Growing new types:

- Scala allows users to grow and adapt the language in the directions they need by defining easy-to-use libraries that feel like native language support.
- Scala lets you add new types that can be used as conveniently as built-in types

- Example: BigInt (Page No. 47)

Scala Language

- The name Scala stands for “scalable language.” The language is so named because it was designed to grow with the demands of its users.

- Technically, Scala is a blend of object-oriented and functional programming concepts.

- Instead of providing all constructs you might ever need in one “perfectly complete” language, Scala puts the tools for building such constructs into your hands

Scala is object oriented:

- Scala is an object-oriented language in pure form: every value is an object and every operation is a method call. For example, when you say 1 + 2 in Scala, you are actually invoking a method named + defined in class Int

- You can define methods with operator-like names

- Scala is more advanced than most other languages when it comes to composing objects. An example is Scala's traits. Traits are like interfaces in Java, but they can also have method implementations and even field

- Methods should not have any side effects. They should communicate with their environment only by taking arguments and returning results.

- For instance, the replace method in Java's String class fits this description. It takes a string and two characters and yields a new string where all occurrences of one character are replaced by the other. There is no other effect of calling replace.

- Methods like replace are called referentially transparent, which means that for any given input the method call could be replaced by its result without affecting the program's semantics

Scala is functional: (Functions are first-class values)

- In a functional language, a function is a value of the same status as, say, an integer or a string.
- You can pass functions as arguments to other functions, return them as results from functions, or store them in variables.

- You can also define a function inside another function
- You can define functions without giving them a name

Immutable data structures are one of the cornerstones of functional programming.

- Immutable data structures are one of the cornerstones of functional programming.
- The Scala libraries define many more immutable data types on top of those found in the Java APIs.
- For instance, Scala has immutable lists, tuples, maps, and sets

Referentially transparent method:
Method can be referentially transparent, if a call to this method may be replaced by its return value.

```
int add(int a, int b)
{
return a + b
}
int mult(int a, int b)
{
return a * b;
}
int x = add(2, mult(3, 4));
```

Here, function call mult(3,4) can be replaced by 3*4. Doing this replacement will not change the result of x. The same way, function call add(2,mult(3,4)) can be replaced by 2+3*4. This replacement will not the change the result of x.

On the other hand, consider the following program:

```
int add(int a, int b)
{
   int result = a + b;
   System.out.println("Returning " + result);
   return result;
}
```

Replacing a call to the add method with the corresponding return value will change the result of the program, since the message will no longer be printed.

Why Scala

- Compatibility
- Brevity
- high-level abstractions
- advanced static typing

Why Scala?

Compatibility: Scala is designed for seamless interoperability with Java.

Scala programs compile to JVM bytecodes.

Scala code can call Java methods, access Java fields, inherit from Java classes and implement Java interfaces.

- Scala heavily re-uses Java types. Scala’s Ints are represented as Java primitive integers of type int,

- Floats are represented as floats, Booleans as Booleans, and so on. Scala arrays are mapped to Java arrays. Scala also re-uses many of the standard Java library types. For instance, the type of a string literal in "abc“ in Scala is java.lang.String, and a thrown exception must be a subclass of java.lang.Throwable

- Scala not only re-uses Java's types, but also dresses them up to make them nicer.

Example: Page number 55.

Scala code can also be invoked from Java

Scala is concise:

Scala programs tend to be short. Scala programmers have reported reductions in number of lines of up to a factor of ten compared to Java.

These might be extreme cases. A more conservative estimate would be that a typical Scala program should have about half the number of lines of the same program written in Java. Fewer lines of code mean not only less typing, but also less effort at reading and understanding programs and fewer possibilities of defects.

Example: Page Number 56

- Scala's type inference is another factor that contributes to its conciseness.

- Scala is high-level: Example: Page no:57

- Scala is statically typed: types checked before run time

- C++ is another scalable language that can be adapted and extended through operator overloading and its template system; compared to Scala it is built on a lower-level, more systems-oriented core.

Scala Installation

Go to http://www.scala-lang.org/downloads and follow the directions for your platform.

You can also use a Scala plug-in for Eclipse, IntelliJ or NetBeans.

Scala interpreter: Type scala in command prompt to use scala interpreter.
Example: Page number 65

Scala interpreter

```
$ scala
Welcome to Scala version 2.11.7
Type in expressions to have them evaluated.
Type :help for more information.
scala>

scala> 1 + 2
The interpreter will print:
res0: Int = 3

scala> println("Hello, world!")
Hello, world!
```

Define Variables

2 kinds: val and var

val - Once initialized, a val can never be reassigned.
var - can be reassigned throughout its lifetime.

Example for val definition:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

- This statement introduces msg as a name for the string "Hello, world!". The type of msg is java.lang.String, because Scala strings are implemented by Java's String class.

Define Variables – Contd.

Type inference: It can figure out type of the variable automatically. You can however specify a type explicitly if you wish.

msg is initialized with a string literal. So, Scala inferred the type of msg to be String.

In Scala, a variable's type is specified after its name, separated by a colon. Example:

Page No 67: Difference between val and var.

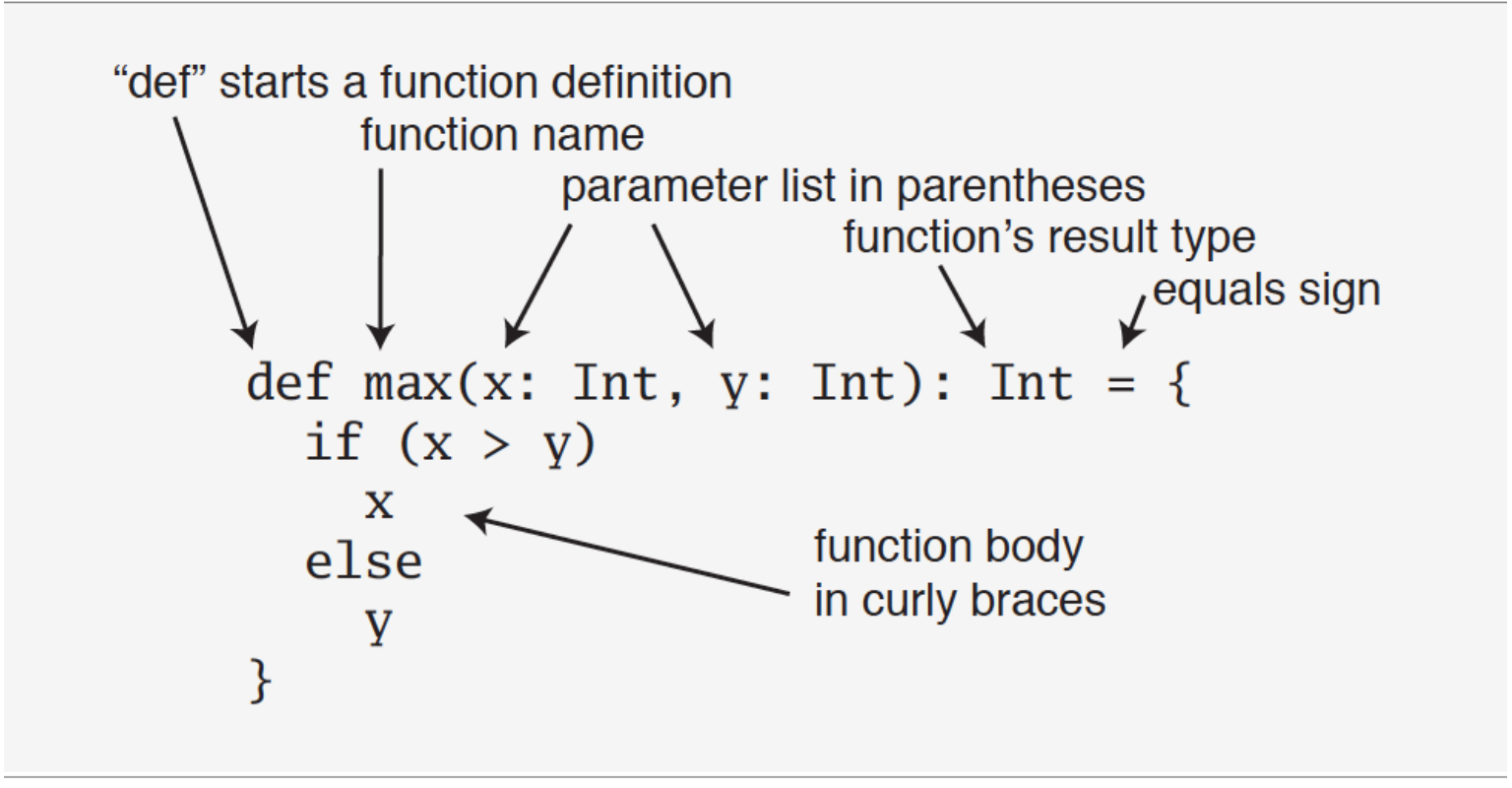
Variables/Values Examples

```
scala> val msg2: java.lang.String = "Hello again, world!"
msg2: String = Hello again, world!

scala> msg = "Goodbye cruel world!"
<console>:8: error: reassignment to val
msg = "Goodbye cruel world!"

scala> var greeting = "Hello, world!"
greeting: String = Hello, world!

scala> greeting = "Leave me alone, world!"
greeting: String = Leave me alone, world!
```

Variables/Values Examples – Contd.

```
scala> val multiLine =
| "This is the next line."
multiLine: String = This is the next line.

scala> val oops =
|
|
You typed two blank lines. Starting a new command.
scala>
```

Define functions in Scala:



max: (Int,Int)Int

The equals sign that precedes the body of a function hints that in the functional world view, a function defines an expression that results in a value.

Functions – Contd.

Specifying result type is not compulsory unless if it is recursive. Compiler can infer result type automatically.

if a function consists of just one statement, you can optionally leave off the curly braces. Thus, you could alternatively write the max function like this:

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y max2: (Int,Int)Int
```

Once you have defined a function, you can call it by name, as in:

```
scala> max(3, 5)
res6: Int = 5
```

Functions – Contd.

- Here's the definition of a function that takes no parameters and returns no interesting result:
- scala> def greet() = println("Hello, world!")

greet: ()Unit

When you define the greet() function, the interpreter will respond with greet: ()Unit.

"greet" is, of course, the name of the function. The empty parentheses indicate the function takes no parameters. And Unit is greet's result type.

Functions – Contd.

- A result type of Unit indicates the function returns no interesting value. **Scala's Unit type is similar to Java's void type**
- Methods with the result type of Unit, therefore, are only executed for their side effects. In the case of greet(), the side effect is a friendly greeting printed to the standard output.

If you wish to exit the interpreter, you can do so by entering :quit or :q.

- **scala> :quit $**

Scala Scripts

- A script is just a sequence of statements in a file that will be executed sequentially.
- Put this into a file named hello.scala:

```
println("Hello, world, from a script!")
```

- Then run: $ scala hello.scala

Scala Scripts – Contd.

- Command line arguments to a Scala script are available via a Scala array

named args.

- Arrays are zero based, and you access an elementby specifying an index in parentheses

```
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

then run:

```
$ scala helloarg.scala planet
```

Hello, planet!

Loop with while; decide with if

To try out a while, type the following into a file named printargs.scala:

```
var i = 0
while (i < args.length) {
        println(args(i))
        i += 1
}
```

$ scala printargs.scala Scala is fun

O/P

Scala

is

fun

i++ or ++i won't work in SCALA

Iterate with foreach and for

- Type the below code into a new file named pa.scala

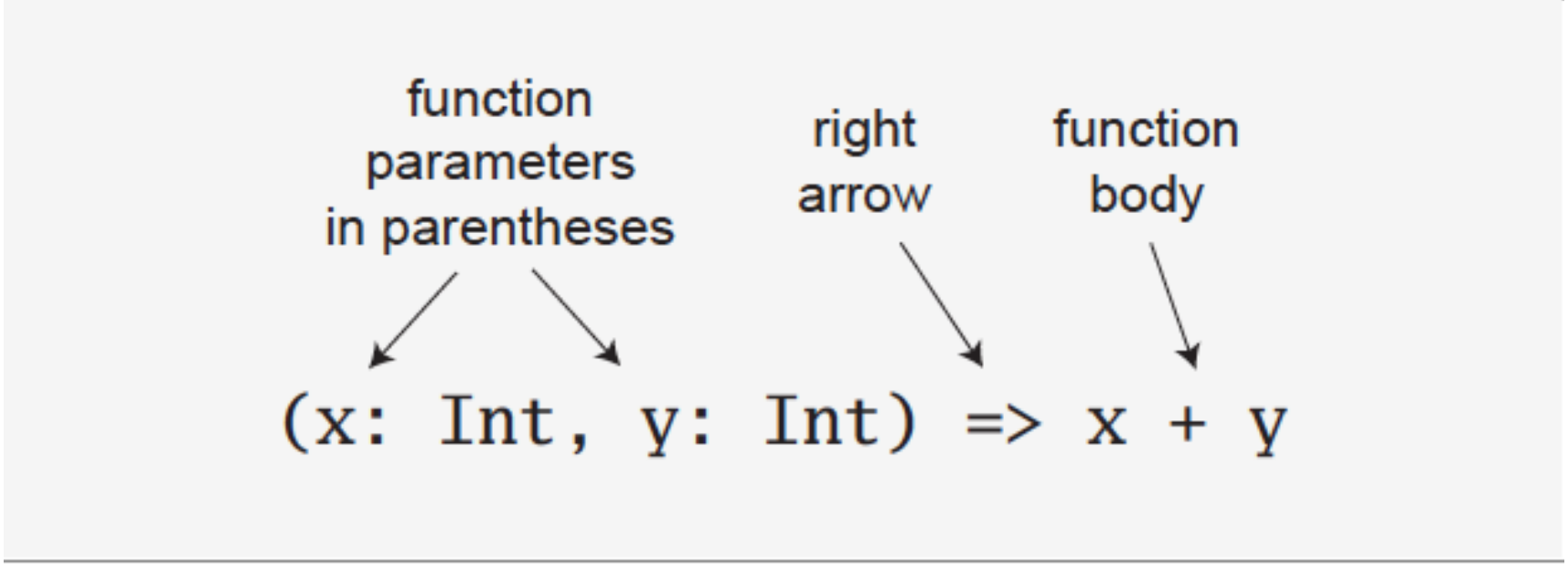
```
args.foreach(arg => println(arg))
OR args.foreach((arg: String) => println(arg))
```

- In this code, you call the foreach method on args and pass in a function.
- The body of the function is println(arg).
- Execute with the command:

```
$ scala pa.scala Concise is nice
You should see:
        Concise
        is
        nice
        In the
```

- args.foreach((arg: **String**) => println(arg))

The syntax of a function literal in Scala.



For

```
for (arg <- args)
println(arg)
```

- The parentheses after the “for” contain arg <args.
- To the right of the <- symbol is the args array.
- To the left of <- is “arg”, the name of a val, not a var.

for each element of the args array, a new arg val will be created and initialized to the element value, and the body of the for will be executed.

If you run the forargs.scala script

$ scala forargs.scala for arg in args

You’ll see:

for
arg
in
args

Chapter 3

new: It can instantiate/create objects.

When you instantiate object, you can parameterize/configure it with values and types.

Example for instantiating an object and parameterize it with the value 12345.

val big = new java.math.BigInteger("12345")

Parameterize arrays with types

Parameterize an instance with both a type and a value:

First type should come in square bracket and followed by the value in parentheses:

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
for (i <- 0 to 2)
  print(greetStrings(i))
```

Arrays in Scala are accessed by placing the index inside parentheses, not square brackets as in Java

We cannot reassign greetStrings to a different array. It will always point to the same Array[String] instance with which it was initialized. But you can change the elements of that over time, so the array itself is Array[String] mutable.

In this for (i <0 to 2), **to** is actually a method

0 to 2 is transformed into the method call (0).to(2)

Why?

Scala does not support operator overloading

- +, -, * and / are like methods.
- When you type 1+2 into the Scala interpreter, you are actually invoking a method named + on the int object 1 and passing in 2 as a parameter.
- It is equivalent to (1).+(2)

Why arrays are accessed with parentheses in Scala?

apply(): When you apply parentheses surrounding one or more values to a variable, Scala will transform the code into an invocation of a method named **apply** on that variable.

So, *greetStrings(i)* is transformed into *greetStrings.apply(i)*. Thus, accessing an element of an array in Scala is simply a method call.

This principle applies to any application of an object to some arguments in parentheses will be transformed to an apply method call.

update(): When an assignment is made to a variable to which parentheses and one/more arguments have been applied, the compiler will transform into an invocation of an update method that takes the arguments in parentheses as well as the object to the right of the equals sign.

greetStrings(0) = "Hello“ will be transformed into greetStrings.update(0, "Hello")

```
val greetStrings = new Array[String](3)
greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")
for (i <0.to(2))
  print(greetStrings.apply(i))
```

Scala treats everything from arrays to expressions as objects with methods

Concise way to create and initialize arrays:

val numNames = Array("zero", "one", "two")

The above code creates a new array of length three, initialized to the passed strings “zero”, “one” and “two”. Here type of the array is Array[String] and that is automatically inferred because we passed strings to it.

val numNames = Array("zero", "one", "two") is equivalent to val numNames = Array.apply("zero", "one", "two")

- If you’re a Java programmer, you can think of this as calling a static method named *apply* on class *Array*

List:

- A Scala array is a mutable sequence of objects that all share the same type.
- We can't change the length of an array after it is instantiated but we can change its element values.
- List is a immutable sequence of objects that all share the same type.
  Java's List is mutable but Scala's List is immutable.

How to create a list in Scala?

```
val oneTwoThree = List(1,2,3)
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new list.")
```

O/P

List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list

:: cons

Cons prepends a new element to the beginning of an existing list and returns the resulting list.

For example, if you run this script:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)

val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Nil is an empty list here. It is required here because :: is defined in class List. But 1, 2 and 3 are int here.

Tuples:

- Tuples are immutable,
- Tuples can contain different types of elements.
- A tuple could contain both an integer and a string at the same time.

- To instantiate a new tuple :

```
val pair = (99, "Luftballons")
```

- To access its elements:

```
println(pair._1)
println(pair._2)
```

The actual type of a tuple depends on the number of elements it contains and the types of those elements.

The type of (99, "Luftballons") is Tuple2[Int, String].
The type of ('u', 'r', "the", 1, 4, "me") is Tuple6[Char, Char, String, Int, Int, String].

Accessing the elements of a tuple
You may be wondering why you can't access the elements of a tuple like the elements of a list, for example, with "pair(0)".

The reason is that a list's apply method always returns the same type, but each element of a tuple may be a different type: _1 can have one result type, _2 another, and so on

Collection classes: Set and Map

Set: Immutable Set(it never changes. But you have still operations that simulate additions, removals or updates, but those operations will in each case return a new collection and leave the old collection unchanged) and Mutable Set(It can be updated/extended in place)

Immutable Set creation example

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

Mutable Set creation example

```
import scala.collection.mutable
val movieSet = mutable.Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

Map:

Creation of mutable Map

```
import scala.collection.mutable
val treasureMap = mutable.Map[Int, String]()
treasureMap += (1 ->"Go to island.")
treasureMap += (2 ->"Find big X on ground.")
treasureMap += (3 ->"Dig.")
println(treasureMap(2))
```

Creation of immutable Map

```
val romanNumeral = Map(1 ->"I", 2 ->"II", 3 ->"III", 4 ->"IV", 5 ->"V")
println(romanNumeral(4))
```

- **One way to move towards a functional style is to try to program without vars**

- Scala is not a pure functional language that forces you to program everything in the functional style. Scala is a hybrid imperative/functional language

- **A balanced attitude for Scala programmers**

Prefer vals, immutable objects, and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.

- Preferring methods without side effects can help make your programs easier to test.

Functional style

```
def printArgs(args: Array[String]): Unit = {
var i = 0
while (i < args.length) {
println(args(i))
i += 1
} }
```

- **TO Write in functional Style**

```
def printArgs(args: Array[String]): Unit = {
for (arg <args)
println(arg)
}
```

or this:

```
def printArgs(args: Array[String]): Unit = {
args.foreach(println)
}
```

This example

What is side effect?

A side effect is generally defined as mutating state somewhere external to the method or performing an I/O action.

A method that is executed only for its side effects is known as a procedure.

A function without side effects or vars.

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```

Read lines from a file: To read lines from a file and prints them out prepended with the number of characters in each line

```
import scala.io.Source

if (args.length > 0) {
  for (line <- Source.fromFile(args(0)).getLines())
      println(line.length + " " + line)
}
else
  Console.err.println("Please enter filename")
```

The expression **Source.fromFile(args(0))** attempts to open the specified file and returns a **Source** object, on which you call **getLines**. The **getLines** method returns an **Iterator[String]**, which provides one line on each iteration, excluding the end-of-line character. The **for** expression iterates through these lines and prints for each the length of the line, a space, and the line itself.

To run: $ scala countchars1.scala countchars1.scala (Show the output in textbook)

Chapter 4
Classes and Objects

Principles of Programming Languages

AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

Scala

Chapter- 1

Scala Language

- The name Scala stands for “scalable language.” The language is so named because it was designed to grow with the demands of its users.
- a blend of object-oriented and functional programming concepts.
- Instead of providing all constructs you might ever need in one “*perfectly complete*” language, Scala puts the tools for building such constructs into your hands
- Founded by Martin Odersky
- Runs on JVM
- Statistically typed language
- Interoperable with Java
- Use Java libraries in Scala code
- Write less and achieve more

Scala Language

- The name Scala stands for “scalable language.” The language is so named because it was designed to grow with the demands of its users.
- a blend of object-oriented and functional programming concepts.
- Instead of providing all constructs you might ever need in one “*perfectly complete*” language, Scala puts the tools for building such constructs into your hands
- Founded by Martin Odersky
- Runs on JVM
- Statistically typed language

Sample Code

def factorial(x: BigInt): BigInt =

*if (x == 0) 1 else x * factorial(x 1)*

Now, if you call factorial(30) you would get:
265252859812191058636308480000000

BigInt is a class defined in Javas standard library NOT a built-in datatype

Scala is object oriented

- Scala is an object-oriented language in pure form
- every value is an object and every operation is a method call.
- For example, when you say 1 + 2 in Scala, you are actually invoking a method named + defined in class Int
- You can define methods with operator-like names
- Scala is more advanced than most other languages when it comes to composing objects.
- An example is Scala's traits. Traits are like interfaces in Java, but they can also have method implementations and even field

Scala is Functional

- In a functional language, a function is a value of the same status as, say, an integer or a string.
- You can pass functions as arguments to other functions, return them as results from functions, or store them in variables.

- You can also define a function inside another function
- You can define functions without giving them a name

Immutable Data Structures

- Immutable data structures are one of the cornerstones of functional programming.
- The Scala libraries define many more immutable data types on top of those found in the Java APIs.
- For instance, Scala has immutable lists, tuples, maps, and sets

Requirements by Functional Programming

- Methods should not have any side effects. They should communicate with their environment only by taking arguments and returning results.

- For instance, the replace method in Java's String class fits this description. It takes a string and two characters and yields a new string where all occurrences of one character are replaced by the other. There is no other effect of calling replace.

- Methods like replace are called referentially transparent, which means that for any given input the method call could be replaced by its result without affecting the program's semantics

Scala is also a functional language in the sense that every function is a value and every value is an object so ultimately every function is an object.

Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying.

```
object Sample
{
    // Define currying function
    def add(x: Int, y: Int) = x + y;

    def main(args: Array[String])
    {
        println(add(20, 19));
    }
}
```

Referentially Transparent Method

Method can be referentially transparent, if a call to this method may be replaced by its return value.

```
int add(int a, int b)
{
return a + b
}
int mult(int a, int b)
{
return a * b;
}
int x = add(2, mult(3, 4));
```

Here, function call mult(3,4) can be replaced by 3*4.
Doing this replacement will not change the result of x. The same way, function call add(2,mult(3,4)) can be replaced by 2+3*4. This replacement will not change the result of x.

On the other hand, consider the following program:

```
int add(int a, int b)
{
   int result = a + b;
   System.out.println("Returning " + result);
   return result;
}
```

Replacing a call to the add method with the corresponding return value will change the result of the program, since the message will no longer be printed.

Why Scala?

- Compatibility
- Almost as fast as Java
- high-level abstractions
- advanced static typing
- Interoperable with Java
- Use Java libraries in Scala code
- Concise (Write less and achieve more)

Scala compatibility with Java

- Scala is designed for seamless interoperability with Java.

- Scala programs compile to JVM bytecodes.

- Scala code can call Java methods, access Java fields, inherit from Java classes and implement Java interfaces.
- Scala heavily re-uses Java types. Scala's Ints are represented as Java primitive integers of type int,

- Floats are represented as floats, Booleans as Booleans, and so on. Scala arrays are mapped to Java arrays. Scala also re-uses many of the standard Java library types. For instance, the type of a string literal in "abc“ in Scala is java.lang.String, and a thrown exception must be a subclass of java.lang.Throwable

Scala Installation

Go to http://www.scala-lang.org/downloads and follow the directions for your platform.

You can also use a Scala plug-in for Eclipse, IntelliJ or NetBeans.

Scala interpreter: Type scala in command prompt to use scala interpreter.

Scala interpreter

```
$ scala
Welcome to Scala version 2.11.7
Type in expressions to have them evaluated.
Type :help for more information.
scala>

scala> 1 + 2
The interpreter will print:
res0: Int = 3

scala> println("Hello, world!")
Hello, world!
```

Basic Syntax

Scala - a collection of objects that communicate via invoking each other's methods.

Object − Objects have states and behaviors. An object is an instance of a class. Example − A dog has states - color, name, breed as well as behaviors - wagging, barking, and eating.

Class − A class can be defined as a template/blueprint that describes the behaviors/states that are related to the class.

Methods − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Fields − Each object has its unique set of instance variables, which are called fields. An object's state is created by the values assigned to these fields.

Closure − A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

Traits − A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Traits are used to define object types by specifying the signature of the supported methods.

•**Case Sensitivity** − Scala is case-sensitive, which means identifier **Hello** and **hello** would have different meaning in Scala.

•**Class Names** − For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

Example − class MyFirstScalaClass.

•**Method Names** − All method names should start with a Lower Case letter. If multiple words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example − def myMethodName()

•**Program File Name** − Name of the program file should exactly match the object name. When saving the file you should save it using the object name

def main(args: Array[String]) – Scala program processing starts from the main() method which is a mandatory part of every Scala Program.

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

```
class HelloWorld
{
    public static void main(String[] args)
     {
       System.out.println("Hello, World!");
      }
}
```

Scala Identifiers

Names used for objects, classes, variables and methods are called identifiers.
A keyword cannot be used as an identifier and identifiers are case-sensitive.
Scala supports four types of identifiers.

- ✓Alphanumeric Identifiers
- ✓Operator Identifiers
- ✓Mixed Identifiers
- ✓Literal Identifiers

1. Alphanumeric Identifiers : Starts with a letter or an underscore, which can be followed by further letters, digits, or underscores. Eg: age, salary, _value, __1_value

2. Operator Identifiers: An operator identifier consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #. Eg: + ++ ::: <?> :>

3. Mixed Identifiers: A mixed identifier consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier. Eg: unary_+, myvar_=

4. Literal Identifiers: A literal identifier is an arbitrary string enclosed in back ticks (` . . . `).Eg: `x` `<clinit>` `yield`

Reserved Words or Keywords

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	Null
object	override	package	private
protected	return	sealed	super
this	throw	trait	Try
true	type	val	Var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Comments

Similar to Java – Single line and Multi line comments

```
object HelloWorld {
   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */
   def main(args: Array[String]) {
      // Prints Hello World
      // This is also an example of single line comment.
      println("Hello, world!")
   }
}
```

Note

Scala is a line-oriented language where statements may be terminated by semicolons (;) or newlines. A semicolon at the end of a statement is usually optional.

A semicolon is required if you write multiple statements on a single line
Eg: val s = "hello"; println(s)

Variable Definition

2 kinds: val and var

val - Once initialized, a val can never be reassigned.
var - can be reassigned throughout its lifetime.

Example for val definition:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

- This statement introduces msg as a name for the string "Hello, world!". The type of msg is java.lang.String, because Scala strings are implemented by Java’s String class.

Variable Definition– Contd…

Type inference: It can figure out type of the variable automatically. You can however specify a type explicitly if you wish.

msg is initialized with a string literal. So, Scala inferred the type of msg to be String.

In Scala, a variable's type is specified after its name, separated by a colon. Example:

```
scala> val msg : String = "Hello, world!"
    msg: String = Hello, world!

scala> println(msg)
Hello, world!
```

Variables/Values Examples

```
scala> val msg2: java.lang.String = "Hello again, world!"
msg2: String = Hello again, world!

scala> msg = "Goodbye cruel world!"
<console>:8: error: reassignment to val
msg = "Goodbye cruel world!"

scala> var greeting = "Hello, world!"
greeting: String = Hello, world!

scala> greeting = "Leave me alone, world!"
greeting: String = Leave me alone, world!
```

Variables/Values Examples – Contd…

```
scala> val multiLine = "This is the next line."
multiLine: String = This is the next line.

scala> val oops =
|
|
You typed two blank lines. Starting a new command.
scala>
```

A line containing only whitespace, possibly with a comment, is known as a blank line, and Scala totally ignores it

Multiple assignments

- val (myVar1: Int, myVar2: String) = Pair(40, “Hello")

- val (myVar1, myVar2) = Pair(40, “Hello")

Datatypes

- Scala has all the same data types as Java, with the same memory footprint and precision.
- Along with these, string, char etc

Byte	8 bit signed value. Range from -128 to 127
Short	16 bit signed value. Range -32768 to 32767
Int	32 bit signed value. Range -2147483648 to 2147483647
Long	64 bit signed value. -9223372036854775808 to 9223372036854775807
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float

Integer and Floating point Literals

```
0
035
21
0xFFFFFFFF
0777L
```

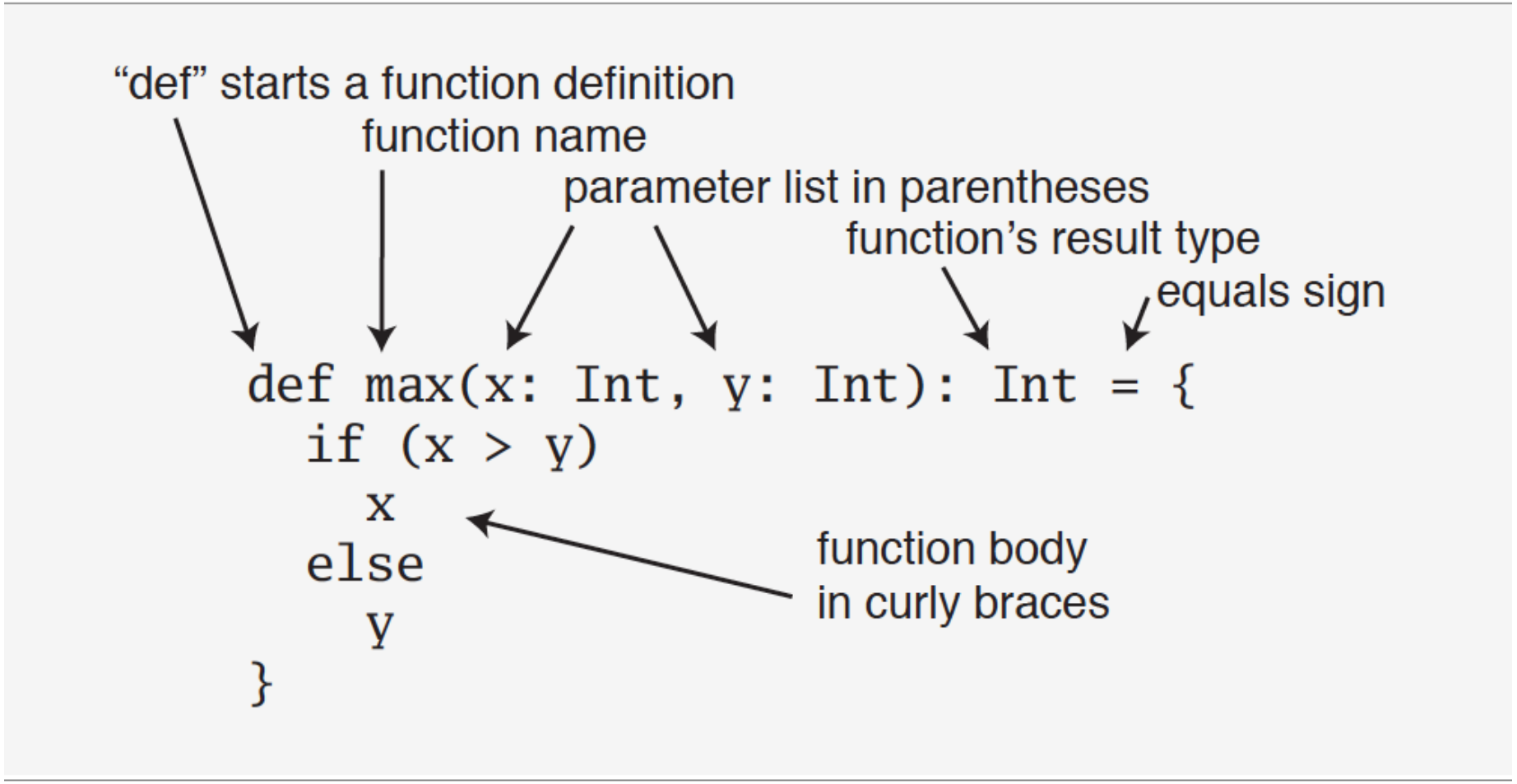
```
0.0
1e30f
3.14159f
1.0e100
.1
```

Functions

- Syntax

```
def functionName ([list of parameters]) : [return type] = {
   function body
   return [expr]
}
```

Define functions in Scala:



max: (Int,Int)Int

The equals sign that precedes the body of a function hints that in the functional world view, a function defines an expression that results in a value.

```
object Demo {
   def main(args: Array[String]) {
      println( "Returned Value : " + addInt(5,7) );
   }

   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

Functions – Contd.

Specifying result type is not compulsory unless if it is recursive. Compiler can infer result type automatically.

if a function consists of just one statement, you can optionally leave off the curly braces. Thus, you could alternatively write the max function like this:

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y max2: (Int,Int)Int
```

Once you have defined a function, you can call it by name, as in:

```
scala> max(3, 5)
res6: Int = 5
```

Functions – Contd.

- Here's the definition of a function that takes no parameters and returns no interesting result:
- scala> def greet() = println("Hello, world!")

greet: ()Unit

When you define the greet() function, the interpreter will respond with greet: ()Unit.

"greet" is, of course, the name of the function. The empty parentheses indicate the function takes no parameters. And Unit is greet's result type.

Functions – Contd.

- A result type of Unit indicates the function returns no interesting value. **Scala's Unit type is similar to Java's void type**
- Methods with the result type of Unit, therefore, are only executed for their side effects. In the case of greet(), the side effect is a friendly greeting printed to the standard output.

If you wish to exit the interpreter, you can do so by entering :quit or :q.

- **scala> :quit $**

Thank You

Principles of Programming Languages

Manju Venugopalan
Dept of CSE, Amrita School of Engg, Bangalore

Scala

Basic Syntax and Coding Conventions

Case Sensitivity − Scala is case-sensitive

Class Names − For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
Example − class MyFirstScalaClass.

Method Names − All method names should start with a Lower Case letter. If multiple words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
Example − def myMethodName()

Program File Name − Name of the program file should exactly match the object name. When saving the file you should save it using the object name and append '**.scala**' to the end of the name.

Example − Assume 'HelloWorld' is the object name. Then the file should be saved as 'HelloWorld.scala'.

def main(args: Array[String]) − Scala program processing starts from the main() method which is a mandatory part of every Scala Program.

First program

```
object HelloWorld {
  /*
   This will print 'Hello World' as the output
  */
  def main(args: Array[String]) {
    println("Hello, world!") // prints Hello World
  }
}
```

Identifiers in Scala

- Names used for objects, classes, variables and methods are called identifiers.
- A keyword cannot be used as an identifier and identifiers are case-sensitive.
- Scala supports four types of identifiers.

Alphanumeric identifiers

Valid:

age12, Ahe_12, _12

Invalid:

$sal, 13abc, -12

Operator Identifiers

An operator identifier consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #

Valid: + ++ ::: <?> :>

Mixed Identifiers

A mixed identifier consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier.

Valid: unary_+ myval_=

Literal Identifiers

A literal identifier is an arbitrary string enclosed in back ticks (` `).

Valid: `y` ` abc` ` deci`

Single and Multiline comments

```
object HelloWorld {
   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */
   def main(args: Array[String]) {
      // Prints Hello World
      // This is also an example of single line comment.
      println("Hello, world!")
   }
}
```

Scala- DataTypes

1	**Byte** **8 bit signed value. Range from -128 to 127**
2	Short 16 bit signed value. Range -32768 to 32767
3	Int 32 bit signed value. Range -2147483648 to 2147483647
4	Long 64 bit signed value. -9223372036854775808 to 9223372036854775807
5	Float 32 bit IEEE 754 single-precision float
6	Double 64 bit IEEE 754 double-precision float
7	Char 16 bit unsigned Unicode character. Enclosed within a single quote

8	String A sequence of Chars enclosed within double quotes Multiline strings to be enclosed in a pair of triple quotes Var a:String = “““ abc deeeeef ”””
9	Boolean Either the literal true or the literal false
10	Unit Corresponds to no value
11	Null null or empty reference
12	Nothing The subtype of every other type; includes no values
13	Any The supertype of any type; any object is of type Any
14	AnyRef The supertype of any reference type

Examples

- Scala>var x: Int=x23
- Scala> var x:Float=23.5f
- Scala> var x:Double =23.5
- Scala> var x:Char='D'
- Scala> var str: String="abcd"
- Scala> var y:Boolean=true

Thank You

Growing new types:

- Scala allows users to grow and adapt the language in the directions they need by defining easy-to-use libraries that feel like native language support.
- Scala lets you add new types that can be used as conveniently as built-in types

- Example: BigInt (Page No. 47)

Scala Language

- The name Scala stands for “scalable language.” The language is so named because it was designed to grow with the demands of its users.

- Technically, Scala is a blend of object-oriented and functional programming concepts.

- Instead of providing all constructs you might ever need in one “perfectly complete” language, Scala puts the tools for building such constructs into your hands

Scala is object oriented:

- Scala is an object-oriented language in pure form: every value is an object and every operation is a method call. For example, when you say 1 + 2 in Scala, you are actually invoking a method named + defined in class Int

- You can define methods with operator-like names

- Scala is more advanced than most other languages when it comes to composing objects. An example is Scala's traits. Traits are like interfaces in Java, but they can also have method implementations and even field

- Methods should not have any side effects. They should communicate with their environment only by taking arguments and returning results.

- For instance, the replace method in Java's String class fits this description. It takes a string and two characters and yields a new string where all occurrences of one character are replaced by the other. There is no other effect of calling replace.

- Methods like replace are called referentially transparent, which means that for any given input the method call could be replaced by its result without affecting the program's semantics

Scala is functional: (Functions are first-class values)

- In a functional language, a function is a value of the same status as, say, an integer or a string.
- You can pass functions as arguments to other functions, return them as results from functions, or store them in variables.

- You can also define a function inside another function
- You can define functions without giving them a name

Immutable data structures are one of the cornerstones of functional programming.

- Immutable data structures are one of the cornerstones of functional programming.
- The Scala libraries define many more immutable data types on top of those found in the Java APIs.
- For instance, Scala has immutable lists, tuples, maps, and sets

Referentially transparent method:
Method can be referentially transparent, if a call to this method may be replaced by its return value.

```
int add(int a, int b)
{
return a + b
}
int mult(int a, int b)
{
return a * b;
}
int x = add(2, mult(3, 4));
```

Here, function call mult(3,4) can be replaced by 3*4. Doing this replacement will not change the result of x. The same way, function call add(2,mult(3,4)) can be replaced by 2+3*4. This replacement will not the change the result of x.

On the other hand, consider the following program:

```
int add(int a, int b)
{
   int result = a + b;
   System.out.println("Returning " + result);
   return result;
}
```

Replacing a call to the add method with the corresponding return value will change the result of the program, since the message will no longer be printed.

Why Scala

- Compatibility
- Brevity
- high-level abstractions
- advanced static typing

Why Scala?

Compatibility: Scala is designed for seamless interoperability with Java.

Scala programs compile to JVM bytecodes.

Scala code can call Java methods, access Java fields, inherit from Java classes and implement Java interfaces.

- Scala heavily re-uses Java types. Scala’s Ints are represented as Java primitive integers of type int,

- Floats are represented as floats, Booleans as Booleans, and so on. Scala arrays are mapped to Java arrays. Scala also re-uses many of the standard Java library types. For instance, the type of a string literal in "abc“ in Scala is java.lang.String, and a thrown exception must be a subclass of java.lang.Throwable

- Scala not only re-uses Java's types, but also dresses them up to make them nicer.

Example: Page number 55.

Scala code can also be invoked from Java

Scala is concise:

Scala programs tend to be short. Scala programmers have reported reductions in number of lines of up to a factor of ten compared to Java.

These might be extreme cases. A more conservative estimate would be that a typical Scala program should have about half the number of lines of the same program written in Java. Fewer lines of code mean not only less typing, but also less effort at reading and understanding programs and fewer possibilities of defects.

Example: Page Number 56

- Scala's type inference is another factor that contributes to its conciseness.

- Scala is high-level: Example: Page no:57

- Scala is statically typed: types checked before run time

- C++ is another scalable language that can be adapted and extended through operator overloading and its template system; compared to Scala it is built on a lower-level, more systems-oriented core.

Scala Installation

Go to http://www.scala-lang.org/downloads and follow the directions for your platform.

You can also use a Scala plug-in for Eclipse, IntelliJ or NetBeans.

Scala interpreter: Type scala in command prompt to use scala interpreter.
Example: Page number 65

Scala interpreter

```
$ scala
Welcome to Scala version 2.11.7
Type in expressions to have them evaluated.
Type :help for more information.
scala>

scala> 1 + 2
The interpreter will print:
res0: Int = 3

scala> println("Hello, world!")
Hello, world!
```

Define Variables

2 kinds: val and var

val - Once initialized, a val can never be reassigned.
var - can be reassigned throughout its lifetime.

Example for val definition:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

- This statement introduces msg as a name for the string "Hello, world!". The type of msg is java.lang.String, because Scala strings are implemented by Java's String class.

Define Variables – Contd.

Type inference: It can figure out type of the variable automatically. You can however specify a type explicitly if you wish.

msg is initialized with a string literal. So, Scala inferred the type of msg to be String.

In Scala, a variable's type is specified after its name, separated by a colon. Example:

Page No 67: Difference between val and var.

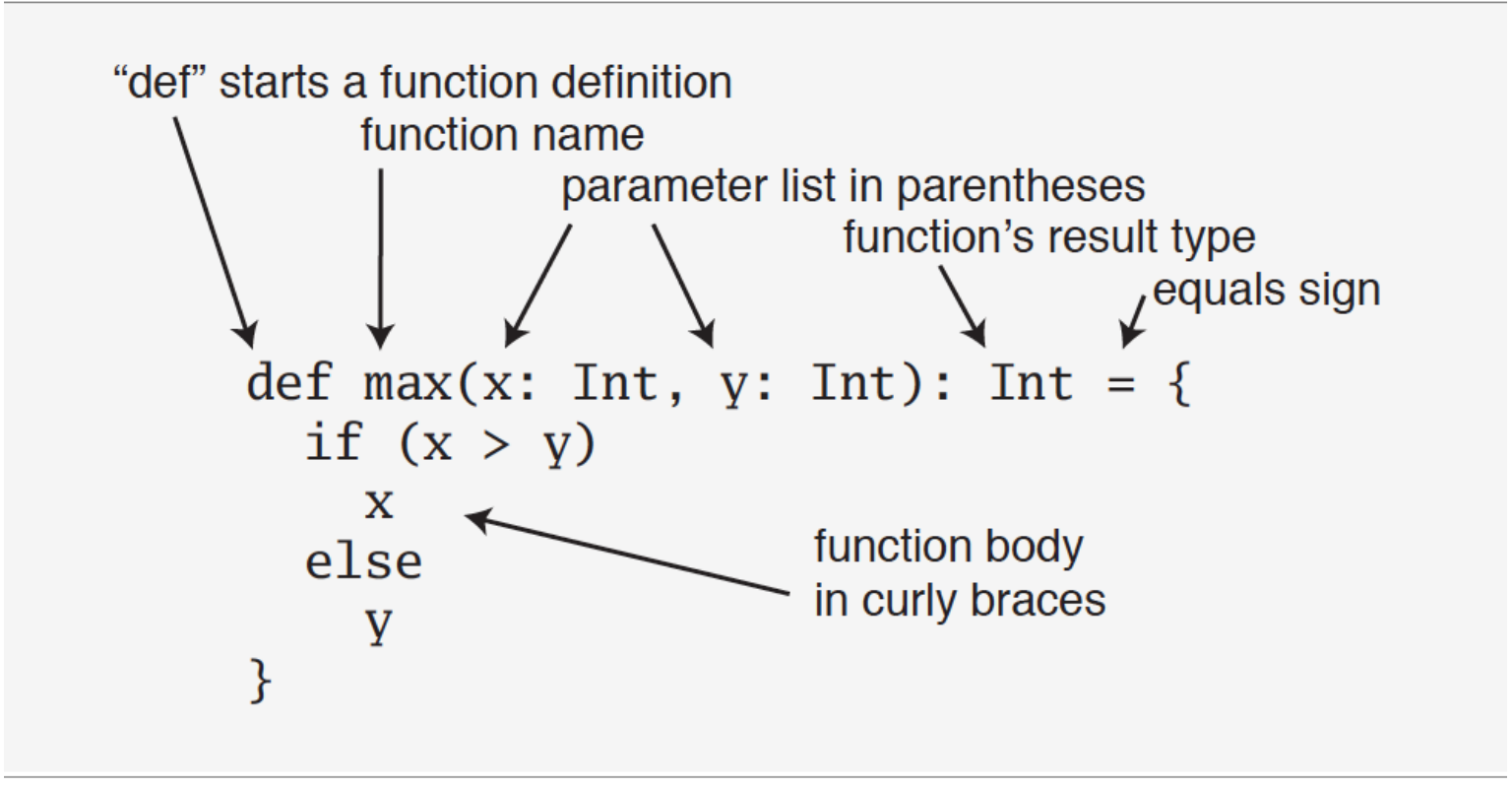
Variables/Values Examples

```
scala> val msg2: java.lang.String = "Hello again, world!"
msg2: String = Hello again, world!

scala> msg = "Goodbye cruel world!"
<console>:8: error: reassignment to val
msg = "Goodbye cruel world!"

scala> var greeting = "Hello, world!"
greeting: String = Hello, world!

scala> greeting = "Leave me alone, world!"
greeting: String = Leave me alone, world!
```

Variables/Values Examples – Contd.

```
scala> val multiLine =
| "This is the next line."
multiLine: String = This is the next line.

scala> val oops =
|
|
You typed two blank lines. Starting a new command.
scala>
```

Define functions in Scala:



max: (Int,Int)Int

The equals sign that precedes the body of a function hints that in the functional world view, a function defines an expression that results in a value.

Functions – Contd.

Specifying result type is not compulsory unless if it is recursive. Compiler can infer result type automatically.

if a function consists of just one statement, you can optionally leave off the curly braces. Thus, you could alternatively write the max function like this:

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y max2: (Int,Int)Int
```

Once you have defined a function, you can call it by name, as in:

```
scala> max(3, 5)
res6: Int = 5
```

Functions – Contd.

- Here's the definition of a function that takes no parameters and returns no interesting result:
- scala> def greet() = println("Hello, world!")

greet: ()Unit

When you define the greet() function, the interpreter will respond with greet: ()Unit.

"greet" is, of course, the name of the function. The empty parentheses indicate the function takes no parameters. And Unit is greet's result type.

Functions – Contd.

- A result type of Unit indicates the function returns no interesting value. **Scala's Unit type is similar to Java's void type**
- Methods with the result type of Unit, therefore, are only executed for their side effects. In the case of greet(), the side effect is a friendly greeting printed to the standard output.

If you wish to exit the interpreter, you can do so by entering :quit or :q.

- **scala> :quit $**

Scala Scripts

- A script is just a sequence of statements in a file that will be executed sequentially.
- Put this into a file named hello.scala:

```
println("Hello, world, from a script!")
```

- Then run: $ scala hello.scala

Scala Scripts – Contd.

- Command line arguments to a Scala script are available via a Scala array

named args.

- Arrays are zero based, and you access an elementby specifying an index in parentheses

```
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

then run:

```
$ scala helloarg.scala planet
```

Hello, planet!

Loop with while; decide with if

To try out a while, type the following into a file named printargs.scala:

```
var i = 0
while (i < args.length) {
        println(args(i))
        i += 1
}
```

$ scala printargs.scala Scala is fun

O/P

Scala

is

fun

i++ or ++i won't work in SCALA

Iterate with foreach and for

- Type the below code into a new file named pa.scala

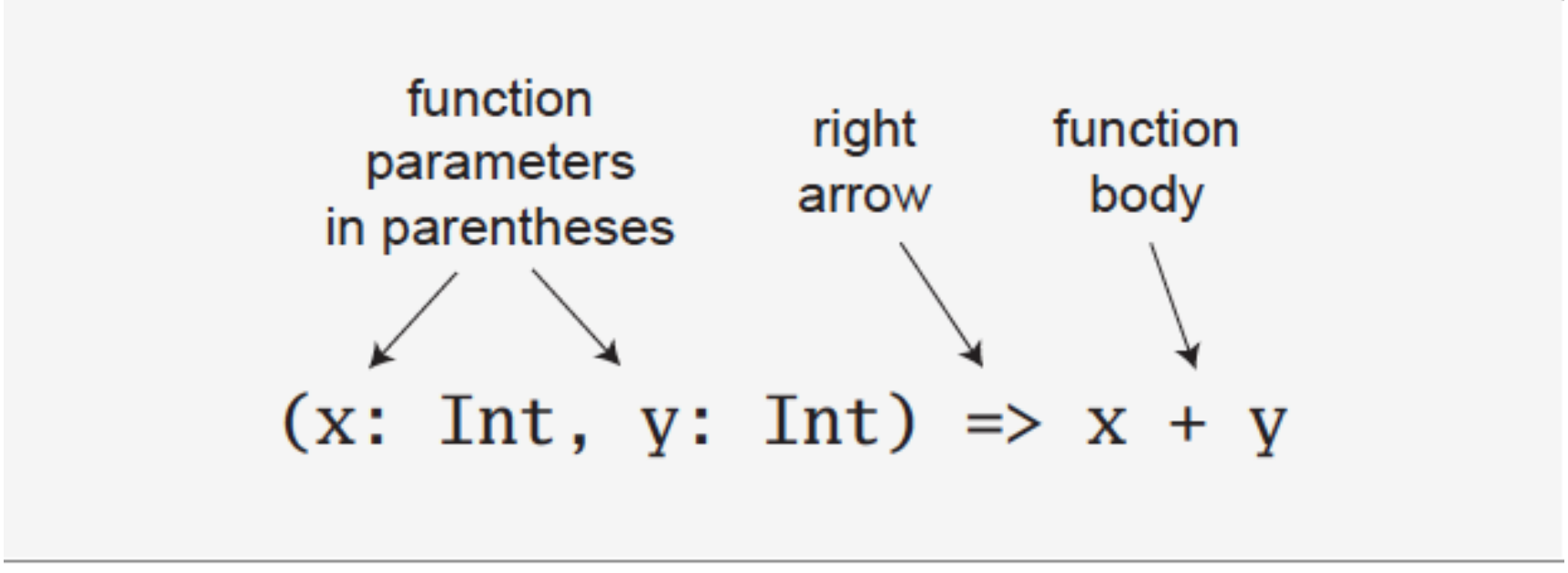
```
args.foreach(arg => println(arg))
OR args.foreach((arg: String) => println(arg))
```

- In this code, you call the foreach method on args and pass in a function.
- The body of the function is println(arg).
- Execute with the command:

```
$ scala pa.scala Concise is nice
You should see:
        Concise
        is
        nice
        In the
```

- args.foreach((arg: **String**) => println(arg))

The syntax of a function literal in Scala.



For

```
for (arg <- args)
println(arg)
```

- The parentheses after the “for” contain arg <args.
- To the right of the <- symbol is the args array.
- To the left of <- is “arg”, the name of a val, not a var.

for each element of the args array, a new arg val will be created and initialized to the element value, and the body of the for will be executed.

If you run the forargs.scala script

$ scala forargs.scala for arg in args

You’ll see:

for
arg
in
args

Chapter 3

new: It can instantiate/create objects.

When you instantiate object, you can parameterize/configure it with values and types.

Example for instantiating an object and parameterize it with the value 12345.

val big = new java.math.BigInteger("12345")

Parameterize arrays with types

Parameterize an instance with both a type and a value:

First type should come in square bracket and followed by the value in parentheses:

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
for (i <- 0 to 2)
  print(greetStrings(i))
```

Arrays in Scala are accessed by placing the index inside parentheses, not square brackets as in Java

We cannot reassign greetStrings to a different array. It will always point to the same Array[String] instance with which it was initialized. But you can change the elements of that over time, so the array itself is Array[String] mutable.

In this for (i <0 to 2), **to** is actually a method

0 to 2 is transformed into the method call (0).to(2)

Why?

Scala does not support operator overloading

- +, -, * and / are like methods.
- When you type 1+2 into the Scala interpreter, you are actually invoking a method named + on the int object 1 and passing in 2 as a parameter.
- It is equivalent to (1).+(2)

Why arrays are accessed with parentheses in Scala?

apply(): When you apply parentheses surrounding one or more values to a variable, Scala will transform the code into an invocation of a method named **apply** on that variable.

So, *greetStrings(i)* is transformed into *greetStrings.apply(i)*. Thus, accessing an element of an array in Scala is simply a method call.

This principle applies to any application of an object to some arguments in parentheses will be transformed to an apply method call.

update(): When an assignment is made to a variable to which parentheses and one/more arguments have been applied, the compiler will transform into an invocation of an update method that takes the arguments in parentheses as well as the object to the right of the equals sign.

greetStrings(0) = "Hello“ will be transformed into greetStrings.update(0, "Hello")

```
val greetStrings = new Array[String](3)
greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")
for (i <0.to(2))
  print(greetStrings.apply(i))
```

Scala treats everything from arrays to expressions as objects with methods

Concise way to create and initialize arrays:

val numNames = Array("zero", "one", "two")

The above code creates a new array of length three, initialized to the passed strings “zero”, “one” and “two”. Here type of the array is Array[String] and that is automatically inferred because we passed strings to it.

val numNames = Array("zero", "one", "two") is equivalent to val numNames = Array.apply("zero", "one", "two")

- If you’re a Java programmer, you can think of this as calling a static method named *apply* on class *Array*

List:

- A Scala array is a mutable sequence of objects that all share the same type.
- We can't change the length of an array after it is instantiated but we can change its element values.
- List is a immutable sequence of objects that all share the same type.
  Java's List is mutable but Scala's List is immutable.

How to create a list in Scala?

```
val oneTwoThree = List(1,2,3)
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new list.")
```

O/P

List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list

:: cons

Cons prepends a new element to the beginning of an existing list and returns the resulting list.

For example, if you run this script:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)

val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Nil is an empty list here. It is required here because :: is defined in class List. But 1, 2 and 3 are int here.

Tuples:

- Tuples are immutable,
- Tuples can contain different types of elements.
- A tuple could contain both an integer and a string at the same time.

- To instantiate a new tuple :

```
val pair = (99, "Luftballons")
```

- To access its elements:

```
println(pair._1)
println(pair._2)
```

The actual type of a tuple depends on the number of elements it contains and the types of those elements.

The type of (99, "Luftballons") is Tuple2[Int, String].
The type of ('u', 'r', "the", 1, 4, "me") is Tuple6[Char, Char, String, Int, Int, String].

Accessing the elements of a tuple
You may be wondering why you can't access the elements of a tuple like the elements of a list, for example, with "pair(0)".

The reason is that a list's apply method always returns the same type, but each element of a tuple may be a different type: _1 can have one result type, _2 another, and so on

Collection classes: Set and Map

Set: Immutable Set(it never changes. But you have still operations that simulate additions, removals or updates, but those operations will in each case return a new collection and leave the old collection unchanged) and Mutable Set(It can be updated/extended in place)

Immutable Set creation example

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

Mutable Set creation example

```
import scala.collection.mutable
val movieSet = mutable.Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

Map:

Creation of mutable Map

```
import scala.collection.mutable
val treasureMap = mutable.Map[Int, String]()
treasureMap += (1 ->"Go to island.")
treasureMap += (2 ->"Find big X on ground.")
treasureMap += (3 ->"Dig.")
println(treasureMap(2))
```

Creation of immutable Map

```
val romanNumeral = Map(1 ->"I", 2 ->"II", 3 ->"III", 4 ->"IV", 5 ->"V")
println(romanNumeral(4))
```

- **One way to move towards a functional style is to try to program without vars**

- Scala is not a pure functional language that forces you to program everything in the functional style. Scala is a hybrid imperative/functional language

- **A balanced attitude for Scala programmers**

Prefer vals, immutable objects, and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.

- Preferring methods without side effects can help make your programs easier to test.

Functional style

```
def printArgs(args: Array[String]): Unit = {
var i = 0
while (i < args.length) {
println(args(i))
i += 1
} }
```

- **TO Write in functional Style**

```
def printArgs(args: Array[String]): Unit = {
for (arg <args)
println(arg)
}
```

or this:

```
def printArgs(args: Array[String]): Unit = {
args.foreach(println)
}
```

This example

What is side effect?

A side effect is generally defined as mutating state somewhere external to the method or performing an I/O action.

A method that is executed only for its side effects is known as a procedure.

A function without side effects or vars.

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```

Read lines from a file: To read lines from a file and prints them out prepended with the number of characters in each line

```
import scala.io.Source

if (args.length > 0) {
  for (line <- Source.fromFile(args(0)).getLines())
      println(line.length + " " + line)
}
else
  Console.err.println("Please enter filename")
```

The expression **Source.fromFile(args(0))** attempts to open the specified file and returns a **Source** object, on which you call **getLines**. The **getLines** method returns an **Iterator[String]**, which provides one line on each iteration, excluding the end-of-line character. The **for** expression iterates through these lines and prints for each the length of the line, a space, and the line itself.

To run: $ scala countchars1.scala countchars1.scala (Show the output in textbook)

Chapter 4
Classes and Objects

Programs on Array

Write a Scala program to sum values of an given array.

```
object Scala_Array {
   def main(args: Array[String]): Unit = {
     var nums = Array(1.2, 1.7, 1.12, 1.16, 1.81, 1.99)
     println("Original Array elements:")
     // Print all the array elements
      for ( x <- nums ) {
         print(s"${x}, ")
       }
     println("\nUsing sum():")
     val result = nums.sum
     println(s"Result: ${result}");
     println("\nUsing for loop:")
     var total = 0.0;
      for ( i <- 0 to (nums.length - 1)) {
         total += nums(i);
      }
      println(s"Result: ${total}");
   }
}
```

Write a Scala program to calculate the average value of an array of element.

```
object Scala_Array {
   def main(args: Array[String]): Unit = {
     var nums = Array(1, 2, 3, 4, 5, 6)
     println("Original Array elements:")
     // Print all the array elements
      for ( x <- nums ) {
         print(s"${x}, ")
       }
     //Sum using for loop
     var total = 0.0;
      for ( i <- 0 to (nums.length - 1)) {
         total += nums(i);
      }
      println(s"\nAverage value of the array elements is:
${total/nums.length}");
   }
}
```

Write a Scala program to find the index of an element in a given Array.

```
object Scala_Array
{
```

```
  def main(args: Array[String]): Unit =
 {
   val colors = Array("Red","Blue","Black","Green","White")
   println("Original Array elements:")
   // Print all the array elements
      for ( x <- colors ) {
         print(s"${x}, ")
       }
   println("\n")
   println("Index of 'Red':", colors.indexOf("Red"))
   println("Index of 'Blue':", colors.indexOf("Blue"))
   println("Index of 'Black':", colors.indexOf("Black"))
   println("Index of 'Green':", colors.indexOf("Green"))
   println("Index of 'White':", colors.indexOf("White"))
  }
}
```

Write a Scala program to remove a specific element from an given array.

```
object Scala_Array
{
  def main(args: Array[String]): Unit =
 {
   val colors = Array("Red","Blue","Black","Green","White")
   println("Original Array elements:")
   // Print all the array elements
      for ( x <- colors ) {
         print(s"${x}, ")
       }
     println("\nReplace some elements with ''/null etc.:")
      colors(0) = ""
      colors(3) = null
   println("Now the Original Array becomes:")
   // Print all the array elements
      for ( x <- colors ) {
         print(s"${x}, ")
       }
  }
}
```

Write a Scala program to reverse an array of integer values.

```
object Scala_Array {
  def test(nums: Array[Int]): Array[Int] = {
    var temp1 = 0
    var temp2 = 0
    var index_position = 0
    var index_last_pos = nums.length - 1
    while (index_position < index_last_pos) {
```

```
    temp1 = nums(index_position)
    temp2 = nums(index_last_pos)
    nums(index_position) = temp2
    nums(index_last_pos) = temp1
    index_position += 1
    index_last_pos -= 1
     }
    nums
  }
   def main(args: Array[String]): Unit = {
    var nums1 = Array(1789, 2035, 1899, 1456, 2013)
    println("Orginal array:")
    for ( x <- nums1) {
       print(s"${x}, ")
     }
    var result1= test(nums1)
    println("\nReversed array:")
    for ( x <- result1) {
       print(s"${x}, ")
     }
    var nums2 = Array(1789, 2035, 1899, 1456)
    println("\nOrginal array:")
    for ( x <- nums2) {
       print(s"${x}, ")
     }
    var result2= test(nums2)
    println("\nnReversed array:")
    for ( x <- result2) {
       print(s"${x}, ")
    }
   }
}
```

Programs on List

Write a Scala program to delete element(s) from a given List.

```
object Scala_List
{
  def main(args: Array[String]): Unit =
 {
   val nums = List(1, 3, 5, 7, 9, 11, 14, 12)
   println("Original list:")
   println(nums)
  //As scala List is immutable, so we can't delete elements from it,
but
  //filter out element(s) as per requirement.
```

```
    println("Filter out 3 from the above list:")
    val nums1 = nums.filter(_ != 3)
    println(nums1)
    println("Filter out numbers which are greater than 10:")
    val nums2 = nums.filter(_ > 10)
    println(nums2)
  }
}
```

Write a Scala program to delete element(s) from a given List.

```
object Scala_List
{
  def main(args: Array[String]): Unit =
 {
    val nums = List(1, 3, 5, 7, 9, 11, 14, 12)
    println("Original list:")
    println(nums)
  //As scala List is immutable, so we can't delete elements from it,
but
  //filter out element(s) as per requirement.
    println("Filter out 3 from the above list:")
    val nums1 = nums.filter(_ != 3)
    println(nums1)
    println("Filter out numbers which are greater than 10:")
    val nums2 = nums.filter(_ > 10)
    println(nums2)
  }
}
```

Write a Scala program to iterate over a list to print the elements and calculate the sum and product of all elements of this list

```
object Scala_List
{
def main(args: Array[String]): Unit =
 {
    //Iterate over a list
    val nums = List(1, 3, 5, 7, 9)
    println("Iterate over a list:")
    for( i <- nums)
    {
     println(i)
    }

    println("Sum all the items of the said list:")
    //Applying sum method
    val result = nums.sum
    println(result)

    println("Multiplies all the items of the said list:")
```

```
    val result1 = nums.product
    println(result1)
  }
}
```

Write a Scala program to find the largest and smallest number from a given list.

```
object Scala_List
{
def main(args: Array[String]): Unit =
 {
    //Iterate over a list
    val nums = List(1, 3, 5, 7, 9, 11, 14, 12)
    println("Original list:")
    println(nums)
    println("Largest number of the said list:")
    println(nums.max)
    println("Smallest number from the said list:")
    println(nums.min)
  }
}
```

Write a Scala program to remove duplicates from a given list.

```
object Scala_List
{
def main(args: Array[String]): Unit =
 {
    val nums = List(1, 3, 5, 2, 7, 9, 11, 5, 2, 14, 12, 3)
    println("Original list:")
    println(nums)
    val result1 = nums.distinct
    println("Unique elements of the said list:")
    println(result1)
    val chars = List("a", "a", "b", "c", "d", "c", "e", "f")
    println("Original list:")
    println(chars)
    val result2 = chars.distinct
    println("Unique elements of the said list:")
    println(result2)
  }
}
```

Write a Scala program to find the first and last element of given list.

```
object Scala_List
{
  def main(args: Array[String]): Unit =
```

```
 {
   val colors = List("Red", "Blue", " Black ", "Green", " White",
"Pink")
   println("Original list:")
   println(colors)
   println("First element of the said list: " + colors.head)
   println("Last element of the said list: " + colors.last)
  }
}
```

Write a Scala program to find the index of an element in a given list.

```
object Scala_List
{
 def main(args: Array[String]): Unit =
 {
  val colors = List("Red","Blue","Black","Green","White")
  println("Original lists:")
  println(colors)
  println("Index of 'Red':", colors.indexOf("Red"))
  println("Index of 'Blue':", colors.indexOf("Blue"))
  println("Index of 'Black':", colors.indexOf("Black"))
  println("Index of 'Green':", colors.indexOf("Green"))
  println("Index of 'White':", colors.indexOf("White"))
 }
}
```

Write a Scala program to find the even and odd numbers from a given list.

```
object Scala_List
{
  def main(args: Array[String]): Unit =
 {
   val nums = List(1, 2, 3, 4, 5, 7, 9, 11, 14, 12, 16)
   println("Original list:")
   println(nums)
   val even_nums = nums.filter(_ % 2 ==0)
   println("Even number of the said list:")
   println(even_nums)
   val odd_nums = nums.filter(_ % 2 != 0)
   println("Odd number of the said list:")
   println(odd_nums)
  }
}
```

Write a Scala program to reverse a given list.

```
object scala_basic {

  def main(args: Array[String]): Unit = {
```

```
   val nums = List(1,2,3,4,5,6,7,8,9,10)
   println("Original List")
   println(nums)
   println("Reversed the said list:")
   println("Using reverse() function:")
   println(nums.reverse)
   println("Using for loop:")
   for(n<-nums.reverse)
     {
      print(n)
      print(" ")
     }
   }
}
```

Write a Scala program to check a given list is a palindrome or not

```
object Scala_List {

  def is_Palindrome[A](list_nums: List[A]):Boolean = {
    list_nums == list_nums.reverse
}

   def main(args: Array[String]): Unit = {
         println("Result: " + is_Palindrome(List(1,2,3,4,3,2,1)));
         println("Result: " + is_Palindrome(List(1,2,3)));
       }
}
```

Write a Scala program to count the number of occurrences of each element in a given list.

```
object Scala_List {

  def list_elemnt_occurrences[A](list1:List[A]):Map[A, Int] = {
      list1.groupBy(el => el).map(e => (e._1, e._2.length))
    }

  def main(args: Array[String]): Unit = {

println(list_elemnt_occurrences(List(1,1,1,2,2,3,6,4,4,6,1,6,2)))
     println(list_elemnt_occurrences(List("Red", "Green", "White",
"Black", "Red", "Green", "Black")))
    }
}
```

Chapter 4
Classes and Objects

Class:
It is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword **new**.

For example, given the class definition:

```
class ChecksumAccumulator {
  // class definition goes here
}
```

A **checksum** is a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage.
A **checksum** is a value used to verify the integrity of a file or a data transfer

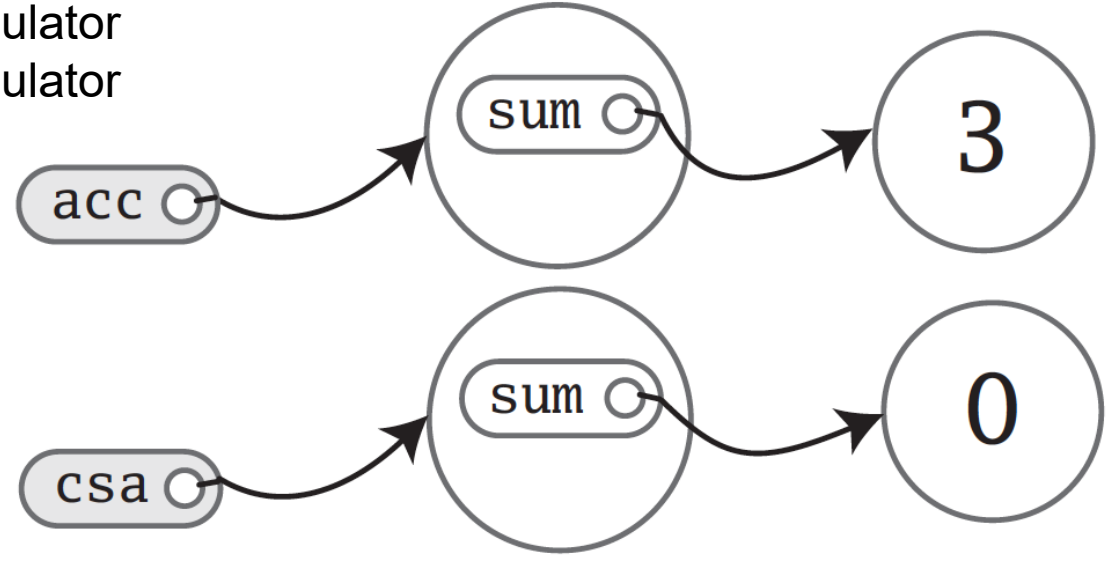
You can create ChecksumAccumulator objects with:
new ChecksumAccumulator

Members: fields and methods which are kept inside a class definition

Fields/instance variables: They can be defined with either **val or var**. They are variables.
Methods: We define methods with **def** and it contains executable code.
Fields hold the state or data of an object, whereas the methods use that data to do the computational work of the object.

Class - Objects

```
class ChecksumAccumulator {
var sum = 0
}

val acc = new ChecksumAccumulator
val csa = new ChecksumAccumulator
```



acc.sum = 3

Private and Public

public is Scala's default access level

return: return is not required to return a value from a method. In the absence of any explicit return statement, a Scala method returns the last value computed by the method. Scala also allows you to write methods that have multiple, explicit returns if that's what you desire.

result type: We can leave off the curly braces if a method computes only a single result expression. We can leave off the result type and Scala will infer it. Although Scala compiler will correctly infer the result types of the methods, it is often better to explicitly provide the result types of public methods declared in a class even when the compiler would infer it for you.

Private Field

```
class ChecksumAccumulator {
private var sum = 0
}
```

Given this definition of ChecksumAccumulator, any attempt to access sum
from the outside of the class would fail

```
val acc = new ChecksumAccumulator
acc.sum = 5 // Won't compile, because sum is private
```

Parameter in Methods are Val

Method parameters in Scala is that they are vals, not vars

```
def add(b: Byte): Unit = {
b = 1 // This won't compile, because b is a val
sum += b
}
```

Semicolon inference

```
val s = "hello"; println(s)
```

A semicolon is required if you write multiple statements on a single line:

Singleton object

- Scala has singleton object instead of Static in Java
- A singleton object definition looks like a class definition except instead of the keyword **class** we use the keyword **object**.
- When a singleton object shares the same name with a class, it is called that class's **companion object**.
- One must define both the class and its companion object in the same source file.
- The class is called the **companion class** of the singleton object.
- A class and its companion object can access each other's private members.

Companion object for class ChecksumAccumulator.

```
// In file ChecksumAccumulator.scala
import scala.collection.mutable
object ChecksumAccumulator {
  private val cache = mutable.Map.empty[String, Int]
  def calculate(s: String): Int =
    if (cache.contains(s))
      cache(s)
    else {
      val acc = new ChecksumAccumulator
      for (c <- s)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s ->cs)
      cs
      }
 }
```

```
// In file ChecksumAccumulator.scala

class ChecksumAccumulator {
private var sum = 0
def add(b: Byte): Unit = { sum += b }
def checksum(): Int = ~(sum & 0xFF)
+ 1
}
```

If you are a Java programmer, one way to think of singleton objects is as the home for any static methods you might have written in Java. You can invoke methods on singleton objects using a similar syntax: the name of the singleton object, a dot, and the name of the method

Example:

You can invoke the method of singleton object calculate ChecksumAccumulator like this:

```
ChecksumAccumulator.calculate("Every value is an object.")
```

Class Vs. Singleton Objects

- Singleton objects cannot take parameters, whereas classes can.
- Because you can’t instantiate a singleton object with the keyword new, you have no way to pass parameters to it.
- A singleton object is initialized the first time some code accesses it.

Standalone object:
A singleton object that does not share the same name with a companion class is called a standalone object.
You can use standalone objects for many purposes, including collecting related utility methods together or defining an entry point to a Scala application

Scala Application

```
// In file Summer.scala

import ChecksumAccumulator.calculate
object Summer {
    def main(args: Array[String]) = {
       for (arg <- args)
            println(arg + ": " + calculate(arg))
      }
}
```

```
$ scalac ChecksumAccumulator.scala Summer.scala
OR
$ fsc ChecksumAccumulator.scala Summer.scala

$ scala Summer of love
```

Chapter 5
Basic Types and Operations

Data Types

1. integral types
    - Byte
    - Short
    - Int
    - Long,
    - Char
2. The integral types plus **Float** and **Double** are called **numeric types**.
3. String

Data Types in Scala Contd.

Basic type	Range
`Byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`Short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`Int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`Long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`Char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`String`	a sequence of `Char`s
`Float`	32-bit IEEE 754 single-precision float
`Double`	64-bit IEEE 754 double-precision float
`Boolean`	`true` or `false`

Literals

- Integer literals
- Number begins with a 0x or 0X, it is hexadecimal(base 16)
- scala> val hex2 = 0x00FF
- hex2: Int = 255
- If an integer literal ends in an L or l, it is a Long; otherwise it is an Int

  scala> val tower = 35L

  tower: Long = 35

  scala> val of = 31l

  of: Long = 31

Literals – Contd.

Floating point literals

```
scala> val big = 1.2345
big: Double = 1.2345

scala> val bigger = 1.2345e1
bigger: Double = 12.345
scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47

scala> val little = 1.2345F
little: Float = 1.2345
scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0
scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

Character literals

```
scala> val a = 'A'
a: Char = A
```

write \u
followed by four hex digits with the code point

```
scala> val d = '\u0041'
d: Char = A
```

Character - Literals

Literal Meaning
\n line feed (\u000A)
\b backspace (\u0008)
\t tab (\u0009)
\f form feed (\u000C)
\r carriage return (\u000D)
\" double quote (\u0022)
\' single quote (\u0027)
\\ backslash (\u005C)

```
scala> val backslash = '\\'
backslash: Char = \
```

Sting Literals

```
println("""Welcome to Ultamix 3000.
             Type "HELP" for help.""")
Welcome to Ultamix 3000.
          Type "HELP" for help.
```

***the leading spaces before the second line are included in the string!

```
println("""|Welcome to Ultamix 3000.
             |Type "HELP" for help.""".stripMargin)

Welcome to Ultamix 3000.
Type "HELP" for help.
```

```
scala> val bool = true
bool: Boolean = true
scala> val fool = false
fool: Boolean = false
```

Boolean literals

```
scala> s"The answer is ${6 * 7}."
res0: String = The answer is 42.
```

String Literals

```
println(raw"No\\\\escape!")
// prints: No\\\\escape!

scala> f"${math.Pi}%.5f"
res1: String = 3.14159
```

```
scala> val pi = "Pi"
pi: String = Pi
scala> f"$pi is approximately ${math.Pi}%.8f."
res2: String = Pi is approximately 3.14159265.
```

Operators are methods

```
scala> val sum = 1 + 2 // Scala invokes 1.+(2)
sum: Int = 3

scala> val s = "Hello, world!"
s: String = Hello, world!
scala> s indexOf 'o' // Scala invokes s.indexOf('o')
res0: Int = 4

scala> s indexOf ('o', 5) // Scala invokes s.indexOf('o', 5)
res1: Int = 8
```

Prefix & Post fix operations

In prefix notation, you put the method name before the object on which you are invoking the method.

for example, the ‘-’ in -7

In postfix notation, you put the method after the object

for example, the “toLong” in “7 toLong”

Prefix and postfix operators are unary: they take just one operand

The only identifiers that can be used as prefix operators are +, - , !, and ~

```
scala> 2.0
// Scala invokes (2.0).unary_res2:
Double = 2.0
scala> (2.0).unary_res3:
Double = 2.0
```

Arithmetic operations

You can invoke arithmetic methods via infix operator notation for:

addition (+)
subtraction (-),
multiplication (*)
division (/) and
remainder (%)

```
scala> 1.2 + 2.3
res6: Double = 3.5
scala> 3 - 1
res7: Int = 2
scala> 'b' - 'a'
res8: Int = 1
```

```
scala> 2L * 3L
res9: Long = 6
scala> 11 / 4
res10: Int = 2
scala> 11 % 4
res11: Int = 3
scala> 11.0f / 4.0f
res12: Float = 2.75
scala> 11.0 % 4.0
res13: Double = 3.0
```

Relational methods

- greater than (>),
- Less than (<),
- greater than or equal to (>=),
- less than or equal to (<=),
- unary ‘!’ operator (the
- unary_! method) to invert a Boolean value.

```
scala> 1 > 2
res16: Boolean = false
scala> 1 < 2
res17: Boolean = true
scala> 1.0 <= 1.0
res18: Boolean = true
scala> 3.5f >= 3.6f
res19: Boolean = false
scala> 'a' >= 'A'
res20: Boolean = true
scala> val untrue = !true
untrue: Boolean = false
scala> val toBe = true
toBe: Boolean = true
scala> val question = toBe || !toBe
question: Boolean = true
```

logical operations

the right-hand side of && and || expressions won't be evaluated if the left-hand side determines the result.

```
scala> def salt() = { println("salt"); false }
salt: ()Boolean
scala> def pepper() = { println("pepper"); true }
pepper: ()Boolean
scala> pepper() && salt()
pepper
salt
res21: Boolean = false
scala> salt() && pepper()
salt
res22: Boolean = false
```

In the first expression, pepper and salt are invoked, but in the second, only salt is invoked.

Bitwise operations

The bitwise methods are:
bitwise-and (&),
bitwise-or (|),
bitwise-xor (^).
The unary bitwise complement operator
(~, the method unary_~) inverts each bit in its operand

```
scala> 1 & 2
res24: Int = 0
scala> 1 | 2
res25: Int = 3
scala> 1 ^ 3
res26: Int = 2
scala> ~1
res27: Int = 2
```

Shift Operator

Scala integer types also offer three shift methods:
shift left (<<),
shiftright (>>), and
unsigned shift right (>>>).
The shift methods, when used in infix operator notation, shift the integer value on the left of the operator by the amount specified by the integer value on the right.

Object equality

To compare two objects

for equality, you can use either == or its inverse !=. Here are a few simple examples:

```
scala> 1 == 2
res31: Boolean = false
scala> 1 != 2
res32: Boolean = true
scala> 2 == 2
res33: Boolean = true
scala> ("he" + "llo") == "hello"
res40: Boolean = true
```

```
scala> List(1, 2, 3) == null
res38: Boolean = false
scala> null == List(1, 2, 3)
res39: Boolean = false
```

```
scala> 1 == 1.0
res36: Boolean = true
scala> List(1, 2, 3) == "hello"
res37: Boolean = false
```

Operator precedence and associativity

Given that Scala doesn't have operators, per se, just a way to use methods
in operator notation
Scala decides precedence based on the first character of the methods used in
Operator notation.

scala> 2 << 2 + 2
res41: Int = 32

<< will have lower precedence than +, and the expression will be evaluated by first invoking the + method, then the << method 2 << (2 + 2). 2 + 2 is 4, by our math, and 2 << 4 yields 32.

Table 5.3 · Operator precedence

(all other special characters)
* / %
+ −
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

Chapter 6
Functional Objects

A specification for class Rational

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3
scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

A specification for class Rational

- Each rational number will be represented by one Rational object.
- When you add two Rational objects, you'll create a new Rational object to hold the sum.

```
class Rational(n: Int, d: Int) {

println("Created " + n + "/" + d)}


scala> new Rational(1, 2)

Created 1/2

val res0: Rational = Rational@77eb5790
```

Reimplementing the toString method

- Class Rational inherits the implementation of toString defined in class java.lang.Object, which just prints the class name, an @ sign, and a hexadecimal number.
- Override the default implementation by adding a method toString to class Rational.

```
class Rational(n: Int, d: Int) {
override def toString = n + "/" + d
}
```

```
scala> new Rational(1, 2)
val res1: Rational = 1/2
```

Checking preconditions

```
class Rational(n: Int, d: Int) {
require(d != 0)
override def toString = n + "/" + d
}

scala> new Rational(1, 0)
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:324)
  ... 33 elided
```

The require method takes one boolean parameter. If the passed value is true, require will return normally. Otherwise, require will prevent the object from being constructed by throwing an IllegalArgumentException

Adding fields

- we’ll define a public add method on class Rational that takes another Rational as a parameter.
- To keep Rational immutable, the add method must not add the passed rational number to itself.
- Rather, it must create and return a new Rational that holds the sum

Adding fields – Contd.

```
class Rational(n: Int, d: Int) { // This won't compile
require(d != 0)
override def toString = n + "/" + d
def add(that: Rational): Rational =
new Rational(n * that.d + that.n * d, d * that.d)
}
```

However, given this code the compiler will complain:

```
<console>:11: error: value d is not a member of Rational
new Rational(n * that.d + that.n * d, d * that.d)

<console>:11: error: value d is not a member of Rational
new Rational(n * that.d + that.n * d, d * that.d)
```

Adding fields – Contd.

```
class Rational(n: Int, d: Int) {
require(d != 0)
val numer: Int = n
val denom: Int = d
override def toString = numer + "/" + denom
def add(that: Rational): Rational =
new Rational(
numer * that.denom + that.numer * denom,
denom * that.denom
)
}
```

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3
scala> oneHalf add twoThirds
res2: Rational = 7/6
```

6.7 Auxiliary constructors

- Constructors other than the primary constructor are called auxiliary constructors
- Auxiliary constructors in Scala start with this(...). The body def of Rational's auxiliary constructor merely invokes the primary constructor

```
class Rational(n: Int, d: Int) {
require(d != 0)
val numer: Int = n
val denom: Int = d
def this(n: Int) = this(n, 1) // auxiliary constructor
override def toString = numer + "/" + denom
def add(that: Rational): Rational =
new Rational(
numer * that.denom + that.numer * denom,
denom * that.denom
) }
```

6.8 Private fields and methods

```
class Rational(n: Int, d: Int) {
require(d != 0)
private val g = gcd(n.abs, d.abs)
val numer = n / g
val denom = d / g
def this(n: Int) = this(n, 1)
def add(that: Rational): Rational =
new Rational(
numer * that.denom + that.numer *
denom,
denom * that.denom
)
```

```
override def toString = numer + "/" + denom
private def gcd(a: Int, b: Int): Int =
if (b == 0) a else gcd(b, a % b)
}
```

6.9 Defining operators

```
class Rational(n: Int, d: Int) {
require(d != 0)
private val g = gcd(n.abs, d.abs)
val numer = n / g
val denom = d / g
def this(n: Int) = this(n, 1)
def + (that: Rational): Rational =
new Rational(
numer * that.denom + that.numer * denom,
denom * that.denom
)
```

```
def * (that: Rational): Rational =
new Rational(numer * that.numer, denom * that.denom)
override def toString = numer + "/" + denom
private def gcd(a: Int, b: Int): Int =
if (b == 0) a else gcd(b, a % b)
}
scala> val x = new Rational(1, 2)
x: Rational = 1/2
scala> val y = new Rational(2, 3)
y: Rational = 2/3
scala> x + y
res7: Rational = 7/6
```

6.10 Identifiers in Scala

- An alphanumeric identifier starts with a letter or underscore,
- Contain letters, digits, or underscores.
- The ‘$’ character also counts as a letter; however, it is reserved for identifiers generated by the Scala compiler.
- Identifiers in user programs should not contain ‘$’ characters,
- Even though it will compile; if they do, this might lead to name clashes with identifiers generated by the Scala compiler
- Scala follows Java’s convention of using camel-case5 identifiers, such as toString and HashSe

6.11 Method overloading

```
class Rational(n: Int, d: Int) {
require(d != 0)
private val g = gcd(n.abs, d.abs)
val numer = n / g
val denom = d / g
def this(n: Int) = this(n, 1)
def + (that: Rational): Rational =
  new Rational(
    numer * that.denom + that.numer * denom,
    denom * that.denom )
def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
```

```
def - (that: Rational): Rational =
new Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom )
def - ( i: Int): Rational =
  new Rational(
  numer - i * denom, denom)
```

-----Contd.--------

Method overloading – Contd.

```
def * (that: Rational): Rational =
  new Rational(numer * that.numer, denom * that.denom)
def * (i: Int): Rational =
  new Rational(numer * i, denom)
def / (that: Rational): Rational =
  new Rational(numer * that.denom, denom * that.numer)
def / (i: Int): Rational =
  new Rational(numer, denom * i)
override def toString = numer + "/" + denom
private def gcd(a: Int, b: Int): Int =
if (b == 0) a else gcd(b, a % b)
}
```

```
scala> val x = new Rational(2, 3)
x: Rational = 2/3
scala> x * x
res12: Rational = 4/9
scala> x * 2
res13: Rational = 4/3
```

6.12 Implicit conversions

```
scala> 2 * r
<console>:10: error: overloaded method value * with
alternatives:
(x: Double)Double <and>
(x: Float)Float <and>
(x: Long)Long <and>
(x: Int)Int <and>
(x: Char)Int <and>
(x: Short)Int <and>
(x: Byte)Int
cannot be applied to (Rational)
2 * r
  ^
```

```
scala> implicit def intToRational(x: Int) =
new Rational(x)
```

Conclusion

- You are able to:
  - add parameters to a class,
  - define several constructors,
  - define operators as methods,
  - customize classes so that they are natural to use
  - defining and using immutable objects

FUNCTIONS AND CLOSURES

Chapter 8

8.1 Methods

- A function is as a member of some object.
- Programs should be decomposed into many small functions that each do a well-defined task.

8.1 Methods

```
import scala.io.Source
object LongLines {
 def processFile(filename: String, width: Int) = {
   val source = Source.fromFile(filename)
   for (line <- source.getLines())
       processLine(filename, width, line)
 }
```

To execute:

$ scala FindLongLines 45 LongLines.scala

O/P: LongLines.scala: def processFile(filename: String, width: Int) = {

```
private def processLine(filename:String,
width: Int, line: String) = {
if (line.length > width)
     println(filename + ": " + line.trim)
  }
}

object FindLongLines {
def main(args: Array[String]) = {
val width = args(0).toInt
for (arg <args.
drop(1))
LongLines.processFile(arg, width)
} }
```

8.2 Local functions

- One can define functions inside other functions.
- Just like local variables, such local functions are visible only in their enclosing block.

```
def processFile(filename: String, width: Int) = {
  def processLine(filename: String,
      width: Int, line: String) = {
       if (line.length > width)
          println(filename + ": " + line.trim)
}
```

```
val source = Source.fromFile(filename)
for (line <-source.getLines()) {
  processLine(filename, width, line)
   }
}
```

Local functions can access the parameters of their enclosing function

```
import scala.io.Source
object LongLines {
  def processFile(filename: String, width: Int) = {
    def processLine(line: String) = {
      if (line.length > width)
        println(filename + ": " + line.trim)
    }
   val source = Source.fromFile(filename)
   for (line <-source.getLines())
     processLine(line)
   }}
```

8.3 First-class functions

- Functions as unnamed literals and then pass them around as values
- A function literal is compiled into a class that when instantiated at runtime is a function value.
- Function literals exist in the source code
- Whereas function values exist as objects at runtime

(x: Int) => x + 1

- The => designates that this function converts the thing on the left (any integer x) to the thing on the right (x + 1). So, this is a function mapping any integer x to x + 1.
- Function values are objects, so you can store them in variables

Scala> var increase = (x: Int) => x + 1

increase: Int => Int = <function1>

scala> increase(10)

res0: Int = 11

First-class functions – Contd.

```
scala> increase = (x: Int) => {
            println("We")
            println("are")
            println("here!")
            x + 1
            }
increase: Int => Int = <function1>
scala> increase(10)
We
are
here!
res2: Int = 11
```

First-class functions – Contd.

- It takes a function as an argument and invokes that function on each of its elements.

```
scala> val someNumbers = List(11, 10, 5, 0, 5, 10)
someNumbers: List[Int] = List(11, 10, 5, 0, 5, 10)
scala> someNumbers.foreach((x: Int) => println(x))
11
10
5
0
5
10
scala> someNumbers.filter((x: Int) => x > 0)
res4: List[Int] = List(5, 10)
```

8.4 Short forms of function literals

```
scala> someNumbers.filter((x: Int) => x > 0)
```

- Leave off the parameter types

```
scala> someNumbers.filter((x) => x > 0)
```

```
res5: List[Int] = List(5, 10)
```

- Scala compiler knows that x must be an integer, because it sees that you are immediately using the function to filter a list of integers
- A second way to remove useless characters is to leave out parentheses around a parameter whose type is inferred

```
scala> someNumbers.filter(x => x > 0)
```

```
res6: List[Int] = List(5, 10)
```

8.5 Placeholder syntax

- We can use underscores as placeholders for one or more parameters

```
scala> someNumbers.filter(_ > 0)
res7: List[Int] = List(5, 10)
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function2>
scala> f(5, 10)
res9: Int = 15
```

- The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on

8.6 Partially applied functions

someNumbers.foreach(x => println(x))

Could be written as:

someNumbers.foreach(println _)

scala> def sum(a: Int, b: Int, c: Int) = a + b + c

sum: (a: Int, b: Int, c: Int)Int

- scala> sum(1, 2, 3)
- res10: Int = 6

Partially applied functions – Contd.

```
scala> val a = sum _
a: (Int, Int, Int) => Int = <function3>

scala> a(1, 2, 3)
res11: Int = 6
```

- The variable named a refers to a function value object.
- This function value is an instance of a class generated automatically by the Scala compiler from sum _, the partially applied function expression.

```
scala> a.apply(1, 2, 3)
res12: Int = 6
```

8.7 Closures

- (x: Int) => x + more
  - *“more” is free variable in this expression; “more” is a variable that’s used inside the expression but not defined inside the expression*
  - *The x variable, by contrast, is a bound variable because it does have a meaning in the context of the function*
- The function value (the object) that’s created at runtime from this function literal is called a closure.
- A function literal with no free variables, such as (x: Int) => x + 1, is called a closed term
- A function literal with free variables, such as (x: Int) => x + more, is an open term
- The resulting function value, which will contain a reference to the captured more variable, is called a closure because the function value is the end product of the act of closing the open term, (x: Int) => x + more

```
scala> more = 9999

more: Int = 9999

scala> addMore(10)

res17: Int = 10009
```

Repeated parameters

- to indicate that the last parameter to a function may be repeated parameter, place an asterisk after the type of the parameter

```
scala> def echo(args: String*) =
            for (arg <args) println(arg)
echo: (args: String*)Unit
scala> echo("one")
one
scala> echo("hello", "world!")
hello
world!
```

- declared as type “String*” is actually Array[String].

Repeated parameters –Contd.

```
scala> val arr = Array("What's", "up", "doc?")
arr: Array[String] = Array(What's, up, doc?)
scala> echo(arr)
<console>:10: error: type mismatch;
found : Array[String]
required: String
echo(arr)
scala> echo(arr: _*)
What's
up
doc?
```

Named arguments

```
scala> def speed(distance: Float, time: Float): Float =
                    distance / time
speed: (distance: Float, time: Float)Float
scala> speed(100, 10)
res27: Float = 10.0
scala> speed(distance = 100, time = 10)
res28: Float = 10.0
```

- Called with named arguments, the arguments can be reversed without changing the meaning:

```
scala> speed(time = 10, distance = 100)
res29: Float = 10.0
```

Default parameter values

```
def printTime(out: java.io.PrintStream = Console.out) =
        out.println("time = " + System.currentTimeMillis())
```

- Call the function as printTime(),
  - *Then out will be set to its default value of Console.out.*
- printTime(Console.err).
  - *Calling the function with an explicit argument*

```
def printTime2(out: java.io.PrintStream = Console.out,  divisor: Int = 1) =
        out.println("time = " + System.currentTimeMillis()/divisor)
```

- Named Argument call
  - *printTime2(out = Console.err)*
  - *printTime2(divisor = 1000)*

8.9 Tail recursion

```
def approximate(guess: Double): Double =
        if (isGoodEnough(guess)) guess
        else approximate(improve(guess))
```

- Functions, which call themselves as their last action, are called **tail recursive**
- A tail-recursive function will not build a new stack frame for each call; all calls will execute in a single frame.

COMPOSITION AND INHERITANCE

Chapter 10

Agenda

- **Composition** means one class holds a reference to another, using the referenced class to help it fulfil its mission.
- **Inheritance** is the superclass/subclass relationship
- Abstract classes,
- Parameter less methods,
- Extending classes,
- Overriding methods and fields,
- Parametric fields,
- Invoking superclass constructors,
- Polymorphism and dynamic binding,

10.1 A two-dimensional layout library

elem(s: String): Element
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2

hello ***
*** world

We are creating the above running example with:
Combinators - composing operators ‘above’ and ‘beside’
“elem” that construct new elements from passed data.
elements are two dimensional rectangles of characters.

10.2 Abstract classes

```
abstract class Element {
        def contents: Array[String]
}
```

- The abstract modifier signifies that the class may have abstract members that do not have an implementation.
- We cannot instantiate an abstract class.- a compiler error

```
scala> new Element
<console>:5: error: class Element is abstract;
   cannot be instantiated
        new Element
```

10.3 Defining parameter less methods

```
abstract class Element {
        def contents: Array[String]
        def height: Int = contents.length
        def width: Int = if (height == 0) 0 else contents(0).length
}
```

10.2 · Defining parameterless methods width and height.

- The width method returns the length of the first line, or if there are no lines in the element, returns zero
- Define methods that take no parameters and have no side effects as parameter less methods (i.e., leaving off the empty parentheses).

The method “def width(): Int “ is defined without parentheses: “def width: Int”

Defining parameter less methods – Contd.

```
abstract class Element {
   def contents: Array[String]
    val height = contents.length
     val width =
         if (height == 0) 0 else contents(0).length
 }
```

- The two pairs of definitions are completely equivalent from a client's point of view.
- Field accesses might be slightly faster than method invocations because the field values are pre-computed when the class is initialized, instead of being computed on each method
- The fields require extra memory space in each Element object

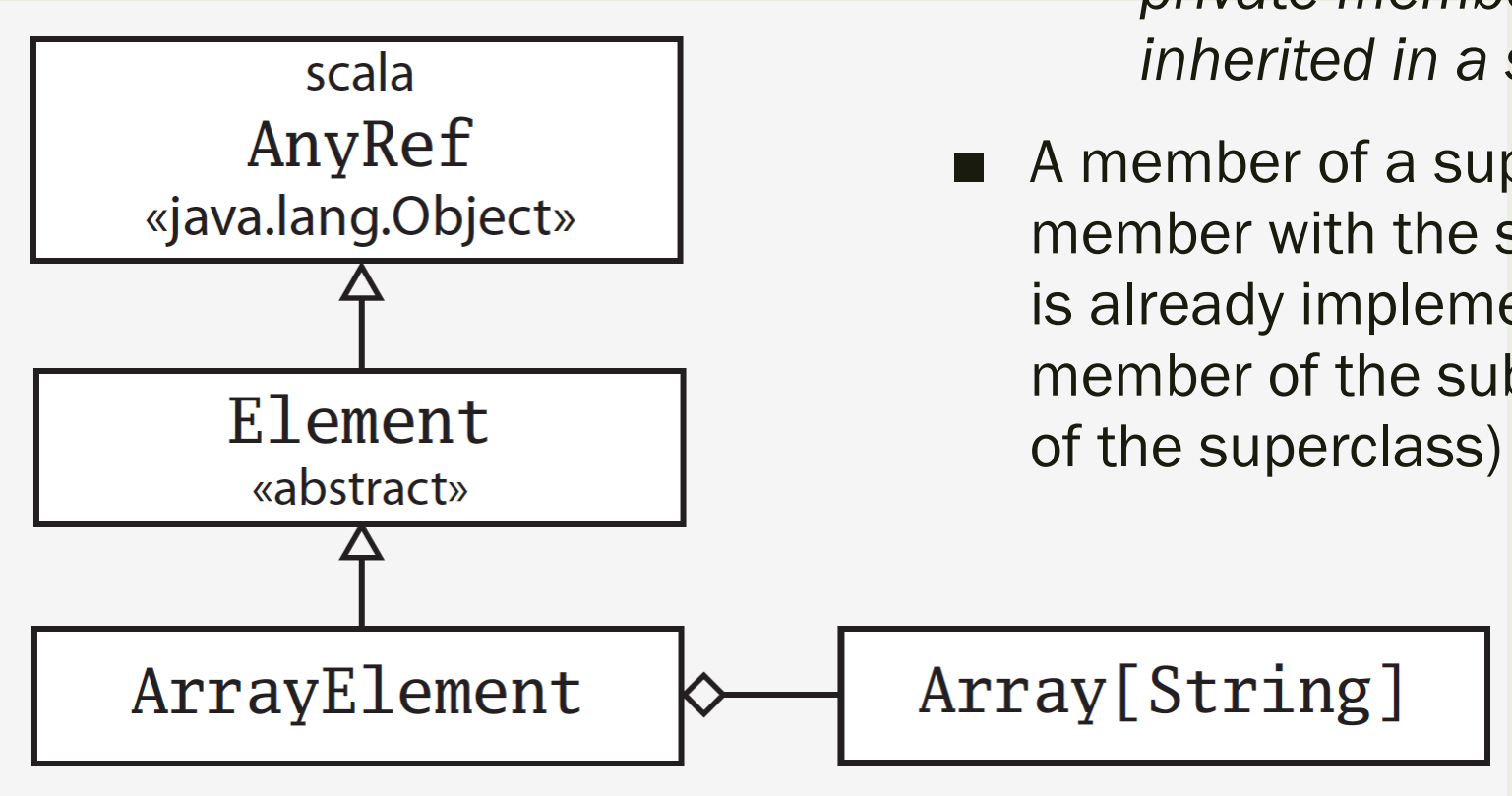
10.4 Extending classes

- To instantiate an element, therefore, we will need to create a subclass that extends Element and implements the abstract contents method.

```
class ArrayElement(conts: Array[String]) extends Element {
        def contents: Array[String] = conts
}
```

- Class ArrayElement inherit all non-private members from class Element
- It makes the type ArrayElement a subtype of the type Element
- ArrayElement is called a subclass of class Element & Element is a superclass of ArrayElement
- Scala compiler implicitly assumes the class extends from scala.AnyRef

Class diagram for ArrayElement.



- Inheritance means
  - *all members of the superclass are also members of the subclass*
  - *private members of the superclass are not inherited in a subclass*
- A member of a superclass is not inherited if a member with the same name and parameters is already implemented in the subclass (the member of the subclass overrides the member of the superclass)

10.5 Overriding methods and fields

- Scala treats fields and methods more uniformly than Java
- Fields and methods belong to the same namespace. Hence it possible for a field to override a parameterless method

```
class WontCompile {
private var f = 0 // Won't compile, because a field
def f = 1 // and method have the same name
}
```

- Parameter and method having same name is accepted in Java but not in Scala
- Java’s four namespaces are fields, methods, types, and packages.
- contrast, Scala’s two namespaces are:
  - *values (fields, methods, packages, and singleton objects)*
  - *types (class and trait names)*

10.6 Defining parametric fields

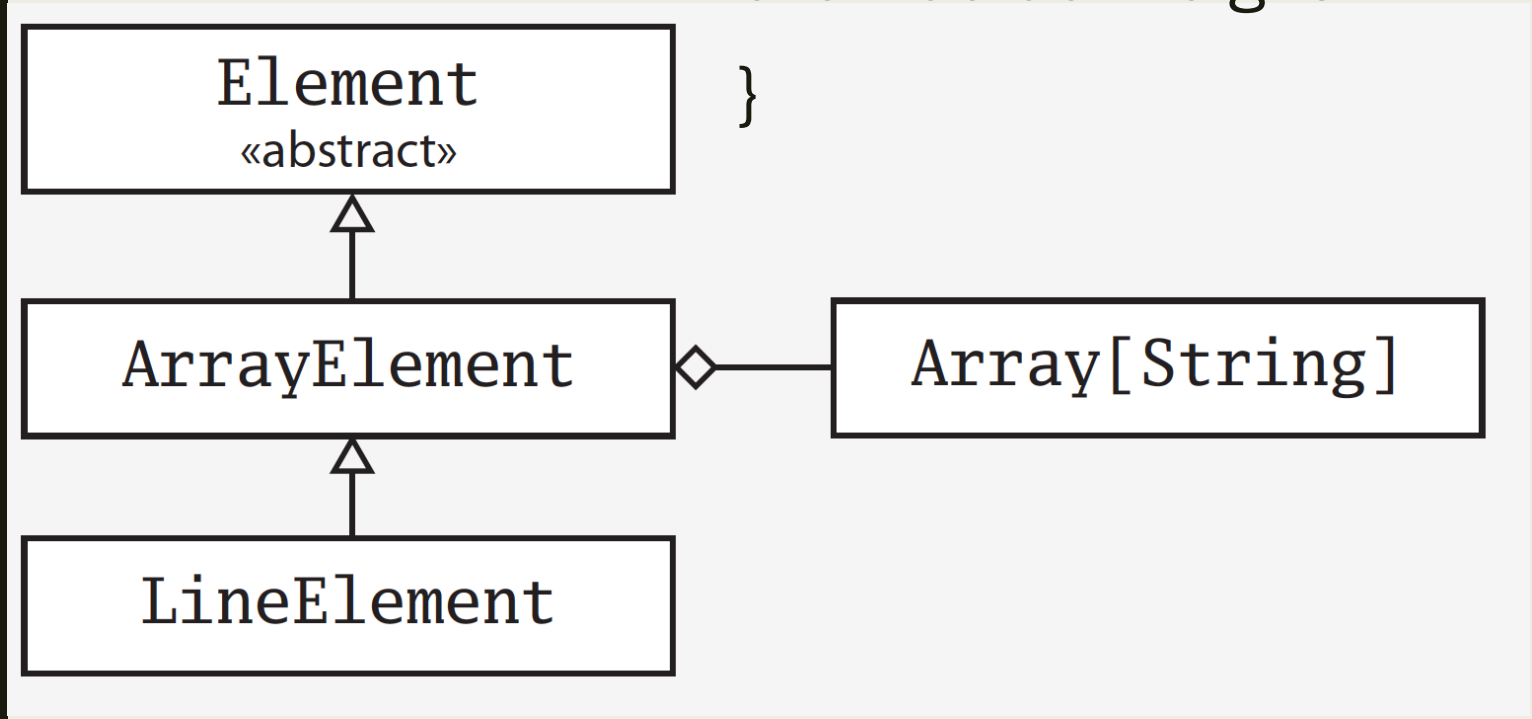
```
class Cat {
val dangerous = false
}
class Tiger(
override val dangerous: Boolean,
private var age: Int
) extends Cat
```

- Tiger's definition is a shorthand for the following alternate class definition

```
class Tiger(param1: Boolean, param2: Int) extends Cat {
override val dangerous = param1
private var age = param2
}
```

10.7 Invoking superclass constructors

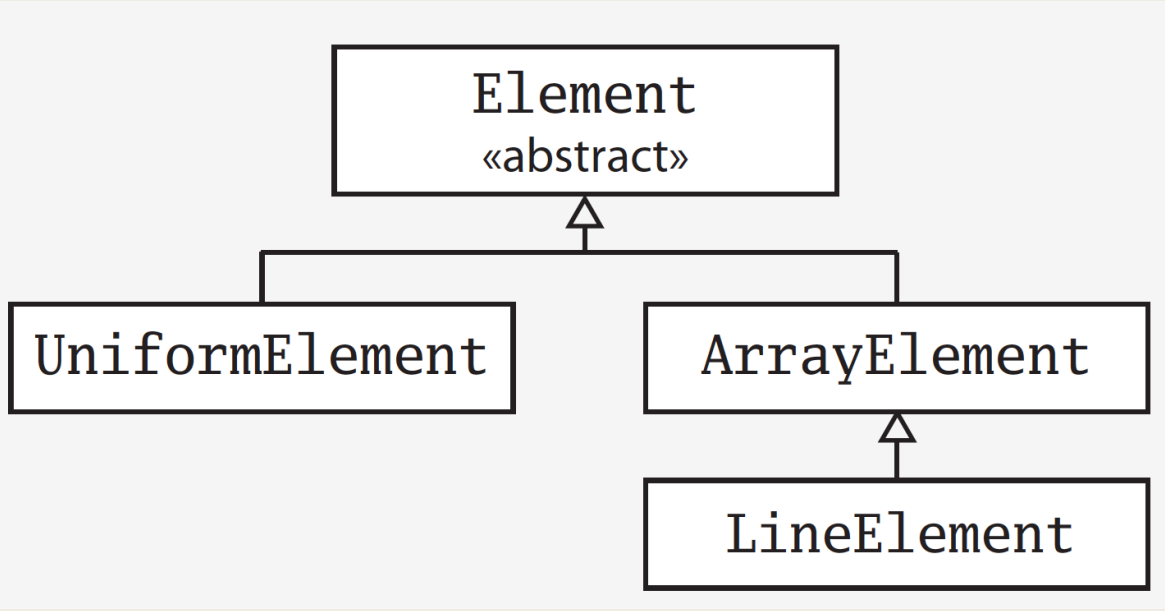
```
class LineElement(s: String) extends
ArrayElement(Array(s)) {

override def width = s.length

override def height = 1

}
```



10.9 Polymorphism and dynamic binding

- Polymorphism - means “many shapes” or “many forms.”

```
class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
) extends Element {
    private val line = ch.toString * width
    def contents = Array.fill(height)(line)
}
```



- val e1: Element = new ArrayElement(Array("hello", "world"))
- val ae: ArrayElement = new LineElement("hello")
- val e2: Element = ae
- val e3: Element = new UniformElement('x', 2, 3)

10.10 Declaring final members

- To ensure that a member cannot be overridden by a subclasses by adding a final modifier to the member

```
class ArrayElement extends Element {
    final override def demo() = {
        println("ArrayElement's implementation invoked")
    }
}
```

```
elem.scala:18: error: error overriding method demo in class ArrayElement of type ()Unit;
method demo cannot override final member
override def demo() = {
final class ArrayElement extends Element {
    override def demo() = {
        println("ArrayElement's implementation invoked")
    }
}
```

10.11 Using composition and inheritance

- Composition and inheritance are two ways to define a new class in terms of another existing class.
- Class ArrayElement also has a composition relationship with Array, because its parametric contents field holds a reference to an array of strings

```
class LineElement(s: String) extends Element {
        val contents = Array(s)
        override def width = s.length
        override def height = 1
}
```