

Fundamentals of Scala

Notes

Background

Scala is a programming language that can run on a JVM and generate and use class files compatible with Java code, allowing extensive interoperability with Java code and libraries. Additionally, a closely related project, Scala.js, provides a compiler, and supporting libraries, that convert Scala source-code into JavaScript, enabling code written in Scala to run in a browser.

Scala supports both object oriented and functional programming paradigms, and has a syntax that borrows elements from the “curly-brace” family of languages. That is, its syntax has similarities with C, C++, Java, JavaScript and others. However, since it adds extensive functional capabilities not provided by those languages, many of the syntax features that support those capabilities are unfamiliar. While Scala 2.13 is perhaps the most commonly used variant of the language, Scala 3 introduced some syntax variants that can simplify code. Perhaps most notably, the ability to control block scoping using indentation (as with Python) rather than with curly braces. Fortunately, almost all code written in Scala 2.13 remains valid in Scala 3. Key Scala 3 syntax variations are described together at the end of these notes.

Interpreted and compiled modes

- The command `scalac` may be run as a compiler, building binary classfiles, and resulting programs run using the `scala` command

```
$ scalac Hello.scala
$ scala Hello
Hello world!
$
```

- The `scala` command by itself enters a “REPL” (read, execute, print loop) allowing interactive experimentation

```
$ scala
```

```
Welcome to Scala version ...
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala>
```

- Scala code can be entered interactively at this point:

```
scala> println("Hello Interactive World!")
```

```
Hello Interactive World!
```

```
scala>
```

- IntelliJ and other IDEs typically provide support for this interactive mode without the need for spawning a new command / terminal window.
- The interpreted mode can be used to create shell scripts / batch files, though this is inevitably a somewhat operating system-dependent process. In a Unix environment, a bourne or bash script might look like the following. Note that the `#!` characters introduce an interpreter binary path, and are a feature of those shells rather than of Scala. By contrast, the `!#` marks the end of the preamble for Scala. Everything that follows the `!#` is treated as Scala script:

```
#!/usr/local/scala-2.12.4/bin/scala
```

```
!#
```

```
val x = 99
```

```
println(s"Value is ${x}")
```

Basics

Some immediate comparisons/contrasts with languages like Java, C/C++ are:

- Scala source files which may contain many top-level elements; packages, classes, and objects.
- Source filenames and directories are not required to map to class and package names, though it's a good convention to adhere to anyway, since it can assist in finding things during maintenance.
- Scala uses packages which can be declared with a statement at the top of the file, or using a block-syntax. For example:

```
package mypackage {  
  // package contents  
}
```

- An `import` statement can be placed anywhere. They are block scoped, and can import anything, including object references, fields, and methods. An underscore is used as a wildcard, and braces can group a number of specific items to be imported from a particular source. For example:

```
import math._  
import math.{Pi,E}
```

In Scala 3, the underscore for import wildcards is replaced with an asterisk, in the manner of Java.

- Many identifiers in Scala appear as though they're part of the core language, but are in fact imported definitions. In Scala an `import` statement can select almost any identifier, even down to methods and fields inside objects. An imported element is available unqualified.
- A large number of such identifiers are pre-imported via a standard import of an object called `scala.Predef`. The `Predef` object supplies

many features, including `Console.out.println`, which is typically used as simply `println`.

- Note that these seemingly magical imports are common, are frequently provided for parts of the standard API and third party libraries, and can be added to your own libraries.
- Semicolons are not generally required unless multiple statements are on a single line. Because of this, be sure that when wrapping a line, the first and second parts do not form legal syntax individually, or they will be parsed that way. For example:

```
def value = 3
val x = 4
  + value
println("x is " + x)
```

Prints `x is 4`

Note: Scala 3 changes to a less flexible (less ambiguous) syntax, and is often controlled by indentation. In that version of the language, this example will print 7. However, if a blank line is placed between the `val x = 4` and the `+ value`, it would again print 4 without reporting any errors.

- Comments are as Java etc.
- The default accessibility in Scala is what other languages commonly refer to as "public". In Scala "public" is *not* a keyword.
- Scala does not have "static" features in a class, nor is `static` a keyword. Instead values and methods that are related to the concept represented by a class but are not related to any particular instance are represented in a "companion object". This is expressly declared as `object` rather than `class`. The object is in the same package—and is given the same name as—the companion class. Classes and objects do not have to exist as companion pairs, but either can be declared individually if desired.

- The `main` method that serves as the entry point to a Scala program is declared in an object. For example:

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

- Scala can, and most commonly does, run on a JVM, in which case the full suite of Java libraries, both core and installed third party ones, may be used directly. Note, however, that if this is done, such code might not execute successfully on other runtime systems (notably Scala.js)

Forms of Identifiers

Scala identifiers take three basic forms

- Letter, letters+digits (\$ and `_` count as letters, but \$ is reserved for compiler-generated ids). Identifiers of this group are broadly familiar to most programming languages including Java. For example:

```
count
_count_of_3
```

- Identifiers of this form must not be identical to Scala keywords
- Sequence of “operator characters” (no embedded alphanumerics). Operator characters are printable characters in the 7 bit ASCII set, excluding alphanumerics, parentheses, brackets, braces, any quotation marks, period, comma, and semicolon. For example:

```
# ::
```

```
->
```

```
^&*%
```

- Almost anything between backquotes, for example:

```
`😊 smiling happy faces !`
```

- Note that the backquotes are not part of the identifier
- This form is primarily intended to allow interoperability with Java which exports some identifiers that are Scala keywords, and must be wrapped in this way. For example:
`java.lang.Thread.`yield``

Types, Literals

- Scala types all mimic pure objects (though the underlying implementation might use primitives for efficiency)
- For example, basic types include: `Boolean`, `Int`, `Float`, `String`
 - Note that the types that map to primitives have capital first letters
- The root type in Scala's hierarchy is `Any`. In Scala 2, this has two immediate subtypes `AnyRef` and `AnyVal`, . In Scala 3, an intermediate type, `Matchable`, exists directly below `Any`.
 - `Any` serves as the most general type (much as `Object` does in Java, except it can refer to the primitive-like types too)
 - `AnyRef` maps directly to `java.lang.Object`
 - `AnyVal` is the parent class of the primitive-like types. In addition Scala allows user defined value types to be defined subclassing this. In such a situation, the class must define exactly one field of primitive type. The type is named, is not the same type as the primitive it is built on (which can be useful for type safety), and can contain methods as normal.
- `Nothing` is a special type that is assignment compatible to all types (it's the "bottom" of the type hierarchy, a subtype of every other type). It can be used to represent the absence of a value where one might be expected (for example a method that throws an exception instead of returning a value.)
- The empty pointer value is `null` , the type of this is `Null`

- Literal forms for numbers are largely as expected. As with Java, default types are `Int` for integral types, and `Double` for floating point types. For example:

```
123
1.23
1.23E+4
1234567890123L
1.23F
```

- Notes:
 - Literals of integral types must not begin with redundant zeroes (this avoids the accidental octal problem often found in C-derived languages).
 - Binary and octal forms are not supported
 - Underscore grouping separators are only supported since Scala 2.13.
- Simple character and String literals are similar to Java and related languages. A character is surrounded with single-quotes, Strings with double-quotes. These literal forms use backslashes for escapes and Unicode. For example:

```
'X'
'\u96E5'
"Hello\n"
"Hello\\not a newline"
```

- A String literal can be created such that special character handling is disabled using the prefix `raw` immediately prior to the opening double quote mark
- `raw"Hello\not a newline"`
- Or using three double quotation marks. In this case, a string literal can contain newline characters directly:

```
"""
Hello there
```


Line 2!"""

- Note that if the string is indented for formatting in the source, those indent characters will also form part of the string. This can be avoided using the vertical bar character and the “stripMargin” operation:

```
"""Hello
  |Line 2
""".stripMargin
```

- Scala provides range types, These can be created with constructor, but also with literals using the words `to` (denoting an inclusive range) and `until` (denoting a range that excludes the upper bound). A step size may be specified using the word “by”. For example:

```
1 to 10
1 until 10
1 to 96 by 5
1 to -10 by -2
1 until 96 by 5
```

Variable Declarations, Type Specification, and Inferencing

Three main forms of “variable” declaration are provided, `val`, `def`, and `var`

- `val` associates an identifier with a value, which must be assigned exactly once
 - This is not a “constant” if the object referred to is mutable
- `def` declares an association between an identifier and some means of obtaining a value. This might be a function/method, or a value. This can also be an abstract association that is satisfied in subclassing (by either a `val` or a `var`)
- `var` associates an identifier with a mutable value. Note that “good Scala style” encourages the functional approach, which prefers

immutable values. This guidance is particularly emphasized where a value's scope is broader, so method locals might be permitted to be mutable, while object fields are more likely immutable.

- In interpreted mode a `val` might behave as if it's mutable, but this is because the interpreter permits re-declaring the identifier.
- Declarations use `val`, `def`, or `var` to introduce the declaration, followed by the identifier.
- The type may be specified after the identifier by placing a colon followed by the typename.
- An assignment that initializes the value (or provides the body of a function) may follow.
- Where a variable is initialized, the type may often be inferred and omitted from the source.
- Example:

```
val count = 3 // inferred to be Int type
```

```
val count: Int = 3 // explicitly Int type
```

- A `val` declaration may not be changed after assignment
- Variable declarations are one per statement, the comma separated list of declarations that's common in other languages is not supported:

```
var x = 2, y = "Hello" // NOT LEGAL IN SCALA
```

Expressions

- Basic arithmetic expressions have forms and operators that are familiar from C/C++/Java. In these, infix operators along with type inferencing and promotions that are generally unsurprising. For example:

```
val x = 2 + 9.2 // x is a Double, value 11.2
```

- The plus operator '+' performs String concatenation and conversions of non String operand types in mixed-type operations

- Less familiar is that *blocks* also generally have an expression value. They adopt the value of the last expression in the block. For example:

```
// b gets the value 4, and Hello is printed
val b = { println("Hello"); 7 } - 3
```

- The `if/else` construct is a conditional *expression*, similar to the ternary operator in C/C++/Java etc. As before, blocks have value. For example:

```
val message = "There are " +
  (if (count > 3) "many" else "a few") + " items"
```

- As with other languages, it's often advisable to use braces for the `if` and `else` subordinate expressions if they are placed on separate lines
 - As with Java, but different from C/C++/JavaScript, the expression to be tested must be a `Boolean` expression.
- Scala 3 allows a variant form of `if/else` which uses the word `then`, instead of parenthesizing the expression being tested.
- Quoted strings can be variable expressions, evaluated at runtime, that describe substitutions (known as "string interpolation") based on expressions embedded in the string. Two forms exist, defined by prefix letters placed immediately prior to the opening double-quote mark. The letter may be `s`, or `f` (both are lowercase). The letter `s` performs a simple expression substitution, whereas using `f` allows a formatted substitution comparable to a `printf` type effect. The expression is prefixed with a dollar symbol and enclosed in braces (like a Unix shell variable) if it is other than a simple identifier. For example:

```
val x = 9; val y = 3
println(s"$x is divisible by $y " +
  s" is ${x % y == 0}")
println(f"3--more precisely ${math.Pi}%7.2f")
```

- Operators in Scala are actually methods on objects, and methods on objects can generally be invoked using an infix syntax (regardless of whether those methods have symbolic or textual names). For a single argument method this translates the first form directly to the second form shown here:

```
val s = "Hello, World!"
val pos = s.indexOf('W') // pos is 7
val pos2 = s indexOf 'W' // equivalent
```

This feature is often used to create the impression of new keywords in the language, or to define domain-specific languages (DSLs).

- Scala allows methods to have symbolic names, and thereby allows programmer-defined operators and operator overloading. Because arbitrary new operators may be invented, the precedence rules are a little more complex than in other languages. Generally, operator precedence is based on the first character of the operator name.

Highest precedence

Special characters not otherwise mentioned

* / %

+ -

:

< >

= !

&

^

|

All letters

Lowest precedence

An exception to this is that assignment operators (+= etc.) take very low priority

- Symbols ending with a colon (for example :: and +:) are right associative when used in infix form.

Arrays, Lists and Tuples

Arrays

- Carry their types as a generic type argument
- May be created with a capacity using
`new Array[Int](count)`
- May be created with immediately specified contents using
`Array[String](item1, item2)`
- May be created with an inferred base type if assigned to an explicitly typed variable, or from the types of the initialization elements:

```
val ar1 = Array("Fred", "Jim") // Array[String]
```

```
val ar2:Array[CharSequence] =  
  Array("Fred", "Jim") // Array[CharSequence]
```

- Elements are accessed at subscripts using parentheses, not square brackets
- Array size, which is fixed after creation, may be determined using either of the elements `length` or `size`

```
val empty: Array[Int] = new Array[Int](5)  
println("size is " + empty.size) // 5  
println("first element is 0 is " + empty(0))  
empty(0) = 99  
println("first element is 0 is " + empty(0))
```

```
val names: Array[String] = Array("Fred", "Jim", "Sheila")
```

- Reading an array uses the `apply(index : Int)` method behind the scenes, this is why Scala uses parentheses, not square brackets for subscript access

- Assignment to an array element uses the method

```
update(index: Int, x: Element)
```

- In general, any application of parentheses directly to an *object* rather than a method will invoke the apply or update method on that object. This is a commonly used syntax shortcut.

Lists

- Generally, Scala's List type is an immutable singly-linked structure, although mutable variants, and other types of sequential structure, exist and may be imported explicitly.
- May be constructed using

```
List("Fred", "Sheila")
```

- May also be constructed using the "cons" operator:

```
"Fred" :: "Sheila" :: Nil
```

Note that `Nil` is a special list element that denotes the end of the list, and `::` is a right-associative operator which is equivalent to "prepend".

This form is exactly equivalent to (but vastly preferred over!):

```
Nil.::("Sheila").::("Fred")
```

- Are commonly accessed via their `head` element and "the rest" via `tail`
- Item-at-index is extracted as with arrays using subscripts in parentheses
- Changes result in a new List

```
val ln = List("Harry", "Sally", "Pally")
println("List is empty? " + ln.isEmpty) // false
println("List contains " + ln.length) // 2
println("Head of list is " + ln.head) // "Harry"
println("Tail of list is " + ln.tail)
// List("Sally", "Pally")
```

Tuples:

- Tuples allow for grouping multiple items in one "bucket" without declaring and instantiating a special class and object.
 - Tuples use a simple parentheses type syntax, which can be on the left side of an assignment.
 - Tuple structures may appear on the left side of an assignment which results in assignment of all the named fields at once. (Although perhaps confusing, this can work with assignment of actual parameters to a single formal parameter of tuple type in a function call).
 - Tuple elements are accessed with a dot syntax, using names `_1`, `_2` etc. The numbers indicate the position of the item being addressed—note these indexes are one-based, not zero-based.

```
val t1 = ("Fred", 12, true)
println(t1._1) // "Fred"
val (a, b, c) = (9, math.Pi, "Ostrich")
println(s"values are a = $a, b = $b, c = $c")
```

- Scala 3 allows tuples to be accessed using parentheses and a zero-based subscript, like arrays.
- Tuples are instances of `Product<n>` traits, these allow:
- Determining the number of elements
- Accessing an element by its computed position
- Obtaining an `Iterator` over the elements

Defining classes

Scala fully supports the notion of classes, which may be defined in hierarchies, as abstract or concrete, and in a special form called a “case

class”. Note that most of the language features described in this section have indentation-structured variations in Scala 3.

- A simple class with no fields or behaviors may be declared like this:

```
class VerySimple
```

- Note that classes, fields, and methods are “public” by default and Scala does not have a `public` keyword

Fields

- The most basic syntax for a field in Scala embeds the declaration in the body of the class:

```
class Person {  
  var name:String = _  
  var address:String = _  
}
```

In this example, the fields are publicly accessible, and mutable.

- Fields must be initialized (though this can be performed in a constructor).
- The underscore explicitly allows the JVM default initialization (which is to zero-like values) to be applied to a `var`.
- If a `val` field is declared, it must be explicitly initialized (remember that a `val` must be initialized exactly once).
- Scala style suggests immutable objects, so `val` fields are common.

Constructors

- The body of the class amounts to the primary constructor, code within it is executed from top to bottom (defining methods and fields as it goes).
 - Arguments to the “primary” constructor are declared after the classname

- Constructor accessibility (which can be reduced) prefixes the argument list

```
class VerySimple private(x: Int) {  
  println(s"Running constructor with arg $x")  
  val myX = x  
}
```

- If a constructor argument is to be copied directly to a field, the argument can be declared as a field directly in the constructor argument list. This is done by adding either a `val` or `var` prefix:

```
class VerySimple(val x: Int) { ...
```

Auxiliary constructors

- Auxiliary constructors are overloaded constructors taking differing argument lists. They must invoke one other constructor first using a `this(arguments)` syntax.
- Accessibility of main and auxiliary constructors may be specified independently

```
class VerySimple private(val x : Int) {  
  def this(x:Int, s:String) = { // not private  
    this(x) // delegate to primary constructor  
    println(s"Auxiliary constructor s = $s")  
  }  
  println(s"Running constructor with arg $x")  
  ...  
}
```

Methods / Functions

- The basic form of a function or method declaration is:

```
def name(arg: ArgType, arg2: ArgType) : ReturnType =  
  body-expression
```

- The parentheses and argument list may be omitted if no arguments are expected.
 - In this case, the invocation *must not* use parentheses either
 - If empty parentheses are used on the declaration, parentheses are optional on the invocation in Scala 2. In Scala 3 the presence or absence of parentheses in the invocation must match the declaration.
 - By convention, you are encouraged to use parentheses if your method mutates internal state or has side-effects. By contrast, omit them if the method is a pure function returning a constant value (e.g. the element count of an immutable data structure).
- The return type may be omitted if:
 - The return type can be inferred from the `body-expression`, and
 - The function is *not* recursive
 - However, it's considered poor style to omit the return type from public methods
- The `body-expression` is an expression that computes the value to be returned by the function call
 - Commonly, this is presented as a block (enclosed in curly braces), but any single expression is legal
 - The expression value of a block is the last expression in the block
- Scala has the keyword `return`, but is generally *not* used. If a method must return before the end of its defining block, then it would be appropriate to use `return`. However, such a premature completion might be considered stylistically poor in its own right.
- Functions may be declared in objects, classes, and other functions. Those declared in objects and classes are generally referred to as methods. Those declared in functions will have closure access to local values in the enclosing method. Closures can capture `val` or `var` locals.

- Scala does not have “static methods” instead such behaviors may be declared in a companion object.
- Methods can use operator-like identifiers for their names.
- Method overloading is supported—the type sequence of overloaded methods must differ.
- Method overriding in subtypes is supported, but the keyword `override` is required
- Scala uses the `this` keyword similarly to other languages as a reference to the object on which a method was invoked.

Example methods:

```
// explicit return type
def sayHello(name : String) : Unit =
    println(s"Hello $name")

// return type inferred
def makeGreet(name: String, handle: String) =
    handle + " " + name

// block expression
def makeGreet(name: String, isFem: Boolean) = {
    val handle = if (isFem) "Ms." else "Mr."
    handle + name // "return value"
}

// override
override def toString: String =
    s"VerySimple, x is $x"
```

Variable Length Argument Lists

Variable length argument lists can be defined

- The variable part must be the last item in the list
- Indicate that the argument is variable-length by following the type with an asterisk.
- The received argument will be some kind of sequence, the precise type depends on the invocation.
- Simple invocation requires only comma separated arguments.
- Note that the single “fixed” argument `y` in this example is illustrative, not required

```
def addUp(y: Int, x: Int *): Int = x.sum + y
println(addUp(1, 2, 3)) // prints 6
```

- Variable length argument lists may be supplied in calls by sequence like structures
 - This is done using the `_*` syntax which “spreads” the values out to fulfill the variable length arguments.
 - This is likely to result in a different type for the receiver, but it will still be sequence-like

```
println(addUp(1, List(2, 3):_*))
```

- A variable length argument lists cannot take default values
- The spreading operation `:_*` only works when the target is a variable length argument list, it cannot spread across multiple individual arguments.

Default Argument Values

- Default argument values may be specified using an assignment, the equals sign comes after the argument’s type specification. The type specification is required even if type could be inferred from the default value:

```
def makeGreet(name: String, isFem: Boolean = true)
```

The methods `apply` and `update`

- A method named `apply` has special use. If defined, it can be invoked implicitly by placing parentheses directly after the reference to the object to which the method belongs.
- Similarly, a method named `update` is used in assignment situations:

```
class ArrayLike {  
  def apply(x: Int): Unit =  
    println(s"called apply with argument $x")  
  def update(idx: Int, x: Int): Unit =  
    println(s"updating index $idx to value $x")  
}
```

```
val e = new ArrayLike  
e(33) // "called apply with argument 33"  
e(33) = 101 // "updating index 33 to value 101"
```

Recursive functions and tail-recursion

- Recursion is the basic mechanism for iteration in a pure functional language, as such it's used quite frequently in Scala code
- A recursive function must explicitly declare its return type
- A recursive function might use a lot of stack space, possibly overflowing and crashing the thread. Scala avoids this if the function is "tail recursive"
- In a tail recursive function the recursive invocation is the very last thing performed by the function.
 - The Scala compiler will identify a tail-recursive function call and convert it into a loop.
 - Tail recursion can be more efficient in execution time.
 - Tail recursion will not overflow the stack no matter how many iterations are needed.

- If a function is intended to be tail-recursive, adding the annotation `@tailrec` will give an error if the compiler determines that it is actually not. This allows easy identification of potentially troublesome recursive methods.
- Converting a function to a tail-recursive implementation often involves adding an accumulator of some kind to pass intermediate results into the recursive call, rather than accumulating the result *after* the recursive call returns.

- This function is not tail recursive; the last operation is the addition, not the recursive `len()` call; the last operation:

```
def len(l:List[Int]): Int = l match {
  case List() => 0
  case h :: t => 1 + len(t)
}
```

- But the nested function here is correctly tail recursive as `accumulateLen()` is the last operation in that function:

```
def trLen(l:List[Int]): Int = {
  @tailrec def accumulateLen(acc: Int, l: List[Int]):
  Int = l match {
    case List() => acc
    case h :: t => accumulateLen(acc + 1, t)
  }
  accumulateLen(0, l)
}
```

Pass By Name Arguments

- Arguments passed to a function invocation are generally evaluated before the method invocation. However, using a “by name” parameter, evaluation can be delayed. This can be valuable if for example the argument might never be used, or to allow exception handling to be performed inside the method rather than at the call site:

```
// msg argument passed "by name"  
// not evaluated at invocation time  
def log(level: Int, msg: => String) : Unit = ...
```

Currying

Where a function takes multiple arguments, these are conventionally defined in a single argument list, but in Scala they can alternatively be presented as multiple argument lists. This is sometimes called “currying”. In this case, it’s usual to put a single argument in each list.

- These two functions are broadly equivalent:

```
def add(a:Int, b:Int): Int = a + b  
def addCurried(a:Int)(b:Int): Int = a + b
```

- Simple invocation follows the parentheses of the definition:

```
add(2, 2)  
addCurried(2)(2)
```

- Note that these particular two functions cannot have the same name, because the type and order of arguments is the same, they are not distinct overloads
- A key use of currying is to support “partial invocation”
 - Partial invocation creates a new function that takes fewer arguments than the original
 - The “missing” arguments have been pre-defined
 - In this example, the type of `addTwo` is expressly declared as “function taking single `Int`, returning `Int`”
 - In this example, the expressly defined type is required, or the compiler will object to “`addCurried(2)`” stating that more arguments are needed

```
def addTwo: Int => Int = addCurried(2)  
println(addTwo(4)) // prints 6
```

- In the absence of expressly defined type in the context, appending an underscore to the partially applied form provides an explicit way of requesting the curried function.

```
def addTwo = addCurried(2)_
```

- From a formal perspective, a curried function of the form

```
def addCurried(a:Int)(b:Int): Int = a + b
```

Is a function that takes an `Int`, and produces a function. That produced function takes an `Int` and in its turn it produces an `Int`.

- Compare that with the following example to which it is functionally identical:

```
def addCurried: Int => Int => Int =  
  (a:Int) => (b:Int) => a + b
```

- Note that if you “manually curry” your functions, you will not need to use the underscore syntax to extract the partially applied function:

```
def addTwo = addCurr(2)_ // for Scala currying  
def addTwo = addCurr(2)  // for manual currying  
// Note the underscore ^ or absence of it
```

- Multiple argument lists are often used to help the type inferencing system.
 - Inferencing works on each argument list individually, working through the argument lists from left to right.
 - Type information is passed from argument list to argument list, left to right, so it's often possible that a curried function works better with behavioral arguments (lambda expressions) than a regular one.
 - This is particularly effective if the lambda expressions are used in argument lists at the right of the group
 - Type inferencing has been improved in Scala 3, so a function with multiple parameters in a single argument list might type-infer

correctly with the Scala 3 compiler even though it is written using Scala 2 syntax and fails with that compiler.

Braces invocation

- Where a function takes a single argument in a set of parentheses, curly braces may be substituted for the parentheses
- This can create a kind of “psuedo-syntax”--that is a function invocation that looks more like a new language feature
 - This style is particularly common when using curried functions

```
def repeat(count: Int)(op: => Unit): Unit = {  
  for (i <- 0 until count) op  
}
```

```
repeat(3) {  
  println("hello")  
}
```

Infix invocation

Methods, regardless of the identifier used, can be invoked using a traditional dotted syntax or an infix form

- Infix notation mandates that the `this` object goes before the method name
- Infix notation is not practical where `this` is not available, such as for functions nested inside other functions
- Infix notation can be used when more than one argument is required; place the arguments in parentheses like a normal argument list

```
// given:  
class Value(val x : Int) {  
  def times(y: Int) = x * y  
  def twoArgs(y: Int, z: Int): Int = x + y + z  
}
```

```
val v = new Value(2)
// infix invocations
println("v times 3 is " + (v times 3))
println("v twoArgs (3, 4) is " +
  (v twoArgs (3,4)))
```

Case Classes

- Scala provides a syntax called a "case class" which allows a more concise syntax for declaring classes with certain common features.
- Case classes are particularly useful in (and get their name from) match expressions, but are entirely valid and useful in their own right
- A case class can be declared with nothing more than a constructor argument list, like this:

```
case class Truck(payload: Int, hardSide: Boolean)
```

- In this example, the `Truck` class has:
 - A primary constructor taking two arguments, and a factory method with the same argument list (this means that "new" is effectively optional—convention prefers *not* using new for these types when using the normal argument list).
 - Two `val` type fields (therefore immutable) with names `payload` and `hardSide`.
 - Standard methods `equals`, `hashCode`, and `toString`, implemented "as expected".
- Case classes can have their own bodies with methods, fields, and auxiliary constructors. They can also override the implementation of the implicit features such as `toString`.

Defining Subclasses

- A class may be defined as a subclass of one other class using the `extends` keyword.

- Note that case classes are prohibited from extending other case classes, but otherwise can take part in inheritance relationships.
- Scala permits only a single class parent, but traits support multiple inheritance of both state and behavior.
- Subclassing may be prevented by using the modifier `final` on the class.
- A class definition may carry the keyword `sealed`, in which case all subclasses must all be defined in the same source file. This permits some useful optimizations and makes it clear that only a specific range of subtypes are supported by the design.
- The parameter list for the parent constructor is appended to the parent classname in the extends clause.
- The parent may be initialized using any *one* of its constructors, either primary or auxiliary.
- Access to parent features is supported using the `super` keyword
- Parent features may be `protected` or `private`, but only `protected` features may be used from the child:

```
sealed class Base private (x: Int) {
  def this(x: Int, b: Boolean) = {
    this(x)
    println("In auxiliary base constructor")
  }
  override def toString: String =
    s"Base, x = $x"
}
```

```
final class Child(x: Int, y: Int) extends
  Base(x + 7, true) {
  override def toString: String =
    s"Child, y = $y, from ${super.toString}"
}
```

Abstract classes

- A class may be declared `abstract`, in this case `val`, `var`, and `def` elements may be left uninitialized (but must be explicitly typed). Any such element must be completed with a real implementation in any non-abstract derived class.
 - Using `def` is generally preferred for abstract elements, as it allows greater flexibility in the implementation.
- An abstract `val` element in a base class requires a concrete `val` implementation
- An abstract `def` element may be implemented using `def`, `var`, or `val`
- An abstract `var` element may be implemented by a `var` or a pair of `def` methods, one a setter, and the other a getter.
 - Getter is named the same as the `var`
 - Setter is the `var` name with `_ =` (underscore-equals) appended
 - The getter/setter approach allows, for example, an easy transition from a simple variable to a range-checked assignment while retaining simple assignment source semantics.

```
abstract class Base {  
  def a : Int  
  var p : Int  
}  
  
class Child(x: Int) extends Base {  
  val a = 9 // def implemented by val  
  def p = 3 // val implemented by get/set pair  
  def p_ = (x: Int) = println(s"Setting p to $x")  
  override def toString: String =  
    s"Child, x = $x, a = $a, p is $p"  
}
```

```
val c = new Child(3)
println(c.p) // prints 3
c.p = 99      // assignment prints 99
```

Traits

Scala provides a mechanism for multiple implementation inheritance, which is the “trait”.

- Traits are similar to abstract classes and also to interfaces in Java and similar languages.
- A trait cannot be instantiated directly, but must be used in the manner of parent class or generalization for defining other types
 - Including anonymous subclasses of the trait
- Traits can have concrete fields and behaviors as well as declaring abstract ones.
 - Identically named concrete, accessible, features from multiple parents will cause compilation errors, which addresses the "diamond problem" often associated with multiple implementation inheritance.
 - This can be resolved by explicitly declaring an override for the name in the type that combines the two indistinguishable elements.
- Traits cannot have constructor arguments but their bodies form an initializing behavior, in the same way that class bodies provide the primary constructor behavior.
- A single class can be assembled with multiple traits
 - If the class also has a parent class, use this form for the declaration:

```
class MyClass extends MyParentClass with
  TraitOne with TraitTwo
```

- If the class only inherits from traits use this form

```
class MyClass extends TraitOne with TraitTwo
```

- A trait may itself specify a parent type (including a class type)

```
trait T extends A ...
```

- Features of the required parent may be overridden and may be accessed using `super`
- Where multiple traits are listed, this can provide a decorator-like chainable behavior--the behaviors are invoked from right to left (based on the `with X with Y` declaration)
- The required parent may be abstract in which case the concrete parent implementation (and with it the meaning of `super`) is supplied when the trait is mixed into to some other class
- If the parent type is abstract, overriding methods must be marked `abstract override`

```
abstract class A { def show: Unit }
```

```
trait T extends A {
```

```
  abstract override def show: Unit = {
    print(" T"); super.show; print(" xT")
  }
```

```
}
```

```
trait U extends A {
```

```
  abstract override def show: Unit = {
    print(" U"); super.show; print(" xU")
  }
```

```
}
```

```
class C extends A {def show:Unit = print(" C") }
```

```
(new C with T).show; // T C xT
```

```
(new C with T with U).show; // U T C xT xU
```

```
(new C with U with T).show; // T U C xU xT
```

- This approach is very flexible, allowing creation of anonymous classes that are made from any combination of base class plus mixed-in traits, however, don't overlook that the traditional OO “decorator” pattern provides additional flexibility in that the order and number of decorating elements may be changed at runtime.
 - If dynamic changes are definitely not required, then the trait mix-in syntax is significantly more compact.
- A trait can require that when it is used, the resulting object has a certain type in its makeup. This is done using the `self` reference:

```
trait Thang:
```

```
  def sayHello:Unit =
    println("Hello from Thang")
```

```
trait Thing:
```

```
  // a Thang must be included in any concrete type
  // that "is a" Thing
  self:Thang =>
  def sayIt():Unit =
```

```
    this.sayHello // known to be valid
```

The above definitions would permit this class to be concrete (because Thang is included:

```
class Success extends Thing with Thang
```

but would reject this, because Thang is not included in the hierarchy:

```
class Fails extends Thing
```

Anonymous classes

- Anonymous classes may be derived and instantiated from concrete or abstract base classes, from traits, or without any explicit parent

```
abstract class GetDoubled {
  def v: Int
```

```

    def get: Int = v * 2
}

// Declare and instantiate an anonymous
// subtype of GetDoubled, defining the
// abstract element v in the process
val v = new GetDoubled{val v = 2};
println(v.get) // prints 4

```

- Without an explicit parent (In which case, the parent will be AnyRef/java.lang.Object):

```
new {def show: Unit = println("Hello!")}.show
```

Note that the above syntax requires an import for Scala 3 to allow the show method to be invoked directly:

```
import reflect.Selectable.reflectiveSelectable
```

Objects (Singletons) and Companion Objects

Scala does not provide a keyword “static”, instead the `object` keyword declares and instantiates a single instance of an anonymous class. The approach avoids the syntactic inconsistencies associated with static variables being members of a class, where the class name is not a valid expression and therefore cannot be used at runtime.

- The result is an object (the defining class name is related to the object name, but is not the same)
- The object behaves fully as an object; elements may be addressed using **this** if desired

```

object OnlyOne {
    val count = 100
}

```



```

val message = "Hello"
def main(args: Array[String]): Unit = {
  println(s"count is $count, " +
    s"message is ${this.message}")
}
}

```

- The name of the object is an expression, with a real value at runtime that can be stored in variables, passed as arguments and return values, and used in larger expressions. This differs from a classname, which is a special "compile-time only" syntax element.
- Features that relate to a "concept as a whole", rather than an individual instance of that concept should be implemented in an `object`. Where some features also relate to particular instances, a class can be declared with the same name. The class and the object are then "companions".
 - Even though the object's class is actually different from the class used to create multiple instances, the two companion types share access to one another's `private` elements.
 - Defining an `apply` method on the companion object is a common pattern for instantiating objects without the keyword `new` in client code:

```

class MyClass private (val x : Int) {
  override def toString: String =
    s"instance of MyClass, x is $x"
}

```

```

object MyClass {
  def apply(x: Int): MyClass = new MyClass(x)
}

```

```
val mc = MyClass(3)
println(mc)
```

Enumerations

- Scala 2 does not have a language element to create values selected from a pre-determined set of legitimate values, but instead uses a library class as the basis of enumerations. This is the `Enumeration` class.
- To define a simple enumeration, create an *object* that extends the class `Enumeration`, and create the required values by declaring the fields, and initializing them using the `Value` method. To provide an identifiable type for the enumeration, capture the type of the `Value` method in a type definition that's enclosed in the enumeration object:

```
object DayOfWeek extends Enumeration {
  type DayOfWeek = Value

  val SATURDAY = Value("Saturday")
  val SUNDAY = Value("Sunday")
  //... more repetitions omitted
}
```

- Given this declaration and appropriate imports, other code can refer to individual values such as: `DayOfWeek.SUNDAY`, and can declare variables such as:

```
val today: DayOfWeek.DayOfWeek =
  DayOfWeek.MONDAY
```

- Scala 3 does provide an enumeration type as part of the language

The match / case Construct

Scala provides a very powerful and flexible construction that is somewhat similar to a case statement in other C-like languages.

- In its simplest form it can match literals

```
val s = "bonjour"
s match {
  case "bonjour" => println("Hello!")
  case "au revoir" => println("Goodbye!")
}
```

- It can also match a default-like situation, capturing the value

```
case x => println(s"What does $x mean?")
```

- If the actual value is not needed on the right hand side, an underscore (Scala's general wildcard) may be used instead

```
case _ => println("No idea what that meant!")
```

- Matches might overlap, and are tested from top to bottom. The first match wins, and no others are tried.
- If a match clause exhausts all cases without matching, an exception is thrown
 - `case x => ...` (or the underscore variant) may be added at the end to pick up unmatched situations
- Matches can match by type, which is generally much preferred to using `target.isInstanceOf[Type]` tests

```
val x:Any = s
x match {
```

```

case i:Int => println(s"int value $i")
case s:String => println(s"string $s")
case _ => println("Some other type")
}

```

- Matches can be “guarded” with `if` clauses, which can benefit from the matched type (note that parentheses are not required around the boolean parameter to the `if` clause, even though normal `if` statements in Scala 2 do require them).

```

case s:String if s.length > 3 =>
  println("A string longer than 3")

```

- For many types (e.g. `List`) a match can extract how the object was created, and/or decompose it. Note that the `::` (“cons”) operator describes adding a new head (the left operand) to an existing list (the right operand). In a match statement, the joined parts are extracted

```

val l = List(1, 2, 3)
l match {
  case head :: tail => println(s"head is $head")
}
// These also match:
case List(x, _, _) =>
  println(s"List with 3 elements, first is $x")
case List(x, _, _*) =>
  println(s"List with 2+ elements, first is $x")

```

- When matches are made with patterns that extract elements from the item being matched, it’s also possible to capture the entire matched item using a “pattern binder”

```

case l @ List(x, _, _) =>
  println(s"List with 3 elements: ${l}")

```

this form would have the entire list printed from the variable `l`.

- A match can be made against an existing (in-scope) variable. This requires a special syntax. Recall that this form:

```
val x = ... // numeric value
x match {
```

```
  case i => println(s"int value $i")
```

“captures” the value that is being matched in the variable `i`. If a variable `i` already exists, this will create a more local variable that *shadows* the original. Instead, this syntax:

```
val x = ... // numeric value
val i = ... // numeric value
x match {
```

```
  case `i` => println(s"x matched $i")
```

does not create a new variable `i`, and matches when the values of `x` and `i` are equal.

Implementing `Function<N>` and `PartialFunction`

- Scala's libraries define a series of types called `Function0...Function22` which are used to describe generic functions (the method is always called `apply`) that take zero through twenty-two generic formal parameters and return a generic result type. The library also declares a single type `PartialFunction` which describes a method that takes a single generic formal parameter and returns a generic result. This type also declares a method `isDefinedAt(x)` which allows the caller to determine whether a meaningful result will be returned. All of these types are known to the compiler which provides a number of special behaviors related to them.

- Note that a `PartialFunction` might throw an exception or simply return a bogus value if called with an argument for which `isDefinedAt` returns false. It's not reliable to assume that a result is valid simply because no exception was thrown.
- If a context (e.g. function actual parameter) requires a `Function<N>` or `PartialFunction`, a match block can be provided directly, without needing to specify the “`x match`” prefix. For example, these two are equivalent:

```
val f2: Function2[Int, Int, String] = {  
  case (x, y) if x + y == 5 => "Five"  
}
```

```
val f3: Function2[Int, Int, String] =  
  (x, y) => (x, y) match {  
    case (a, b) if a + b == 5 => "Five"  
  }
```

Programmer-defined match targets

Matching against class type can be used to define and group variant behavior, achieving a similar goal to method overriding. In this case however the variations are placed in a “manager” code block, unrelated to the particular classes. This *might* be a cleaner, lower maintenance, design in some circumstances, such as when combinations of related types interact in ways that are properly managed from *outside* the instances. In such a situation, one often finds a conventional OO approach leads to a combinatorial explosion of “classification” and “action” behaviors for every class. This is often better handled on the outside and a `match` statement can be very elegant for this.

- Case classes allow easy definition of classes that are optimized for use with matching

```
sealed class Transporter()
sealed case class Car(seats:Int) extends
    Transporter
sealed case class BoxVan(length:Int, width:Int,
height:Int) extends Transporter

val t: Transporter = new BoxVan(3,4,5)
t match {
    case Car(s) =>
        println(s"Item on passenger seat of $s")
    case BoxVan(l, w, h) =>
        println(s"Van capacity ${l*w*h}")
}
```

- Scala also allows any class to be built in a way that integrates with matching by providing an “extractor” behavior.

Handling Exceptions

- Scala uses an exception mechanism broadly similar to other languages
- Exceptions are objects, and they have type which can be used to control how they are handled
- Exceptions may be thrown by any code, and that code does not have to declare that it does so. That is, Scala exceptions differ from Java in that there is no “declare or handle” rule and no concept of checked exceptions.
 - This can cause friction if Scala libraries are used from Java, but an annotation allows the generated binary code to include the information required by Java to allow it to enforce its rules about

checked exceptions. This is an example of the `@throws` annotation:

```
@throws (classOf[IllegalStateException])
```

- **Note** that `classOf[Type]` is equivalent to the Java code `Type.class` and provides a compile-time reference to the defining class of the named type.
- Handling exceptions can be performed with a `try catch finally` structure although functional style prefers an alternative, monad-like, approach using classes such as `Try` or `Either`.
- The single `catch` block is not type-specific, does not define a formal parameter, but is a construction that looks and behaves like `match` to partition the various catchable behaviors

```
def mightWork:Unit = {  
  val result = math.random  
  if (result < 0.34)  
    throw new SQLException("DB broke!")  
  else if (math.random < 0.67)  
    throw new IOException("IO broke!")  
  else throw new RuntimeException  
    ("That shouldn't happen!")  
}  
  
try {  
  mightWork  
} catch {  
  case e:SQLException =>  
    println(s"Yikes ${e.getMessage}")  
  case e:IOException =>  
    println(s"Uh oh ${e.getMessage}")  
  // this would catch unhandled exceptions  
  // case e:Throwable =>
```



```
//      println(s"Groan ${e.getMessage}")
} finally {
    println("In finally block")
}
println("Finishing up.")
```

- Like a `match` block, this structure executes only the first applicable match, so the more specific exceptions should be listed first
- Notice that the type specification `Throwable` applied to the last option is not required, but provides for a stylistic symmetry.
 - A warning is might be issued if this is omitted
 - If the exception value isn't needed, the underscore may be used instead of an actual identifier
- If the `match` construction does not catch an exception that's thrown, then that exception propagates out of the current method (unless there's an outer `try/catch` construction wrapped around this one)

Iterating with `for`

- The keyword `for` exists and creates iteration over certain types.
- Two modes exist, the idiomatically preferred form generates a value and is called a *for-expression*.
- An alternate form provides an imperative looping construct.
- The structure can be made up from up to four types of component:
 - Generators--any `for` construction must start with a generator. Generators create sequences of data items from a source, such as a collection or range. A generator is of the form **<pattern> <- <source>**
 - Filters--which remove some items from the stream. A filter is of the form **if <boolean-expression>**

- Variable bindings--which define/assign to an intermediate variable in the body of the iteration. A binding declares and initializes a new identifier, but does not use the `val` prefix. It has the form **`<identifier> = <expression>`**
- A `yield` expression--which creates a final result value for the entire iteration. The presence of `yield` makes the structure a `for-expression`, and omitting it creates an imperative `for-loop`. This has the form **`yield <expression>`**
- A `for-expression` must start with a generator, but may have any number of filters, variable bindings, and additional generators subsequently.
- The generators, filters, and variable binding elements follow the `for` keyword and are grouped using either parentheses or curly braces.
 - If parentheses are used, the elements must be separated with semicolons (because parentheses disable semicolon inferencing)
 - If curly braces are used, the elements may be placed on separate lines and semicolons are generally not required.

Examples

- The following expression creates an `IndexedSequence` containing the numbers 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. The boldface element is the single generator, notice that the `yield` keyword is followed by an expression `x * 2`:

```
for ( x <- 1 to 10 ) yield x * 2
```

- If multiple generators are used, the later ones iterate faster than the earlier ones. This example produces a sequence of the tuples (1,1) (2,1) (2,2) (3,1) (3,2) (3,3) :

```
for (x <- 1 to 3; y <- 1 to x) yield (x, y)
```

- Filters may be added which discard elements, so the following example produce the sequence of tuples (1,2) (1,3) (2,1) (2,3) (3,1)

(3,2). The boldface code is the filter (note that no parentheses are necessary on the filter construction

```
for (x <- 1 to 3; y <- 1 to 3; if x != y) yield (x, y)
```

- Variable bindings capture intermediate expression values which can then be used again later in the overall for expression. In this example, the variable binding is in boldface, and the code produces the output:

Hello starts with H

Bonjour starts with B

```
(for {  
  w <- List("Hello", "Bonjour", "欢迎", "ᠠᠨᠣᠷᠤᠨ")  
  c1 = w.charAt(0)  
  if c1 <= 'Z'  
} yield s"$w starts with $c1") foreach println
```

Imperative `for` constructions

- If no `yield` expression is added to a `for` construction, an imperative structure is created.
- In this case, the `for` construction should be followed by a single statement or a block of code. This will be executed for each data item that gets to the “end” of the sequence of elements in the `for` construct.
- Variables bound in the `for` construct will be available in the following block or statement. This might be a mildly surprising scope, since the variable is accessible outside the curly brace block that surrounded the point where the variable came into existence.
- In this example, the boldface element (a single statement in this case) is executed repeatedly. This prints a series of message of the form “**x** is *<value>*” with *<value>* set to each of the values 1 through 10 in turn:

```
for (x <- 1 to 10) println(s"x is $x")
```

Expansion of `for` constructions

- A `for` construct is entirely translated to an alternate representation that uses `flatMap`, `map`, and `withFilter`
- This translation only requires that the underlying types used in the generators and conditions provide those methods, without regard to implementing any particular trait or having any particular class parentage.

Function literals (lambda expressions)

Scala provides function literals (sometimes called either anonymous functions, or lambda expressions). The basic syntax is likely to be broadly familiar for anyone who has used this kind of feature in C#, Java, JavaScript, but does differ somewhat from Python and some others.

- Function literals simply define an argument list, followed by an arrow symbol, followed in turn by the expression that constitutes a function body for the anonymous function
 - Note that the arrow is formed with an equals sign, like JavaScript and C#, rather than a minus sign like C++/Java

```
(x: Int) => x * 2
```

Several variations are possible depending on context

- If the context defines the type, the argument type might be omitted
 - Note that in this example that the form “`Int => Int`” is a type specification indicating “function taking single `Int` argument and returning `Int`”

```
def doubleIt : Int => Int = (x) => x * 2
```

- If a function literal takes a single argument, without type notation, the parentheses on the argument list may be omitted

```
def doubleIt : Int => Int = x => x * 2
```

- If the types of all arguments can be reliably inferred and arguments are used exactly once each, in the order they would appear in the formal parameter list (note that's a lot of conditions to meet!), then a shorter form might be supported ("might" because sometimes this does not work when expected as a result of ambiguities—don't start out writing this form, but consider reducing verified code to it if it will simplify reading the code).
 - In this form, provide only the function body, but do *not* specify any argument list
 - Omit the arrow symbol
 - Arguments are picked up using underscores
 - The underscores must be used to pick up the arguments “in order” (left to right)
 - Note that if the order of use of the arguments doesn't conveniently conform to the order in the invocation (which is determined by context) then this form is not likely to be beneficial

```
def sumThree:(Int, Int, Int) => Int = _ + _ + _
```

```
def showThree:(Int, Int, Int) => String =
  "values " + _.toString + ", " +
  _.toString + ", " + _.toString
```

- A block expression may be used for the function body

```
def addAndShowThree:(Int, Int, Int) => Int =
  (a, b, c) => {
    println(s"values are $a, $b, $c")
    a + b + c
  }
```

```
}
```

- Tuples and function arguments can get confusing, it's tempting to try to use a destructuring assignment but this won't work, because the target is mistaken for a multiple argument list.
- For example, in this example, the `map` operation takes an argument that is a function that operates on the `List` elements (which are tuples of type `(Int, Int)`) and returns the sum of the two elements in each tuple. The `map` operation uses this to build a new `List` containing single `Int` values

```
val lt : List[(Int, Int)] = List((1, 2), (2, 1))  
lt map (t => t._1 + t._2) foreach println // OK
```

```
lt map((a,b) => a + b) foreach println // FAILS
```

- In the failure case, the expression `(a,b) => a + b` in the argument to `map` defines a function that takes two arguments, not a function that deconstructs a single tuple into two simple variables
- A destructuring can be performed using a `match` operation:

```
lt map (_ match {case (a,b) => a + b}) foreach println
```

- This can be further shortened to:

```
lt map {case (a,b) => a + b} foreach println
```

Implicit conversions, and implicit arguments

- Widening conversions (e.g. `Int` to `Long`) are automatic and as expected.

- Scala allows programmer, and library-supplied, conversions too. These conversions can be made automatic in the source using implicit conversions. These conversions can be controlled/overruled by the user of the conversion.
- Implicit conversions are often used to integrate new data types with pre-existing types. For example, allowing the effect of adding a complex type with a simple `Int`.
 - Scala 3 provides a more direct, and cleaner, mechanism for this particular type of integration using a feature called an *extension*.
- In addition to default arguments to functions, a function can offer a caller-configurable defaulting mechanism called implicit arguments. This kind of defaulting is often delegated to imports, allowing dynamic selection of the default value based on user preferences. Further, these defaults are context sensitive and might be different at different parts of the source code.

Conversion of an argument

- If a function takes type `Y`, it may be called with a type `X` provided a function taking an argument of type `X` and returning `Y` is:
 - Declared in some namespace,
 - Declared as an `implicit`, and
 - In scope as a single name. E.g.

```
object DefConv {
  implicit def iToS (i: Int): String =
    "Implicitly converted " + i
  def doStuff(s: String): Unit = {
    println("doStuff with " + s)
  }
  def main(args: Array[String]): Unit = {
    val v: Int = 99;
```

```
// Needs String, has Int, iToS converts
doStuff(v)
}
}
```

Conversion of the receiver

- If a method is invoked on an object (the “receiver”), but that object does not provide the method, it is possible the compiler might create a new object that does provide the desired method.
 - Receiver conversion requires an implicit function that converts the original receiver type to the type that contains the method to be invoked.
 - This example illustrates how this feature can integrate new classes with an existing API.
 - Note that in the example, the `+` (plus) operation invoked in the last line is the one defined in the `Count` class, not the normal “addition of `Int`” version

```
class Count(val c: Int) {
  def + (other: Count): Count =
    new Count(c + other.c)
}
```

```
implicit def intToCount(x:Int): Count =
  new Count(x)
```

```
val c1 = new Count(3)
// make Count(7), then call Count.+ method
val sum = 7 + c1
```


Implicit Arguments

- If a function argument is marked as implicit, it takes on a behavior similar to a default value, but more flexible. If, at the point of invocation, that argument is omitted, and a value of the same type is available as a single identifier, and that identifier is labeled implicit, then that implicit value will be used as the argument to the invocation.
- Notice that this creates a situation in which the caller can decide whether to apply a default-like value to the argument, and to control what value is applied.
- It is common for library functions to also offer an implicit value for ease of use.

```
package MyLibrary {  
  object ImpArg {  
    object Implicits {  
      implicit val defaultValue = "Hello World"  
    }  
    def myFunction(implicit s: String): Unit =  
      println(s)  
  }  
}
```

```
import MyLibrary.ImpArg.myFunction  
myFunction("Bonjour le monde")  
import MyLibrary.ImpArg.Implicits.defaultValue  
myFunction
```

- When using implicit arguments it is good practice to create a special type for the argument. In this example, having an implicit value of type

`String` would make it very easy for the default that is implied to be applied to the wrong function.

Implicit classes

- In some situations it's desirable to create a wrapper class (that holds “utility” methods) and simultaneously define an implicit conversion to that wrapper.
 - This happens frequently when providing implicit arguments, to avoid arguments of common types like `String` being defined as implicit
 - It can also happen when arranging better interactions with new extension code and existing libraries.
- This situation is supported by a specific syntax called an implicit class
 - It is more restricted in application allowing conversion from only a *single* type
 - In this case the intended receiver *class* is declared as implicit
 - The implicit class must define a single-argument constructor.
 - The compiler performs the conversion using the class' constructor

```
implicit class Count(val c: Int) {  
  def combine(other: Count): Count =  
    new Count(c + other.c)  
}
```

```
val c1 = new Count(3)  
val sum = 7 combine c1
```

Implicit conversion Rules

Implicit conversions are subject to rules to avoid accidental conversions:

- Implicits cannot be top-level.
- Functions must be *marked* `implicit` to be used.

- A conversion must be completed with a *single* implicit conversion, not a sequence of conversions.
- An implicit conversion will never be used if an *explicit* conversion is coded.
- The conversion must be in scope as a single identifier (this is usually achieved using imports).
 - In support of this, it's usual to place implicits in a subpackage of the package containing the affected features. This subpackage is sometimes named `implicits` or `Preamble`

Generics

- Declarations (classes, methods, variables, etc.) may all make use of Scala's generic type-consistency checking mechanism
- Declare generic type variables in square brackets.
- Place the declaration immediately after the identifier and prior to any argument list:

```
class Bucket[T] (items: T*) ...
def fetchAt[T] (idx: Int, List[T]): T ...
```

- After declaration, generic type variables may be used in place of specific type names, and the compiler will verify consistency of usage
- Declarations may be constrained. To require that the type represented by variable `T` must be assignment compatible with another type `X`, use the syntax:

```
[T <: X]
```

- To declare that `T` must be assignment compatible from `X`:

```
[T >: X]
```

- In either case, `X` may be a concrete type, or another properly declared generic type variable.

- If a class declaration is intended to be covariant with respect to a generic type, the generic type declaration should be prefixed with a plus sign. If contravariance is required, prefix with a minus sign.

```
Class BucketProcessor[+T, -U]
```

- The Scala compiler will check that uses made of types are compatible with their declared variances

Functional Patterns

- Adapter/decorator: A function that takes a list of parameters of certain types may be wrapped using another function-factory that pre-applies certain arguments, or converts arguments to its own call in type, value, or both prior to passing them to the original method. Such wrapping might (depending somewhat on the particular form of declaration and use) be referred to as:
 - Pre-application
 - A curried function
 - An adapter
 - A decorator
- Monads
 - Scala collections implement the core monad behavior `flatMap`, they also provide a broad range of related and derived behaviors including: `filter`, `fold` (left and right), `map`, and `reduce` (left and right)
- `Option` is a possibly-absent value monad. There are two case-class implementations: `Some`, which contains a real value, and `None`, which does not.
- A monad-like class called `Try` allows execution of code that might throw an exception.
 - The factory for this takes a by-name argument, so that it is executed inside the `Try` behavior, allowing the exception to be

caught and handled inside the `Try` factory, rather than prior to the invocation of that factory method:

`Try[T] (expr: => T)`

- The constructed `Try` object will be one of two case classes, `Success`, or `Failure`. Each of these contains the relevant data, either the result of the successful execution, or the exception that was caught.
- Try objects provide a number of ways to get at the data they contain including: `get`, `getOrElse`, `orElse`, `recover`, `recoverWith`,
- A `Try` can be converted to an `Either` which is an abstract class that also has two concrete subtypes, `Left`, and `Right`. A `Left` contains an indication of why the operation failed and a `Right` contains the successful result of the operation.
- `Stream / LazyList`
 - Scala's `LazyList` class provides a monadic structure that supports lazily computed infinite sequences.
 - This class was previously called `Stream`, but was renamed to `LazyList` with the release of Scala 2.13
 - A `LazyList` can also be extracted from a regular collection, though in this case, the data already exist, so only the computation of intermediate values along the monad “pipeline” will be lazy.
 - Scala `Stream` objects are not parallel, so computations in them are handled by a single CPU
 - Apache Spark is built on top of a very similar stream concept. Spark allows execution of pipeline steps to be distributed across multiple threads on a multi-core architecture, and across distributed machines if appropriately configured.
- `I/O with Scala`
 - Scala has very limited IO libraries, instead it relies heavily on the core Java APIs in a JVM implementation.

- The Scala library provides a convenient means of accessing files for reading, which is the `scala.io.Source` object

Using Scala In a Maven Project

Scala can be used alongside Java in a maven project. To do this, the scala plugin can be used to compile the scala source files, and the scala library should be included in the resulting binary. The following is an example of maven pom.xml contents to achieve this.

```
<!-- Make Scala available in project.  
Pay extra attention to the version -->  
<dependencies>  
  <dependency>  
    <groupId>org.scala-lang</groupId>  
    <artifactId>scala-library</artifactId>  
    <version>2.12.4</version>  
  </dependency>  
</dependencies>  
  
<build>  
  <plugins>  
    <!-- This plugin compiles Scala files -->  
    <plugin>  
      <groupId>net.alchim31.maven</groupId>  
      <artifactId>scala-maven-plugin</artifactId>  
      <version>3.3.1</version>  
      <executions>  
        <execution>  
          <id>scala-compile-first</id>  
          <phase>process-resources</phase>  
          <goals>  
            <goal>add-source</goal>  
            <goal>compile</goal>  
          </goals>  
        </execution>  
        <execution>  
          <id>scala-test-compile</id>  
          <phase>process-test-resources</phase>  
          <goals>
```

```

        <goal>testCompile</goal>
    </goals>
</execution>
</executions>
</plugin>
<!-- This plugin compiles Java files -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.7.0</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
    <executions>
        <execution>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<!-- This plugin adds all dependencies to JAR file during 'package'
command. Be sure to set the 'mainClass' tag.-->
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.1.0</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <mainClass>REPLACE WITH MAIN CLASS FQCN</mainClass>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>

```

```
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

Scala 2 to Scala 3 Changes

Scala 3 refines the language. For the most part, mainstream Scala 2 syntax is still valid in Scala 3. Some key differences are:

- Scala 3 introduces an indentation-controlled block structuring. Braces can still be used, but generally the preferred form is to start the block with a colon (or with equals for a method body), then use consistent indentation on the lines contained in the block. This is very much like Python.

e.g.

```
class MyClass(val x: Int):
  println("running constructor")
  def show(y: Int): Unit =
    println("in method show")
    println(s"x is ${x}, and y is ${y}")
  override def toString: String =
    s"MyClass, x is ${x}"
```

- Scala 2 uses the underscore as a wildcard for imports. In Scala 3, the asterisk replaces this.
- As with Scala 2, methods declared without parentheses must be invoked *without* parens, but in Scala 3 those declared with empty parens must be invoked with empty parens.

- Scala 2 conditional operator uses parens around the conditional expression, but does not use the keyword "then". In Scala 3, the preferred form does not require parens around the condition, but uses "then" to delimit the end of the test. Similarly the while loop has a no-parens on the test form that uses the keyword "do" to delimit the end of the test.

```
var x = 4
while x > 0 do
  println(s"x is ${x}")
  x -= 1

if x != 0 then
  println("Well, that's a surprise!")
else
  println("x is zero, as expected")
```

- Scala 3 provides a simple enum syntax:

```
enum Days:
  case SATURDAY, SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY
```

- In Scala 2, tuples are accessed using their `_1`, `_2`... fields, and were indexed using the `productElement` accessor method. These still work in Scala 3, but they also provide an `apply` method that is equivalent to `productElement`, so they can be accessed like any other indexed sequence using a zero-based subscript. Tuples are still immutable (no `update` or `updated` method is provided).
- Scala 3 allows an `end` marker, which might be useful for facilitating visually tracing the end of a larger indentation-specified block:

```
class MyClass(val x: Int):
  def show(y: Int): Unit =
```

```

println(s"x is ${x}, and y is ${y}")
if x > y then
  println("x is bigger")
end if
end show
end MyClass

```

- Note that the indentation of the `end` clause (exactly below the indentation of the element that is being ended, is syntactically required.
- Function declarations are closed with `end <function-name>`, like this:

```

def doStuff():Unit =
  println("doing stuff")
end doStuff

```

- A `for` expression may be ended with either a `do` or `yield` clause

```

for
  x <- 1 to 10
  if x % 2 == 0
    y = x * x
do println(s"y is ${y}")

```