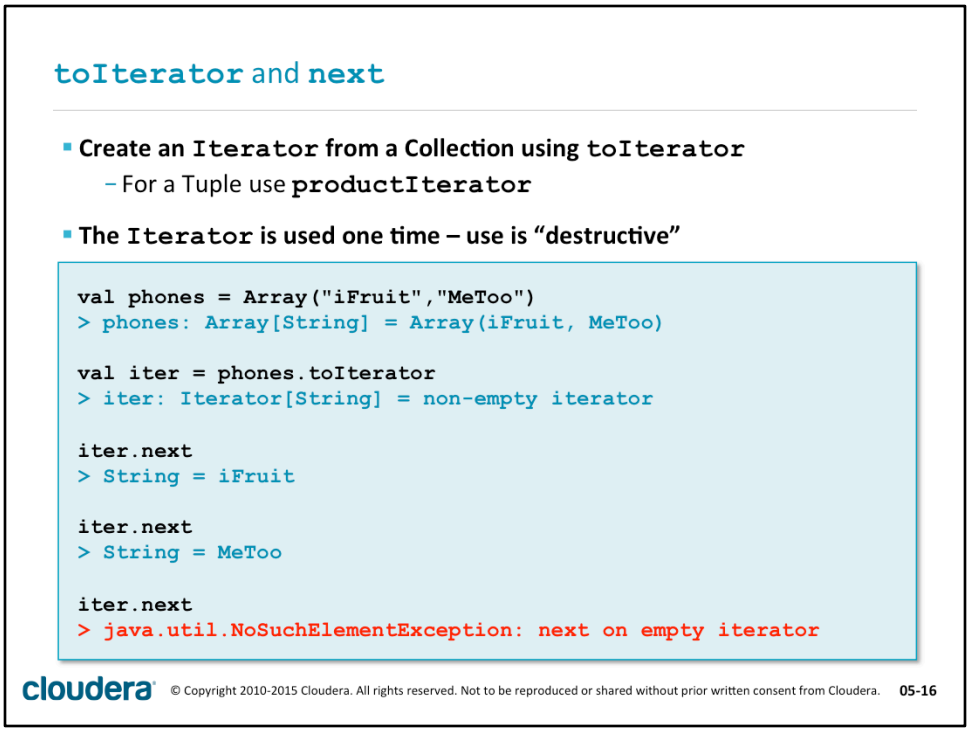
Chapter Topics

Flow Control

- Looping
- **Iterators**
- Functions
- Passing Functions
- Collection Iteration Methods
- Pattern Matching
- Conclusion
- Hands-On Exercises: Flow Control



While loops require a testable changing value to determine when the loop should terminate; therefore, they require variables –vars– which are contradictory to Scala's design principles.
Therefore, you will likely see much less use of while in Scala code.



Iterators are "traversable once". Once an iterator item is used for anything – even .toString – its reference is consumed.

It does not change the underlying Collection.

If you need to loop on the same Collection again, create a new Iterator.

Using Iterators in a `while` Loop

```
val titanicPhones =
List("1000","1100","2000","2100","2200","2300","2400","2500"
,"3000","4000","DeckChairs")

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

print(iter.next)
> 1000
print(iter.next)
> 1100

while(iter.hasNext) {
  print(iter.next + " ")
}
> 2000 2100 2200 2300 2400 2500 3000 4000 DeckChairs
```

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

Every time iter.next is used, it returns the current item and moves on to the next. Once an item is accessed, that part of the iterator is consumed, and you can never go back without creating a new
iterator.

Other `Iterator` Methods

- **Some key methods for working with Iterators**
  - **`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 only elements for which the expression is true
  - **`foreach(`*`function`*`)`** – execution **`function`** for each item in the iterator

cloudera

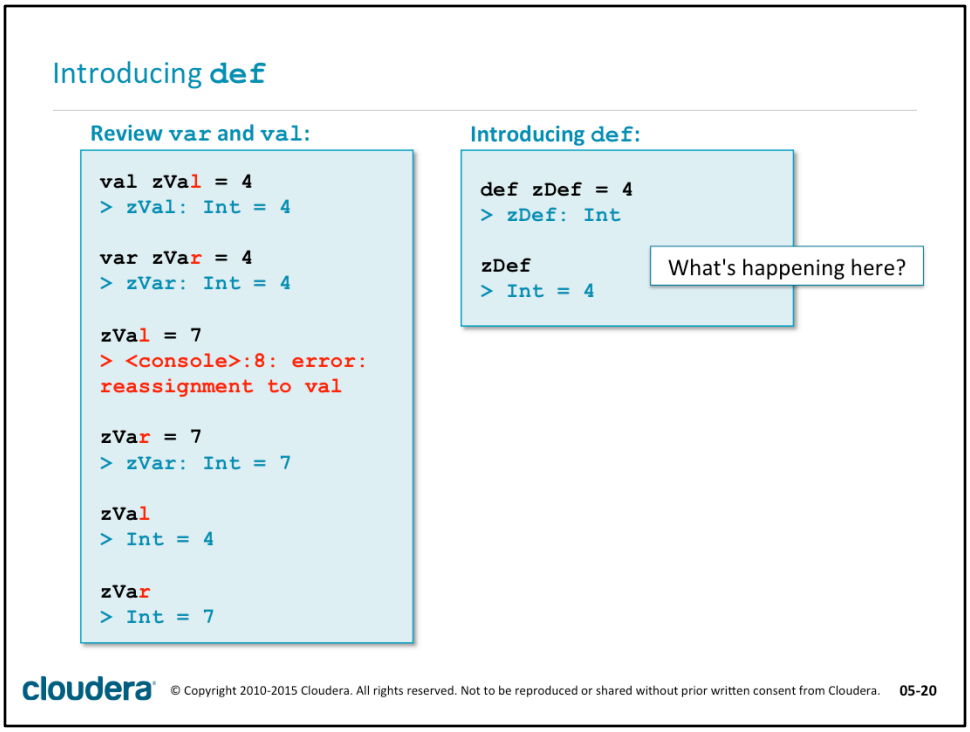
These methods are not covered in this course in detail because they aren't used much in Spark, but they are helpful for understanding.

Many other iterator methods are available, many of them similar to collection methods such as contains, map, min and so on. There are also methods for convert to a collection such as toSet, toList and so on.

Chapter Topics

Flow Control

- Looping
- Iterators
- **Functions**
- Passing Functions
- Collection Iteration Methods
- Pattern Matching
- Conclusion
- Hands-On Exercises: Flow Control

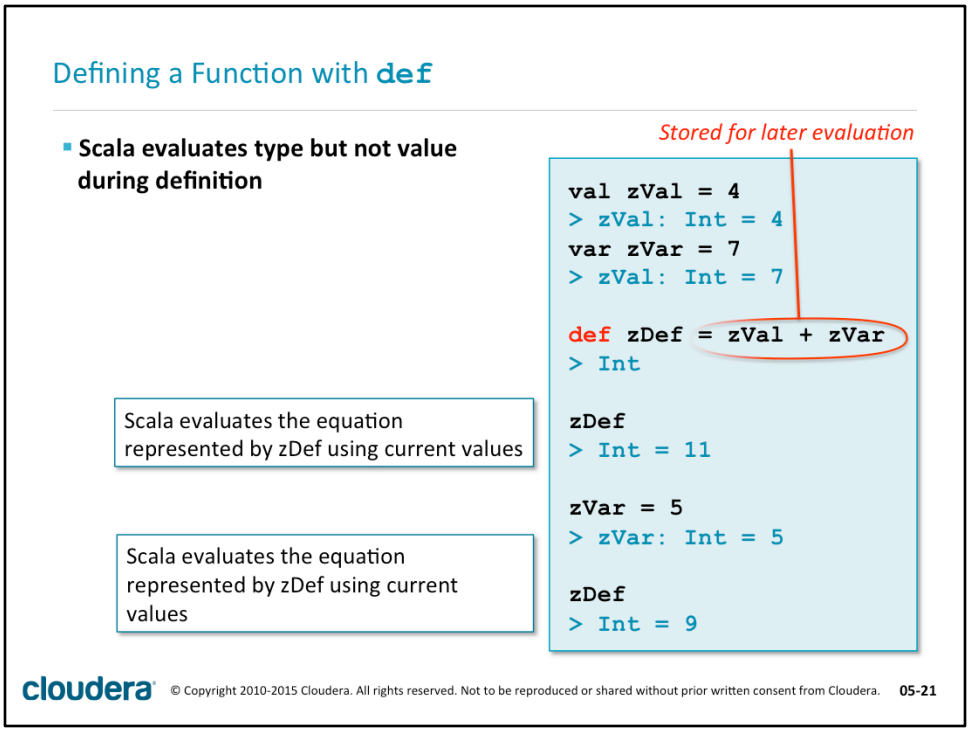


In Scala, unlike other languages, def has a special relationship to val and var.

Introduce the concept of *lazy evaluation* using **def**. This highlights the integration of, and special relationship between, data and functions in Scala.

Cover this slide carefully. The purpose of this slide is to show the relationship between the var and val keywords (data) and the def keyword used to define functions.
The key distinction in Scala is that the literal on the right side of the equal sign is evaluated at assignment time for var and val, but with def, the literal on the right side of the equal sign is stored and only evaluated when required. So, essentially, def causes *lazy* evaluation. This is an extremely clever implementation of functions, and it is from this very simple and powerful approach that Scala gains much of its power for Functional Programming.

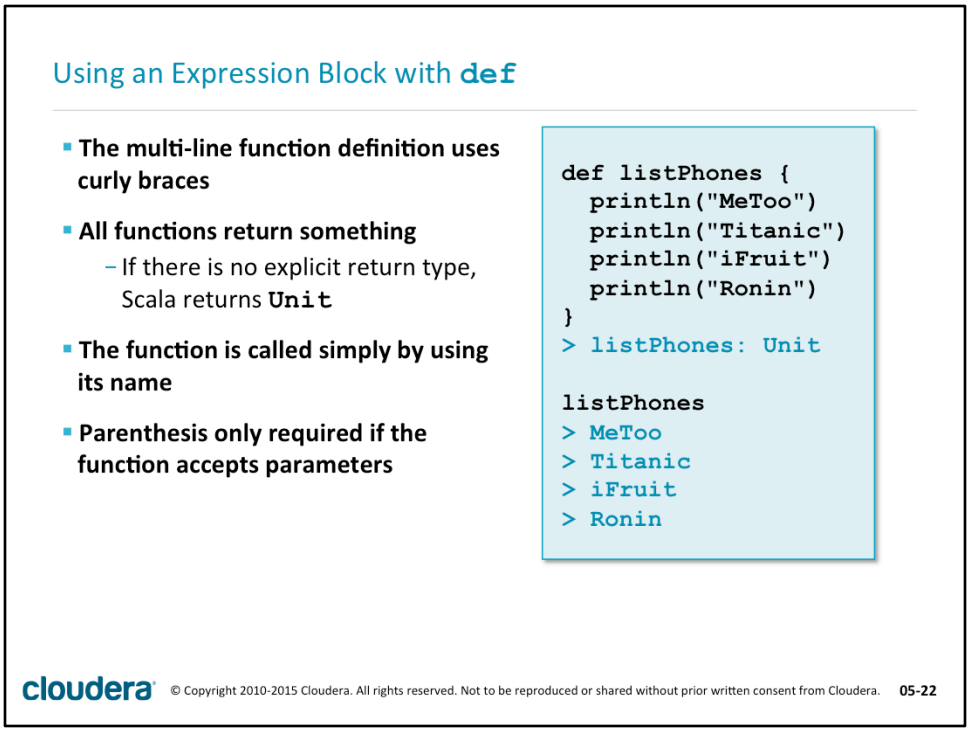
"What's happening here?"
zDef is storing the Integer literal "4". It evaluates the type during assignment. But it does not evaluate the value until it is called on to use it.
So… after def zDef = 4, we see that it has evaluated the type to an Int, but unlike var and val, the result currently has no value. Only when zDef is used, does Scala evaluate the literal string "4" and determine that it is an Int of value 4.



A function is evaluated when it is used.

The previous slide illustrated lazy evaluation with an integer literal. To make the point more explicit, we are going to store an equation for later evaluation.
The point is, the value of the function is not determined at assignment time, as it would be with val and var, but at the time that it is used.

Because zDef stores the literal "zVal + zVar", it is only evaluated on output. So changing the value of zVar causes the evaluation to result in a different number.
Also, notice that when zDef was defined, Scala was able to infer type, and it was already assigned a type of Int. So type inference is not necessarily lazy, only the value evaluation.
In some cases, Scala may not know what the type will be at definition time, and may have to infer type at evaluation or may error and require explicit type declaration.



Curly braces delimit an expression block, which is used to make multi-line functions. This function has no parameters – how do we pass parameters? Next section.

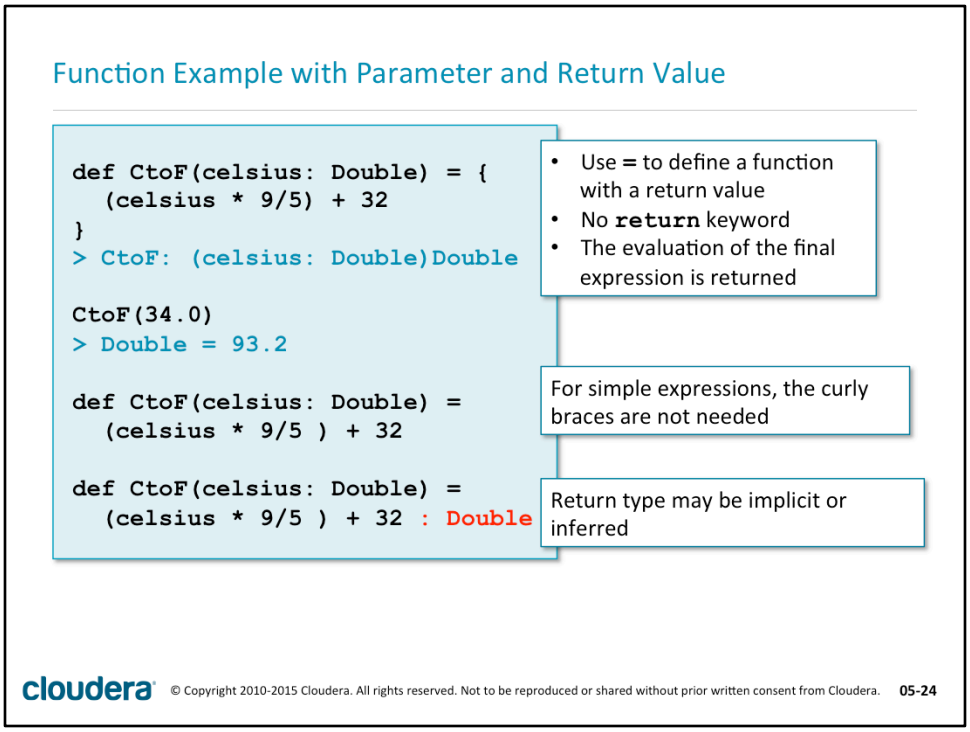
This slide also introduces the concept of "Unit" type.

Scala design is mathematically inspired. In "C" you'd have a 'void' return type. In Scala, the "Unit" return type is actually a *variable*.
It's a "non-instantiable" variable, which means there can only ever be one of them, hence the name *Unit*. So, at least theoretically, there is a solitary Unit variable someplace within Scala at a specific memory location, and this Unit variable can never take on any other value, and there can never be more of them. This distinction makes no practical difference in code. It's still returning 'no value' which is discarded by the code that invokes the function. Why not use 'void' in keeping with other languages and common usage? Because it wouldn't be mathematically correct in concept. In Scala, you have to sometimes expect to learn things that make no practical difference in common code usage for the sake of mathematical purity.

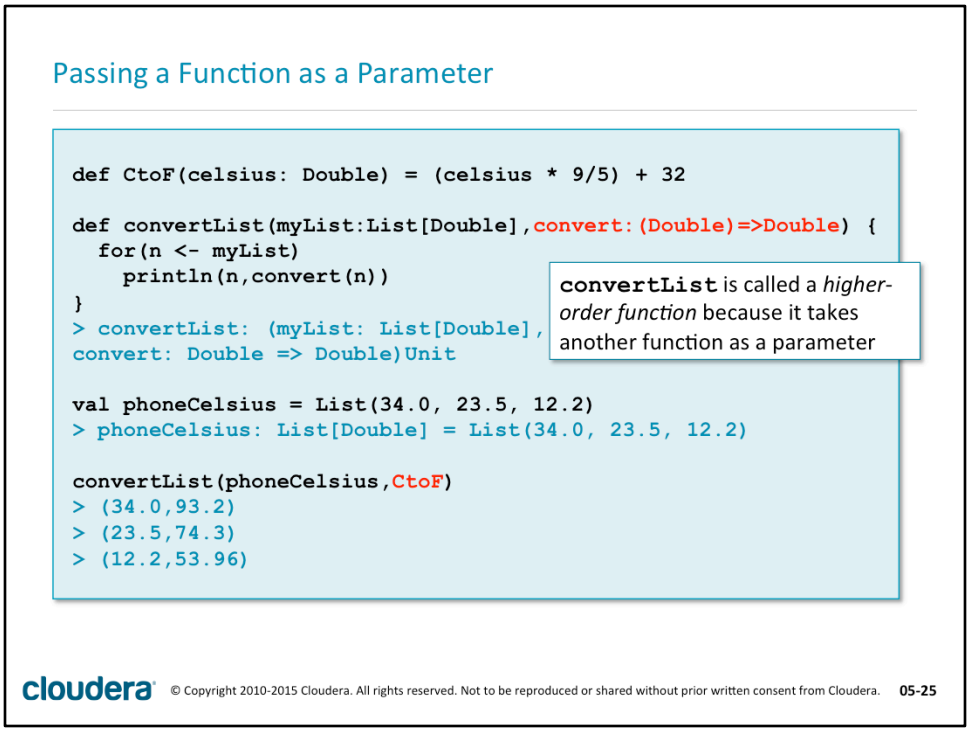
Chapter Topics

Flow Control

- Looping
- Iterators
- Functions
- **Passing Functions**
- Collection Iteration Methods
- Pattern Matching
- Conclusion
- Hands-On Exercises: Flow Control



This example returns a value. How do we explicitly specify the return type?
Equals sign – return value
Colon – explicit type definition



In this example, we are going to define a general program that converts each value in a list into another value.
To do this we will pass a list of phone temperatures in Celsius to the convertList function, along with a conversion method, CtoF.

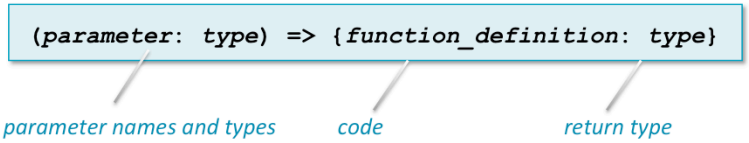
In this example, we are declaring a prototype for a function that will be passed as a parameter to the higher-order function convertList.
The notation internal-name: input-type => return-type.

phoneCelsius is a list of phone temperatures reported in Celsius
CtoF is a conversion function that will take a Celsius temperature as an input and return a Fahrenheit temperature

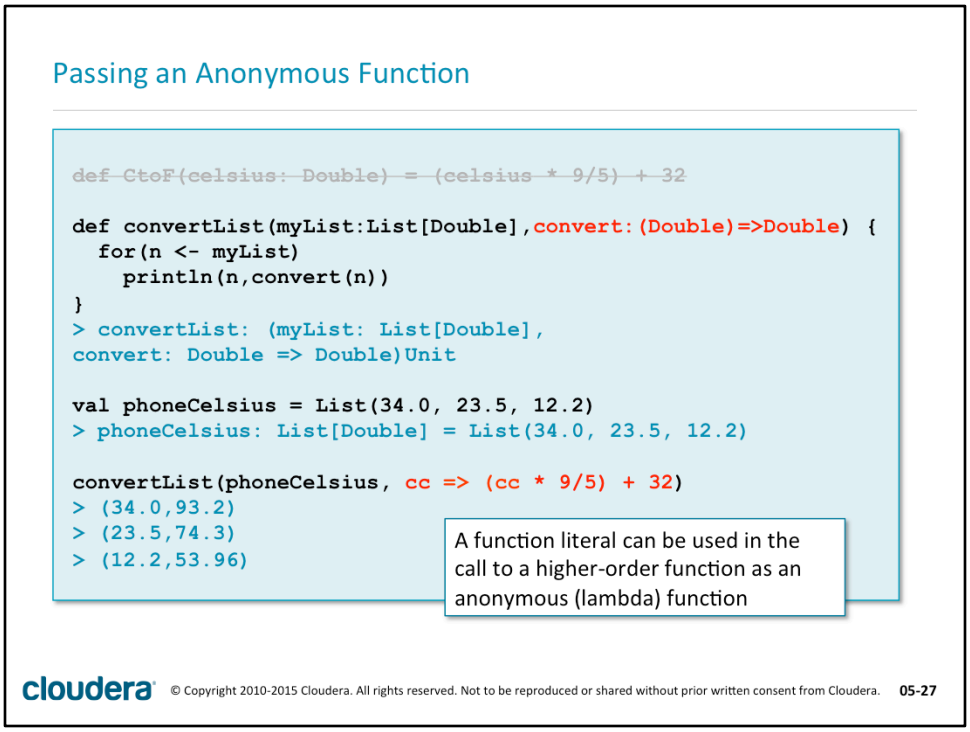
convertList is a higher-order function. It's higher-order *because* it is defined to accept a parameter as a function.
convertList doesn't know 'CtoF' when it is being defined. So it is going to use the internal name of 'convert' to refer symbolically to whatever function it has been passed.
Note: CtoF is the external name of the function. When it is called in convertlist, CtoF is given the internal "code" name of "convert". Later, when convert is called, it is referring to CtoF.
The parameter definition tells us (and Scala) that whatever function is passed must require one input parameter of type Double, and must return type Double.

(Fortunately, CtoF meets this criteria) Look at the definition of CtoF … (celsius: Double)Double means one input parameter of type Double and return type Double.

Function Literals



- **An alternate syntax for defining functions**
- **Does not require a function name or label**
- **A way to define a function "inline", embedded in other code**
  - Example: Within a parameter list to another function
- **Also known as**
  - Lambda functions
  - Anonymous functions



In this example we've done away with CtoF altogether. Instead, we are using a Function Literal right in the call to convertList.
This is handy for functions that will only be used one time. This function literal is never given a name (ie with def) or assigned to a variable; so it is 'anonymous'.

convertList still calls the function by the internal name of "convert". However, the function is defined anonymously – outside of convertList, it has no name.

This is one of the key concepts of this class. Anonymous Functions are used extensively in Cloudera classes to pass function values to Spark for distributed execution.

It is quite common in Scala to define multi-line anonymous functions using { } right in the call to higher-order functions, rather than using named functions.

Chapter Topics

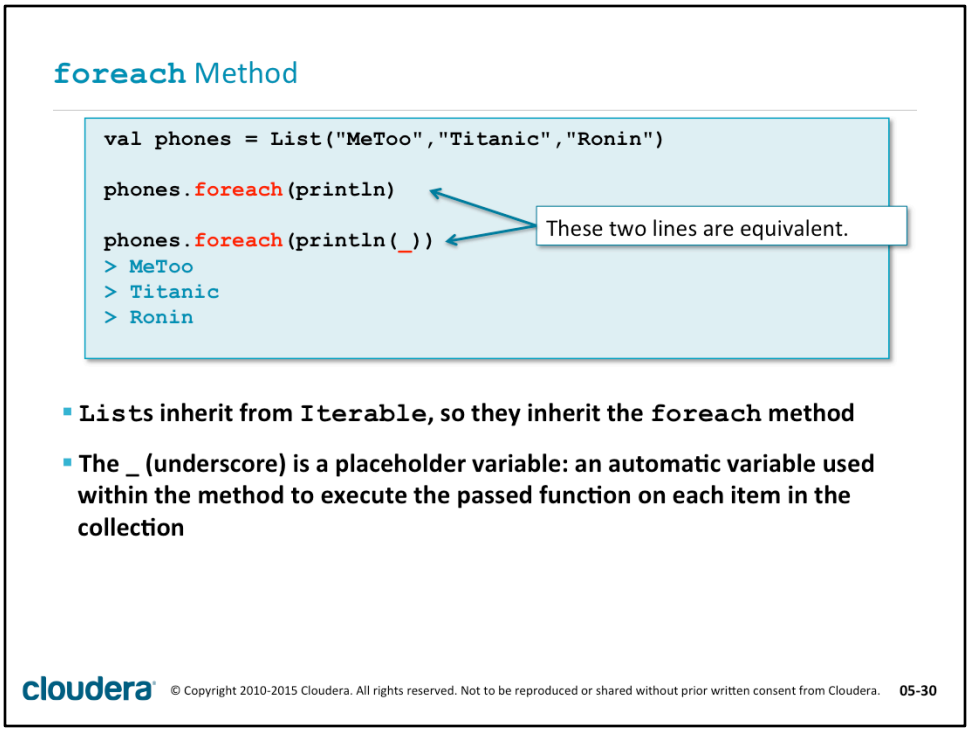
Flow Control

- Looping
- Iterators
- Functions
- Passing Functions
- **Collection Iteration Methods**
- Pattern Matching
- Conclusion
- Hands-On Exercises: Flow Control

Higher-Order Collection Methods

- **Common Collection Methods**
  - `foreach`
  - `map`
  - `filter`

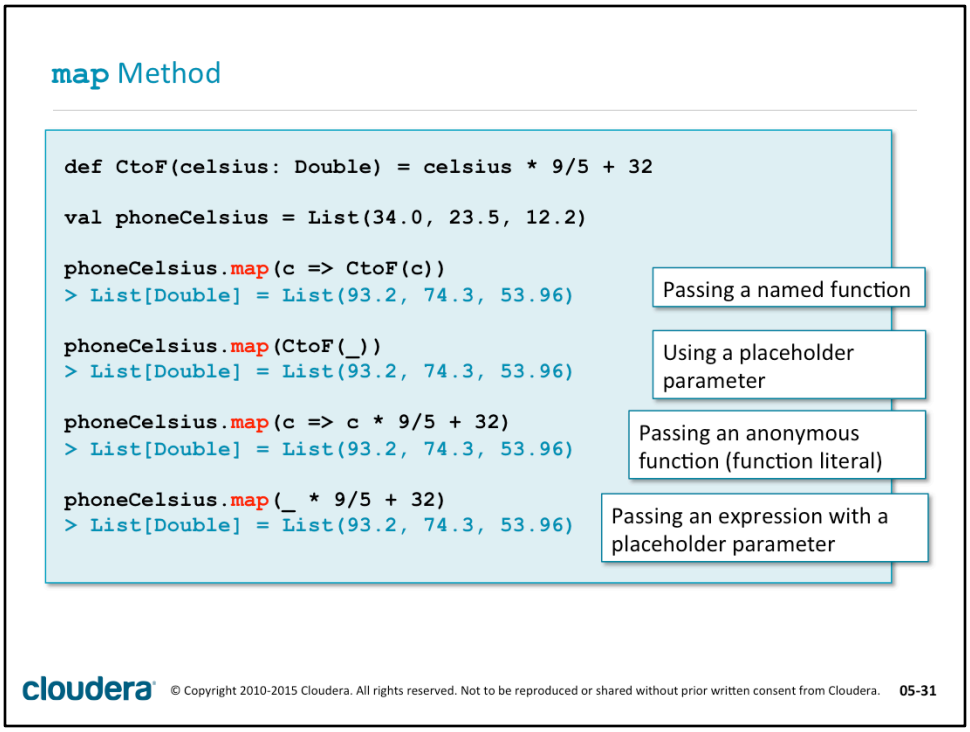
When possible, for scalability, use a Collection's methods and have the *framework perform the iteration* rather than explicitly iterating in the local program



foreach is a higher-order function.
The placeholder value relates foreach to println. foreach is going to iterate over the values in object phones, and each item will be passed to println.
Notice that there is no for loop or while loop here, no iterator. Instead, the looping is implied by the method.

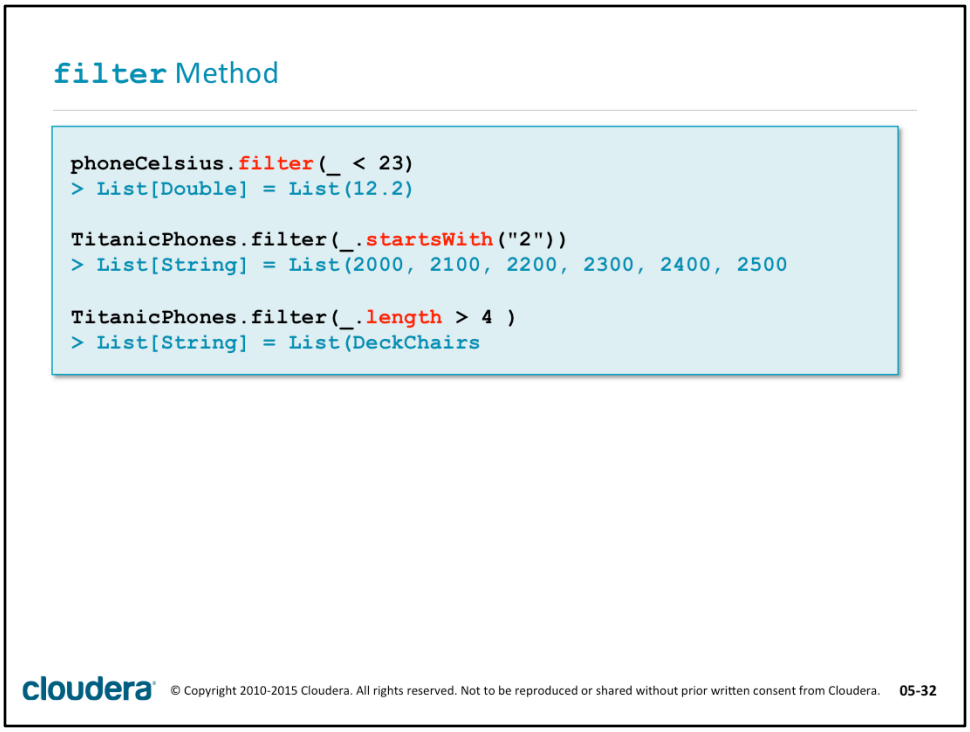
Note that we already saw the use of foreach in previous chapters, using the shortcut syntax of simply passing the name of the function. Scala assumes in that case that the iterated value (e.g. the string in the list) should be passed to the function. Here we see that that the explicit way to do the same thing is to specifically pass "_".

In Scala it is common to use functions (and recursion) to generate iteration or looping behaviors without explicit *while* loops.

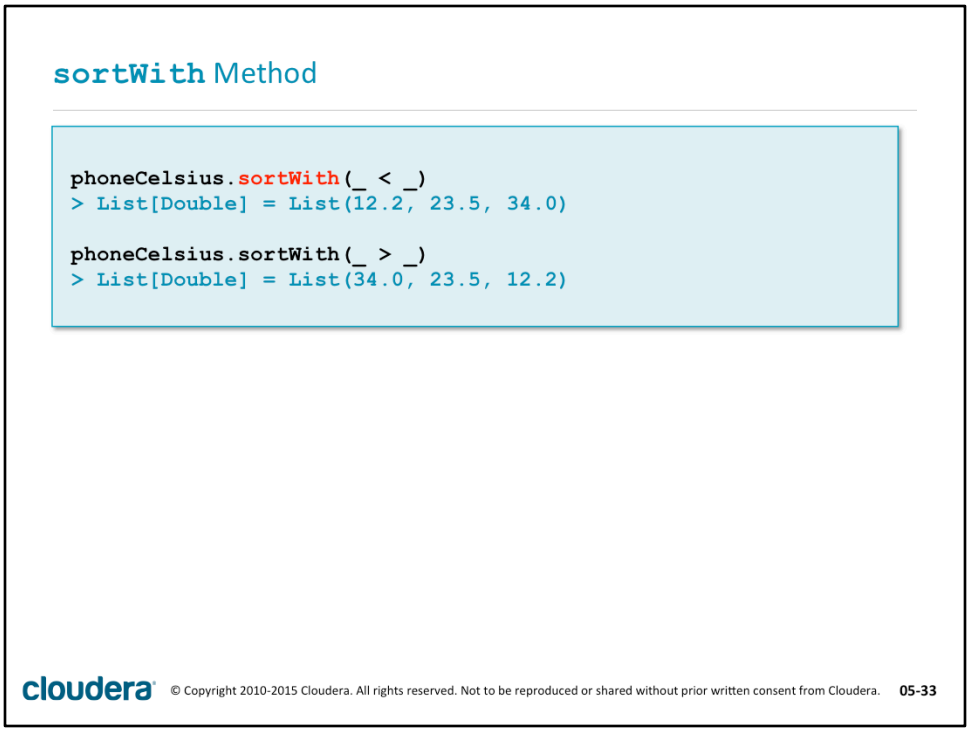


This example shows four different ways to do the same thing, which is to convert a list of Fahrenheit temperatures to a list of Celsius temperatures.
Applying a function or process to every item in a collection to generate a new collection is a common activity. So Scala provides the .map method.

Notice how this last line combines everything we've learned.
The data hierarchy gives us List[Int] as an object with associated methods.
One of those methods is .map
We can pass map an anonymous function that uses a placeholder value to represent each item in the collection.
Looping over the collection is implied.

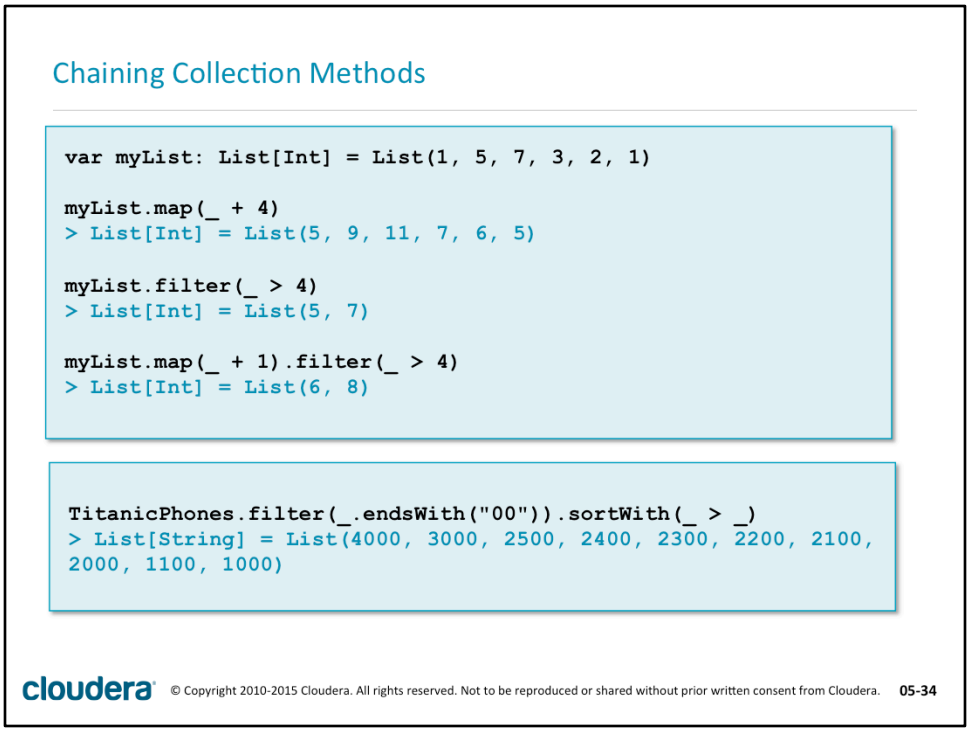


Notice that the placeholder object in this case refers to a String, so we can call string methods like .startsWith and .length on the placeholder.



Note that this example shows *two* placeholder variables. the sortWith uses the passed in function to compare two elements, so the anonymous function you specify should take two parameters. The first _ refers to the first parameter, the second to the second parameter (and so on, with any number of parameters.)

Note that this can be confusing. What if the expression needed to use the first parameter twice, for instance? You would not be able to use the placeholder syntax in that case. You would need to explicitly define an anonymous function with named parameters, and use the first one twice.



What's happening here?
The underscore is introduced as a method for addressing a temporary variables during Traversal.

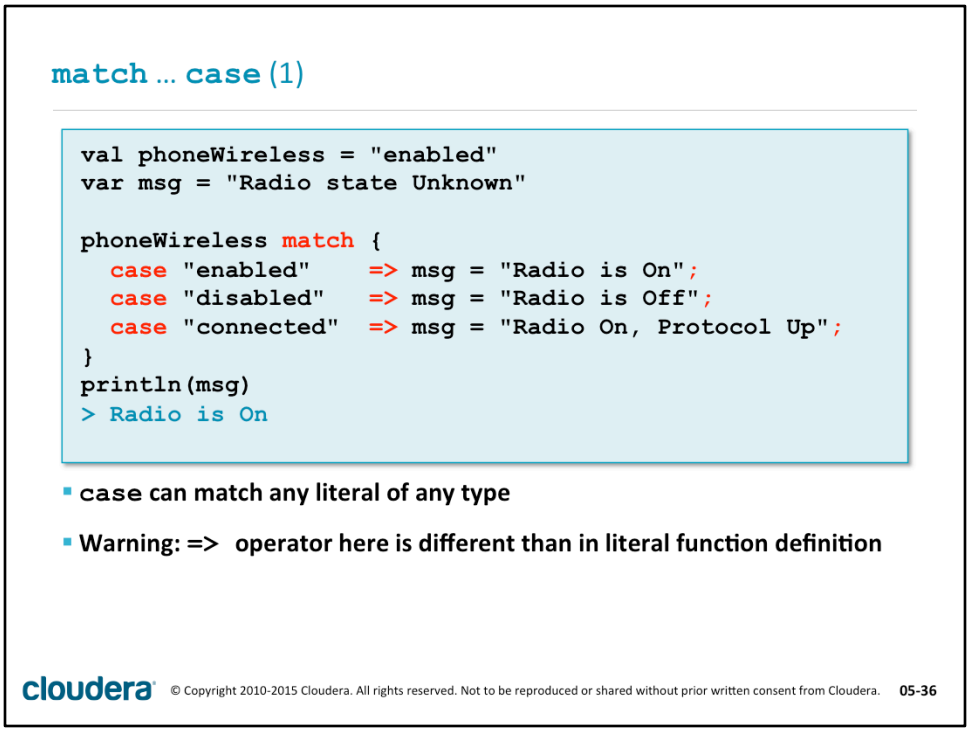
Remember, traversal and iteration are functionally equivalent. But traversal happens within the framework, outside of the program, and the program passes a request to the framework to perform an activity on all the element ins the collection. Iteration, in contrast, returns each element of the collection, and the program is responsible to perform the activity on each element.

So, L1 in the example could be iterated, and the program could be responsible for adding 4 to each value. But this means the program is responsible for scaling and optimizing the action rather than the framework which has visibility into distributed resources and competing resource requests, that the program does not have. So instead, L1.map requests that Scala and its supporting framework (Spark, for example) will process each element in L1 (traverse) applying the function of adding 4 to each element. The underscore '_' represents the temporary variable used by the framework to traverse the collection.

Chapter Topics

Flow Control

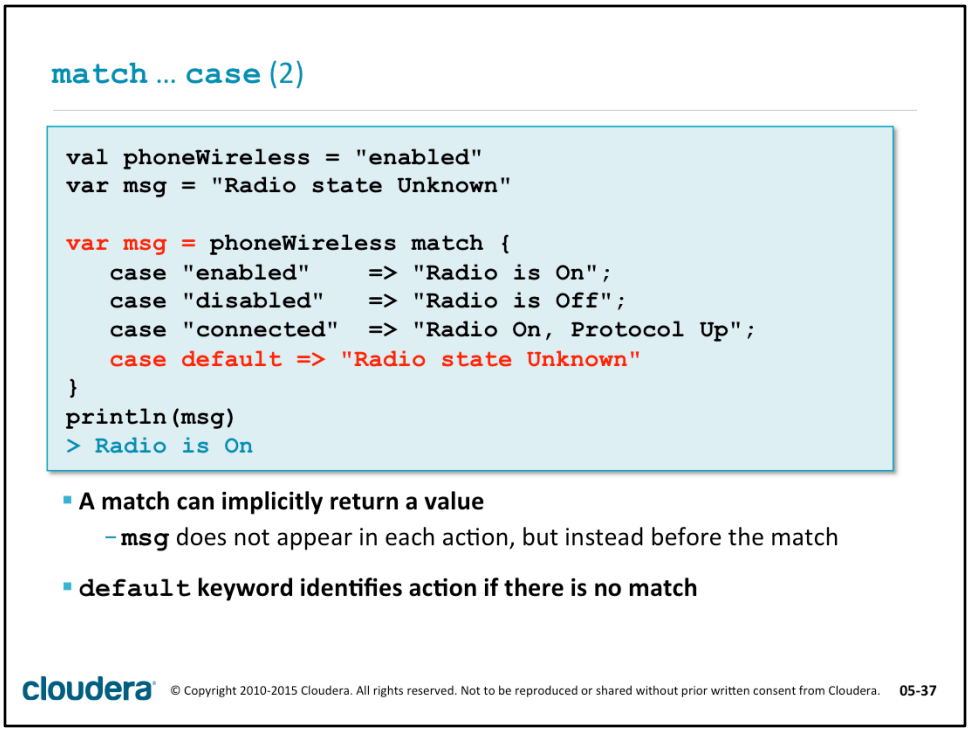
- Looping
- Iterators
- Functions
- Passing Functions
- Collection Iteration Methods
- **Pattern Matching**
- Conclusion
- Hands-On Exercises: Flow Control



Interpreting the phoneWireless state using match…case

Differences compared with match-case in other languages:
1. The name of the variable comes before **match**, instead of the more familiar **match(** variable **)**
2. The case matching literal immediately follows the case keyword, instead of the more familiar **case :** *literal*
3. The matching case is related to the action by '=>'; but this is not the literal function definition.
For example "enabled" => println(_); Should print "enabled" if this was a function literal. Instead, it generates an error. So '=>' is reused here.
4. Each line requires a terminating semicolon.
5. There is no **break** keyword. In other languages, each test and action is processed. After a match, processing continues until a 'break' occurs.
6. In Scala, case can match any kind of literal, not just a number.
7. In Scala, a match can return a value. (See second example)

Instructor Value Add (verify)
- Uppercase matches to Type
- Lowercase is assumed to be an assignment to a new variable
- backticks `y` to refer to an existing variable



Interpreting the phoneWireless state using match…case

Differences compared with match-case in other languages:
1. The name of the variable comes before **match**, instead of the more familiar **match(** variable **)**
2. The case matching literal immediately follows the case keyword, instead of the more familiar **case :** *literal*
3. The matching case is related to the action by '=>'; but this is not the literal function definition.
For example "enabled" => println(_); Should print "enabled" if this was a function literal. Instead, it generates an error. So '=>' is reused here.
4. Each line requires a terminating semicolon.
5. There is no **break** keyword. In other languages, each test and action is processed. After a match, processing continues until a 'break' occurs.
6. In Scala, case can match any kind of literal, not just a number.
7. In Scala, a match can return a value. (See second example)

Instructor Value Add (verify)
- Uppercase matches to Type
- Lowercase is assumed to be an assignment to a new variable
- backticks `y` to refer to an existing variable

Option

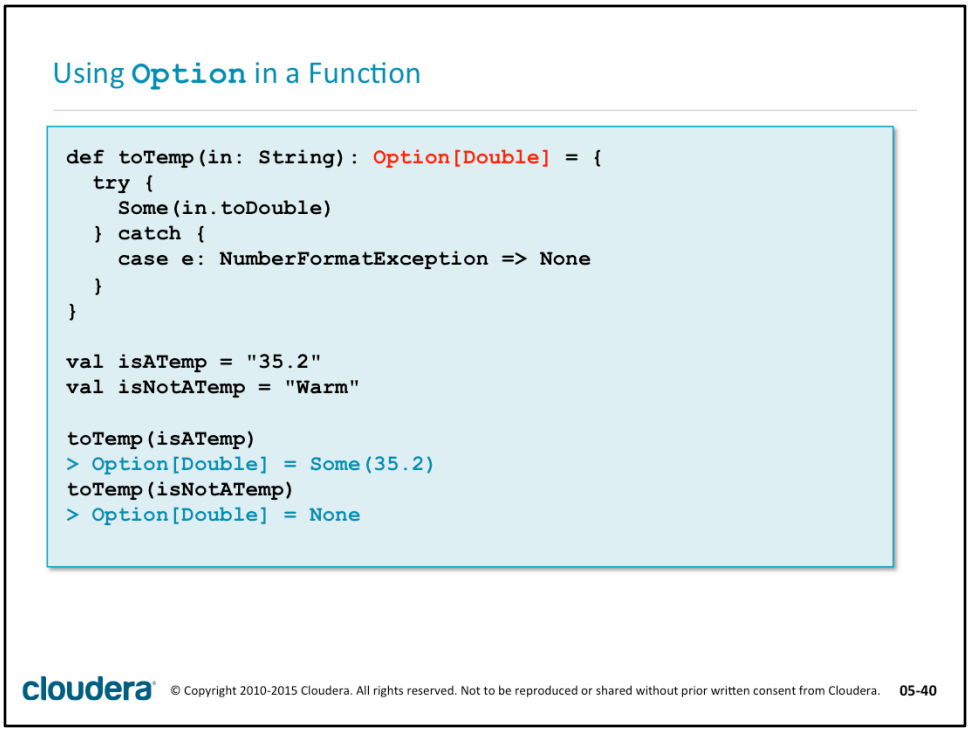
- An `Option` is a special type with a value of `Some(n)` or `None`
- An `Option` can be used to "wrap" a function that would potentially throw an error if it produced an illegal value
- If the value is good, then it is returned wrapped in `Some`
- Some API functions to return an `Option` so that the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

`Some` and `None` classes both extend `Option`.

getOrElse

```
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
```

- **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



The function toTemp converts a String containing a phone Temperature to a Double floating point number.
However, what if the string that is passed is a text string and doesn't contain a numerical Temperature?
In this case, if there is a value, it will be returned wrapped in a "Some". And if the string is not valid, then it will return a None.

`Option` with `match` and `case`

```
toTemp(isATemp) match {
  case Some(i) => println(i)
  case None    => println("Not a Temperature")
}
> 35.2

toTemp(isNotATemp) match {
  case Some(i) => println(i)
  case None    => println("Not a Temperature")
}
> Not A Temperature
```

This is a common use of **`Option`** in APIs. The API will return a value encapsulated in a **`Some`** / **`None`** (**`Option`**) so that the caller can take appropriate action.

Chapter Topics

Flow Control

- Looping
- Iterators
- Functions
- Passing Functions
- Collection Iteration Methods
- Pattern Matching
- **Conclusion**
- Hands-On Exercises: Flow Control

Essential Points

- **Imperative Programming vs Functional Programming techniques**
  - Scala supports IP; You need to know when to use Imperative Programming and when to use Functional Programming techniques
    - *Caveat: Java "intuition" can lead to poor quality Scala code*
  - Looping: Scala provides while and for comprehension and ranges, but, use iterative methods instead for scalability
  - Iterators: Scales to distributed data, doesn't stage in memory – but accesses data when it is used
  - Functions: Delayed evaluation, assignment of functions to variables – Scala supports Higher Order Functions (HOFs)
  - Function Literals: technique for coding Anonymous Functions (Lambda Functions) in Scala
  - Collection Methods: Common methods in Scala Collections – consider these before using Imperative Programming techniques
  - Pattern Matching: Behaves differently from "switch" in other languages

cloudera 05-43

Using Scala versus Java programming techniques

Coding Scala like it was Java – will work – BUT… that produces poor quality Scala code and *misses* the scalability value/purpose of Scala

Data Example: Scala inherits Java collections. But use Scala's collections instead

Especially true for Flow Control techniques

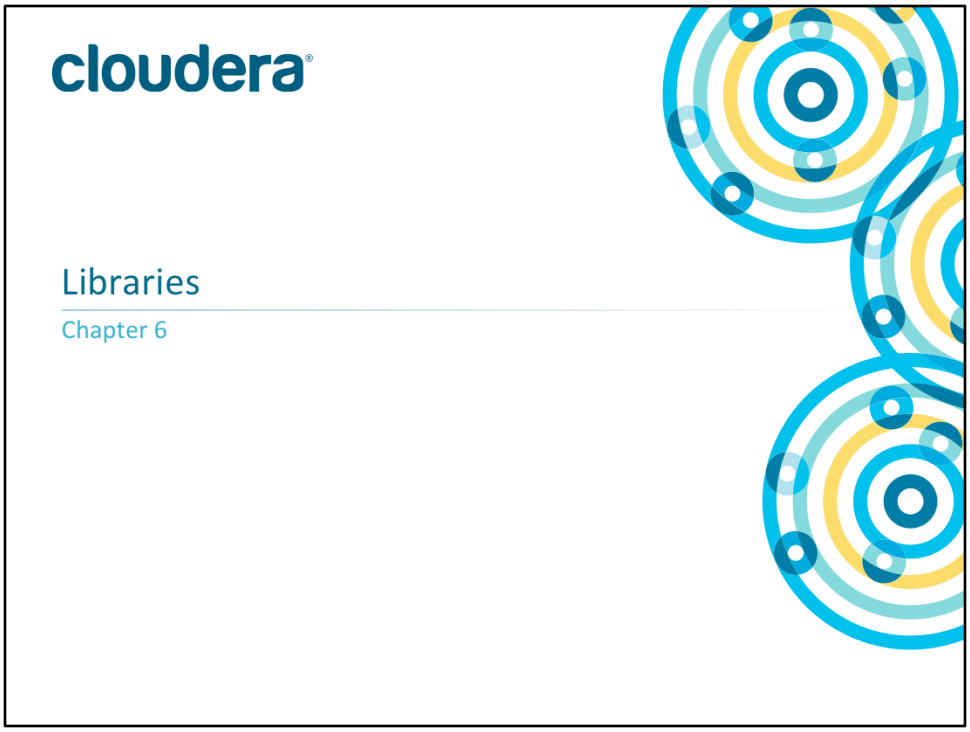
Chapter Topics

Flow Control

- Looping
- Iterators
- Functions
- Passing Functions
- Collection Iteration Methods
- Pattern Matching
- Conclusion
- **Hands-On Exercises: Flow Control**

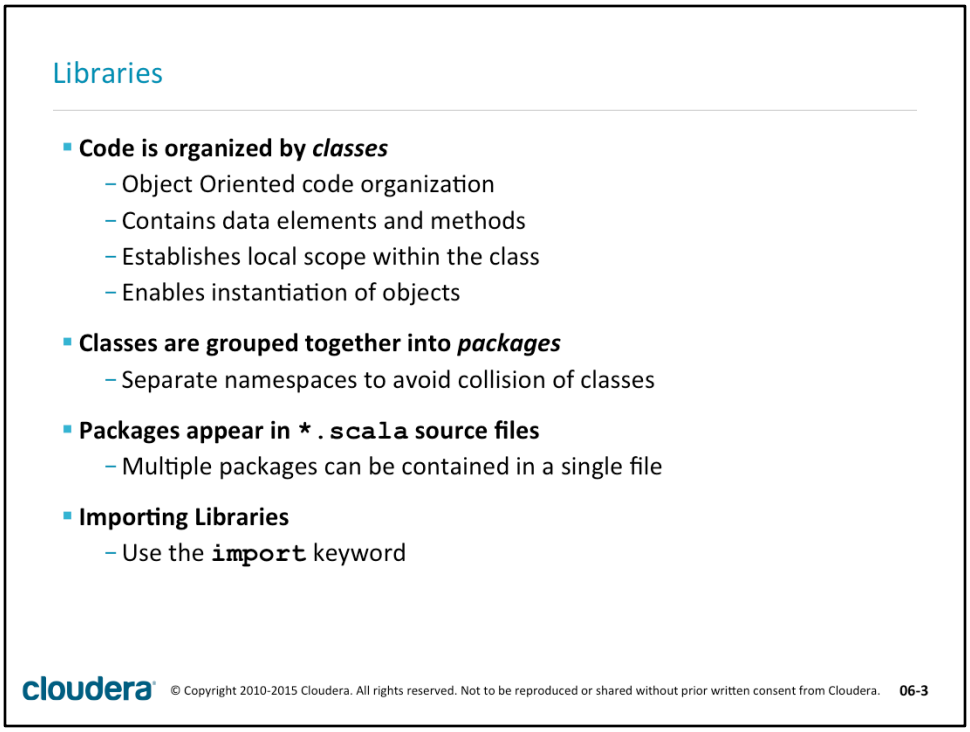
Hands-On Exercise: Flow Control

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



Course Chapters

- Introduction
- Scala Basics
- Variables
- Collections
- Flow Control
- **Libraries**
- Conclusion

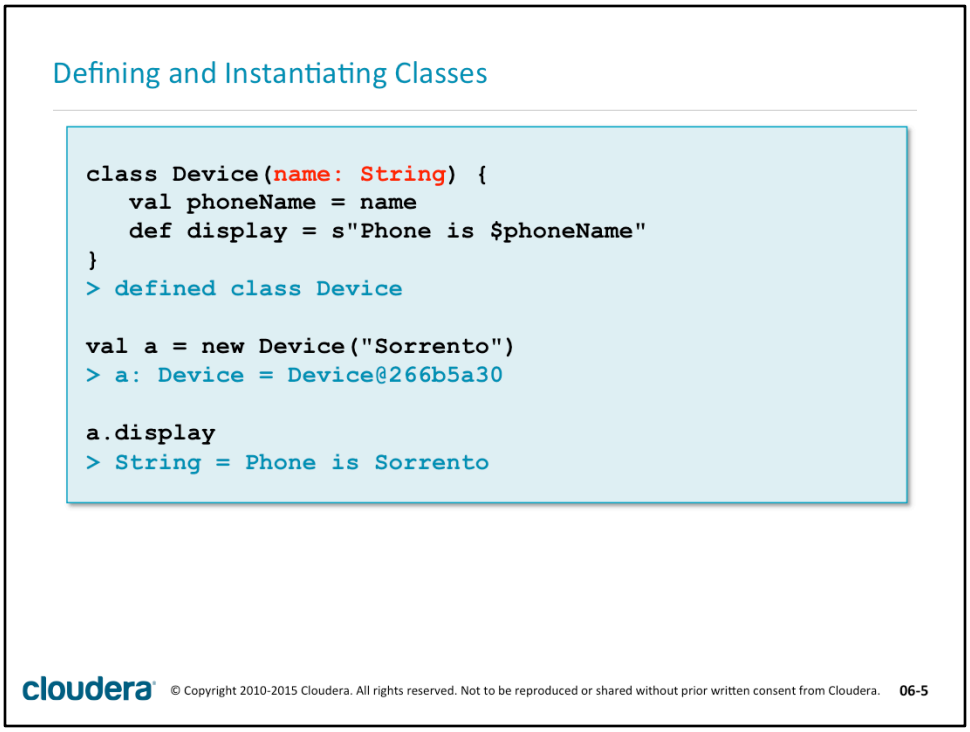


Why do Java (and Scala) us reverse domain names?
1) It ensures that there will not be collisions between namespaces when code from multiple organizations is used together.
2) Reversing the domain name creates a general-to-specific hierarchy within the name that continues into the file structure and then the packages within the file.
So it streamlines the algorithm that needs to access the libraries, because it can be written from general to specific.
3) Easier parsing and DNS lookup

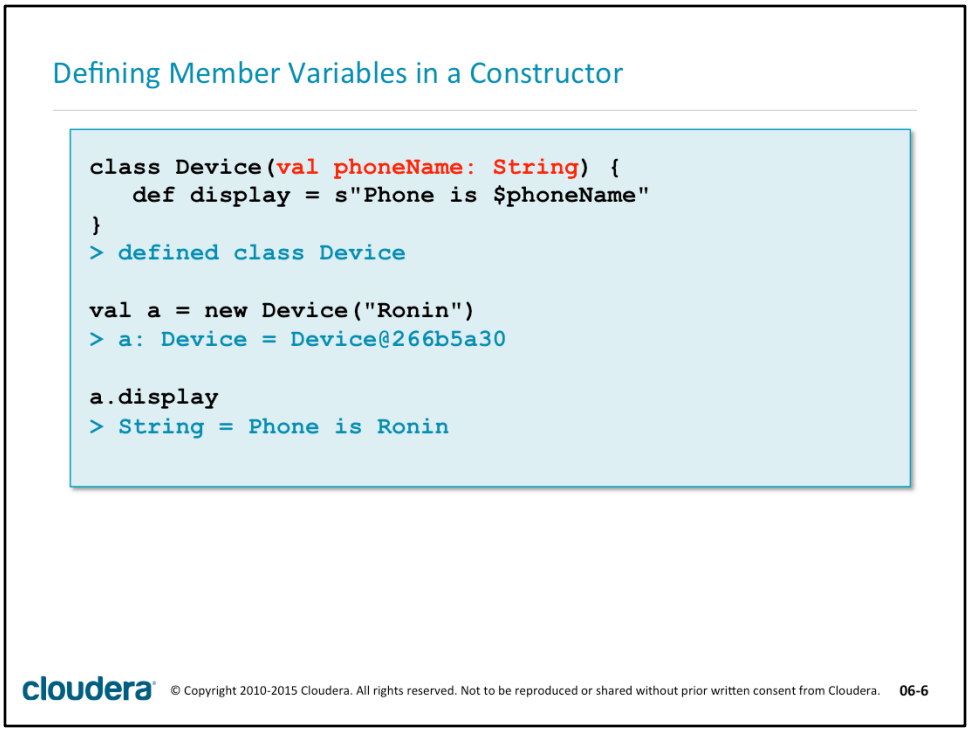
Chapter Topics

Libraries

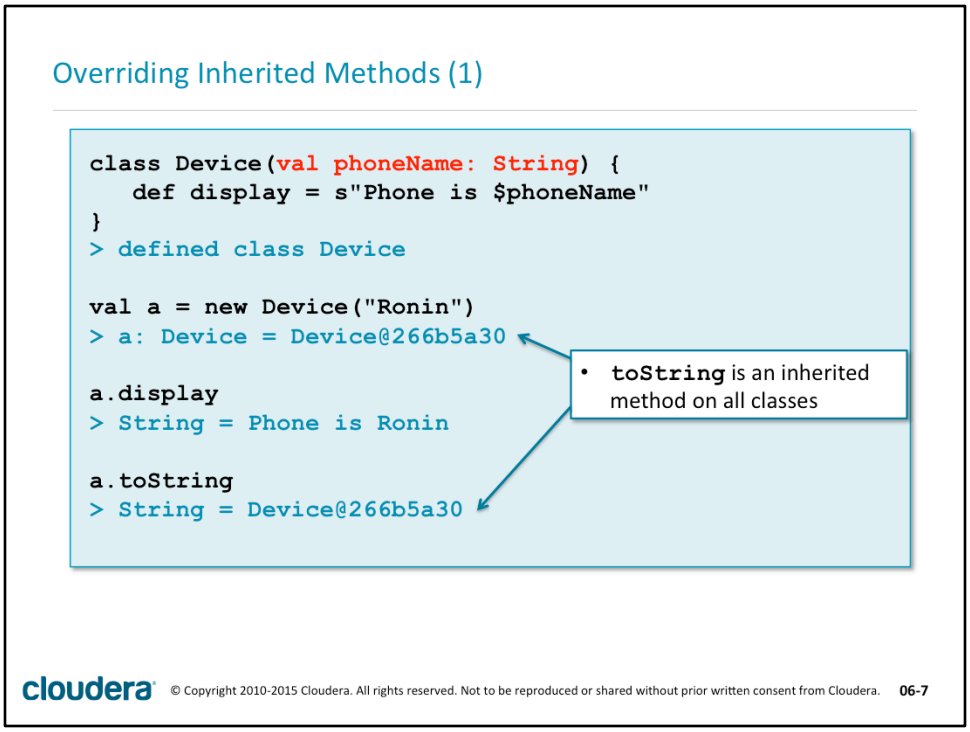
- **Classes and Objects**
- Packages
- Import
- Conclusion
- Hands-On Exercises: Libraries



In this example, 'name' is an input parameter. It only has scope while the object is being constructed. After it is instantiated, name will no longer be addressable. So in this example, name is reassigned to val phoneName, which is part of the object and has persistent addressability.



In this case, we are using phoneName with the val keyword in the parameter list. This is a shortcut to get Scala to store the value directly into that variable.

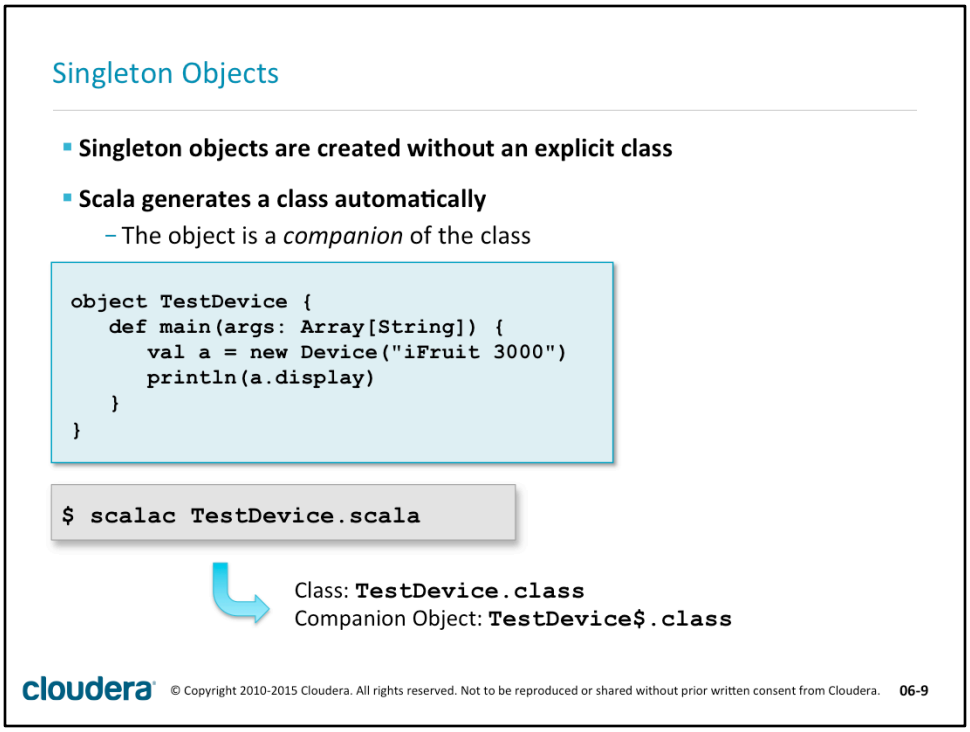


As in Java, toString is a method on all classes.

In Scala, toString is used, among other places, to display an identifier for an object when the object is instantiated in the shell. As with Java, the default toString implementation isn't very helpful, so how do we override it?

Overriding Inherited Methods (2)

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

val b = new Device("Titanic")
> b: Device = Cool
```

"companion object refers to any object with the same name as a class. These must be defined in the same file, if the class is defined explicitly instead of implicitly as shown in this example.

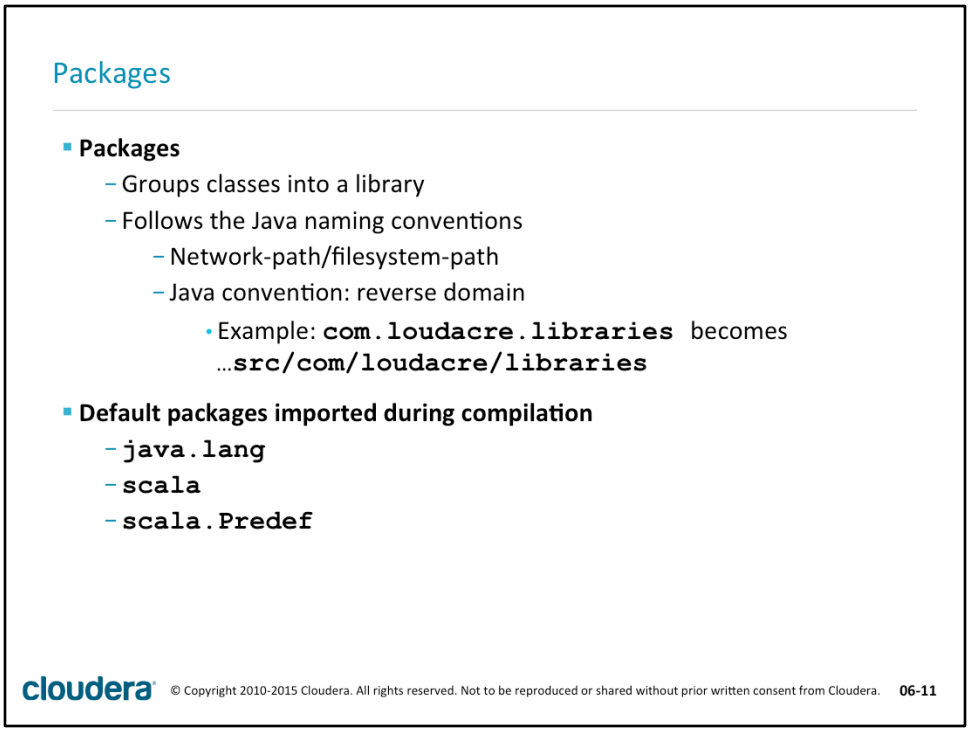
Note that objects can extend classes. For example, an object can add or override a class's method.

Note: static is not a keyword in Scala. Instead, all members that would be static in Java, including classes, should go in a singleton object instead. They can be referred to with the same syntax, imported piecemeal or as a group, and so on.

Chapter Topics

Libraries

- Classes and Objects
- **Packages**
- Import
- Conclusion
- Hands-On Exercises: Libraries



Instructor Value Add

Why do Java (and Scala) us reverse domain names?
1) It ensures that there will not be collisions between namespaces when code from multiple organizations is used together.
2) Reversing the domain name creates a general-to-specific hierarchy within the name that continues into the file structure and then the packages within the file.
So it streamlines the algorithm that needs to access the libraries, because it can be written from general to specific.
3) Easier parsing and DNS lookup

Creating 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
```

File: **com/loudacre/phonelib/Device.class**

Importing a Simple Package

File: **TestDevice.scala**

```
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)
   }
}
```

```
$ scalac TestDevice.scala
```

Files: **com/cloudera/training/TestDevice.class**
com/cloudera/training/TestDevice$.class

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
```

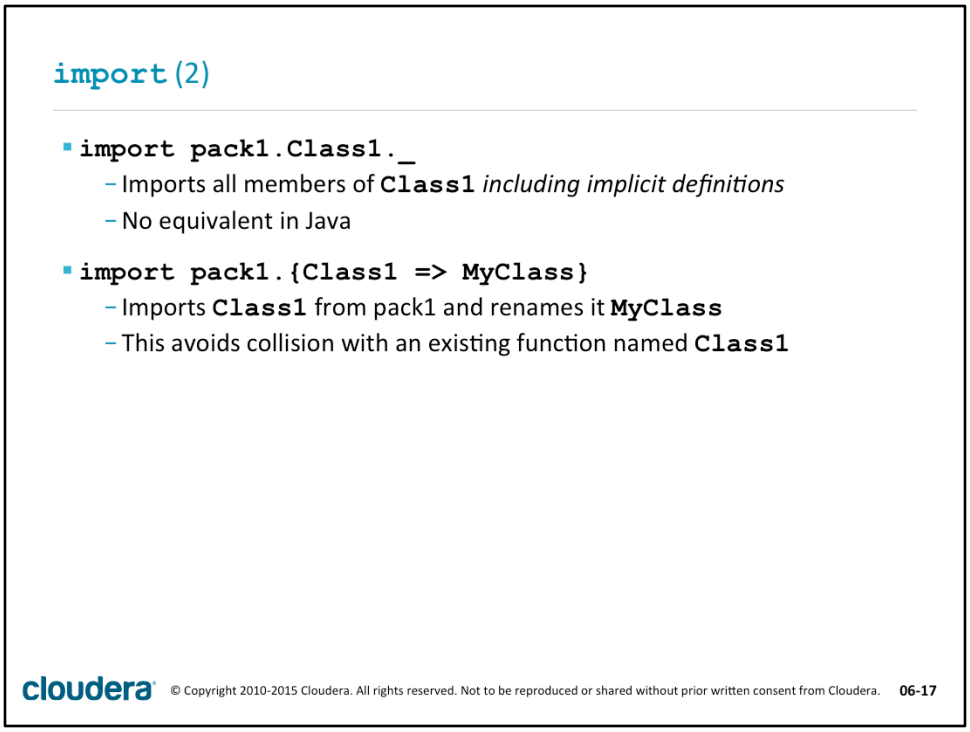

Chapter Topics

Libraries

- Classes and Objects
- Packages
- **Import**
- Conclusion
- Hands-On Exercises: Libraries

`import` (1)

- **`import pack1._`**
  - Imports all classes from **`pack1`**
  - Equivalent to Java **`import *.pack1`**
- **`import pack1._, pack2._`**
  - Imports all classes from **`pack1`** and from **`pack2`**
- **`import pack1.Class1`**
  - Imports only **`Class1`** from **`pack1`**
- **`import pack1.(Class1, Class3}`**
  - Imports only **`Class1`** and **`Class3`** from **`pack1`**



This syntax is particularly important in Spark:
`import pack1.Class1._`

Because the essential Spark libraries are imported like:
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

Students ask what the second one is for, what does it do that the first does not. The answer is that the second imports implicit member definitions for the class, which are used to do automatic/implicit conversions. Without it, class cast errors abound.

Chapter Topics

Libraries

- Classes and Objects
- Packages
- Import
- **Conclusion**
- Hands-On Exercises: Libraries

Essential Points

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

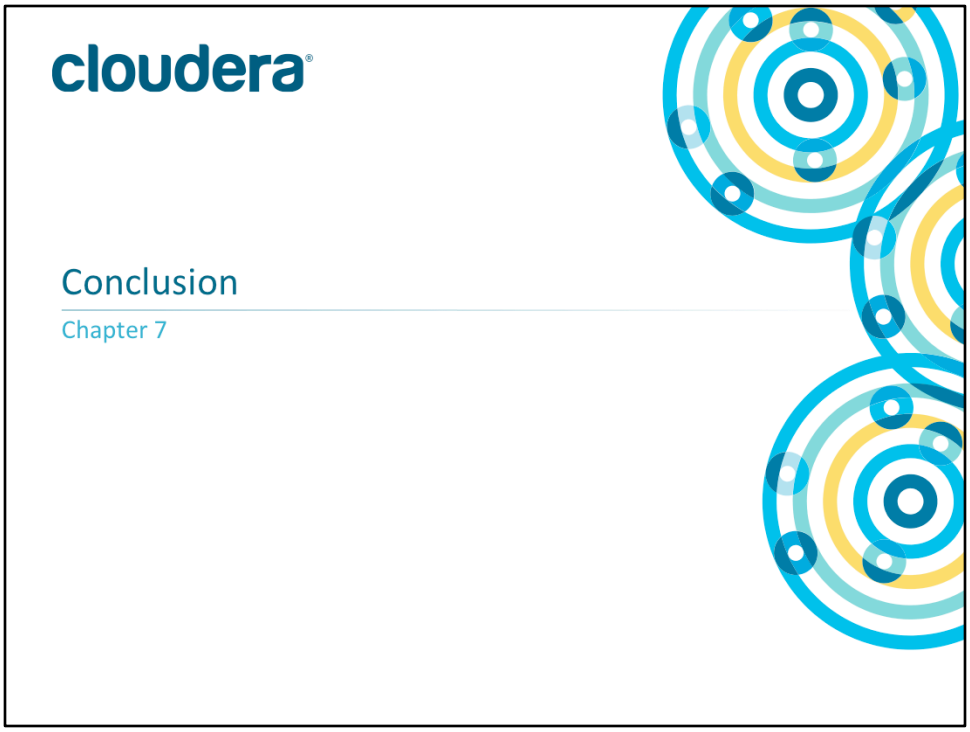
Chapter Topics

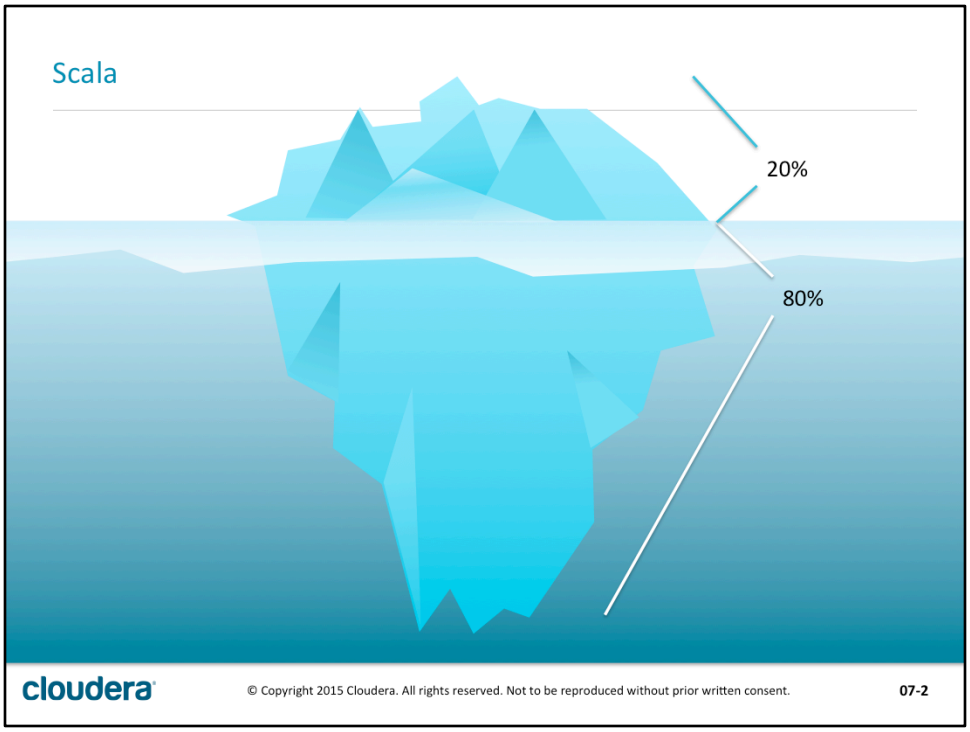
Libraries

- Classes and Objects
- Packages
- Import
- Conclusion
- **Hands-On Exercises: Libraries**

Hands-On Exercise: Libraries

- **In this exercise, you will create a library, import it, and call it from a main program**
  - Please refer to the Hands-On Exercise Manual for instructions





This diagram should now have more meaning.
We spent the class learning how to work with the 20% of the surface to make the 80% work or us.
An when you add Spark or Yarn underneath – the leverage of Scala grows LARGE.

Chapters

- Introduction
- Scala Basics
- Variables
- Collections
- Flow Control
- Libraries
- **Conclusion**

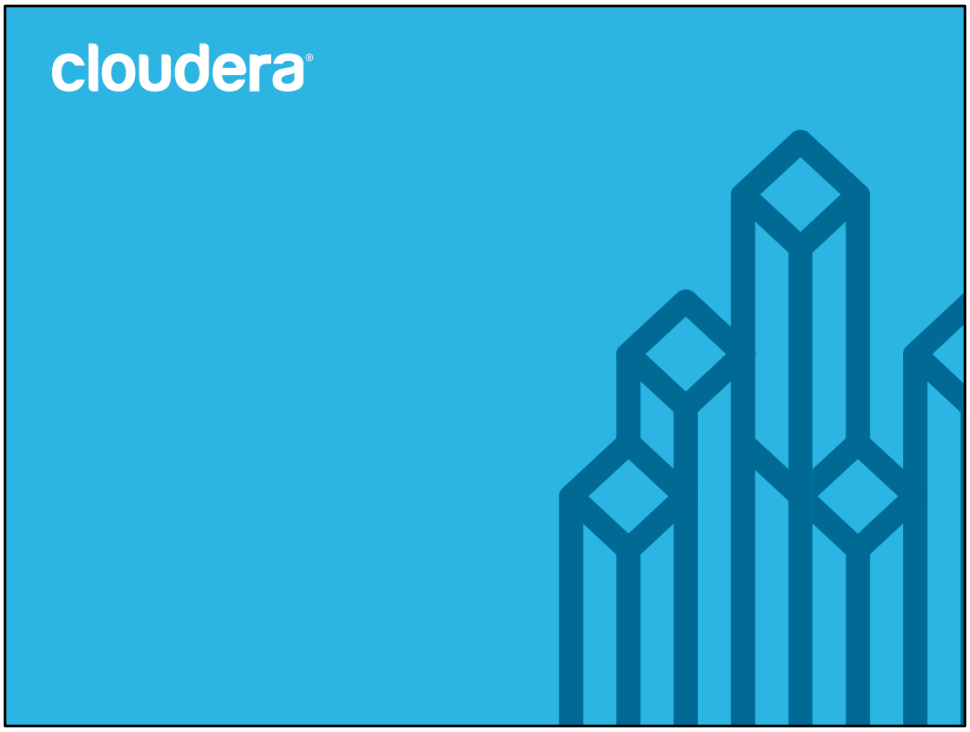
cloudera

Course Objectives

In this course you learned

- **What Scala is and how it differs from other languages such as Java or Python**
- **Key Scala concepts such as data types, collections and program flow control**
- **How to implement both imperative and functional programming solutions in Scala**
- **How to work with Scala classes, packages and APIs**

cloudera



Just Enough Scala: Hands-On Exercises

Instructor Guide

General Notes

Points to note while working in the VM

1. The VM is set to automatically log in as the user `training`. If you log out, you can log back in as the user `training` with the password `training`.

2. If you need it, the root password is `training`. You may be prompted for this if, for example, you want to change the keyboard layout. In general, you should not need this password since the `training` user has unlimited `sudo` privileges.

3. In some command-line steps in the exercises, you will see lines like this:

```
$ hdfs dfs -put united_states_census_data_2010  \
/user/training/example
```

   The dollar sign ($) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (for example, `[training@localhost training_materials]$`) but this is omitted from these instructions for brevity.

   The backslash (`\`) at the end of the first line signifies that the command is not completed, and continues on the next line. You can enter the code exactly as shown (on two lines), or you can enter it on a single line. If you do the latter, you should *not* type in the backslash.

4. Although most students are comfortable using UNIX text editors like vi or emacs, some might prefer a graphical text editor. To invoke the graphical editor from the command line, type `gedit` followed by the path of the file you wish to edit. Appending `&` to the command allows you to type additional commands while the editor is still open. Here is an example of how to edit a file named `myfile.txt`:

```
$ gedit myfile.txt &
```

Directories, Files and Data used in the exercises

1. All files and data used in the course are in the course directory:

    - `/home/training/training_materials/jes`

2. Under the course directory you will find the following directories:

    - `data` – data files used in the exercises or course example code
    - `examples` – programs and Scala shell code snippets used in the course presentation
    - `solutions` – programs (.scala) and Scala shell code snippets (.scala.txt) with solutions to the exercises in this Manal

Working in the Scala Shell

3. Scala ships with an interactive interpreter called the Scala shell. It is an REPL (Read-Evaluate-Print Loop) interpreter. Most of the exercises in this guide can be completed from the Scala shell without editing external files.

4. Start the Scala shell from the command line by typing the command `scala`

5. To see a list of commands that are directed to the Scala shell, type `:help`. In general, commands directed to the Scala shell begin with a colon.

6. The Scala shell provides command completion and command hints when the **`[TAB]`** key is pressed.

7. To quit the shell, enter `:quit`.

8. For code exercises that rely on complex text strings a separate `.txt` file has been provided so that you can copy and paste the code without formatting concerns.

9. If you wish, you can usually copy code and strings from this Hands-On Exercises manual and paste them into your terminal sessions.

Working with Scala Program Files

10. Exercises can also be performed by entering the code into a program file. The program is then compiled and run. Some exercises may require programming in files rather than using the interactive shell. (For example, the exercises associated with *libraries*).

11. Files with the suffix `.scala` are Scala program files.

12. To create and edit `.scala` files you can use the graphical editor (gedit) installed on the class exercise machine, or you can use a variation of the vi text editor if preferred. (Other common text editors are also installed and available.)

13. To compile `.scala` files use the `scalac` command in a terminal window:

```
$ scalac filename.scala
```

    That will generate a directory structure containing `.class` files.

14. Run the program using the `scala` command in a terminal window:

```
$ scala ClassName
```

15. You cannot edit a file without leaving the Scala shell. Rather than repeatedly entering and exiting the shell, it is useful to have one terminal window open for the Scala shell and another terminal window open to the native OS shell for programming, compiling, and execution.

16. You can load and run a program from a file within the Scala shell without leaving the shell by using `:load <filename>`

Hands-On Exercise: Using Scala

Files and Data Used in this Exercise

Solutions: `Shell-exploration.scala.txt`
`HelloScala.scala`

Data file:
`/home/training/training_materials/jes/data/loudacre.log`

In this exercise you will explore the course training materials, and then start the Scala shell and use it interactively. You will also write, compile and run a Scala program and practice with basic Scala file I/O.

Exploring the Course Materials

17. Open a terminal window.

18. Change to the directory containing the exercise files. Use the Unix `ls` command to examine the contents of this directory and its subdirectories. All of the files you need for the exercises are here. Also, there are solutions to all of the exercises that you can view if you need help.

```
$ cd ~/training_materials/jes/
$ ls
$ ls data
$ ls examples
$ ls solutions
```

Working in the Scala Shell

19. Open a new terminal window and enter:

```
$ scala
```

The Scala shell is an REPL (Read-Evaluate-Print Loop) shell. It provides a method for having an interactive session with Scala. You will use this to explore some of the basic principles of the Scala language. The Scala shell also provides *directive* commands. A directive command is a command preceded by the colon `:command` that causes the Scala shell to process the command. For example, `:quit` will cause the Scala shell to exit.

20. Create an immutable variable called `ab` and assign the integer value `25` to it:

```
scala> val ab = 25
ab: Int = 25
```

Note that the name, type and value of the variable are displayed as confirmation.

21. Enter the variable as a command:

```
scala> ab
res0: Int = 25
```

Note two things: first, that a variable entered without any command returns its value, because all commands in Scala are expressions that return a value; and that the return value is assigned to an automatically created variable such as `res0`.

22. Use the built-in `print` function to display the value of ab:

```
scala> print(ab)
25
```

23. Try entering a string:

```
scala> "Hello, Scala"
res2: String = Hello, Scala
```

Note that entering a literal like this also returns a value, and assigns that value to a result variable. (The number of the res variable will depend on how many result variables Scala has created in this session.)

24. Try entering an invalid command, such the print function without the final parenthesis:

```
scala> print(ab
     |
```

25. Note the indented vertical bar prompt. This means Scala detected that the command is not yet complete, and you can continue typing. Or, if you want to cancel entry of the command, hit ENTER twice:

```
scala> print(ab
     |
     |
You typed two blank lines.  Starting a new command.
```

26. Finally, try referencing an invalid variable:

```
scala> print(nonesuch)
<console>:8: error: not found: value nonesuch
              print(nonesuch)
                    ^
```

Note that the caret (^) indicates the point in the entered code where the error was encountered. This can help you determine which part of a complex line of code is causing the syntax error.

Writing, Compiling and Running a Scala Program

27. Using an editor of your choice, create a new file called `HelloScala.scala` in the course exercise directory: `~/training_materials/jes/`.

    - **Tip:** If you are not familiar with any Linux text editors, consider using **gedit**, a simple graphical text editor, which you can start by using the desktop icon, or by entering `gedit &` on the terminal command line.

28. Enter the following code:

```
object HelloScala {
  def main(args: Array[String]) {
    println("Hello, Scala!")
  }
}
```

29. Save the program file `HelloScala.scala`.

30. In a terminal window, change to the course exercise directory where you created the new Scala program file.

31. Compile the program using the `scalac` compiler:

```
$ scalac HelloScala.scala
```

    Notice that compilation of the source file produces a `.class` file with the name of the object.

32. Run the program by specifying the main object name:

```
$ scala HelloScala
```

Introducing Loudacre

Loudacre is a (fictional) mobile phone carrier. In many of the exercises in this course you will use Loudacre device status data from mobile phones.

Whenever a phone has to do a soft or hard restart, it gathers error information and sends it back to Loudacre's central facility where the data is collected for analysis.

Start by examining the log file `~/training_materials/jes/data/loudacre.log` using an editor of your choice.

This is a typical line of data from the data file:

```
2014-03-15:10:10:31,Titanic 4000,
1882b564-c7e0-4315-aa24-228c0155ee1b,58,36,39,31,15,
0,TRUE,enabled,enabled,40.69206648,-119.4216429
```

Here are the data fields shown in that line:

- Date Time: `2014-03-15:10:10:31`
- Model name and number: `Titanic 4000`
- Unique Device ID: `1882b564-c7e0-4315-aa24-228c0155ee1b`
- Device temperature (Celsius): `58`
- Ambient temperature (Celsius): `36`
- Battery available (percent): `39`
- Signal strength (percent): `31`
- CPU utilization (percent): `15`
- RAM memory usage (percent): `0`
- GPS Status (enabled=TRUE/disabled=FALSE): `TRUE`
- Bluetooth status (enabled/disabled/connected): `enabled`
- WiFi status (enabled/disabled/connected): `enabled`
- Latitude: `40.69206648`

- Longitude: `-119.4216429`

Reading and Displaying a Data File

33. Enter the following program in the Scala shell.

```
scala> import scala.io.Source
scala> Source.
fromFile("/home/training/training_materials/jes/data/lo
udacre.log").
foreach(print)
```

Hands-On Exercise: Variables

Files Used in this Exercise

Solutions:

```
Mutability-solved.scala.txt
Numeric-solved.scala.txt
Operators-solved.scala.txt
AdvMath-solved.scala.txt
Strings-solved.scala.txt
Boolean-solved.scala.txt
```

Exploring Mutability, Reassignment and Redefinition of Variables

34. Try creating an immutable variable and reassigning it. What happens?

```
scala> val ambientTemp = 19
scala> ambientTemp = 47.0
```

35. Try redefining the variable instead:

```
scala> val ambientTemp = 27.0
```

36. Try creating a mutable variable with an inferred type. What happens when you try to change the value to a new value of the same type? What about a value of a different type?

```
scala> var deviceTemp = 21
scala> deviceTemp = 35.2
scala> deviceTemp = 45
```

37. Try redefining the variable using a different inferred type:

```
scala> var deviceTemp = 33.1
```

38. Define some different variables with an explicitly declared type:

```
scala> val phoneModel: String = "Sorrento"
scala> val temp: Int = 30
scala> val location: Double = 33.1765
```

The exploration is recorded as a text file in `Mutability-solved.scala.txt`.

Working with Numeric Variables

39. Try the following explorations in converting Celsius temperatures to Fahrenheit temperatures using the Scala shell to learn how Scala works with integer and floating point variables.

```
scala> val tempC = 27
scala> val c_to_f = 9/5
scala> val tempF = tempC * c_to_f + 32
```

Note that the variables and final result are all integers.

40. Trying repeating the calculation using floating point numbers instead:

```
scala> val c_to_f = 9.0/5.0
scala> val tempF = tempC * c_to_f + 32
```

41. You can find the type of the resulting object using `getClass`:

```
scala> println( tempC.getClass, tempF.getClass )
```

The exploration is recorded as a text file in `Numerical-solved.scala.txt`

Using Scala Operators

Loudacre reports CPU utilization in percentages. Use Scala interactively to explore operators.

42. Calculate the average of two CPU utilization percentages:

```
scala> val cpuT1 = 17
scala> val cpuT2 = 38
scala> val averageCPU = (cpuT1 + cpuT2) / 2
```

43. The calculation returned integer results. Try assigning the variable type explicitly and note the implicit conversion:

```
scala> val averageCPU: Double = (cpuT1 + cpuT2) / 2
```

44. Print the result of the calculation. Note the implicit conversion of Double to String using the `String` + concatenation operator.

```
scala> println("Average CPU: " + averageCPU)
```

45. New software could reduce CPU utilization by 12%. Using in-line operations, display the expected new CPU results after the improvement has been implemented.

```
scala> val reduction = 12/100
scala> println("Expected CPU: "+(cpuT2-
(cpuT2*reduction)))
```

46. Increment CPU utilization at time 1 by 1% using the increment operator +=. What happens?

```
scala> cpuT1 += 1
```

47. How would you fix this problem? (If necessary, review the solution.)

48. Explore casting between different variable types using conversion methods.

```
scala> val cpuT1: Int = 35
scala> val cpuT2: Double = 37.23
scala> cpuT1.toDouble
scala> cpuT2.toInt
scala> cpuT1.toString + cpuT2.toString
```

49. What other methods are available on the Int type? Enter `cpuT1.` and press `[TAB]` to see a list.

The exploration is recorded in `Operators-solved.scala.txt`.

Importing and Using Advanced Mathematical Functions

50. Use Scala interactively to explore the math functions.

51. Math functions that are integrated in many other languages are in separate built-in libraries in Scala. Try using the `sqrt` function.

```
scala> sqrt(192)
```

52. The library must be imported prior to use:

```
scala> import scala.Math
scala> Math.sqrt(192)
```

The completed program is in `AdvMath-solved.scala.txt`.

Working with Strings

53. Use the following single string to represent one record in the `loudacre.log` file.

    - Note: The record string is available to cut-and-paste in `data/sample_one_record.txt`

```
scala> val record="2014-03-15:10:10:31,Titanic
4000,1882b564-c7e0-4315-aa24-
228c0155ee1b,58,36,39,31,15,0,TRUE,enabled,enabled,40.6
9206648,-119.4216429"
```

54. Print out the record and the length of the record. The length should be 132 characters.

55. Use the `contains` method to determine whether the model name (`Titanic`) is in the record.

56. Use `indexOf` to locate the index of the first character of Titanic.

57. Use the `substring` method to assign the model name to a new variable called `name`.

58. Use the `toUpperCase` method to turn the name into uppercase.

The completed program is in `Strings-solved.scala.txt`.

Booleans

59. Extract the Bluetooth and Wifi fields from the record string.

    Each field can contain three values: `enabled`, `disabled`, and `connected`. Find the start of the first field with `indexOf`, then use `substring` to extract the string into new `val bluetooth`. Use `indexOf` with the position starting parameter to find the next instance and extract it into new `val wifi`.

60. Use == to determine if Bluetooth and Wifi are ON and print the results.

61. Use == to determine if Wifi is connected to an access point.

The completed program is in `Boolean-solved.scala.txt`.

Optional: Viewing the Scala API Documentation

62. Start Firefox using the icon in your virtual machine's top menu.

63. Click the **Scala API** bookmark, or open local file URI:
    `file:///usr/share/scala-docs/api/scala-library/index.html`

64. On the left is a list of packages and the classes they contain. The first package displayed is the main `scala` package, containing the default set of types. Select one or more of the types covered in this chapter, such as `Any`, `Double` or `Boolean`, and review the API.

This is the end of the exercise.

Hands-On Exercise: Collections

Files Used in this Exercise

Solutions: `TuplestoLists-solved.scala.txt`
`ModelToBrand.scala`

In this exercise you will practice working with various types of Collections and Tuples.

Working with Tuples and Lists

65. Create the following Tuple in the Scala shell. Verify its type with `getClass`.

```
scala> val phoneTuple =
("Titanic","3000","disabled","connected")
scala> phoneTuple.getClass
```

66. Convert the Tuple to a List:

```
scala> val phoneList =
phoneTuple.productIterator.toList
```

67. Using the `[TAB]` key, determine which type – the Tuple or the List – has more methods?

```
scala> phoneTuple.[TAB]
scala> phoneList.[TAB]
```

The completed program is in `TuplestoLists-solved.scala.txt`.

Working with Maps

Loudacre support has received a report containing the model name of the phone in error, but the brand name has been omitted. They requested a command line tool to enable reverse lookup of the brand name based on the model name.

In this section you will create a command line tool that will use a `Map` Collection to look up phone model numbers and return their corresponding manufacturer brand names.

68. Using an editor of your choice, create a file called `ModelToBrand.scala` to define a main object for your program.

    Copy in this stub code to get you started with command line arguments:

```
object ModelToBrand {
  def main(args: Array[String]) {
    if (args.length > 0) {
      println("Model name entered: " + args(0))
    }
  }
}
```

69. The `Map` should account for the following relationships:

    iFruit brand: Models 1, 2, 3, 3A, 4A, 5

    Ronin brand: Models S1, S2, S3

    Sorrento brand: Models F01L, F11L, F21L, F23L, F33L, F41L

70. If the model is not recognized, print out "`Record not found`".

71. If the program is called with no input arguments (that is, if the length of the args array is 0), then print out a list of keys (models) and a list of values (brands).

    - **Hint**: Generate a list of the `Map`'s values and convert it to a `Set`. Because the `Set` Collection ensures unique values, it will automatically remove the duplicates.

The completed program is in `ModelToBrand.scala`.

Optional: Viewing the Collections Hierarchies in the Scala API Documentation

72. In Firefox, Click the **Scala API** bookmark, or open local file URI: `file:///usr/share/scala-docs/api/scala-library/index.html`

73. On the left, scroll down to the `scala.collection.immutable` package. (Hint: click **display packages only** to view just the packages; click **display all entities** to return to a view of all packages and their contents.)

74. Review the API for some of the collection types discussed in this chapter, such as `List` or `Map`.

This is the end of the Exercise.

Hands-On Exercise: Flow Control

Files and Data Used in this Exercise

Solutions:
- IterationExploration-solved.scala.txt
- ExploreIterator-solved.scala.txt
- FileIterator-solved.scala.txt
- Split-exploration.scala.txt
- ParseLines1.scala
- ParseLines2.scala
- FilterLines.scala

Data file:

/home/training/training_materials/jes/data/loudacre.log

In this exercise, you will explore flow control and program structure in Scala.

Comparing Approaches to Iteration

In this section, you will process a collection of floating point number. The numbers represent phone charge percentages, and you will convert them to milliamps per house (mAh).

You will repeat the same task using different approaches in order to compare them.

75. Start by defining a function to convert percentage to mAh:

```
scala> def mAh(percent: Double) =
(percent/100) * 5400
```

76. Define a list of phone battery charge percentages

```
scala> val phoneBattery = List(82.3, 31.6, 72.5, 64.7)
```

77. Classic iterative approach – use a `for` loop with a range to iterate over the list and calculate and print the results:

```
scala> for(i <- 0 until phoneBattery.length)
println(phoneBattery(i) + " percent = " +
mAh(phoneBattery(i)))
```

78. Iteration with a generator – instead of using a counting variable with a range, use the `List` with the `for` generator:

```
scala> for(percent <- phoneBattery) println(percent + "
percent = " + mAh(percent))
```

79. List comprehension – instead of having the `for` loop print out each value, have it return a new `List` containing the converted values by using `for`/`yield`:

```
scala> val myList = for(percent <- phoneBattery) yield
mAh(percent)
```

80. Now you can loop through the resulting list to display the values any number of ways, such as:

```
scala> myList.foreach(println)
```

81. The `map` method can be used as a convenience method for `for`/`yield`:

```
scala> val myList = phoneBattery.map(percent =>
mAh(percent))
```

82. Try using a placeholder parameter in the above map call as a shortcut:

```
scala> val myList = phoneBattery.map(mAh(_))
```

83. Or as an even more succinct shortcut:

```
scala> val myList = phoneBattery.map(mAh)
```

84. Anonymous function – another option, rather than defining and using the named function mAh, is to pass the conversion function as an anonymous function:

```
scala> val myList = phoneBattery.map(percent => percent
/ 100 * 5400)
```

85. You can also use a placeholder parameter in an anonymous function:

```
scala> val myList = phoneBattery.map(_ / 100 * 5400)
```

This exploration is saved in `IterationExploration-solved.scala.txt`.

Using Iterators

In this section you will be explore a List and an Iterator in the Scala shell.

86. Create a list called `myBrands` with the following values: `iFruit`, `MeToo`, `Titanic`, `Ronin`, `Sorrento`:

```
scala> val myBrands =
List("iFruit","MeToo","Titanic","Ronin","Sorrento")
```

87. Now create an iterator `myIt` that refers to `myBrands`. Does it have items left in the iterator?

```
scala> var myIt = myBrands.toIterator
scala> myIt.hasNext
```

88. Use the following code to step through the iterator twice. What do you think will happen? Will the second time work the same as the first?

```
scala> while(myIt.hasNext) println(myIt.next)
scala> while(myIt.hasNext) println(myIt.next)
```

89. Create a new iterator, then check how many items are in the iterator using `size`. Do you predict this method will consume the iterator or not? Check by calling `hasNext`.

```
scala> myIt = myBrands.toIterator
scala> myIt.size
scala> myIt.hasNext
```

The completed exploration is in `ExploreIterator-solved.scala.txt`.

Using `for` with a File Source Iterator

90. In an earlier exercise, you saw how to read and display the content of a file using code like this:

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

In this exercise you will continue working with the same data file, in different ways.

91. Enter this code:

```
scala> import scala.io.Source
scala> val fname =
"/home/training/training_materials/jes/data/loudacre.log"
scala> var fsource = Source.fromFile(fname)
```

92. What type is `fsource`? Use `getClass` to find out.

93. Use a `for` loop to display the first 500 characters of the file:

```
scala> for (x <- 1 to 500) print(fsource.next)
```

94. Try repeating the 500 count for loop again. Is the output the same as the first time?

95. `fsource` is a character-based iterator over the file. Use the `getLines` method to return a *line*-based iterator:

```
scala> var flines = Source.fromFile(fname).getLines
```

96. Print out the first line of the file:

```
scala> print(flines.next)
```

97. Use `flines` to display the remaining lines in the file.

    - Hint: There are two ways to do this. Either is equally valid.

98. What happens if you try to call `flines.next` now that you have already iterated through the whole file?

The solution is `FileIterator-solved.scala.txt`.

Parsing String Input Using `split`

The next task is to split the string that is returned by `line` from the previous section. Print out the number of fields that result from the use of split, and print out the Device IDs (the third field) of all the records.

- Use the `String.split` method to separate the record into individual fields using the comma delimiters
- Print out the third field (the device ID field) for each record

99. Start by exploring a single line of data in the Scala shell. Use the following single string to represent one record in the `loudacre.log` file.

    - Note: The record string is also available to cut-and-paste in `data/sample_one_record.txt`

```
scala> val record="2014-03-15:10:10:31, Titanic 4000,
1882b564-c7e0-4315-aa24-228c0155ee1b,
58,36,39,31,15,0,TRUE,enabled,enabled,40.69206648, -119
.4216429"
```

100. Now split the string into its respective fields, using the comma (`,`) as the delimiter character:

```
scala> val fields=record.split(',')
```

101. What data type is returned by `split`? Using `fields`, determine how many fields are in the line.

The solution is in `Split-exploration.scala.txt`.

102. Now write, compile and run a main program that displays the Device ID from each line of data in the `loudacre.log` data file.

Note that there are many approaches to accomplishing this task. Compare your solution with two possible solutions: `ParseLines1.scala` and `ParseLines2.scala`.

Using `filter` to Limit Processing

103. There are six instances of "Titanic 4000" model phones in the Loudacre log file. Modify your solution to the last section (or start with the provided solution) to print out the device IDs for these six records only. Use the `Iterator.filter` method to limit processing to online those lines containing Titanic 4000 phone records.

You should get the following IDs:

```
27a418dc-9f20-4cb9-9c69-92d16c596bad
3d88c332-61fa-46f0-a171-84a8b02dd52a
c9b5d5d4-19a2-4259-bff7-e915d967d3f4
54994404-ab72-4e5f-a6bc-0d485b3806f2
57d50d31-ce40-440d-a2b3-a041eb9a29f5
1882b564-c7e0-4315-aa24-228c0155ee1b
```

Compare your solution with the solution in `FilterLines.scala`.

This is the end of the Exercise.

Hands-On Exercise: Libraries

Files and Data Used in this Exercise

Solutions: `Phone.scala`
`LogUtils.scala`
`PhoneTest.scala`

Data file:
`/home/training/training_materials/jes/data/loudacre.log`

In this exercise, you will write a program to process a set of device logs and display the device ID and model.

In order to do this you will create a class to represent a Phone device, and a utility object for parsing log lines.

This exercise makes use of various techniques you have previously worked with in the course. Feel free to refer to previous solutions while completing this exercise.

104. In your exercise working directory, create a singleton object called `LogUtils` with the following characteristics:

    - In package `com.loudacre`
    - A function `getDevID`: Given a line from a device log file, return the Device ID string
    - A function `getModel`: Given a line from a device log file, return the Model name

    A sample solution is in `LogUtils.scala`.

105. Compile the `LogUtils` object.

106. Start a Scala shell instance in the exercise working directory.

107. Test the `LogUtils` class:

```
scala> import com.loudacre.LogUtils
scala> val record="2014-03-15:10:10:31,Titanic
4000,1882b564-c7e0-4315-aa24-
228c0155ee1b,58,36,39,31,15,0,TRUE,enabled,enabled,40.6
9206648,-119.4216429"
scala> LogUtils.getDevId(record)
scala> LogUtils.getModel(record)
```

108. Once you have confirmed the object methods work correctly, return to your exercise working directory and crate a class called `Phone` with the following characteristics:

- In package `com.loudacre`
- Two `String` member variables: `devId` and `model`
- The constructor takes `devId` and `model` as parameters and sets the member variables
- Overrides the `toString` method to display `devId:model` as the object identifier

A sample solution is in `Phone.scala`.

109. Compile the `Phone` class.

110. Use the Scala shell to test instantiation of a `Phone` object and the `Phone.toString` method.

111. Write a main program class called `PhoneTest` with the following characteristics:

- The program takes a command line parameter with the path name of a log file
- The program prints out an error and usage message of the correct number of parameters are not provided

- The program creates a `List` of `Phone` instances, one for each line in the log file, using the `LogUtils` methods you wrote above to find the device ID and model name
- The program displays the contents of the `List`

A sample solution is in `PhoneTest.scala`.

112. Compile `PhoneTest.scala` and test it with the `loudacre.log` file.

This is the end of the Exercise.