

Classes



Lesson Objectives

- After completing this lesson, you should be able to:
 - Create and instantiate classes in Scala
 - Describe how arguments are passed to Scala class instances
 - Outline the lifespan of class parameters in a Scala class instance

What is a Class?

- A class is a description of a Type
 - Embodies state in an instance of a class
 - Represents behavior for how that state can be transformed
 - Is not concrete until it has been “instantiated” via a call to its constructor via the “new” keyword
 - Multiple instances of a class can exist

A Simple Scala Class

```
scala> class Hello
defined class Hello

scala> new Hello()
res0: Hello = Hello@33bd6867

scala> res0.toString
res1: String = Hello@33bd6867
```

The Primary Constructor

- Each class gets a primary constructor automatically
 - Defines the “signature” of how to create an instance
 - The body of the class is the implementation of the constructor

The Primary Constructor

```
scala> class Hello {  
      |   println("Hello")  
      | }  
defined class Hello  
  
scala> new Hello()  
Hello  
res0: Hello = Hello@7f68f33c
```

Class Parameters

- You can pass values into an instance of a class using one or more parameters to the constructor
 - You must specify the type of the parameter
 - The values are internally visible for the life of the class instance
 - They cannot be accessed from outside of the class instance

Class Parameters

```
scala> class Hello(message: String) {  
      |   println(message)  
      | }  
defined class Hello
```

```
scala> new Hello("Hello, world!")  
Hello, world!  
res0: Hello = Hello@5daef86b
```

```
scala> new Hello  
<console>:9: error: not enough arguments for constructor Hello:  
Unspecified value parameter message.
```

Class Parameters are not Accessible

```
scala> class Hello(message: String)
defined class Hello
```

```
scala> new Hello("Hello, world!")
res0: Hello = Hello@1946d5dc
```

```
scala> res0.message
<console>:9: error: value message is not a member of Hello
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Create and instantiate classes in Scala
 - Describe how arguments are passed to Scala class instances
 - Outline the lifespan of class parameters in a Scala class instance

Immutable and Mutable Fields



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe the difference between mutable and immutable fields
 - Create fields in Scala classes
 - Describe the difference between class parameters and fields
 - Outline how to promote class parameters to fields

What is a Field?

- A value encapsulated within an instance of a class
 - Represents the state of an instance, and therefore of an application at a given time
 - Is accessible to the outside world, unless specified otherwise

Fields versus Parameters

- Parameters are passed to a class and are only visible within a class
- Fields exist in the body of the class, and are accessible to outsiders

Immutable Fields

```
scala> class Hello {  
      |   val message: String = "Hello"  
      | }  
defined class Hello
```

```
scala> (new Hello).message  
res0: String = Hello
```

Mutable Fields

```
scala> class Hello {  
      |   var message: String = "Hello"  
      | }  
defined class Hello
```

```
scala> val hello = new Hello  
hello: Hello = Hello@3617a35c
```

```
scala> hello.message = "Hello, world!"  
hello.message: String = Hello, world!
```

Immutable or Mutable?

- Immutable fields cannot be changed and are therefore “threadsafe” in a multithreaded environment, such as the JVM
- Mutable fields can be useful, but require diligence to ensure that multiple threads cannot update the field at the same time

Use Immutable By Default

- It is easier to reason about immutable fields and classes that only contain immutable fields
- Scala makes all class parameters immutable by default

Specify Types

- Scala has “type inference”
- It is a good habit to be specific about types anyway

```
scala> class Hello {  
      |   val message = "Hello"  
      | }  
defined class Hello
```

```
scala> (new Hello).message  
res0: String = Hello
```



Promoting Class Parameters

- If you want to make a parameter passed to a class constructor into a publicly visible field, add the `val` keyword in front of it
- The Scala compiler will generate an accessor method for you, and other class instances can now ask for the current state of the promoted field

Promoting Class Parameters

```
scala> class Hello(val message: String)
defined class Hello

scala> val hello = new Hello("Hello, world!")
hello: Hello = Hello@59d941d7

scala> hello.message
res0: String = Hello, world!
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe the difference between mutable and immutable fields
 - Create fields in Scala classes
 - Describe the difference between class parameters and fields
 - Outline how to promote class parameters to fields

Methods



Lesson Objectives

- After completing this lesson, you should be able to:
 - Implement methods in Scala
 - Describe evaluation order of methods versus fields in Scala
 - Outline how infix notation works in Scala

What is a Method?

- A method describes behavior within a class
 - Are something that can be called to do work
 - Where transformations to internal state can take place
 - May take parameters as inputs, and may return a single value
 - Should specify their return type
 - More correctness
 - Faster compilation

A Simple Scala Method

```
scala> def hello = "Hello"  
hello: String
```

```
scala> def echo(message: String): String = message  
echo: (message: String)String
```

Why Methods Instead of Fields?

- Methods can look like fields
- Methods are evaluated at the time they are called
- Methods are re-evaluated every time they are called
- Fields are only evaluated at the time the class is constructed, and if immutable, only one time

Infix Notation

- Methods are called on an instance of a class
- Scala permits methods to be called with no “.” or parentheses, if the method takes only one argument
- This is flexible syntax that supports powerful DSLs
- For readability, you should not use this feature

Infix Method Calling

```
scala> "Martin Odersky".split(" ")  
res0: Array[String] = Array(Martin, Odersky)  
  
scala> "Martin Odersky" split " "  
res1: Array[String] = Array(Martin, Odersky)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Implement methods in Scala
 - Describe evaluation order of methods versus fields in Scala
 - Outline how infix notation works in Scala

Default and Named Arguments



Lesson Objectives

- After completing this lesson, you should be able to:
 - Utilize default argument values in Scala class constructors and methods
 - Leverage named arguments to only pass certain values

Default Argumets

- Allows the developer to specify a value to use for a constructor or method when none is passed by the caller, and omit values that are frequently the same
- Reduces boilerplate in application source code because you don't have to “overload” methods with different signatures

Default Arguments

```
def name(first: String = "", last: String = ""): String =  
    first + " " + last
```

```
scala> name("Martin")  
res0: String = Martin
```

Best Practice

- If you have a mixture of default arguments and those that do not have a default value, put the arguments without defaults first

```
def name(first: String,  
        last: String,  
        middle: String = ""): String =  
    first + " " + middle + " " + last
```

Named Arguments

- Leading arguments can be omitted if they have default values
- You can specify only the values you want to pass

Named Arguments

```
scala> name(last = "Odersky")
```

```
res0: String = Odersky
```

```
scala> name(first = "Martin", last = "Odersky")
```

```
res0: String = Martin Odersky
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Utilize default argument values in Scala class constructors and methods
 - Leverage named arguments to only pass certain values

Objects



Lesson Objectives

- After completing this lesson, you should be able to:
 - Create Singleton objects in Scala
 - Describe the difference between a class and an object in Scala
 - Outline usages for objects in Scala applications
 - Start a Scala application

What is an Object?

- The Singleton Pattern
 - Defines a single instance of a class that cannot be recreated within a single JVM instance
 - Can be directly accessed via its name

A Simple Scala Object

```
object Hello {  
  def message = "Hello!"  
}
```

```
scala> Hello.message  
res0: java.lang.String = "Hello!"
```

Why is this Useful?

- Many languages permit the definition of “static” fields and methods
- These are globally available within the runtime, such as a JVM
- They are not related to specific instances of a class

When are Objects Used?

- Class Factories
- Utility methods
- Constant definitions

A Simple Object

```
object Hello {  
  val oneHourInMinutes: Int = 60  
  
  def createTimeFromMinutes(minutes: Int) =  
    new Time(  
      minutes / oneHourInMinutes,  
      minutes % oneHourInMinutes)  
}
```

Starting a Scala Application

```
object Hello {  
  def main(args: Array[String]): Unit =  
    println("Hello!")  
}
```

Starting a Scala Application

```
$ scala -cp target/scala-2.11/classes/ Hello  
Hello
```

```
> run  
[info] Running Hello  
Hello  
[success] Total time: 0 s, completed Jul 20, 2012 6:00:20 PM
```



Lesson Summary

- Having completing this lesson, you should be able to:
 - Create Singleton objects in Scala
 - Describe the difference between a class and an object in Scala
 - Outline usages for objects in Scala applications
 - Start a Scala application

Accessibility and Companion Objects



Lesson Objectives

- After completing this lesson, you should be able to:
 - Leverage accessibility keywords to manage visibility of methods and fields
 - Describe the role companion objects play in Scala
 - Outline how to use companion objects

Accessibility

- We can use keywords to limit the visibility of methods and fields in class instances
 - **public**, the default
 - **private**, limiting visibility only to yourself
 - **protected**, unimportant for now

Accessibility

```
class Hello {  
  private val message: String = "Hello!"  
}
```

```
class Welcome {  
  val message: String = "Hello!"  
}
```

Companion Objects

- If a Singleton object and a class share the same name and are located in the same source file, they are called companions
- A companion class can access private fields and methods inside of its companion object

Companion Objects

- This is a great way to separate static members (fields, constants and methods) that are unrelated to a specific instance from those members that are related to a specific instance of that class

Companion Objects

```
object Hello {  
    private val defaultMessage = "Hello!"  
}  
  
class Hello(message: String = Hello.defaultMessage) {  
    println(message)  
}
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Leverage accessibility keywords to manage visibility of methods and fields
 - Describe the role companion objects play in Scala
 - Outline how to use companion objects

Case Classes and Case Objects



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe when to use case classes and case objects instead of regular classes and objects
 - Outline the differences between case classes and objects compared to regular classes and objects

What is a Data Class?

- Some classes represent specific data types in a “domain”
- Imagine an online store:
 - Customers
 - Accounts
 - Orders
 - Inventory

What is a Service Class?

- Some classes represent work to be performed in an application
- They know what to do, but they do not hold the data themselves
- When an application calls the service, they pass the data to the service, and the service transforms it in some way
- Examples:
 - Persistence
 - Loggers
 - Calculation engines

Case Classes

- A representation of a data type that removes a lot of boilerplate code
 - Generates JVM-specific convenience methods
 - Makes every class parameter a field
 - Is immutable by default
 - Performs value-based equivalence by default

Case Classes

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)  
defined class Time
```

```
scala> val time = Time(9, 0)  
res0: Time = Time(9, 0)
```

```
scala> time == Time(9)  
res1: Boolean = true
```

```
scala> time == Time(9, 30)  
res2: Boolean = false
```

```
scala> time.hours  
res3: Int = 9
```

Case Objects

- If a case class is an instance-based representation of a data type, a case object is a representation of a data type of which there can only be a single instance
- If you try to create a case class with no parameters, it is stateless and should be a case object

Case Objects

```
scala> case class Time
<console>:1: error: case classes without a parameter list are
use either case objects or case classes with an explicit '()'
case class Time
      ^
```

```
scala> case object Time
defined object Time
```

```
scala> Time.toString
res0: String = Time
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe when to use case classes and case objects instead of regular classes and objects
 - Outline the differences between case classes and objects compared to regular classes and objects

Apply and Unapply



Lesson Objectives

- After completing this lesson, you should be able to:
 - Illustrate the difference between a type and a term
 - Describe how the apply method works in both objects and classes
 - Outline how unapply works

Types versus Terms

- A type is a description of a concept in an application
 - A class is a type
- A term is a concrete representation of a type
 - Any class instance (including an object) is a term
 - A method is a term, as it is also concrete and “callable”

Calling a Term

- Like some other languages, Scala allows you to “call” a term without specifying the method you want to call on it

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)  
defined class Time
```

```
scala> val time = Time(9, 0)  
res0: Time = Time(9, 0)
```

How Did That Work?

- When you create a case class, the compiler generates a companion object for the class for you
- Calling **Time**(9, 0) is actually calling the companion object Time and delegating to the **apply()** method inside of it.

```
scala> Time(9)
res0: Time = Time(9,0)
```

```
scala> Time.apply(9)
res1: Time = Time(9,0)
```

An Example of apply

```
object Reverse {  
  def apply(s: String): String =  
    s.reverse  
}
```

```
scala> Reverse("Hello")  
res0: String = olleH
```

Another Example of apply

```
scala> Array(1, 2, 3)  
res0: Array[Int] = Array(1, 2, 3)
```

```
scala> res0(0)  
res1: Int = 1
```

Unapply Deconstructs a Case Class

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)
defined class Time

scala> val time = Time(9, 0)
time: Time = Time(9,0)

scala> Time.unapply(time)
res2: Option[(Int, Int)] = Some((9,0))
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Illustrate the difference between a type and a term
 - Describe how the apply method works in both objects and classes
 - Outline how unapply works

Synthetic Methods



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe how the Scala compiler generates functionality for you
 - Explain what the synthetic **`equals()`**, **`hashCode()`**, **`toString()`** and **`copy()`** methods do
 - Outline how you would use immutable case classes in a program where state is changing

Coding and Maintenance are Expensive

- Writing and maintaining the source code required by the JVM for simple data classes is difficult
- To support the features of case classes, a comparable Time class in Java would be over 70 lines of code

What are Synthetic Methods?

- Scala's compiler generates this “boilerplate” for you
- The implementations are rock solid and proven
 - `equals()`
 - `hashCode()`
 - `toString()`
 - `copy()`

`equals ()`

- This method is required by the JVM, but the default implementation only compares whether an instance of the class is the exact same instance
- Scala provides value-based equivalence, allowing you to compare whether two different instances of a class have the same state

equals()

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)
defined class Time
```

```
scala> val time = Time(9, 0)
time: Time = Time(9,0)
```

```
scala> time == Time(9)
res0: Boolean = true
```

```
scala> time == Time(9, 30)
res1: Boolean = false
```

hashCode ()

- This method is required for any class that you might want to put into a hashed collection, such as a HashMap or HashSet

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)
defined class Time
```

```
scala> Time(9, 45).hashCode()
res0: Int = 471971180
```



toString()

- This method is required by the JVM, but the default implementation prints out a virtual representation of the instance location in memory
- The synthetic **toString()** provided by Scala's case class shows you the values inside of the class
- You can override this to make it even better

toString()

```
scala> class Time(hours: Int = 0, minutes: Int = 0)  
defined class Time
```

```
scala> new Time()  
res0: Time = Time@4d591d15
```

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)  
defined class Time
```

```
scala> Time()  
res1: Time = Time(0,0)
```



`copy()`

- This method is not required by the JVM
- The synthetic `copy()` provided by Scala's case class allows you to remain immutable and use “snapshots” of case classes when state needs to change

copy()

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)  
defined class Time
```

```
scala> var time = Time(9, 45)  
time: Time = Time(9,45)
```

```
scala> time = time.copy(minutes = 50)  
time: Time = Time(9,50)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe how the Scala compiler generates functionality for you
 - Explain what the synthetic **equals()**, **hashCode()**, **toString()** and **copy()** methods do
 - Outline how you would use immutable case classes in a program where state is changing

Immutability and Thread Safety



Lesson Objectives

- After completing this lesson, you should be able to:
 - Understand basic thread safety in the JVM
 - Describe the importance of immutability in multithreaded applications
 - Outline how to use snapshots to preserve thread safety with case classes



What is Thread Safety




- The JVM has a well-defined memory model with specific guarantees
- There are two concerns:
 - **Synchronize-With:** Who is able to change state and in what order (locks)
 - **Happens-Before:** When to publish changes on one thread to all other threads (memory barriers)



Names versus Values

val me = **new** Person("Jamie", "Allen")

 Name  Value

		VALUE	
		Mutable	Immutable
NAME	var		
	val		

The Left Side of the Equals Sign

- Represents a pointer to the current value
- We want this to be “final” as much as possible, using a **val**
- Reassignment to a new value is possible when using a **var**

The Right Side of the Equals Sign

- Represents the value of the current state
- This should always be immutable, meaning that the class instance contains only fields that are defined as **val**
- If not, you must protect the state and who can change it using Mutually Exclusive Locks

Using a `var` for Snapshots

- Allows us to keep the value on the right side of the equals immutable, but still change our current state by replacing what the `var` points to with another instance
- The case class `copy()` method will help you do this

@volatile

- The **@volatile** annotation must be used when you follow the snapshot strategy, to ensure that all threads see your updates
- The case class **copy()** method will help you do this

@volatile

```
scala> case class Customer(firstName: String = "",  
                             lastName: String = "")  
defined class Customer  
  
scala> @volatile var customer = Customer("Martin", "Odersky")  
customer: Customer = Customer(Martin,Odersky)  
  
scala> customer = customer.copy(lastName = "Doe")  
customer: Customer = Customer(Martin,Doe)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Understand basic thread safety in the JVM
 - Describe the importance of immutability in multithreaded applications
 - Outline how to use snapshots to preserve thread safety with case classes

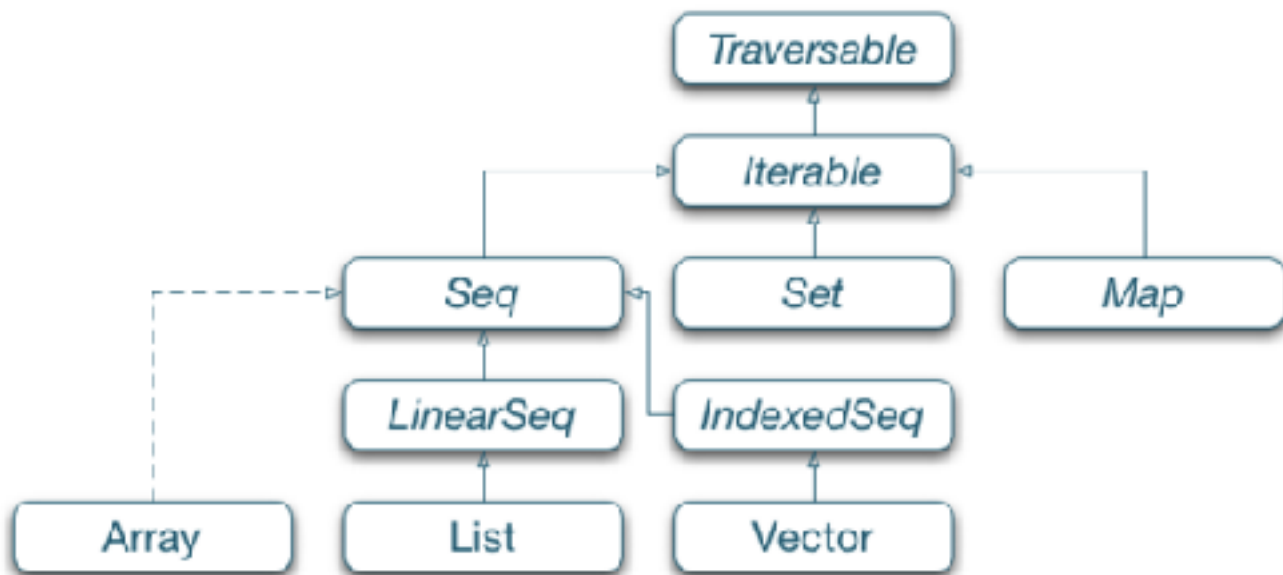
Collections Overview



Lesson Objectives

- After completing this lesson, you should be able to:
 - Understand the structure of the Scala collections hierarchy
 - Describe how to apply functions to data in collections
 - Outline the basics of structural sharing
 - Illustrate the performance characteristics of different data structures

Scala Collections



Higher Order Functions

- Scala is a functional programming language
- We apply functions to data inside of containers
- There are many higher order functions available to you across the collections library

Higher Order Functions

```
scala> 1 to 10
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> res0.map(n => n + 1)
res1: scala.collection.immutable.IndexedSeq[Int] =
  Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

Structural Sharing

- When you create a collection, it is an aggregation of references to individual values in the Java heap
- If you use immutable collections of immutable values, those references can be shared between collection instances

Structural Sharing

```
scala> List(1, 2, 3, 4, 5)
res0: List[Int] = List(1, 2, 3, 4, 5)

scala> res0 :+ 6
res1: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> 0 +: res1
res2: List[Int] = List(0, 1, 2, 3, 4, 5, 6)
```

Performance

COLLECTIONS

Performance Characteristics

The previous explanations have made it clear that different collection types have different performance characteristics. That's often the primary reason for picking one collection type over another. You can see the performance characteristics of some common operations on collections summarized in the following two tables.

Performance characteristics of sequence types:

	head	tail	apply	update	prepend	append	insert
immutable							
List	O	O	L	L	O	L	-
Stream	O	O	L	L	O	L	-
Vector	eO	eO	eO	eO	eO	eO	-
Stack	O	O	L	L	O	O	L
Queue	eO	eO	L	L	L	O	-
Range	O	O	O	-	-	-	-
String	O	L	O	L	L	L	-

<http://docs.scala-lang.org/overviews/collections/performance-characteristics.html>



Lesson Summary

- Having completing this lesson, you should be able to:
 - Understand the structure of the Scala collections hierarchy
 - Describe how to apply functions to data in collections
 - Outline the basics of structural sharing
 - Illustrate the performance characteristics of different data structures

Sequences and Sets



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe the various kinds of sequence collections and their properties
 - Outline the desirable properties of the Vector collection type
 - Describe the properties of set collections

What is a Sequence?

- An ordered collection of data
- Duplicates are permitted
- May or may not be indexed
- Array, List, Vector
- The apply method on an instance is a lookup

Array

- A fixed size, ordered sequence of data
- Very fast on the JVM
- Values are contiguous in memory
- Indexed by position

Array

```
scala> Array(1, 2, 3, 4, 5)
res0: Array[Int] = Array(1, 2, 3, 4, 5)

scala> res0(2)
res1: Int = 3

scala> res0(5) = 6
java.lang.ArrayIndexOutOfBoundsException: 5
... 33 elided
```

List

- A linked list implementation, with a value and a pointer to the next element
- Theoretically unbounded in size
- Poor performance as data could be located anywhere in memory, and must be accessed via “pointer chasing”

List

```
scala> List(1, 2, 3, 3, 4)
res0: List[Int] = List(1, 2, 3, 3, 4)

scala> res0.distinct
res1: List[Int] = List(1, 2, 3, 4)

scala> res0
res2: List[Int] = List(1, 2, 3, 3, 4)

scala> res0 :+ 5
res3: List[Int] = List(1, 2, 3, 3, 4, 5)

scala> 0 +: res1
res4: List[Int] = List(0, 1, 2, 3, 4)
```

Vector

- A linked list of 32 element arrays
- 2.15 billion possible elements
- Indexed by hashing
- Good performance across all operations without having to copy arrays when more space is needed

What is a Set?

- A “bag” of data, where no duplicates are permitted
- Order is not guaranteed
- HashSet, TreeSet, BitSet, KeySet, SortedSet, etc
- The apply method on an instance checks to see if the set contains a value

Set

```
scala> Set(1, 2, 3, 3, 4)
res0: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> res0.getClass
res1: Class[_ <: scala.collection.immutable.Set[Int]] =
  class scala.collection.immutable.HashSet$HashTrieSet

scala> res0 + 5
res2: scala.collection.immutable.Set[Int] = Set(1, 5, 2, 3, 4)

scala> res2 + 2
res3: scala.collection.immutable.Set[Int] = Set(1, 5, 2, 3, 4)
```

Set

```
scala> Set(1, 2, 3, 4)
res0: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> res0(5)
res1: Boolean = false

scala> res0(2)
res2: Boolean = true

scala> res0.toSeq
res3: Seq[Int] = ArrayBuffer(1, 2, 3, 4)

scala> res3.toSet
res4: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe the various kinds of sequence collections and their properties
 - Outline the desirable properties of the Vector collection type
 - Describe the properties of set collections

Option



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe the relevance of Option in the Scala type system
 - Outline how to use Option in your types

Algebraic Data Types (ADTs)

- A distinct set of possible types
- Intuition:
 - Days of the week
 - Binary light switches

Option

- Not a collection, but a container
- An ADT representing the existence of a value
- **Some** is the representation of a value
- **None** is the representation of the absence of a value
- Allows us to avoid **null** on the JVM

Option

```
scala> Option("Jamie")  
res1: Option[String] = Some(Jamie)
```

```
scala> res1.get  
res2: String = Jamie
```

```
scala> res1.getOrElse("Jane")  
res3: String = Jamie
```

Option

```
scala> Option(null)
res0: Option[Null] = None

scala> res0.get
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:347)
  at scala.None$.get(Option.scala:345)
  ... 33 elided

scala> res0.getOrElse("Foo")
res2: String = Foo
```

Option

- Option allows us to create APIs where the possible absence of value is encoded in the type system
- We can then perform behavior without asking whether or not the value is **null** in advance

Option

```
scala> case class Customer(  
      first: String = "",  
      middle: Option[String] = None,  
      last: String = "")  
  
defined class Customer  
  
scala> Customer("Martin", last = "Odersky")  
res0: Customer = Customer(Martin,None,Odersky)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe the relevance of Option in the Scala type system
 - Outline how to use Option in your types

Tuples and Maps



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe what a tuple is and how they are used
 - Outline how to deconstruct tuples
 - Describe the properties of a map

Tuples

- A loose aggregation of values into a single container
- Can have up to 22 values in Scala
- Are always used when you see parentheses wrapping data without a specific type

Tuples

```
scala> Tuple2(1, "a")  
res0: (Int, String) = (1,a)
```

```
scala> Tuple2(1, 2)  
res1: (Int, Int) = (1,2)
```

```
scala> (1, "a")  
res0: (Int, String) = (1,a)
```

Tuples

- Can be accessed using a 1-based accessor for each value
- Can be deconstructed into names bound to each value in a tuple

Tuples

```
scala> val tuple = (1, "a", 2, "b")  
tuple: (Int, String, Int, String) = (1,a,2,b)
```

```
scala> tuple._3  
res0: Int = 2
```

```
scala> val (first, second, third, fourth) = tuple  
first: Int = 1  
second: String = a  
third: Int = 2  
fourth: String = b
```

Tuple2

- Frequently called a pair
- Have a unique syntax for values

Tuple2

```
scala> (1, "a")  
res0: (Int, String) = (1,a)
```

```
scala> (2 -> "b")  
res1: (Int, String) = (2,b)
```

```
scala> (3 -> "c" -> 4)  
res2: ((Int, String), Int) = ((3,c),4)
```

Unapply Deconstructs a Case Class

```
scala> case class Time(hours: Int = 0, minutes: Int = 0)
defined class Time

scala> val time = Time(9, 0)
time: Time = Time(9,0)

scala> Time.unapply(time)
res2: Option[(Int, Int)] = Some((9,0))
```

Maps

- A grouping of data by key to value, which are tuple “entries”
- Allows you to index values by a specific key for fast access
- Common implementations: HashMap, TreeMap

Maps

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5)

scala> 'a' to 'g'
res1: scala.collection.immutable.NumericRange.Inclusive[Char]
  = NumericRange(a, b, c, d, e, f, g)

scala> res0.zip(res1)
res2: scala.collection.immutable.IndexedSeq[(Int, Char)] =
  Vector((1,a), (2,b), (3,c), (4,d), (5,e))

scala> res2.toMap
res3: scala.collection.immutable.Map[Int,Char] =
  Map(5 -> e, 1 -> a, 2 -> b, 3 -> c, 4 -> d)
```

Maps

```
scala> Map(1 -> "a", 2 -> "b", 3 -> "c")
res0: scala.collection.immutable.Map[Int,String] =
  Map(1 -> a, 2 -> b, 3 -> c)

scala> res0(1)
res1: String = a

scala> res0(4)
java.util.NoSuchElementException: key not found: 4
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
  ... 33 elided
```

Maps

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

```
scala> map(1)  
res0: String = a
```

```
scala> map.get(9)  
res1: Option[String] = None
```

```
scala> map.getOrElse(1, "z")  
res2: String = a
```

```
scala> map.getOrElse(9, "z")  
res3: String = z
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe what a tuple is and how they are used
 - Outline how to deconstruct tuples
 - Describe the properties of a map

Higher Order Functions



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe the application of functions to data
 - Outline basic usages of higher order functions in Scala

Higher Order Functions

- A function which takes another function
- Typically describes the “how” for work to be done in a container
- The function passed to it describes the “what” that should be done to elements in the container

map

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5)

scala> res0.map(number => number + 1)
res1: scala.collection.immutable.IndexedSeq[Int] =
  Vector(2, 3, 4, 5, 6)

scala> res0.map(_ + 1)
res2: scala.collection.immutable.IndexedSeq[Int] =
  Vector(2, 3, 4, 5, 6)
```

flatMap

```
scala> List("Scala", "Python", "R")
res0: List[String] = List(Scala, Python, R)

scala> res0.map(lang => lang + "#")
res1: List[String] = List(Scala#, Python#, R#)

scala> res0.flatMap(lang => lang + "#")
res2: List[String] =
  List(S, c, a, l, a, #, P, y, t, h, o, n, #, R, #)
```

filter

```
scala> List("Scala", "Python", "R", "SQL")  
res0: List[String] = List(Scala, Python, R, SQL)  
  
scala> res0.filter(lang => lang.contains("s"))  
res1: List[String] = List(Scala, SQL)
```

foreach

```
scala> List(1, 2)
res0: List[Int] = List(1, 2)

scala> res0.map(println)
1
2
res1: List[Unit] = List((), ())

scala> res0.foreach(println)
1
2
```

forall

```
scala> List("Scala", "Simple", "Stellar")  
res0: List[String] = List(Scala, Simple, Stellar)  
  
scala> res0.forall(lang => lang.contains("s"))  
res1: Boolean = true  
  
scala> res0.forall(lang => lang.contains("a"))  
res2: Boolean = false
```

reduce

```
scala> 1 to 5
```

```
res0: scala.collection.immutable.Range.Inclusive =  
  Range(1, 2, 3, 4, 5)
```

```
scala> res0.reduce((acc, cur) => acc + cur)
```

```
res1: Int = 15
```

```
scala> res0.reduce(_ + _)
```

```
res2: Int = 15
```

```
scala> List[Int]().reduce((acc, cur) => acc + cur)
```

```
java.lang.UnsupportedOperationException: empty.reduceLeft  
... 37 elided
```



fold, foldLeft, foldRight

```
scala> 1 to 5
```

```
res0: scala.collection.immutable.Range.Inclusive =  
  Range(1, 2, 3, 4, 5)
```

```
scala> res0.foldLeft(0){ case (acc, cur) => acc + cur}  
res1: Int = 15
```

```
scala> List[Int]().foldLeft(0){ case (acc, cur) => acc + cur}  
res2: Int = 0
```

product

```
scala> 1 to 5  
res0: scala.collection.immutable.Range.Inclusive =  
  Range(1, 2, 3, 4, 5)  
  
scala> res0.product  
res1: Int = 120
```

exists

```
scala> 1 to 5  
res0: scala.collection.immutable.Range.Inclusive =  
  Range(1, 2, 3, 4, 5)
```

```
scala> res0.exists(num => num == 3)  
res1: Boolean = true
```

```
scala> res0.exists(num => num == 6)  
res2: Boolean = false
```

find

```
scala> 1 to 5  
res0: scala.collection.immutable.Range.Inclusive =  
  Range(1, 2, 3, 4, 5)  
  
scala> res0.find(num => num == 3)  
res1: Option[Int] = Some(3)  
  
scala> res0.find(num => num == 6)  
res2: Option[Int] = None
```

groupBy

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5)

scala> res0.groupBy(num => num % 2)
res1: Map[Int,scala.collection.immutable.IndexedSeq[Int]] =
  Map(1 -> Vector(1, 3, 5), 0 -> Vector(2, 4))
```

takeWhile and dropWhile

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5)

scala> res0.takeWhile(num => num < 3)
res1: scala.collection.immutable.Range = Range(1, 2)

scala> res0.dropWhile(num => num < 3)
res2: scala.collection.immutable.Range = Range(3, 4, 5)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe the application of functions to data
 - Outline basic usages of higher order functions in Scala

For Expressions



Lesson Objectives

- After completing this lesson, you should be able to:
 - Understand the relationship between for expressions and higher order functions
 - Describe the usage of for expressions

Composition Is Hard

- Trying to put together multiple higher order functions into a single expression is difficult
- For expressions are syntactic sugar that simplifies the work of coding a multi-stage transformation

Composing HOFs

```
scala> val myNums = 1 to 3
myNums: Range.Inclusive = Range(1, 2, 3)

scala> myNums.map(i => (1 to i).map(j => i * j))
res0: IndexedSeq[IndexedSeq[Int]] =
  Vector(Vector(1), Vector(2, 4), Vector(3, 6, 9))

scala> myNums.flatMap(i => (1 to i).map(j => i * j))
res1: IndexedSeq[Int] = Vector(1, 2, 4, 3, 6, 9)
```

For Expressions

```
scala> val myNums = 1 to 3
myNums: Range.Inclusive = Range(1, 2, 3)

scala> for {
  |   i <- myNums
  |   j <- 1 to i
  | } yield i * j
res0: IndexedSeq[Int] = Vector(1, 2, 4, 3, 6, 9)
```

Syntax

- Must start with the **for** keyword
- Must have generators, using the **<-** arrow
- The **yield** keyword dictates whether or not a new value is returned

Syntax

- Syntactic sugar over **map**, **flatMap**, **withFilter** and **foreach**
- Higher Order Functions have rules
 - If I **map** over a List, I will get a List
 - The first generator of a for expression follows the same rule
- Can have guard conditions to apply filters

Filtering

```
scala> val myNums = 1 to 3
myNums: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3)

scala> for {
  | i <- myNums if i % 2 == 1
  | j <- 1 to i
  | } yield i * j
res0: scala.collection.immutable.IndexedSeq[Int] =
  Vector(1, 3, 6, 9)
```

Definitions

```
for {  
  time <- times  
  hours = time.hours if hours > 12  
} yield (hours - 12) + "pm"  
// Result: Vector[String] = Vector(1pm, 2pm)
```

Effectful Usages

```
for (n <- 1 to 3) println(n)  
(1 to 3).foreach(n => println(n))
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Understand the relationship between for expressions and higher order functions
 - Describe the usage of for expressions

Pattern Matching



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe how to use pattern matching to handle different values in different ways
 - Outline how case classes and ADTs help in pattern matching
 - Illustrate how to extract values from tuples

What is Pattern Matching?

- Many languages have the concept of **switch/case**
- Pattern matching is similar, but can be applied across many different types of data
- Can be embedded within other expressions as a way of cleanly expressing conditional logic

The match Keyword

```
def isCustomer(someValue: Any): Boolean = {  
  someValue match {  
    case cust: Customer => true  
    case _ => false  
  }  
}
```

Usage

```
scala> case class Customer(first: String = "",  
                             last: String = "")
```

```
defined class Customer
```

```
scala> Customer("Martin", "Odersky")
```

```
res0: Customer = Customer(Martin,Odersky)
```

```
scala> isCustomer(res0)
```

```
res1: Boolean = true
```

```
scala> isCustomer("Martin Odersky")
```

```
res2: Boolean = false
```



Pattern Matching is Flexible

- You can match on many different kinds of values
 - Literal values, like “12:00”
 - Use guard conditions to be more specific
 - Match on only some parts of a value
 - More specific cases must come first, more general last
 - If you use the `_` or a simple name with no type, both match on everything

Exhaustiveness

- When you see the **case** keyword, pattern matching is in play
- Case classes and ADTs provide compile-time exhaustiveness checking that all possible conditions have been have been met

Pattern Matching Tuple Values

```
scala> val tuple = (1, "a", 2, "b")  
tuple: (Int, String, Int, String) = (1,a,2,b)
```

```
scala> tuple._3  
res0: Int = 2
```

```
scala> val (first, second, third, fourth) = tuple  
first: Int = 1  
second: String = a  
third: Int = 2  
fourth: String = b
```

Pattern Matching HOF Arguments

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5)

scala> res0.reduce((acc, cur) => acc + cur)
res1: Int = 15

scala> res0.foldLeft(0){ case (acc, cur) => acc + cur }
res2: Int = 15
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe how to use pattern matching to handle different values in different ways
 - Outline how case classes and ADTs help in pattern matching
 - Illustrate how to extract values from tuples

Handling Options



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe how to pattern match on an Option
 - Outline how to use higher order functions on an option to avoid null-checking
 - Illustrate how to use for comprehensions to work with Option

Pattern Matching an Option

```
def getMiddleName(value: Option[String]): String = {  
  value match {  
    case Some(middleName) => middleName  
    case None => "No middle name"  
  }  
}
```

Pattern Matching an Option

```
scala> case class Customer(first: String = "",  
                           middle: Option[String] = None,  
                           last: String = "")
```

```
scala> val martin = Customer("Martin", last = "Odersky")  
martin: Customer = Customer(Martin,None,Odersky)
```

```
scala> getMiddleName(martin.middle)  
res0: String = No middle name
```

Pattern Matching an Option

```
scala> case class Customer(first: String = "",  
                             middle: Option[String] = None,  
                             last: String = "")
```

```
scala> val jane = Customer("Jane", Option("D."), "Doe")  
jane: Customer = Customer(Jane,Some(D.),Doe)
```

```
scala> getMiddleName(jane.middle)  
res1: String = D.
```

HOFs and Option

```
scala> Option("Martin")  
res0: Option[String] = Some(Martin)  
  
scala> res0.map(name => println("Yay, " + name))  
Yay, Martin  
res1: Option[Unit] = Some(())  
  
scala> res0.foreach(name => println("Yay, " + name))  
Yay, Martin  
  
scala> None.foreach(name => println("Yay, " + name))
```

For Expressions and Option

```
scala> val martin = Option("Martin")  
martin: Some[String] = Some(Martin)
```

```
scala> val jane = Option("Jane")  
jane: Some[String] = Some(Jane)
```

```
scala> for {  
  |   m <- martin  
  |   j <- jane  
  | } yield (m + " is friends with " + j)  
res1: Option[String] = Some(Martin is friends with Jane)
```



For Expressions and Option

```
scala> val martin = Option("Martin")  
martin: Some[String] = Some(Martin)
```

```
scala> val noValue = None  
noValue: None.type = None
```

```
scala> for {  
  |   m <- martin  
  |   n <- noValue  
  | } yield (m + " is friends with " + n)  
res2: Option[String] = None
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe how to pattern match on an Option
 - Outline how to use higher order functions on an option to avoid null-checking
 - Illustrate how to use for comprehensions to work with Option

Handling Failures



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe the usage and importance of a Try
 - Describe how to pattern match on a Try
 - Outline how to use higher order functions on a Try
 - Illustrate how to use for comprehensions to work with Try

JVM Exceptions

- They represent runtime failures for various reasons
 - NullPointerException (Runtime)
 - ClassCastException (Runtime)
 - IOException (Checked)
 - When one occurs, control is “thrown” back within a thread stack to whomever “catches” it

Catching an Exception

```
def toInt(s: String): Int =  
  try {  
    s.toInt  
  } catch {  
    case _: NumberFormatException => 0  
  }
```

Idiomatic Scala and Exceptions

- In Scala, we do not believe in this approach, as it represents a possible “side effect”
 - We want everything in our code to be pure
 - When we interact with libraries or services that may fail, we “wrap” the call in a Try to capture the failure



Wrapping a Call in Try

```
scala> import scala.util.{Try, Success, Failure}  
import scala.util.{Try, Success, Failure}
```

```
scala> Try("100".toInt)  
res0: scala.util.Try[Int] = Success(100)
```

```
scala> Try("Martin".toInt)  
res1: scala.util.Try[Int] =  
  Failure(java.lang.NumberFormatException:  
    For input string: "Martin")
```

Pattern Matching on Try

```
scala> import scala.util.{Try, Success, Failure}
import scala.util.{Try, Success, Failure}

scala> def makeInt(s: String): Int = Try(s.toInt) match {
  |   case Success(n) => n
  |   case Failure(_) => 0
  |   }
makeInt: (s: String)Int

scala> makeInt("35")
res2: Int = 35

scala> makeInt("James")
res3: Int = 0
```

Higher Order Functions and Try

```
scala> import scala.util._  
import scala.util._  
  
scala> def getScala: Try[String] = Success("Scala")  
getScala: scala.util.Try[String]  
  
scala> val scala = getScala  
scala: scala.util.Try[String] = Success(Scala)  
  
scala> scala.map(s => s.reverse)  
res0: scala.util.Try[String] = Success(alacS)
```

Higher Order Functions and Try

```
scala> import scala.util._  
import scala.util._  
  
scala> def getOuch: Try[String] =  
  Failure(new Exception("Ouch"))  
getOuch: scala.util.Try[String]  
  
scala> val ouch = getOuch  
ouch: scala.util.Try[String] =  
  Failure(java.lang.Exception: Ouch)  
  
scala> ouch.map(s => s.reverse)  
res0: scala.util.Try[String] =  
  Failure(java.lang.Exception: Ouch)
```



For Expressions and Try

```
scala> Success("Scala").map(_.reverse)
res0: scala.util.Try[String] = Success(alacS)

scala> for {
  |   language <- Success("Scala")
  |   behavior <- Success("rocks")
  | } yield s"$language $behavior"
res1: scala.util.Try[String] = Success(Scala rocks)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe the usage and importance of a Try
 - Describe how to pattern match on a Try
 - Outline how to use higher order functions on a Try
 - Illustrate how to use for comprehensions to work with Try

Handling Futures



Lesson Objectives

- After completing this lesson, you should be able to:
 - Describe how to use Futures to perform work asynchronously
 - Describe how to pattern match on the result of a Future
 - Outline how to use higher order functions on the result of a Future
 - Illustrate how to use for comprehensions to work with Futures

Futures

- Allow us to define work that may happen at some later time, possibly on another thread
- Futures return a Try of whether or not the work was successfully completed

ExecutionContext

- To use a Future, you must provide a thread pool that the Future can use to perform the work
- I can use an **implicit val** to declare it one time and automatically apply it to all usages within a scope

ExecutionContext

```
import scala.concurrent.ExecutionContext
import java.util.concurrent.ForkJoinPool

implicit val ec: ExecutionContext =
  ExecutionContext.fromExecutor(new ForkJoinPool())
```

Timeout

- Futures can have a defined amount of time before they “time out”, or fail because they have taken too long to do their work or be scheduled
- Scala has a nice DSL for creating such time-based values

Timeout

```
scala> import scala.concurrent.duration._  
import scala.concurrent.duration._
```

```
scala> implicit val timeout = 1 second  
timeout: scala.concurrent.duration.FiniteDuration = 1 second
```

Required Imports

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext
import java.util.concurrent.ForkJoinPool
import scala.util.Failure
import scala.util.Success
import scala.concurrent.duration._
```

Wrapping a Call in a Future

```
val f: Future[Int] = Future {  
  inventoryService.getCurrentInventory(1234567L)  
}
```

Pattern Matching on Future

```
scala> val f: Future[Int] = Future { 1 + 2 + 3 }  
f: scala.concurrent.Future[Int] =  
  scala.concurrent.impl.Promise$DefaultPromise@8b96fde  
  
scala> f.onComplete {  
  | case Success(i) => println("onComplete Success: " + i)  
  | case Failure(f) => println("onComplete Failure: " + f)  
  | }  
  
onComplete Success: 6
```

Higher Order Functions and Futures

```
scala> val g: Future[Int] = Future { 1 + 2 + 3 }  
g: scala.concurrent.Future[Int] = ...  
  
scala> g.map(result => println(Mapping: " + result))  
Mapping: 6
```

Higher Order Functions and Futures

```
val g: Future[Int] = Future { Thread.sleep(4000); 5 }  
g: scala.concurrent.Future[Int] = ...  
  
scala> g.map(result => println(Mapping: " + result))
```

For Expressions and Futures

```
val usdQuote = Future {  
  connection.getCurrentValue(USD) }  
val chfQuote = Future {  
  connection.getCurrentValue(CHF) }  
val purchase = for {  
  usd <- usdQuote  
  chf <- chfQuote if isProfitable(usd, chf)  
} yield connection.buy(amount, chf)
```

Lesson Summary

- Having completing this lesson, you should be able to:
 - Describe how to use Futures to perform work asynchronously
 - Describe how to pattern match on the result of a Future
 - Outline how to use higher order functions on the result of a Future
 - Illustrate how to use for comprehensions to work with Futures