CSC 735 – Data Analytics

Introduction to Scala

Goal

- Our goal here is to learn enough Scala to write Spark code

What is Scala?

- Scala is a general-purpose programming language
- Concise – like Python
- Scala source code compiles to Java bytecode that runs on a Java virtual machine
- Language interoperability with Java

Features of Scala

- Scala ≡ Scalable Language
- Good support for functional programming
  - high-order functions, immutable values, lazy evaluation, optimization, pattern matching
- Good support for object-oriented programming
- A strong type system
- Implicits
  - code that is concise and easier to understand

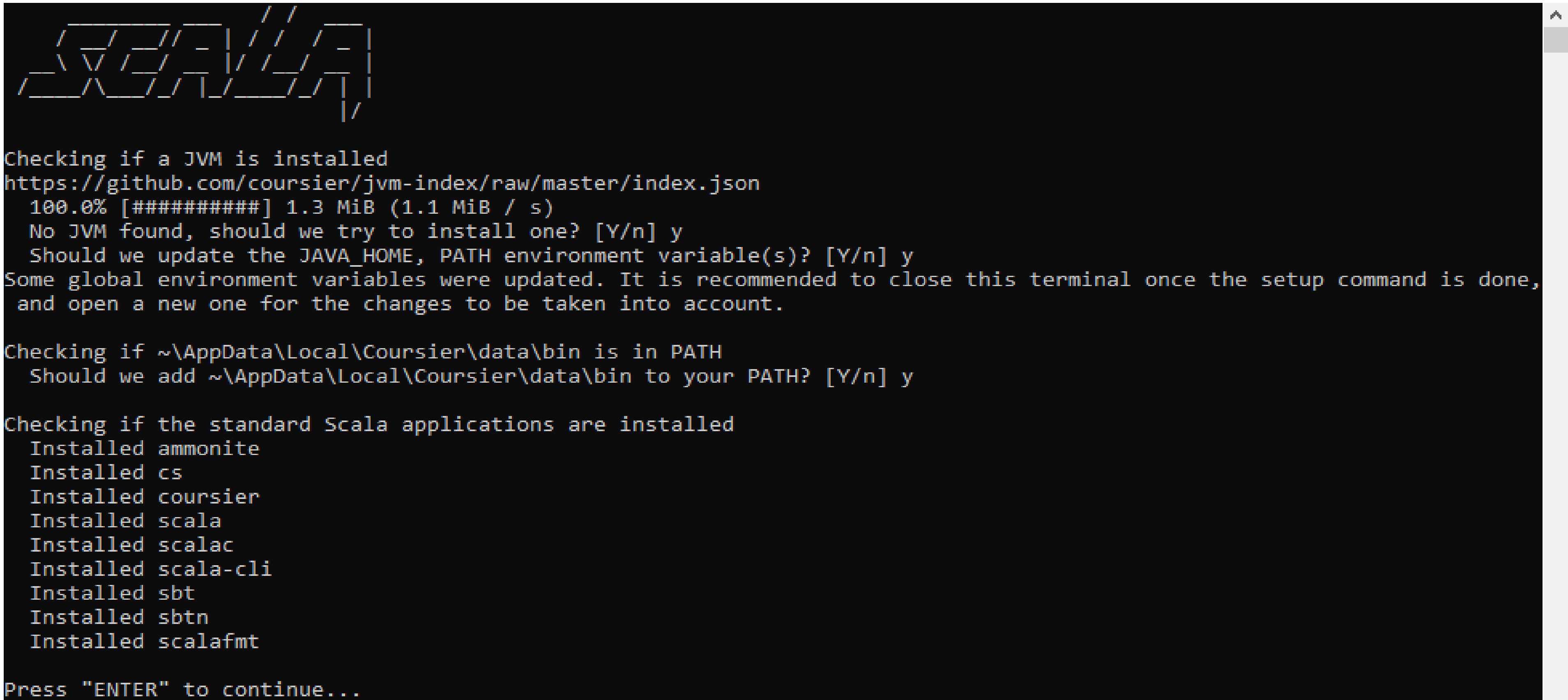
Why Learn Scala for Big Data

- Provides a boost to your professional career
- Write robust code with few bugs
- Spark is written in Scala
- Best support for Spark
- Faster Spark code

Installing Scala

- Make sure you have Java 8 or newer
- Download the Scala Binaries from

  https://www.scala-lang.org/download/



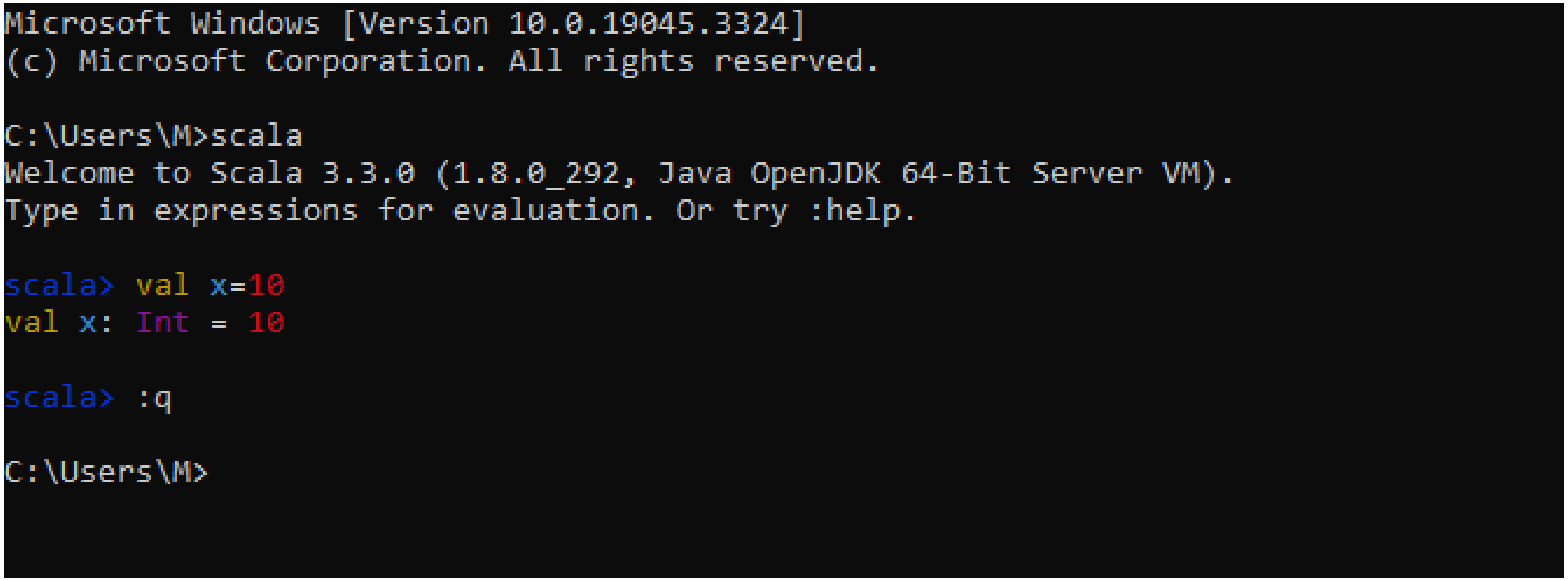
```
     ________ ___   / /  ___
    / __/ __// _ | / /  / _ |
  __\ \/ /__/ __ |/ /__/ __ |
 /____/\___/_/ |_/____/_/ | |
                          |/

Checking if a JVM is installed
https://github.com/coursier/jvm-index/raw/master/index.json
  100.0% [##########] 1.3 MiB (1.1 MiB / s)
  No JVM found, should we try to install one? [Y/n] y
  Should we update the JAVA_HOME, PATH environment variable(s)? [Y/n] y
Some global environment variables were updated. It is recommended to close this terminal once the setup command is done,
 and open a new one for the changes to be taken into account.

Checking if ~\AppData\Local\Coursier\data\bin is in PATH
  Should we add ~\AppData\Local\Coursier\data\bin to your PATH? [Y/n] y

Checking if the standard Scala applications are installed
  Installed ammonite
  Installed cs
  Installed coursier
  Installed scala
  Installed scalac
  Installed scala-cli
  Installed sbt
  Installed sbtn
  Installed scalafmt

Press "ENTER" to continue...
```

Installing Scala

- Place the scala\bin subdirectory to system path
- For windows ~\AppData\Local\Coursier\data\bin

Using Scala

- To start Scala REPL, type scala at the command prompt
- To quit, type :quit, or :q
- REPL ≡ Read, Evaluate, Print, Loop



```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\M>scala
Welcome to Scala 3.3.0 (1.8.0_292, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> val x=10
val x: Int = 10

scala> :q

C:\Users\M>
```

Alternative Way of Using Scala

- Use a text editor to type your code as a singleton object
- Assume we saved the following program as HelloWorld.scala

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

- Then, use the command prompt and the following command to compile

```
c:\>scala<u>c</u> HelloWorld.scala
```

- necessary bytecode files will be created. To execute

```
c:\>scala HelloWorld
```

Alternative Way of Using Scala (cont.)

```
object MyFirstScalaProgram extends App {
    println("Hello World")
}
```

- Assume file saved as HelloWorld2.scala
- Use this command to compile

  scalac HelloWorld2.scala
- What is the name of the executable?
- What is the command to run?

Alternative Way of Using Scala (cont.)

```
object MyFirstScalaProgram extends App {
    println("Hello World")
}
```

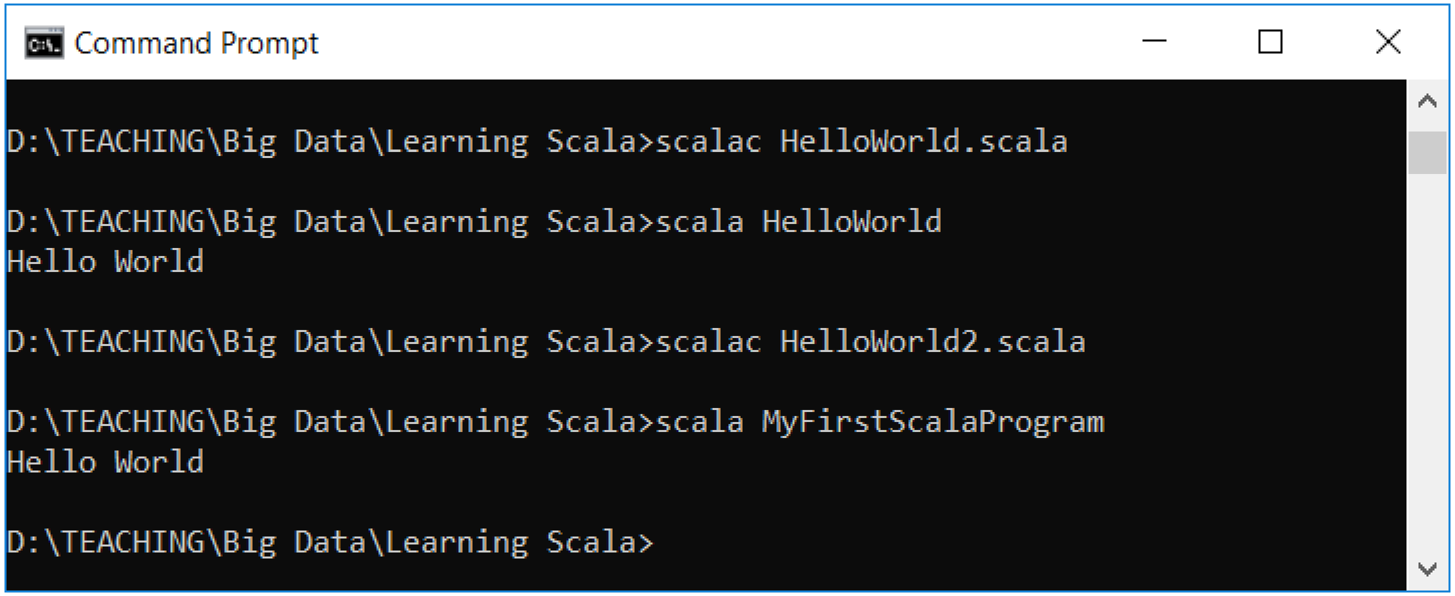
- Assume file saved as HelloWorld2.scala
- Use this command to compile

```
scalac HelloWorld2.scala
```

- What is the name of the executable?
- What is the command to run?

```
scala MyFirstScalaProgram
```

Compiling and Running A Scala Program



```
D:\TEACHING\Big Data\Learning Scala>scalac HelloWorld.scala

D:\TEACHING\Big Data\Learning Scala>scala HelloWorld
Hello World

D:\TEACHING\Big Data\Learning Scala>scalac HelloWorld2.scala

D:\TEACHING\Big Data\Learning Scala>scala MyFirstScalaProgram
Hello World

D:\TEACHING\Big Data\Learning Scala>
```

Using an IDE with Scala

- You could use Eclipse or IntelliJ
- Instructions for IntelliJ at https://docs.scala-lang.org/getting-started-intellij-track/getting-started-with-scala-in-intellij.html
- For Eclipse
  - Download from scala-ide.org
  - Instructions at http://scala-ide.org/docs/current-user-doc/gettingstarted/index.html

Basic Types

Variable Type	**Description**
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	true or false

Basic Types (cont.)

- Scala has 7 numeric types and a Boolean type
- Each type in Scala is implemented as a class
- We can invoke methods on numbers
  - 1.toString() //yields the string "1"
  - 99.44.toInt //yields 99
  - 1.to(10) // yields the Range(1, 2, 3, ..., 10)
  - 2.3.getClass.getSimpleName //res26: String = double

Variables

- mutable vs. immutable

Variables

- mutable vs. immutable
- Use **var** to declared a mutable variable

Variables

- mutable vs. immutable
- Use **var** to declared a mutable variable

```
var x = 10
```

Variables

- mutable vs. immutable
- Use **var** to declared a mutable variable

```
var x = 10
x = 20
```

Variables

- mutable vs. immutable
- Use **var** to declared a mutable variable

```
var x = 10
x = 20
```

- Use **val** to declare an immutable variable

Variables

- mutable vs. immutable
- Use **var** to declared a mutable variable

```
var x = 10
x = 20
```

- Use **val** to declare an immutable variable

```
val x = 10
```

Variables

- mutable vs. immutable
- Use **var** to declared a mutable variable

```
var x = 10
x = 20
```

- Use **val** to declare an immutable variable

```
val x = 10
x = 20  //error
```

Remarks

Remarks

- semicolon at the end of a statement is optional

Remarks

- semicolon at the end of a statement is optional

```
val y = 10;
val y = 10  // Equivalent
```

Remarks

- semicolon at the end of a statement is optional

```
val y = 10;
val y = 10  // Equivalent
```

- the compiler infers type wherever possible

Remarks

- semicolon at the end of a statement is optional

```
val y = 10;
val y = 10  // Equivalent
```

- the compiler infers type wherever possible

```
val y = 10
```

Remarks

- semicolon at the end of a statement is optional

```
val y = 10;
val y = 10  // Equivalent
```

- the compiler infers type wherever possible

```
val y = 10
val y: Int = 10
```

Remarks

- semicolon at the end of a statement is optional

```
val y = 10;
val y = 10  // Equivalent
```

- the compiler infers **type** wherever possible

```
val y = 10
val y: Int = 10
```

- Scala is a statically typed language, so everything has a type

Lazy Values

- When a **val** is declared as lazy, its initialization is deferred until it is needed

Lazy Values

- When a **val** is declared as lazy, its initialization is deferred until it is needed
- Syntax:
  lazy val x = 10

Lazy Values

- When a **val** is declared as lazy, its initialization is deferred until it is needed
- Syntax:
  lazy val x = 10
- Initialization will take place when x is used
  print(x + 5)
  val y = x + 2

Lazy Values

- When a **val** is declared as lazy, its initialization is deferred until it is needed
- Syntax:
  lazy val x = 10
- Initialization will take place when x is used
  print(x + 5)
  val y = x + 2
- Notice: only vals can be lazy

Arithmetic and Operator Overloading

- Scala has the same arithmetic operators as other languages: + - * / %

Arithmetic and Operator Overloading

- Scala has the same arithmetic operators as other languages: + - * / %
- In Scala, operators are actually methods

Arithmetic and Operator Overloading

- Scala has the same arithmetic operators as other languages: + - * / %
- In Scala, operators are actually methods

```
a + b
```

is a shorthand for

```
a.+(b)
```

Arithmetic and Operator Overloading

- Scala has the same arithmetic operators as other languages: + - * / %
- In Scala, operators are actually methods

```
a + b
```

is a shorthand for

```
a.+(b)
```

- In general, these are equivalent:

```
a method b
a.method(b)
```

Arithmetic and Operator Overloading

- Scala has the same arithmetic operators as other languages: + - * / %
- In Scala, operators are actually methods

  a + b

  is a shorthand for

  a.+(b)
- In general, these are equivalent:

  a method b

  a.method(b)
- So, 1.to(10) can be written as 1 to 10

Arithmetic and Operator Overloading

- Scala has the same arithmetic operators as other languages: + - * / %
- In Scala, operators are actually methods

  a + b

  is a shorthand for

  a.+(b)

- In general, these are equivalent:

  a method b

  a.method(b)

- So, 1.to(10) can be written as 1 to 10
- There is no ++ or – – in Scala

More about Calling Methods

- When calling a method that has no parameters, don’t use parentheses after method’s name
- Ex: the method **sorted**, yields a new string with the letters in sorted order

  "Bonjour".sorted // Yields the string "Bjnooru“

More about Calling Methods

- When calling a method that has no parameters, don’t use parentheses after method’s name
- Ex: the method **sorted**, yields a new string with the letters in sorted order

  "Bonjour".sorted // Yields the string "Bjnooru“
- The rule of thumb is that a parameter-less method that doesn’t modify the object has no parentheses

Importing Packages

- Serve same purpose as packages in Python & Java or namespaces in C++
- Allow us to avoid naming conflicts and to write shorter syntax without any prefix
- To print $\sqrt{4}$

- import scala.math.sqrt
  ```
  print(sqrt(4))  //with an import statement
  ```
- To import everything from a package use _
  ```
  import scala.math._
  ```
- To import more than one member from a package use
  ```
  import scala.math.{max, min, cos, Pi}
  ```

Importing Packages

- Serve same purpose as packages in Python & Java or namespaces in C++
- Allow us to avoid naming conflicts and to write shorter syntax without any prefix
- To print $\sqrt{4}$

  print(scala.math.sqrt(4)) //w.o. an import statement
- import scala.math.sqrt

  print(sqrt(4)) //with an import statement
- To import everything from a package use _

  import scala.math._
- To import more than one member from a package use

  import scala.math.{max, min, cos, Pi}

Importing Packages (cont.)

- We can rename members:

```
import scala.math.{max => maximum}
print(maximum(2, 3))
```

Importing Packages (cont.)

- We can rename members:

```
import scala.math.{max => maximum}
print(maximum(2, 3))
```

- Every Scala program **implicitly** starts with

```
import java.lang._
import scala._
import Predef._
```

Importing Packages (cont.)

- We can rename members:

```
import scala.math.{max => maximum}
print(maximum(2, 3))
```

- Every Scala program **implicitly** starts with

```
import java.lang._
import scala._
import Predef._
```

- if a package starts with scala., you can omit the scala prefix

Importing Packages (cont.)

- We can rename members:

  ```
  import scala.math.{max => maximum}
  print(maximum(2, 3))
  ```

- Every Scala program **implicitly** starts with

  ```
  import java.lang._
  import scala._
  import Predef._
  ```

- if a package starts with scala., you can omit the scala prefix

  math.sqrt is as good as **scala.math.sqrt**

The apply Method

- Ex: val s = "Hello";
- s(4) //yields 'o'
- Overloaded form of the () operator,
  which is implemented with the method **apply**

- s(4) is a shortcut for

  s.apply(4)
- In the class StringOps, you find a method
  def apply(n: Int): Char

The apply Method

- Likewise, BigInt("1234567890") is a shortcut for BigInt.apply("1234567890")

The apply Method

- Likewise, BigInt("1234567890") is a shortcut for BigInt.apply("1234567890")
- It yields a new BigInt object
  BigInt("1234567890") * BigInt("112358111321")

The apply Method

- Likewise, BigInt("1234567890") is a shortcut for BigInt.apply("1234567890")
- It yields a new BigInt object
  BigInt("1234567890") * BigInt("112358111321")

The apply Method

- Likewise, BigInt("1234567890") is a shortcut for BigInt.apply("1234567890")
- It yields a new BigInt object
  BigInt("1234567890") * BigInt("112358111321")
- Using the apply method of a class is a common Scala idiom for constructing objects
- For example, Array(1, 4, 9, 16) returns an array

Constructs have Values

In Java or C++

- An expressions has a value
  - 2 + 3
- A statement carries an action
  - if statement, assignment statement

Constructs have Values

In Java or C++

- An expressions has a value
  - 2 + 3
- A statement carries an action
  - if statement, assignment statement
- In Scala, almost all constructs have values
  - an if expression has a value
  - a block has a value—the value of its last expression

Constructs have Values

In Java or C++

- An expressions has a value
  - 2 + 3
- A statement carries an action
  - if statement, assignment statement
- In Scala, almost all constructs have values
  - an if expression has a value
  - a block has a value—the value of its last expression
- Benefit: concise and more readable code

Conditional Statements

- If/else statements have same syntax as in Java/C++
- In Scala, an if/else has a value, namely the value of the expression that follows the if or else

```
if (x > 0) 1 else -1
```

Conditional Statements

- If/else statements have same syntax as in Java/C++
- In Scala, an if/else has a value, namely the value of the expression that follows the if or else

```
    if (x > 0) 1 else -1
    val s = if (x > 0) 1 else -1
```

This has the same effect as

```
if (x > 0) s = 1 else s = -1
```

Conditional Statements

- If/else statements have same syntax as in Java/C++
- In Scala, an if/else has a value, namely the value of the expression that follows the if or else

```
    if (x > 0) 1 else -1
    val s = if (x > 0) 1 else -1
```

This has the same effect as

```
if (x > 0) s = 1 else s = -1

val weather = if (temperature > 85) "hot"
else "not hot"
```

Type/Class Any

- In Scala, every expression has a type

Type/Class Any

- In Scala, every expression has a type
- if (x > 0) 1 else -1
  has type Int

Type/Class Any

- In Scala, every expression has a type
- if (x > 0) 1 else -1
  has type Int
- What is the type of
  `if (x > 0) "positive" else -1`

Type/Class Any

- In Scala, every expression has a type
- if (x > 0) 1 else -1
  has type Int
- What is the type of
  if (x > 0) "positive" else -1
- The type of a mixed-type expression is the common supertype of both branches

Type/Class Any

- In Scala, every expression has a type
- if (x > 0) 1 else -1
  has type Int
- What is the type of
  `if (x > 0) "positive" else -1`
- The type of a mixed-type expression is the common supertype of both branches
- The common supertype of String and Int is called **Any**

The Inheritance Hierarchy of Scala Classes

8.11 ■ The Scala Inheritance Hierarchy 101

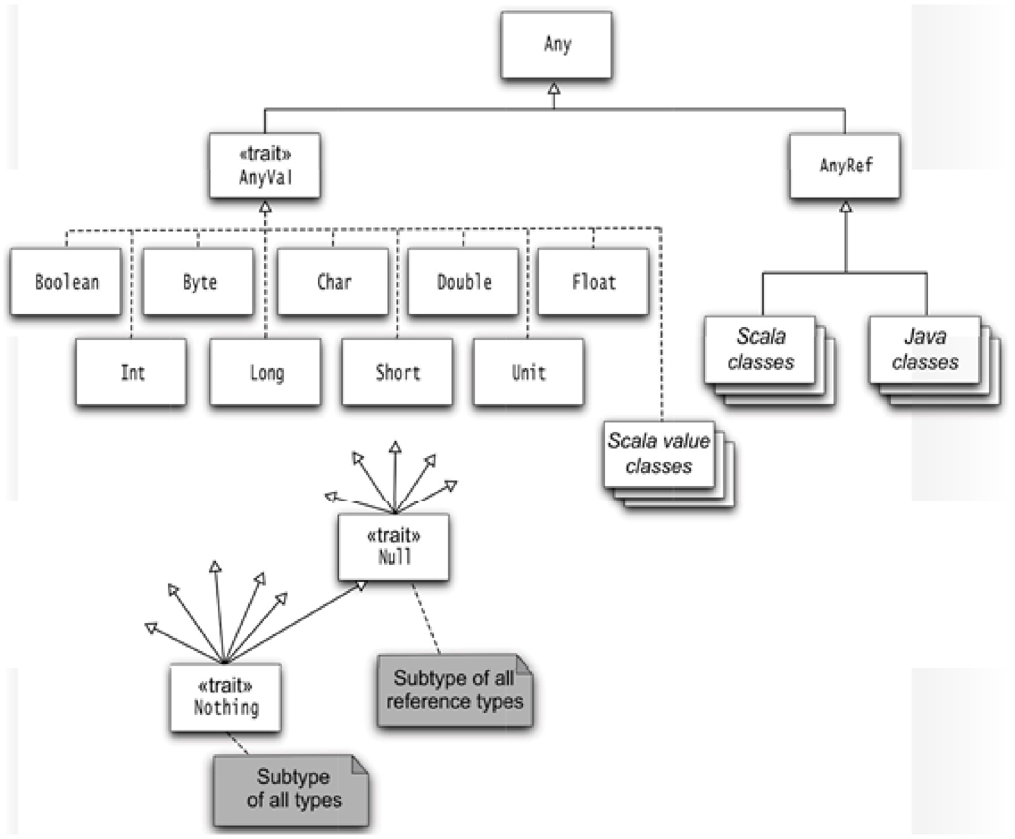


Figure 8–1 The inheritance hierarchy of Scala classes

Type/Class Unit

- What is the type of
  if (x > 0) 1

Type/Class Unit

- What is the type of
  if (x > 0) 1
- This if statement could yield no value

Type/Class Unit

- What is the type of
  if (x > 0) 1
- This if statement could yield no value
- Since every expression should have a value, Scala introduces a class **Unit** that has one value, “no value”, written as ()

Type/Class Unit

- What is the type of
  if (x > 0) 1
- This if statement could yield no value
- Since every expression should have a value, Scala introduces a class **Unit** that has one value, “no value”, written as ()
- The above if statement is equivalent to

Type/Class Unit

- What is the type of
  if (x > 0) 1
- This if statement could yield no value
- Since every expression should have a value, Scala introduces a class **Unit** that has one value, "no value", written as ()
- The above if statement is equivalent to

  if (x > 0) 1 else ()

Type/Class Unit

- What is the type of
  if (x > 0) 1
- This if statement could yield no value
- Since every expression should have a value, Scala introduces a class **Unit** that has one value, “no value”, written as ()
- The above if statement is equivalent to

  if (x > 0) 1 else ()
- Think of () as a placeholder for “no useful value,” and of Unit as an analog of **void** in Java/C++

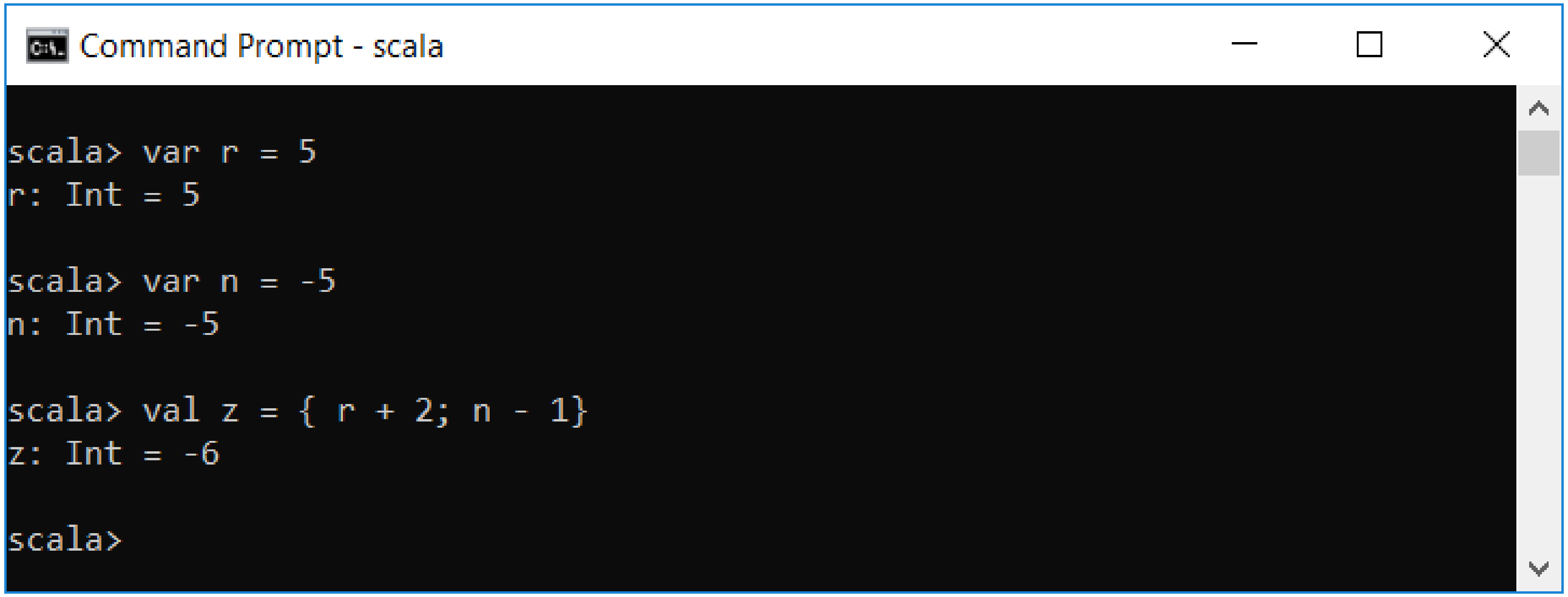
Type/Class Unit

- What is the type of
  if (x > 0) 1
- This if statement could yield no value
- Since every expression should have a value, Scala introduces a class **Unit** that has one value, “no value”, written as ()
- The above if statement is equivalent to

  if (x > 0) 1 else ()
- Think of () as a placeholder for “no useful value,” and of Unit as an analog of void in Java/C++
- The supertype of Int and Unit is **AnyVal**

Block Expressions and Assignments

- {} makes a block of code
- The value of a block is that of the last expression inside it



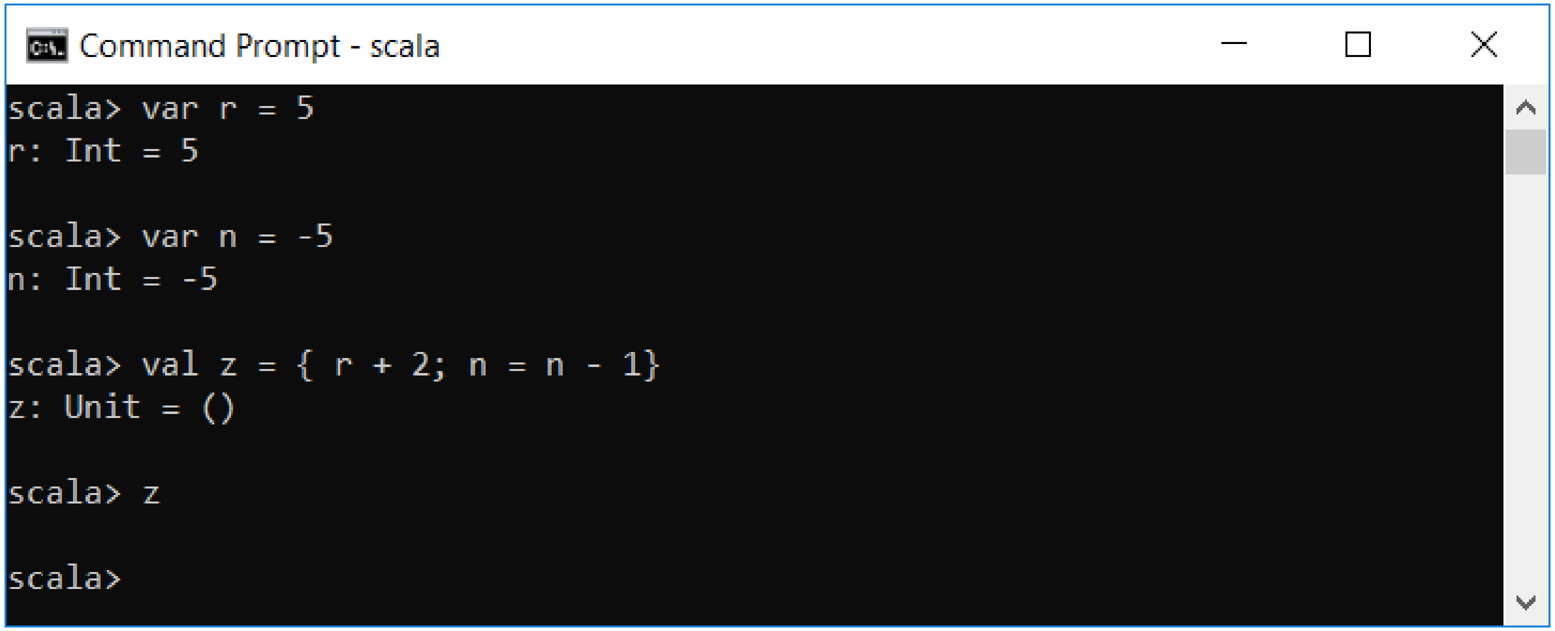
```
scala> var r = 5
r: Int = 5

scala> var n = -5
n: Int = -5

scala> val z = { r + 2; n - 1}
z: Int = -6

scala>
```

Block Expressions and Assignments

- In Scala, assignments have no value (i.e., Unit value)
- So, if we have a block that ends with an assignment statement, that block has a Unit value

Block Expressions and Assignments

- In Scala, assignments have no value (i.e., Unit value)
- So, if we have a block that ends with an assignment statement, that block has a Unit value



```
scala> var r = 5
r: Int = 5

scala> var n = -5
n: Int = -5

scala> val z = { r + 2; n = n - 1}
z: Unit = ()

scala> z

scala>
```

Remark on Chained Assignments

- x = y =1

Remark on Chained Assignments

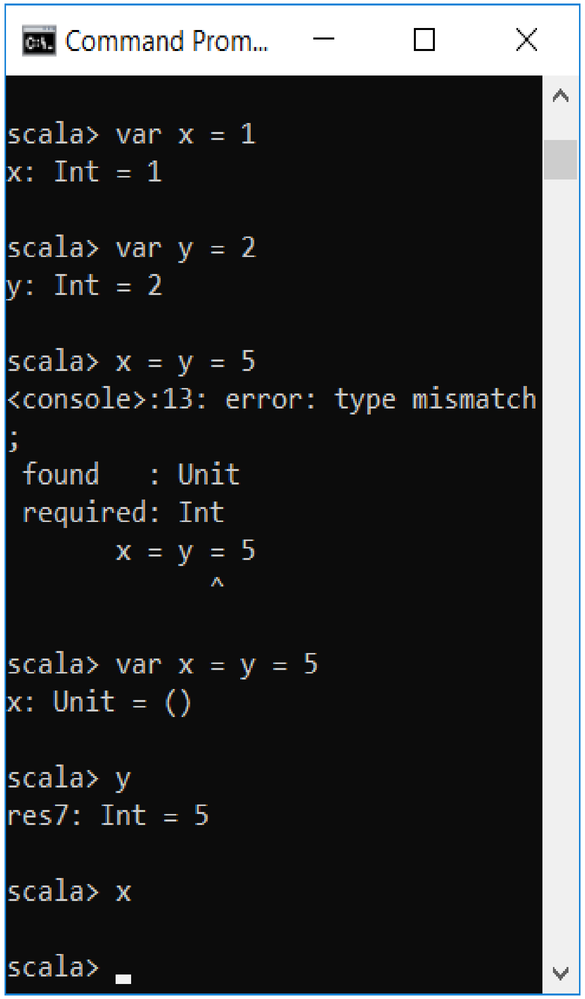
- Do not use chain assignments in Scala

  x = y = 1 // No
- The value of y = 1 is ()
- The expression y = 1 has Unit value
- If syntax allows it, x would have a Unit value

Remark on Chained Assignments

- Do not use chain assignments in Scala

  x = y = 1 // No
- The value of y is 1
- The expression y = 1 has Unit value
- If syntax allows it, x would have a Unit value



```
scala> var x = 1
x: Int = 1

scala> var y = 2
y: Int = 2

scala> x = y = 5
<console>:13: error: type mismatch
;
 found   : Unit
 required: Int
       x = y = 5
             ^

scala> var x = y = 5
x: Unit = ()

scala> y
res7: Int = 5

scala> x

scala>
```

Input and Output

- print() and println()

```
val x = 3; val y = 5
println(x + y) //outputs: 8
```

- Scala has printf() with a C-style syntax

```
val name = "Mark"; val age = 5
printf("Hello %4s! Your are %5d years old.\n", name, age);
//Hello Mark! Your are     5 years old.
```

String Interpolations

- We can also use string interpolation
  1. The **f** Interpolator (f-Strings)
  - simple formatted strings, all variable references should be followed by a **printf-style format string**
- A formatted string can contain expressions and format directives

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age +
0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be    5.50 years old.
```

String Interpolations (cont.)

2. With prefix **s**, a string can contain expressions but not format directives. Escape sequences are evaluated.

```
val name = "Mark"; val age = 5
```

String Interpolations (cont.)

- With prefix s, a string can contain expressions but not format directives. Escape sequences are evaluated.

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years
old.%n")
```

String Interpolations (cont.)

- With prefix s, a string can contain expressions but not format directives. Escape sequences are evaluated.

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years
old.%n")
//Hello, Mark! In six months, you'll be    5.50 years old.
```

String Interpolations (cont.)

- With prefix s, a string can contain expressions but not format directives. Escape sequences are evaluated.

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be    5.50 years old.
print(s"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
```

String Interpolations (cont.)

- With prefix s, a string can contain expressions but not format directives. Escape sequences are evaluated.

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be    5.50 years old.
print(s"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be 5.5%7.2f years old.%n
```

String Interpolations (cont.)

- With prefix s, a string can contain expressions but not format directives. Escape sequences are evaluated.

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be    5.50 years old.
print(s"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be 5.5%7.2f years old.%n
print(s"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.\n")
```

String Interpolations (cont.)

- With a prefix of **raw**, neither escape sequences nor format directive are evaluated

```
val name = "Mark"; val age = 5
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be    5.50 years old.
print(s"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
//Hello, Mark! In six months, you'll be 5.5%7.2f years old.%n
print(s"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.\n")
Hello, Mark! In six months, you'll be 5.5%7.2f years old.

print(raw"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.\n")
Hello, Mark! In six months, you'll be 5.5%7.2f years old.\n~
scala>
```

Reading Input

- import **scala.io.StdIn**
- ReadLine
- To read a numeric, Boolean, or character value, use readInt, readDouble, readByte, readShort, readLong, readFloat, readBoolean, or readChar
- The **readLine** method, but not the other ones, takes a **prompt** string:

```
import scala.io.StdIn
val name = StdIn.readLine("Enter your name: ")
print("Enter your age: ")
val age = StdIn.readInt()
println(s"Hello, ${name}! Next year, you will be ${age + 1}.")
```

Loops

- Scala has the same while and do while loops as in Java/C++

```
var i = 5; var summation = 0
while (i > 0){
   summation += i
   i -=1
}
print(summation) //15
// 5 + 4 + 3 + 2 + 1
```

Loops

- Scala has the same while and do while loops as in Java/C++

```
var i = 5; var summation = 0
while (i > 0){
   summation += i
   i -=1
}
print(summation) //15
// 5 + 4 + 3 + 2 + 1
```

```
var i = 0; var summation = 0
while {
  i += 1
  i<=5
} do (summation += i)
print(summation) //15
// 1 + 2 + 3 + 4 + 5
```

Loops - for

- Scala does not have C++/Java for loop
- Instead, one can use this kind of loop

```
for (i <- 1 to n)
   do something
```

```
for (i <- expr)
   do something
```

- no val or var before the variable in the for loop
- type of the variable is the type of the elements of the collection
- scope of loop variable is until the end of the loop

Loops – for (Examples)

```
for (i <- 1 to 5)
  print(i + " ")
// 1 2 3 4 5
```

Loops – for (Examples)

```
for (i <- 1 to 5)
  print(i + " ")
// 1 2 3 4 5
```

```
val s = "ABC"
for (ch <- s)
  print(ch + " ")
//A B C
```

Loops – for (Examples)

```
for (i <- 1 to 5)
  print(i + " ")
// 1 2 3 4 5
```

```
val s = "ABC"
for (ch <- s)
  print(ch + " ")
//A B C
```

```
val s = "ABC"
var result = 0
for (i <- 0 to s.length - 1)
  result += s(i)
print("result is " + result)
//result is 198  "65 + 66 + 67"
```

Loops – for (Examples)

```
for (i <- 1 to 5)
  print(i + " ")
// 1 2 3 4 5
```

```
val s = "ABC"
for (ch <- s)
  print(ch + " ")
//A B C
```

```
val s = "ABC"
var result = 0
for (i <- 0 to s.length - 1)
  result += s(i)
print("result is " + result)
//result is 198  "65 + 66 + 67"
```

```
val s = "ABC"
var result = ""
for (i <- 0 to s.length - 1)
  result += s(i)
print("result is " + result)
//result is ABC
```

For Comprehension

- if the body of the **for loop** starts with yield, the loop constructs a collection of values, one for each iteration:

```
for (i <- 1 to 10) yield i % 3
// Yields Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
////res104:scala.collection.immutable.IndexedSeq[Int] =
Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

- This type of loop is called a for comprehension