A Concise Introduction to Scala

General

- Scala is based on Java, and compiles to the JVM (or to .NET).
  - Scala textbooks, and this paper, generally assume a knowledge of Java.
- Scala is fully Object-Oriented (OO) -- there are no primitives.
- Scala fully supports functional programming.
- Scala is statically typed like Java, but the programmer has to supply type information in only a few places; Scala can infer type information.
- Scala can be interpreted or compiled.
- Scala is pronounced “skah-lah,” not “skay-lah.”

Getting Scala

As of October 2016, these were the relevant web sites:

- Java (either JDK or just the JRE is enough): https://www.java.com/en/download/
- Scala: http://www.scala-lang.org/download/ (I recommend the Scala IDE, based on Eclipse)
  - Scala API docs:
    - Online, http://www.scala-lang.org/api/current/#package
    - Download, http://scala-lang.org/documentation/api.html (Library API)
- ScalaTest, for unit testing: http://www.scalatest.org/
  - ScalaTest API docs
    - Online: http://www.scalatest.org/scaladoc
    - Download: ???
- ScalaFX, for writing GUIs: https://github.com/scalafx/scalafx
  - ScalaFX API docs:
    - Online, http://www.scalafx.org/api/8.0/index.html#package
    - Download, ???

Running Scala

In a Java application, you need a `public static void main(String[] args)` method. In a Scala application, you need a `def main(args: Array[String])` method.

How to run	Code	Results/Comments

On the command line as a REPL (Read-Eval-Print-Loop).	`dave% scala`
	
	`Welcome to Scala version 2.7.7.final (Java ↙`
	`  HotSpot(TM) 64-Bit Server VM, Java 1.6.0_17).`
	`Type in expressions to have them evaluated.`
	`Type :help for more information.`
		No `main` method is required in this case.

	`scala> 2+2`
	`res0: Int = 4`
	`scala> exit`	

As a script.	`println("Hello from a script.")`	`dave% scala HelloScript.scala`
		`Hello from a script.`
		
		Scala automatically wraps your lines in a class and a `main` method, so don't do this yourself.

As an application.	`object HelloApp extends App {`
	`  println("Hello, World!")`
	`}`
		`dave% scalac hello.scala`
		`dave% scala HelloApp`
		`Hello, World!`
		
		File name doesn't have to be the same as the application name. The `App` trait wraps your code in a `main` method.

As a single object.	`object HelloObject {`
	`  println("In object")`
	
	`  def main(args: Array[String]) {`
	`  println("In main")`
	`  }`
	`}`
		`dave% scalac HelloProgram.scala`
		`dave% scala HelloObject`
		`In object`
		`In main`

As a program.	`class HelloClass {`
	`  def inClass() {`
	`  println("In class")`
	`  }`
	`}`
	
	`object HelloObjectPlusClass {`
	`  println("In object")`
	
	`  def main(args: Array[String]) {`
	`  println("In main")`
	`  val c = new HelloClass`
	`  c.inClass()`
	`  }`
	`}`
		`dave% scalac HelloWithClass.scala`
		`dave% scala HelloObjectPlusClass`
		`In object`
		`In main`
		`In class`
		
		The object ***may not*** have the same name as the class!
		
		`scalac` is the compiler. You can use `fsc` instead, which remains in memory and so is faster (after the first use).

Identifiers

Scala has four types of identifiers:

Alphanumeric identifiers:

- Composed of letters, underscores, and digits, beginning with a letter or underscore.
- The '`$`' counts as a letter, but is reserved for use by Scala.
- Follows Java's camel case conventions, except "real" constants (not `val`s) are camel case and begin with a capital.
- Examples: `abc123`, `myVar`, `myVal`, `myMethod`, `MyClass`, `MyObject`, `Pi`, `E`.

Operator identifiers:

- Composed of operator characters (most punctuation marks).
- May not include: letters or digits, or `( ) [ ] { } ' " _ . , ; , ` `
- Operators that end in a colon (`:`) are right-associative, all others are left-associative.
- Examples: `+  =>  <?>  :::`

Mixed identifiers:

- Composed of an alphanumeric identifier, an underscore, and an operator identifier.
- Examples: `unary_+` , `myVar_=`

Literal identifers:

- Composed of any arbitrary string enclosed in backticks (the `` ` `` character).
- Examples: `` `Hello, World!` ``, `` `class` ``

Declarations

All variables must be declared, with either `var` (if the value may change) or `val` (for constants). In Scala, `val` is preferred, unless you have good reason to use `var`. If given an initial value in the declaration, the variable's type is inferred and need not be explicitly stated (but it may be). If explicitly stated, the type comes after the variable and a colon, for example, `var q: Boolean = true`.

The types of variables must be declared:

- When the variable is the formal parameter of a method.
- When the variable is declared but not initialized; this can only occur within a class.

Type parameters (generics) are enclosed in square brackets, for example, `Array[Int]`.

Within a method, variables must be given an initial value. Variables within a class (and not within a method) may optionally be given an initial value.

When initial values are *not* given, `new` is required: `val ary = new Array[Int](5)`
and default values of zero, `false`, or `null` are used.
When initial values *are* given, `new` is not allowed: `val ary2 = Array(3, 1, 4, 1, 6)`

The type of a function is written with the double-headed arrow, `=>` or the Unicode version, `⇒`. Except in the case of a single argument, parentheses are required:

- A function of no arguments: `() => return_type`
- A function of one argument: `arg_type => return_type` **or** `(arg_type) => return_type`
- A function of two or more arguments: `(arg_type_1, ..., arg_type_N) => return_type`

Data types

Scala has no primitives, only objects.

- `Any`
  - `AnyVal`
    - `Boolean`, `Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double` (these are as in Java)
    - `Unit` (has only a single value, `()`; returned by functions that “don't return anything.”)
  - `AnyRef` (corresponds to `Object` in Java)
    - All java.* reference types
    - `ScalaObject`
      - All `scala.*` reference types, including `Array` and `List`
  - *`Null`* (bottom of all `AnyRef` objects)
- *`Nothing`* (bottom of `Any`)

Numbers

`Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double` are considered to be numeric types. The default type for integers is `Int`, and the default type for "real" numbers is `Double`. The less accurate forms are converted to `Int` or `Double` when arithmetic is performed using them.

All numbers are stored internally in a binary representation, but they may be written in various ways.

- Decimal **integers** use the digits `0` to `9`, with an optional `+` or `-` prefix.
- Decimal **long** integers are written as integers with an `l` or `L` suffix. Due to its resemblance to the digit `1`, use of the letter `l` is strongly discouraged.
- **Octal** integers use the digits `0` to `7`, beginning with a `0`, and an optional `+` or `-` prefix.
- **Hexadecimal** integers use the digits `0` to `9` and the letters `A` to `F` (may be lowercase), and an optional `+` or `-` prefix.
- Decimal **double**-precision numbers use the digits `0` to `9`, with an optional `+` or `-` prefix and an optional

exponent. The exponent consists of an `e` or `E`, an optional sign, and one to three decimal digits.

- Decimal **single**-precision numbers are written as an integer or double-precision number with an `f` or `F` suffix.

Operations on numbers include `+` (addition), `-` (subtraction or negation), `*` (multiplication), `/` (division), and `%` (modulus). Operations on integers also include `<<` (left shift), `>>` (right shift with sign extension), `>>>` (right shift with zero fill), `&` (bitwise and), `|` (bitwise or), and `^` (bitwise exclusive or).

Strings

A `String` may be enclosed in double quotes, `"..."`, or in triple double quotes, `"""..."""`. The latter is a raw string (that is, `\` does not "escape" characters) and may contain newlines.

If a triply-quoted string contains whitespace and the `'|'` character at the beginning of each line after the first, the method `stripMargin` will remove these characters. This can be use to prevent long strings in the code from messing up the code indentation.

A `Symbol` is an interned string. A symbol literal starts with a single quote, `'`, followed by a letter, then zero or more letters and digits.

Lists

Lists are the most valuable data type in any functional language, including Scala.

Much of the following information is from http://anders.janmyr.com/2009/10/lists-in-scala.html.

Lists are:

- Immutable and persistent. Operations that return lists do not modify the original list, but generally share structure with it.
- Homogeneous. All the elements of a list are the same type. The type of the empty list is `List[Nothing]`.

When it is necessary to specify the type of a list, write it as `List[`*`Type`*`]`. A literal list is written as `List(`*`value`*`, ..., `*`value`*`)`.

Fundamental (efficient) list methods are:

- *`list`*`.head` returns the first element in the list.
- *`list`*`.tail` returns the rest of the list, minus the head.
- *`value`* `::` *`list`* (`::` is pronounced “cons”) puts the value as the new head of the list. It is right-associative.
  - The empty list is written as `List()` or as `Nil`.
  - `List(1, 2, 3)` is equivalent to `1 :: 2 :: 3 :: Nil`.
- *`list`*`.isEmpty` checks if the list is empty.

Functions defined in terms of the above include:

- *`list`*`.length` returns the number of elements in the list.
- *`list`*`.last` returns the last element of the list.
- *`list`*`.init` returns a list with the last element removed.
- *`list`*`.take(`*`n`*`)` returns a list of the first n elements. Especially useful for lazy lists.
- *`list`*`.drop(`*`n`*`)` returns a list with the first n elements removed.
- *`list1`* `:::` *`list2`* appends the two lists.
- *`list`*`.reverse` returns a list in the reverse order.
- *`list`*`.splitAt(`*`n`*`)` returns the tuple `(`*`list`* `take` *`n`*`,` *`list`* `drop` *`n`*`)`.
- *`list1`*`.zip(`*`list2`*`)` returns a list of tuples, where the first element of the tuple is taken from *`list1`* and the second from *`list2`*. The length of the result is the length of the shorter list.
- *`list`*`.mkString(`*`string`*`)` converts each element of the list into a string, and concatenates them with the *`string`* in between elements.
- *`list`*`.distinct` returns a list with duplicated elements removed.
- *`listOfLists`*`.flatten` takes a list of lists of elements and returns a list of elements.

Higher-order functions

The following functions are described as operations on lists, but most of them will operate on many types of sequences.

- `list.map(function)` returns a list in which the function of one argument has been applied to each element.
- `listOfLists.flatMap(function)` returns a list in which the function of one argument has been applied to each element of each sublist. Removes one "level" of nesting.
- `list.filter(predicate)` returns a list of the elements of the given list for which the predicate is true.
- `list.filterNot(predicate)` returns a list of the elements of the given list for which the predicate is false.
- `list.partition(predicate)` returns a tuple of two lists: a list of values that satisfy the predicate, and a list of those that do not.
- `list.foldLeft(value)(_ function _)` applies the function to each pair of elements of `value::list`, using each function result as the new first argument to the function, and returns the final value.
- `(value /: list)(_ function _)` is an alternate way of writing `foldLeft`.
- `list.foldRight(value)(_ function _)` applies the function to each pair of elements of the list with the value appended, starting from the right end of the list, using each function result as the new second argument to the function, and returns the final value.
- `(list :\ value)(_ function _)`is an alternate way of writing `foldRight`.
- `list.find(predicate)` returns, as `Some(value)`, the first value in the list that satisfies the predicate, or `None` if no such value is found.
- `list.takeWhile(predicate)` returns a list of the values at the front of the given list that satisfy the predicate, stopping short of the first value that does not.
- `list.dropWhile(predicate)` returns a list omitting those values at the front of the given list that satisfy the predicate.
- `list.span(predicate)`returns the tuple `(list takeWhile predicate, list dropWhile predicate)`.
- `list.forall(predicate)` checks if every element of the list satisfies the predicate.
- `list.exists(predicate)` checks if any element of the list satisfies the predicate.
- `list.sortWith(comparisonFunction)` sorts a list using the two-parameter comparison function. `list.sortWith(_ < _)` sorts the list in ascending order.
- `list.groupBy(function)` returns a Map of *keys* → *values*, where the *keys* are the results of applying the function to each list element, and the *values* are a List of values in `list` such that applying the function to that *value* yields that *key*. For example, `List("one", "two", "three").groupBy(x => x.length)` gives `Map(5 -> List(three), 3 -> List(one, two))`.

Tuples

A *tuple* consists of from 2 to 22 comma-separated values enclosed in parentheses. Tuples are immutable. To access the *n*-th value in a tuple `t`, use the notation `t._n`, where `n` is a literal integer in the range 1 (*not* 0!) to 22. There may be a spaces around the dot, but not between `_` and `n`.

Sets

Sets are immutable by default. Mutable sets may be imported from `scala.collection.mutable`. There are a great many operations on sets, of which `contains`, `isEmpty`, `intersect`, `union`, and `diff` are just a few.

Maps

Maps are immutable by default. Mutable maps may be imported from `scala.collection.mutable`. A map is represented as a list of pairs, that is, `Map((key, value), ..., (key, value))`. A more expressive syntax for writing the same thing is `Map(key -> value, ..., key -> value)`.

The basic operations on maps are (1) getting the value associated with a key, and (2) for mutable maps, associating a value with a key. Most list operations can also be used on maps.

- `map.get(key)` returns (as an `Option`) the value associated with the key.
- `map.getOrElse(key, default)` returns the value associated with the key, or if the key is not in the map, the default value.
- `map.put(key, value)` for mutable maps only, associates the value with the key, and returns (as an `Option`) the old value.

Options

Options are used when an operation may or may not succeed in returning a value. Options are parameterized types, so one may have, for example, an `Option[String]` type. The possible values are `Some(value)`, where the value is of the correct type, or `None`, for the case where no value has been found.

Although a few operations are defined for `Option` types, it is far more common to use a `match` expression to extract the value, if one exists.

Arrays

Arrays are mutable. The advantage of arrays is fast access to random elements. One disadvantage is that the operations `==` and `hashCode` use the Java definitions, that is, identity rather than equality. Unless random access is needed, consider using Lists instead.

An array may be created by listing its values:

- `val ary = Array(2, 3, 5, 7, 11)`

or by explicitly giving its type, and using new to indicate its size:

- `val ary: Array[Int] = new Array(100)`

The basic operations on arrays are getting a value from it, storing a value in it, and getting its length. Note the use of parentheses instead of brackets.

- *`array`*`(`*`n`*`)` returns the *n*-th value (0-based) in the array.
- *`array`*`(`*`n`*`) = `*`value`* puts the *value* in the *n*-th location (0-based) of the array.
- *`array`*`.length` returns the number of elements in the array.

Multidimensional arrays

To create a multidimensional array, `import Array._`, then call either

- `ofDim[`***`element_type`***`](`***`dimensions`***`)`. or
- `Array.fill(`*`dimensions`*`)(`*`value`*`)`

The elements of a multidimensional array can be accessed using the syntax ***`arrayName`***`(`***`firstIndex`***`)(`***`secondIndex`***`)...(`***`lastIndex`***`)`.

Expressions

If expressions

- `if (`*`condition`*`) `*`expression`*
- `if (`*`condition`*`) `*`expression`*` else `*`expression`*
- `if (`*`condition`*`) `*`expression`*` else if (`*`condition`*`) `*`expression`*` ...`
- `if (`*`condition`*`) `*`expression`*` else if (`*`condition`*`) `*`expression`*` ... else `*`expression`*

The value of the if expression is the value of the expression that is chosen; or, if no condition is satisfied and there is no `else` clause, the value is the `Unit` value, `()`.

For expressions (known in other languages as list comprehensions)

The *for expression* is used to create and return a sequence of results. The syntax is `for (`*`seq`*`) yield `*`expression`*, where:

- *`seq`* is a semicolon-separated sequence of *`generators`*, *`definitions`*, and *`filters`*, beginning with a generator.
  - A *`generator`* has the form *`variable`*` <- `*`list`*, where the list may be any expression resulting in a list.
  - A *`definition`* has the form *`variable`*` = `*`expression`* (the keywords `var` and `val` are not used here).
  - A *`filter`* has the form *`if condition`*.
- Where possible, the type of the result will be the same type as the *`seq`*.

Examples:

- `for (v <- list) yield v` returns a copy of `list`.
- `for (v <- list) yield f(v)` is equivalent to `list map f`.
- `for (v <- list if p(v)) yield v` is equivalent to `list filter p`.
- `for (v <- lst; w = 2 * v; x <- lst; if x < w) yield (v, w, x)`

Match expressions

Syntax:

```
expression match {
    case pattern1 => expression1
    case pattern2 => expression2
    ...
    case patternN => expressionN
}
```

The `match` expression uses pattern matching to select a case, then the value of the `match` expression is the value of the corresponding *expression*. Patterns may be:

- A **literal value** of the same type as the *expression*, or a subtype of it.
- A **variable**, which will match any value (and may be used in the corresponding $expression_i$).
- An **underscore**, which will match anything. This is commonly used as the last "catch-all" pattern.
- A **sequence**, such as `List(a, _, c)`, where the components are patterns.
- A **tuple**, where the components are patterns.
- An **option type**, `Some(x)` or `None`.
- A **typed pattern**, such as `s: String` or `m: Map[_, _]`. (For parameterized types such as `List` and `Map`, you cannot specify the types of their components.)
- The name of a **case class**, where patterns are used in place of the constructor parameters.
- A **regular expression**. Regular expressions are not covered here.
- Any of the above followed by a **pattern guard** (the word `if` followed by a Boolean expression), for example, `n: Int if n > 0`.

Cases are tried in the order in which they occur. When a case is selected, the code for that case, and only for that one case, is executed.

Statement types

Scala does not have "statements," it has *expressions*, and every expression has a value. However, some expressions are executed purely for their side effects, and return the `Unit` value, `()`, which is essentially meaningless. Such expressions are basically the same as statements in other languages.

Syntax	Example	Comments
`{ expressions }`	`{ temp = a; a = b; b = temp }`	A *compound expression* consists of one or more expressions enclosed in braces. The value of a compound expression is the value of the last expression evaluated within it; all other expressions are evaluated only for their side effects. The value of a compound expression may or may not be Unit.
`var name: type = value`	`var abc = 5`	Variables must be given initial values, except when within a class.
`var name: type`	`var abc: Int`	Allowable within a class.
`val name: type = value`	`val five = 5`	`val` declares a constant, which must be given a value; explicit type information is allowed.

		but redundant. A `val` name for an object is constant in the sense that the name cannot be reassigned; but the contents of the object may be modified.
name = *expression*	abc = 10	Values may be assigned only to `var` variables.
name += *expression*	abc += 1	All the various *op*= operators are allowed. `++` and `--` do not exist in Scala.

if (*condition*) *expression*
else if (*condition*) *expression*
else *expression*
	if (x < 0) {
	  println(x)
	  x = -x
	}
	else x = x + 1
		Zero or more `else if` clauses may be used, and the final `else` is optional. Braces can be used to group several expressions into a single compound expression. The value of the `if` is the value of the last expression evaluated within it (which may be Unit).
		
		If there is no final `else`, the *type* of the expression is `Any`, regardless of whatever value it may have.

while (*condition*) *statement*	var x = 1 while (x < 1000) x *= 2	Nothing unusual to note. However, Scala has no `break` or `continue` statements.
do { *statements* } while (*condition*)	var x = 1 do { x *= 2 } while (x < 1000)	Nothing unusual to note. However, Scala has no `break` or `continue` statements.
for (*variable* <- *list*) *statement*	for (x <- ary) println(x)	The parenthesized expression may contain tests, assignments, and additional generators (*variable* <- *list* expressions).
for (*variable* <- *min* to *max*) *statement*	for (i <- 1 to 10) println(i)	`to` is a method of `RichInt`. `to` is inclusive; `until` excludes *max*.
for (*variable* <- *min* to *max* if *condition*) *statement*	for (i <- 1 to 15 if i % 2 == 0) print(i)	`for` comprehensions may have *guards*.
return *expression*	return true	Returns a value from a method. Use of `return` requires that the return type of the method be explicitly stated.
throw new *Exception*	throw new IllegalArgumentException	As in Java.
try { *expressions* } catch {	try { val f = new FileReader("input.txt")	The *Exception* is the exception type, and the

`  case name: Exception => {...}`
`  ...`
`  case name: Exception => {...}`
`} finally {...}`
	`} catch {`
	`  case ex: FileNotFoundException => {`
	`  println("Missing file exception")`
	`  }`
	`  case ex: IOException => {`
	`  println("IO Exception")`
	`  }`
	`  finally {`
	`  println("Exiting finally...")`
	`  }`
	`}`
		name is a variable holding the exception that occurred.
		
		As in Java, you should have one or both of `catch` and `finally`; usually you won't want both.
		
		This example is from TutorialsPoint.

Methods

Syntax:

- [override] [access] def *methodName*(*arg*: *type*, ..., *arg*: *type*) { *body* } // returns Unit
- [override] [access] def *methodName*(*arg*: *type*, ..., *arg*: *type*): *returnType* = { *body* }

If overriding an inherited method, the keyword `override` is required.

access may be `private`, `protected`, or public by default.

The type of each argument must be specified explicitly.

The return type of a method can usually be omitted; it must be declared:

- When there is an explicit `return` statement.
- When the method is recursive.
- When the method calls another method with the same name.
- When the inferred return type is more general than desired, for example, `Any`.

For reasons of clarity, it is usually best to declare the return type.

Braces around the body may be omitted if the body consists of a single expression, whose value is to be returned.

Methods may be overloaded, as in Java.

The parameters of a method or function are treated as if they were defined by `val`; they cannot be reassigned a new value.

Function literals

Syntax: `(arg1: Type1, ..., argN:TypeN) => expression`

Passing function literals as arguments

Within an enclosing argument list, the parentheses around the parameter list can usually be omitted.
Example: `"aBcDeF".map(x => x toLower)`

If each parameter is used only once, and the parameters appear in the expression in the same order as in the parameter list, the parameter list may be omitted, and underscores may be used in the expression.
Example: `"aBcDeF".map(_ toLower)`

Passing methods as functions

Methods can often be passed as if they were functions, for example, `"abc" map println` works. In other cases, the method name must be followed by an underscore to convert it to a "partial function," for example, `"abc" map (println _)`.

Methods are functions that belong to objects, and as such, can manipulate the fields of their object. Methods that do this should *not* be used as functions.

Program structure: Classes, Objects, Traits and Scripts

Scripts

A *script* is a list of commands that can be interpreted directly by the Scala system. Scala wraps the script in a `main` method and executes it immediately. The syntax for running a script is simply `scala myScript.scala`.

Classes

A *class* describes objects, as in Java. A class may extend one other class, and it may include any number of traits. A class has a primary constructor, which is part of the class declaration itself.

`class Foo { ... }`
: By default, a class extends `AnyRef`.

`class Foo(name: String) { constructor_body }`
: The primary constructor is part of the class header. Parameters are put after the class name, and the constructor body is the entire class body.

`class Foo(a: Int, var b: Int, val c: Int) { ... }`
: Constructor parameters are handled as follows. Automatically, all parameters are automatically declared as private fields; `a` has private getters and setters; `var b` has public getters and setters; and `val c` has a public getter. A setter method for a parameter `v` has the name `v_=`.

`def this(parameters1) { this(parameters2); ... }`
: Auxiliary constructors are defined with `this`; the first statement in the auxiliary constuctor must be a call to some other constructor.

`class Foo(a: Int, b: Int) extends Bar(a + b) { ... }`
: When extending a class, any base class (=superclass) parameters are provided immediately.

`class Dog(name: String) extends Animal with Friend { ... }`
: A comma-separated list of traits can follow the keyword `with`. Any uninitialized `var`s and `val`s defined by the trait must be initialized by the class.

Case classes

A *case class* is defined by adding the keyword `case` before the word `class`. This results in additional methods being added to the class.

- Scala provides an `apply` method, which allows you omit the word `new` when constructing a new object of the class.
- Scala provides an `unapply` method, which allows you use case classes in pattern matching.
- Getters and setters for the constructor parameters are provided.
- Scala provides a nicer `toString` method and `==` and `hashCode` methods, based on the constructor arguments.

Objects

In addition to creating objects from classes, you can declare *objects*. An object is declared like a class, but with the keyword `object` instead of `class`; also, an object cannot take parameters. Whereas a `class` declaration describes a blueprint for objects, an `object` declaration declares a single object.

A program, other than a script, must contain an object with a `main(args: Array[String])` method.

When you define a class and create objects from it, you use the keyword `new`:

```
scala> class Dog(name: String)
  defined class Dog

scala> val a = new Dog("Fido")
  a: Dog = Dog@a80370d
```

But when you create an object "literal", you don't use `new`, because these are defined as case classes.

```
scala> val t = Tuple(1, "apple")
t: (Int, java.lang.String) = (1,apple)

scala> val l = List(1, "apple")
l: List[Any] = List(1, apple)

scala> val s = Set(1, "apple")
s: scala.collection.immutable.Set[Any] = Set(1, apple)
```

Traits

Traits are declared like classes, used like Java interfaces, and may contain fully-defined methods. In the declaration of a class (or object), if the superclass is mentioned, the syntax is class *ClassName* extends *SuperClass* with *Trait1*,...,*TraitN*, otherwise the syntax is *ClassName* extends *Trait1* with *Trait2*,...,*TraitN*

Input/Output

Console

```
import scala.io.StdIn.readLine

val data = readLine // Reads a single line (as a String) from the Console
val data = readLine(prompt) // Prints the prompt, then reads a line
```

Note: In the REPL, readLine does not display the text being entered.

```
print(data)    // Prints to the Console without a newline
println(data)  // Prints to the Console with a newline
println        // Just prints a newline
```

Text files

Finding a file to read or write

Scala.swing is no longer supported.

```
import scala.swing.FileChooser, java.io.File
val file = new File(fileName)   // with a relative or absolute path
val chooser = new FileChooser
val response = chooser.showOpenDialog(null)  // to read a file
val response = chooser.showSaveDialog(null)  // to write a file
if (response == FileChooser.Result.Approve) { val file = chooser.selectedFile }
```

Reading from a text file

```
import scala.io.Source, scala.io.Source._
val stream = Source.fromFile(file or fileName)
for (char <- stream) { process each character }
for (lines <- stream.getLines) { process each line }
val all = stream.getLines.toList         // as a list of strings
val all = stream.getLines.mkString       // as a single string
val all = stream.getLines.mkString("\n") // with newlines
stream.close
```

Writing to a text file

```
import java.io.PrintWriter
val stream = new PrintWriter(file or fileName)
stream.print(data)
stream.println(data)
stream.println
stream.close
```

Scalatest

It is extremely important to have compatible versions of the software, otherwise you may get errors or simply have the program refuse to run.

Here are two setups that have worked for me:

Fall 2015	Fall 2016

• Scala IDE 4.0.0-vfinal-2014 (Juno) • Scala Library container [2.11.4] • JRE System Library [JavaSE-1.7] • scalatest_2.11-2.2.1.jar	• Scala IDE 4.4.1-v-2016e (Neon) • Scala Library container [2.11.8] • JRE System Library [JavaSE-1.8] • scalatest_2.11-2.2.4.jar

And a working test file:

```
package fractionTest

import org.scalatest.FunSuite
import Fraction._

class ExampleTests extends FunSuite {

  val f1 = Fraction(1, 2)
  val f2 = Fraction(1, 4)

  test("Test multiplication") {
    assert(f1 * f1 == f2)
  }

  test("Test bad multiplication") {
    assert(f1 * f2 == f2)
  }
}
```

GUI Programs

While Java Swing can still be used from Scala, Java is moving from Swing to JavaFX, and Scala is following suit. Scala Swing is no longer being developed and is probably incompatible with current versions of Scala.