INDICE

1 Something about Types[1]

1.1 Why types are needed in programming languages?

Instead of asking the question what is a type? we ask why types are needed in programming languages. To answer this question we look at how types arise in several domains of computer science and mathematics. Consider, for example, the following untyped universes:

- Bit strings in computer memory
- S-expressions in pure Lisp
- λ-expressions in the λ-calculus
- Sets in set theory

The most concrete of these is the universe of bit strings in computer memory. *Untyped* actually means that there is only one type, and here the only type is the *memory word*, which is a *bit string* of fixed size. This universe is untyped because everything ultimately has to be represented as bit strings: characters, numbers, pointers, structured data, programs, etc. When looking at a piece of raw memory there is generally no way of telling what is being represented. The meaning of a piece of memory is critically determined by an external interpretation of its contents.

As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes:

- In computer memory, we distinguish *characters* and *operations*, both represented as **bit strings**.
- In Lisp, some **S-expressions** are called *lists* while others form *legal programs*.
- In λ-calculus some **functions** are chosen to represent *boolean values*, others to represent *integers*.
- In set theory some **sets** are chosen to denote *ordered pairs*, and some sets of ordered pairs are then called *functions*.

Untyped universes of computational objects decompose naturally into subsets with uniform behavior. Sets of objects with uniform behavior may be named and are referred to as *types*:

- For example, all integers exhibit uniform behavior by having the same set of applicable operations.
- Functions from integers to integers behave uniformly in that they apply to objects of a given type and produce values of a given type.

[1] Extracted from "On Understanding Types, Data Abstraction, and Polymorphism" of Luca Cardelli and Peter Wegner.

But, although in the untyped system we have types informally defined (subsets of objects with uniform behavior), they aren't part of the system, and for this reason it is very easy to violate the type distinctions we have just created:

- In the computer memory to apply an operator over another operator instead of a character (for example the *bit-wise boolean or* over another operator).
- In the λ-calculus, to apply a conditional over a non-boolean value.

In order to avoid the possible type violations of an untyped system it's necessary a typed system. In order to have a typed system it's necessary to have a type system[2], ie. *a way to declare explicitly the type of an system's object*, and that is usually done by imposing a static type structure on programs: *types are associated with constants, operators, variables, and function symbols*. Thus, the type system impose constraints which help to enforce correctness[3].

Objects of a typed system have a type, which gives them an internal data representation[4] that respects the expected properties of the type. This data representation is chosen to make easy to perform expected operations on the objects that belong to the type.

1.2 Static and Strong Typing

Programming languages in which the *type of every expression*[5] *can be determined by static program analysis*[6] are said to be statically typed.

Dynamic typing, *determines the type-safety of operations at runtime*; in other words, types are associated with runtime values rather than textual expressions[7]. Among other things, this may permit a single variable to refer to values of different types at different points in the program execution. However, type errors cannot be automatically detected until a piece of code is actually executed

Static typing is a useful property, but the requirement that all variables and expressions are bound to a type at compile time is sometimes too restrictive. It may be replaced by *the weaker requirement that all expressions are guaranteed to be type-consistent* although the type itself may be statically unknown; this can be generally done by

[2] The possibility of having representations of higher-level concepts and their interactions.
[3] The types impose constraints on object interaction, which prevent objects from inconsistent interaction with other objects.
[4] The type may be viewed the armor that protects this underlying untyped representation from arbitrary or unintended use. The type provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects. In an untyped system untyped objects are naked in that the underlying representation is exposed for all to see. Violating the type system involves removing the protective armor and operating directly on the naked representation.
[5] A type inference system can be used to infer the types of expressions when little or no type information is given explicitly. In languages like Pascal and Ada, the type of variables and function symbols is defined by redundant declarations. In languages like ML, explicit declarations are avoided wherever possible and the system may infer the type of expressions from local context.
[6] When the program is compiled.
[7] As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions.

introducing some run-time type checking. Languages in which all expressions are type-consistent are called strongly typed languages[8].

Static typing allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type-consistent. It facilitates early detection of type errors and allows greater execution-time efficiency. It enforces a programming discipline on the programmer that makes programs more structured and easier to read. But static typing may also lead to a loss of flexibility and expressive power by prematurely constraining the behavior of objects to that associated with a particular type.

[8] Note that every statically typed language is strongly typed but the converse is not necessarily true. Strong typing prevents mixing operations between mismatched types. In order to mix types, you must use an explicit conversion (variables cannot be coerced to unrelated types). Conversely, weak typing means that you can mix types without an explicit conversion (language will do an implicit coercion, or cast, of the types when used).

2 Identifiers in Scala

Scala support four forms of identifier formation:

- Alphanumeric identifier, starts with a letter or underscore, which can be followed by further letters, digits, or underscores[9]:
    - Scala follows Java's convention of using camel-case identifiers:
        - Avoid use underscores in identifiers.
        - Camel-case names of *fields*, *method parameters*, *local variables*, and *functions* should start with lower case letter, for example: **length**, **flatMap**.
        - Camel-case names of *classes* and *traits* should start with an upper case letter, for example: **BigInt**, **List**.
    - Scala departs from Java in the constant[10] identifiers:
        - In Java, the convention is to give constants names that are all upper case, with underscores separating the words, such as **X_OFFSET**. The Scala convention the convention is that the first character should be upper case, such as **XOffset**.
- Operator identifier, consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #[11]:
    - Examples the valid operators identifiers: + ++ **:::** **<?>** **:->**
        - The Scala compiler will internally "mangle" operator identifiers to turn them into legal Java identifiers[12] with embedded **$** characters. For instance, the identifier **:->** would be represented internally as **$colon$minus$greater**.
- Mixed identifier, consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier. For example, **unary_+** .
- Literal identifier, is an arbitrary string enclosed in back ticks (` . . `). Some examples of literal identifiers are: **`x`** **`<clinit>`** **`yield`** :
    - The idea is that you can put any string that's accepted by the runtime as an identifier between back ticks. The result is always a Scala identifier.
    - This works even if the name contained in the back ticks would be a Scala reserved word:
        - A typical use case is accessing the static **yield** method in Java's **Thread** class. You cannot write **Thread.yield()** because **yield** is

[9] 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.
[10] In Scala, the word constant does not just mean **val**. Even though a **val** does remain constant after it is initialized, it is still a variable.
[11] More precisely, an operator character belongs to the Unicode set of mathematical symbols(Sm) or other symbols(So), or to the 7-bit ASCII characters that are not letters, digits, parentheses, square brackets, curly braces, single or double quote, or an underscore, period, semi-colon, comma, or back tick character.
[12] There is a small difference between Java and Scala. In Java, the input **x <- y** would be parsed as 4 lexical symbols, so it would be equivalent to **x < - y**. In Scala, **<-** would be parsed as a single identifier, giving **x <- y**. If you want the first interpretation, you need to separate the < and the **-** characters by a space.

a reserved word in Scala. However, you can still name the method in back ticks, e.g., **Thread.`yield`()**.

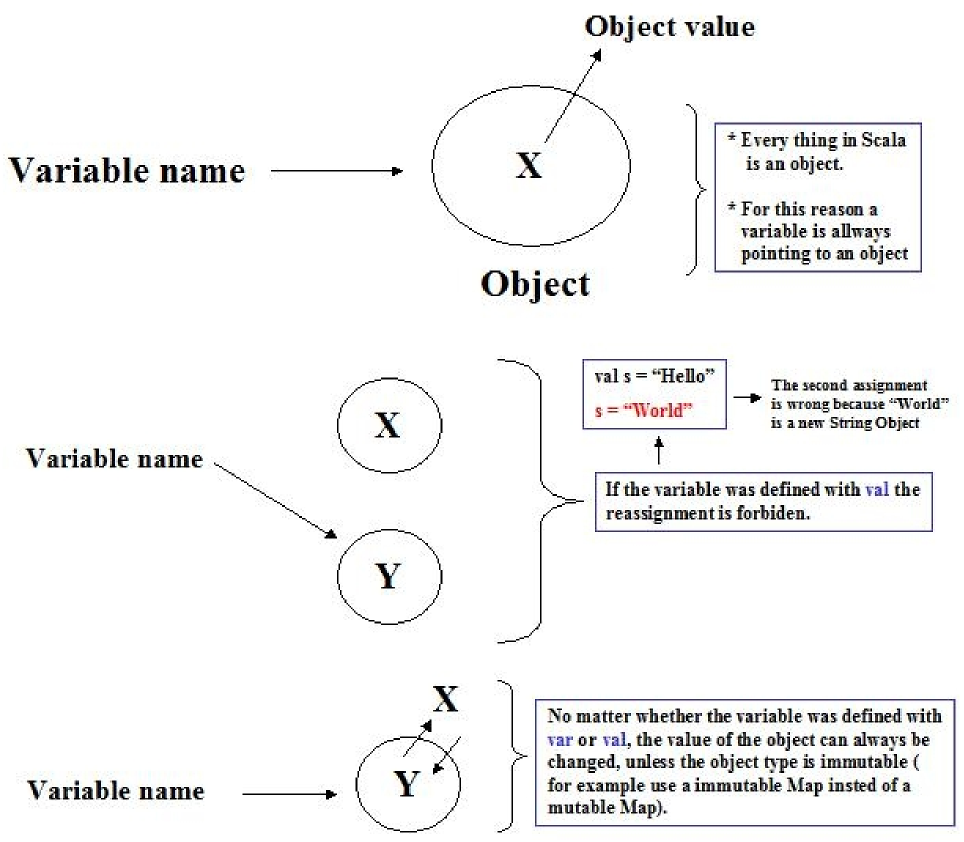
3 Variables

3.1 Variables

There are two types[13]:

- Immutable: a variable defined with the key **val**[14]
    - **val msg = “Hello World”**
    - There are an special type of **val** variables: the **lazy val**
        - The **lazy val** variables are calculated once, the first time the variable is accessed. You would use a **lazy val** if the variable may not be used and the cost of calculating it is very long.

- Mutable: a variable defined with the key **var**
    - **var msg: String = “Hello World”**[15]



* Every thing in Scala is an object.

* For this reason a variable is allways pointing to an object

val s = “Hello”
s = “World”

The second assignment is wrong because “World” is a new String Object

If the variable was defined with val the reassignment is forbiden.

No matter whether the variable was defined with var or val, the value of the object can always be changed, unless the object type is immutable (for example use a immutable Map insted of a mutable Map).

3.2 Variable Declaration

Variable definitions can be <u>code blocks</u> as well. This comes in handy when defining val

[13] In the picture we are showing that a variable is pointing to an object (a class instance), that it’s true because in Scala <u>all is allways a class instance of some class</u>.
[14] A **val** is similar to a final variable in Java. Once initialized, a val can never be reassigned.
[15] The variable type isn’t obligatory because the Scala compiler can make type inference.

variables, and the logic required to compute the value is non-trivial:

```
val x3: String = {
        val d = new java.util.Date()
        d.toString()
}
```

3.3 Variable scope

Variable declarations in Scala programs have a scope[16] that defines where you can use the name.

You can define a variable in an inner scope that has the same name as a variable in an outer scope[17]:

```
val a = 1;
{
        val a = 2
        println(a) // This prints 2
}
println(a)  // This prints 1
```

In a Scala program, an inner variable is said to shadow **a** like-named outer variable.

[16] The most common example of scoping is that curly braces (**{..}**) generally introduce a new scope.
[17] One difference to note between Scala and Java is that unlike Scala, Java will not let you create a variable in an inner scope that has the same name as a variable in an outer scope.

4 Working with classes and instances

4.1 *Something about of Object-Oriented Programming theory*

4.1.1 Data abstraction[18]

Abstraction refers to the act of representing essential features without including the background details or explanations.

Data abstraction refers to a range of techniques for defining and manipulating data in an abstract fashion. Abstract data is useful because the conceptual view of the properties of data are often very different from the properties of the data's detailed representation in a computational system.

A general view of abstract data is based on the notion of abstract *constructors* and abstract *observations*. The constructors create or instantiate abstract data, while the observations retrieve information about abstract data. The behavior of a data abstraction is specified by giving the value of each observation applied to each constructor[19]:

- As an example, consider a data abstraction for *integer lists*:
    - The constructors: are **nil**, which constructs an empty list, and **adjoin**, which takes a list and an integer, and forms a new list with the integer added to the front of the list argument.
    - The observers: are **null?**, **head**, and **tai**l. **Null?** is a predicate that returns true if its argument is the empty list; i.e. if it is equal to **nil**. **Head** returns the first integer in a non-empty list. **Tail** returns the rest of a non-empty list.
    - The data abstraction behavior:

Observations	**Constructor of s**	
	nil	**Adjoin(s',n)**
null?(s)	true	false
head(s)	error	n
tail(s)	error	s'

4.1.2 Abstract Data Types[20]

The Abstract Data Types (ADTs) are organized around the abstract observations. Each observation is implemented as an operation upon a concrete representation derived from the abstract constructors. The constructors are also implemented as operations that create values in the representation type. The representation is shared among the operations, but hidden from clients of the ADT.

[18] From "Object-Oriented Programming Versus Abstract Data Types" of William R. Cook.
[19] That is: a data abstraction may be defined by listing the value of each observation on each constructor.
[20] From "Object-Oriented Programming Versus Abstract Data Types" of William R. Cook.

For the case of *integer lists* one possible ADT is the following:

Representation:

- **list = NIL | CELL of integer * list**[21]

Operations:

- **nil = NIL**
- **adjoin(x : integer, l : list) =**
  CELL(x, l)
- **null?(l : list) = case l of**
  NIL => true
  CELL(x, l)) false
- **head(l : list) = case l of**
  NIL => error
  CELL(x, l') => x
- **tail(l : list) = case l of**
  NIL => error
  CELL(x, l ') => l'

The ADTs are created to solve specific data-oriented problems. The ADTs are often called user-defined data types, because they allow programmers to define new types that resemble primitive data types. Just like the primitive type **Integer** with operations +, −, *, etc., an ADT has a type domain, whose representation is hidden to clients, and a set of operations defined on the domain.

4.1.3 Polymorphism[22]

4.1.4 Object-oriented Programming

PDA is organized around the constructors of the data abstraction. The observations become the attributes, or methods, of the procedural data values. Thus a procedural data value is simply defined by the combination of all possible observations upon it.

The difference between PDA and ADT concerns how they organize and protect the implementation of a data abstraction. The choice of organization has a tremendous effect on the flexibility and extensibility of the implementation. Viewing ADTs and PDA as orthogonal ways to organize a data abstraction implementation provides a useful starting point for comparing the advantages and disadvantages of the two paradigms. The decomposition used, either by constructor or observer, determines how easy it is to add a new constructor or observer. Adding a new feature that clashes with

[21] It's the concrete data respresentation that is hidden to the ADT's users. The value NIL is a constant and CELL is a record wich contains a pair of an integer and a list.
[22] From "On Understanding Types, Data Abstraction, and Polymorphism" of Luca Cardelli and Peter Wegner.

the organization is difficult. In addition, the tight coupling and security in an ADTs makes it less extensible and flexible, but supports verification and optimization. The independence of PDA implementations has the opposite effect. However, in several cases where PDA might be expected to have difficulty, the problems are lessened by the use of inheritance and subtyping.

Objects are the basic run-time entities in an object-oriented system. Programming problem is analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code.

Objects incorporate both data and operations. In this way they mimic things in the real world, which have properties (data) and behaviors (operations). Objects that hold the same kind of data and perform the same operations form a class.

An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable

Let's consider behavior, which is often understood as an ability to send an object a message to which it replies. To put in other terms, we invoke a dispatcher function of the object and pass it an indicator of a message along with other parameters. The result of that function is the reply. As far as the external world is concerned, this dispatcher function is the representation of the object -- it *is* the object. Well, an object has to encapsulate and maintain a state -- thus the dispatcher function should be a closure. The identity aspect depends: if the dispatcher function is a pure function, or if it can mutate its own state ("closed" variables) in response to some messages.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes.

Storing data and functions in a single unit (class) is encapsulation. Data cannot be accessible to the outside world and only those functions which are stored in the class can access it.

he difference between object orientation and using ADTs is that ADTs are created to solve specific data-oriented problems, and object orientation encompasses objects that manipulate ADTs, along with other objects that provide the basis for the underlying system. Most modern programming language textbooks refer to Lists, Stacks, Queues, Arrays and Trees as examples of ADTs

These languages support abstract data types (Adts) and not classes, which
provide inheritance and polymorphism

[Coad 91] provides another model:

Object-Oriented = Classes and Objects
+ Inheritance
+ Communication with messages

The discussion had turned to closures and objects, and which could be considered "richer" or "more powerful" or "more fundamental.":

The response is: NONE:
http://blogs.claritycon.com/blogs/sean_devlin/archive/2009/04/26/epiphany-closures-are-objects-objects-are-closures.aspx

In 1992 Ken Dickey wrote a fascinating paper "Scheming with Objects" [ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/swob.txt] It starts as follows "There is a saying--attributed to Norman Adams--that **'Objects are a poor man's closures**.' In this article we discuss what closures are and how objects and closures are related, show code samples to make these abstract ideas concrete, and implement a Scheme Object System which solves the problems we uncover along the way." His paper really implements an OO system in Scheme.

Objects are a poor man's closures: esto significa que los objetos son una forma pobre" de closure

How do we define that a paradigm can model another one? After all one can write an interpreter of any real language using any other Turing-complete language. We don't want to say that the C language models functional and OO paradigms only because compilers of functional and OO languages use C as their target languages:

Matthias Felleisen wrote a paper exactly on that subject: "On the Expressive Power of Programming Languages" [http://www.cs.rice.edu/CS/PLT/Publications/] He came up with an interesting, formal definition that lets you tell when one language is strictly more expressive than the other. He _proved_ for example that a call-by-value lambda-calculus is strictly less expressive than call-by-name lambda-calculus:

4.2 *Classes, fields and methods*

Inside a class definition, you place fields and methods, which are collectively called members:

- Fields, which you define with either val or var, are variables that refer to objects:
  - The fields hold the state, or data, of an object[23].

[23] Collectively, an object's instance variables make up the memory image of the object.

- Fields are also known as instance variables, because every instance gets its own set of the variables.

- Methods, which you define with **def**, contain executable code:
  - The methods use the field's data to do the computational work of the object.

```
class ChecksumAccumulator {
        private[24] var sum = 0
        def add(b: Byte) { sum += b }[25]
        def checksum(): Int = ~(sum & 0xFF) + 1
}
```

In the above example, **add** and **checksum** are methods, and **sum** is a field.

4.3 Defining a class

4.3.1 Class parameters

Defining a class

```
class Rational(n: Int, d: Int)[26]
```

A class is defined with the **class** keyword, and **Rational** is the name of the new class. The identifiers **n** and **d** in the parentheses after the class name are called class parameters. This parameters are a kind of class'fields, but are only available to the methods inside of the object to which the parameter belongs[27].

Defining parametric fields

It's possible to combine a parameter and field definition in a single parametric field definition, as follows:

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

Note that now the **n** parameter is prefixed by **val**. This is a shorthand that defines at the same time a parameter and an unreassignable field with the same name. The field is initialized with the value of the parameter. You can also prefix a class parameter with **var**, in which case the corresponding field would be reassignable. Finally, it is possible to add modifiers such as **private**, **protected**, or **override** to these parametric fields:

```
class Cat { val dangerous = false }
```

[24] **Public** is Scala's default access level. Private fields can only be accessed by methods defined in the same class (prevent outsiders from accessing the fields directly)
[25] Here is using the notation for procedures, i.e. methods/functions with side effects (in this case the side effect is that **sum** is reassigned).
[26] If a class doesn't have a body, you don't need to specify empty curly braces (though you could, of course, if you wanted to).
[27] If inside of the **Rational** class the method **add** is defined as follow: **def add(t: Rational): Rational = new Rational(n * t.d + t.n * d, d * t.d)**, the compiler will complain about **t.d** and **t.n**, because it's only possible access to the values of the **t**'s parameters (using theirs parameters names **n** and **d**) inside of **t**.

class Tiger(override val dangerous: Boolean, private var age: Int) extends Cat

4.3.2 Primary constructors

Primary constructor

The Scala compiler will gather up these two class parameters and create a primary constructor that takes the same two parameters:

- The Scala compiler will compile any code you place in the class body, which isn't part of a field or a method definition, into the primary constructor:
  - **class Rational(n: Int, d: Int) { println("Created "+ n +"/"+ d) }**
    - the Scala compiler would place the call to **println** into Rational's primary constructor.
  - **val r = new Rational (1,2)**
    - This create the new **Rational** and print to screen **"Created ½"**.

Invoking superclass constructors

The class **Ratio** extends **Rational** as follows:

class Ratio (n: Int, d: Int, val r: Double) extends Rational (n, d) { .. }

Since **Ratio** extends **Rational**, and **Rational**'s primary constructor takes two parameters, **Ratio** needs to pass these arguments to the primary constructor of its superclass. To invoke a superclass constructor, you simply place the argument or arguments you want to pass in parentheses following the name of the superclass:

- Then, the **Ratio**'s primary constructor automatically invokes the **Rational**'s primary constructor as its first action[28].

4.3.3 Auxiliary constructors

Sometimes you need multiple constructors in a class. In Scala, constructors other than the primary constructor are called auxiliary constructors.

In order to avoid infinite recursion, Scala requires each auxiliary constructor to invoke another constructors defined before it:

- Auxiliary constructors in Scala start with **def this(...)** .
- The constructor invoked may be either another auxiliary constructor or the primary constructor[29], and must be the first statement in the auxiliary constructor's body:

[28] Thus, in a Scala class, only the primary constructor can invoke a superclass constructor.
[29] The net effect of this rule is that every constructor invocation in Scala will end up eventually calling the primary constructor of the class. The primary constructor is thus the single point of entry of a class.

    - In other words, the first statement in every auxiliary constructor in every Scala class will have the form **this(. . .)**

- Additional processing can occur after this call.

```
class Rational(n: Int, d: Int) {
        require(d != 0)
        // auxiliar constructor
        def this(n: Int, str: String) = {
                this(n, 1)  //here is calling the primary constructor
                println("Auxiliar constructor” + str) //additional processing
        }
}
```

4.3.4 Parameterless methods

There are two ways to declare a method without parameters:

- Parameterless methods: the method is declared without parentheses:
    - The convention is declare a parameterless method, when it have no side effects
- Empty-paren methods: the methods is defined with empty parentheses:

Scala is very liberal when it comes to mixing parameterless and empty-paren methods:

- In particular, you can override a parameterless method with an empty-paren method, and vice versa.
- You can also leave off the empty parentheses on an invocation of any function that takes no arguments[30].

The use of parameterless methods supports the uniform access principle, which says that client code should not be affected by a decision to implement an attribute as a field or method[31].

4.3.5 Abstract classes

Scala allows the declaration of abstract classes:

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

The **abstract** modifier signifies that:

[30] However, it is recommended to still write the empty parentheses when the invoked method has side effects.
[31] If the method is parameterless, the call to the method looks the same that the call to a field, then, the client’s code is not affected if you for example decides to change the member from method to a field.

- The class is abstract, as a result, you cannot instantiate an abstract class.
- The class may have abstract methods[32], i.e. methods without implementation (i.e., no equals sign or body), as for example the method **contents**:
    - Unlike Java, no **abstract** modifier is allowed on method declarations:
        - Methods that do have an implementation are called concrete, as the method **width**:
            - Class **Element** declares the abstract method **contents**, and defines the concrete method **width**.

4.3.6 Inheritance and composition

Inheritance and composition enables code reuse.

Extending classes

You can use the **extends** modifier to subclass an existent class:

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

The **extends** clause has two effects:

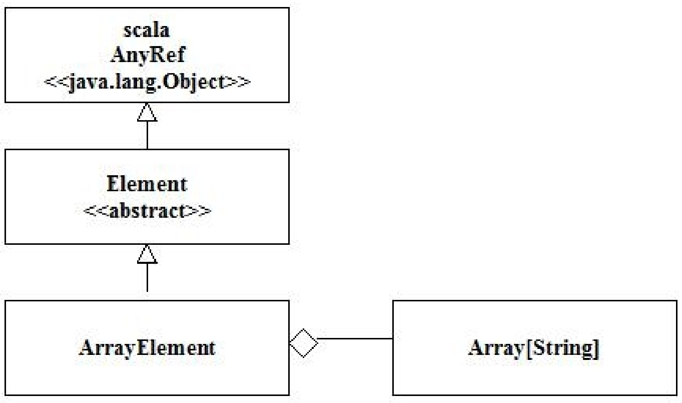
- It makes class **ArrayElement** inherit all non-private members[33] from class **Element**.
- It makes the type **ArrayElement** a subtype of the type **Element**:
    - Given **ArrayElement** extends **Element**, class **ArrayElement** is called a subclass of class **Element**. Conversely, **Element** is a superclass of **ArrayElement**.

If you leave out an **extends** clause, the Scala compiler implicitly assumes your class extends from **scala.AnyRef**, which on the Java platform is the same as class **java.lang.Object**. It's possible extends only one class.

Inheritance vs. composition

Composition[34] and inheritance are two ways to define a new class in terms of another existing class.

[32] A class with abstract members must itself be declared abstract.
[33] 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 (overrides) in the subclass.
[34] Composition means one class holds a reference to another, using the referenced class to help it fulfill its mission.



In the figure above the class **ArrayElement** inherits from the class **Element** and composites the **Array** class:

- Composition: if what you're after is primarily code reuse, you should in general prefer composition to inheritance:
  - Only inheritance suffers from the fragile base class problem, in which you can inadvertently break subclasses by changing a superclass.
- Inheritance: in order to decide if to use inheritance you need to ask two things:
  - It's necessary to model an is-as relationship?: If yes, then inheritance is the right option.
  - The clients will want to use the subclass type as a superclass type?: If yes, then inheritance is the right option.

4.3.7 Polymorphism and dynamic binding

4.3.8 Overriding methods and fields

Scala has just two namespaces for definitions:

- values (fields, methods, packages, and singleton objects)[35]
- types (class and trait names)

The above situation makes it possible for a field to override a parameterless method.

By default, each Scala class inherits the implementation of **toString** defined in class **java.lang.Object**, which just prints the class name, an @ sign, and a hexadecimal number[36]. You can override the default implementation by adding a method **toString** to class Rational, like this:

[35] This means that *fields*, *methods*, *packages* and *singleton objects* share the same namespace, and this means that for example *fields* and *methods* aren't allowed to share the same name in the same class.
[36] The result of **toString** is primarily intended to help programmers by providing information that can be used in debug print statements, log messages, test failure reports, and interpreter and debugger output.

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

The **override** modifier in front of a method definition signals that a previous method definition is overridden. Scala requires such a modifier for all members that override a concrete member in a parent class.

4.3.9 Checking preconditions

A precondition is a constraint on values passed into a method or constructor, a requirement which callers must fulfill. One way to do that is to use **require**[37]:

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

Above, a precondition was added over the primary constructor, that **d** must be non-zero.

4.3.10 Method overloading

Two or more methods can have the same name as long as their full signatures are unique:

- The signature includes the:
    - type name (class or trait name)
    - the list of parameters with types
    - method's return value.

However, there is an exception to this rule due to type erasure (JVM only):

```
// This example WON'T COMPILE
object Foo {
    def bar(list: List[String]) = list.toString
    def bar(list: List[Int]) = list.size.toString
}
```

You'll get a compilation error on the second method because the two methods will have an identical signature after type erasure[38].

4.3.11 Self reference

[37] The **require** method is defined in standalone object, **Predef**.

[38] After type erasure both method will take a parameter of the type **List** (instead of **List[String]** and **List[Int]**).

The keyword **this** refers to the object instance on which the currently executing method was invoked, or if used in a constructor, the object instance being constructed.

```
class Test {
        private val g = 0;
        def sum(g: Int) = this.g + g
}
```

In the above example, **this** allow discriminate between the **g** field of the **Test** class and the **g** local variable of the **sum** method.

4.3.12 Defining operators

The current implementation of Rational addition is OK, but could be made more convenient to use. You might ask yourself why you can write: **x + y** if **x** and **y** are integers or floating-point numbers, but you have to write: **x.add(y)** or at least: **x add y** if they are rational numbers[39]. It's possible, only it's necessary to replace **add** by the usual mathematical symbol +. This is straightforward, as + is a legal identifier in Scala:

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

4.3.13 Implicit conversions

Now that you can write **r * 2** [40], you might also want to swap the operands, as in **2 * r**. Unfortunately this does not work yet, because **2 * r** is equivalent to **2.*(r)**, so it is a method call on the number **2**, which is an integer. But the **Int** class contains no multiplication method that takes a **Rational** argument.

There is a way to solve this problem in Scala: You can create an implicit conversion that automatically converts integers to rational numbers when needed. Try adding this line in the interpreter:

- **implicit def intToRational(x: Int) = new Rational(x)**

[39] Assuming that there is an **add** method in the class **Rational**.
[40] Assuming that **r** is a variable with the **Rational** type.

The **implicit** modifier in front of the method tells the compiler to apply it automatically in a number of situations. With the conversion defined, typing in the line command **2*r** will work.

Note that for an implicit conversion to work, it needs to be in scope. If you place the implicit method definition inside class **Rational**, it won't be in scope in the interpreter.

4.4 Instantiate a class

When you instantiate a class[41] in Scala using **new**, you can configure (parameterize) the instance with values (between parentheses) and types (between square brackets):

- Value: **val big = new java.math.BigInteger("12345")**
- Type and value: **val greetStrings = new Array[String](3)**[42]
  - The type of the new array is String and the array size is 3.

When you instantiate a class (i.e. creates an object), the runtime sets aside some memory to hold the image of that object's state—i.e., the content of its variables.

4.5 All in Scala is a class instance[43]

Scala achieves a conceptual simplicity by treating everything[44], from arrays to expressions, as class instances with methods.

Given the following statements:

```
val ar = Array("z", "o", "t")
for (i <- 0 to 2)
  println(ar(i)[45])
ar(0) = "y"
```

The first statement is creating an array instance (pointed by the variable **ar**) of the type **String** with 3 elements:

- **A general Scala rule is**: when you apply to a variable parentheses surrounding one or more values, Scala will transform the code into an invocation of a method named **apply** on that variable[46]:
  - So, **Array("z", "o", "t")** is calling a factory method **apply** which creates and returns the new array[47].

[41] An object is a class instance.
[42] The statement is semantically equivalent to **val greetStrings: Array[String] = new Array[String](3)**, because the Scala compiler can make type inference.
[43] Every value is an object (and every function is a value, including methods).
[44] In Scala, everything (except a method) is an instance of a class.
[45] Here **ar(i)** gets transformed into **ar.apply(i)**, using the first general Scala rule.
[46] Of course this will compile only if the type of the variable defines an **apply** method.
[47] This **apply** method takes a variable number of arguments and is defined on the **scala.Array** companion object.

The second statement print each element of the array **ar**:

- **A general Scala rule is**: if a method takes only one parameter, you can call it without a dot or parentheses:
    - The sentence **0 to 2** is a shorcut of **(0).to(2)**, because **0** and **2** are instances of the Scala class **Int**, and I'm using the **to** method of the instance **0**.

The third statement updates[48] the first element of the array **ar**:

- **A general Scala rule is**: when an *assignment* is made to a variable to which parentheses and one or more arguments have been applied, the compiler will transform that 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:
    - So, **ar(0) = "y"** gets transformed into **ar.update(0,"y")**.

4.6 Singletons

Classes in Scala cannot have static members[49], instead, Scala has singleton objects[50]:

```
import scala.collection.mutable.Map

object ChecksumAccumulator {
        private val cache = Map[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
                }
}
```

A singleton object definition looks like a class definition, except instead of the keyword **class** you use the keyword **object**:

[48] I can't assign to **ar** a different array (because **ar** is a **val**), but I can change the individual values of the array object because the arrays are mutable, Although you can't change the length of an array after it is instantiated, you can change its element values. For an immutable sequence of objects that share the same type you can use Scala's **List** class.

[49] I.e. the posibility to call the member without instantiate the class. In this sense Scala is more object-oriented than Java.

[50] A special kind of class that has one and only one instance, a singleton is a first-class object (you can think of a singleton object's name, therefore, as a "name tag" attached to the object).

- When a singleton object shares the same name with a class[51], it is called that class's companion object, and the class is called the companion class of the singleton object:
    - You must define both the class and its companion object in the same source file.
    - A class and its companion object can access each other's private members.
- 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:
        - collecting related utility methods together,
        - defining an entry point to a Scala application,
        - implementing the singleton pattern, etc.
- Defining a singleton object doesn't define a type (at the Scala level of abstraction)[52]. Given just a definition of object ChecksumAccumulator, you can't make a variable of type ChecksumAccumulator:
    - You can invoke methods on singleton objects using the name of the singleton object, a dot, and the name of the method:
        - **ChecksumAccumulator.calculate("Every value is an object.")**
- Singleton objects cannot take parameters:
    - Because you can't instantiate a singleton object with the **new** keyword, you have no way to pass parameters to it.
    - Each singleton object is implemented as an instance of a synthetic class[53] referenced from a static variable:
        - A singleton object is initialized the first time some code accesses it.

4.7 Visibility scope

There are two kinds of "users" of a type[54]:

- Derived types[55],
- Code that works with instances of the type.

[51] In the example the statement **val acc = new ChecksumAccumulator** is using the class defined in the section 3.1 .
[52] Then isn't allowed to extend a singleton object, but a singleton object can inherit from a class or trait (you can invoke its methods via these types, refer to it from variables of these types, and pass it to methods expecting these types).
[53] The name of the synthetic class is the object name plus a dollar sign. Thus the synthetic class for the singleton object named **ChecksumAccumulator** is **ChecksumAccumulator$**.
[54] Class or trait.
[55] Derived types usually need more access to the members of their parent types than users of instances do.

The keywords that modify visibility, such as **private** and **protected**, appear at the beginning of declarations. You'll find them[56] before the **class** or **trait** keywords for types, before the **val** or **var** for fields, and before the **def** for methods[57].

Name	Keyword	Description
public	None	Public members and types are visible everywhere, across all boundaries.
protected	**Protected**	Protected members are visible to the defining type and to derived types. Protected types are visible only within the same package.
private	**Private**	Private members are visible only within the defining type. Private types are visible only within the same package.
scoped protected	**protected[**scope**]**	Visibility is limited to *scope*, which can be a package, type, or this (meaning the same instance, when applied to members, or the enclosing package, when applied to types).
scoped private	**private[**scope**]**	Synonymous with scoped protected visibility.

Visibility scopes

The following examples will be over *field* members. *Method* and *type* declarations behave the same way:

- Public visibility: any declaration without a visibility keyword is "public", meaning it is visible everywhere. There is no public keyword in Scala. This is in contrast to Java, which defaults to public visibility only within the enclosing package.
- Protected visibility: is for the benefit of implementers of child types, who need a little more access to the details of their parent types. Any member declared with the protected keyword is visible only to the defining type, including other instances of the same type and any derived types. When applied to a type, protected limits visibility to the enclosing package[58].
- Private visibility: Private visibility completely hides implementation details, even from the implementers of derived classes. Any member declared with the private keyword is visible only to the defining type, including other instances of the same type. When applied to a type, private limits visibility to the enclosing package.

[56] You can't apply any of the visibility modifiers to packages. Therefore, a package is always public, even when it contains no publicly visible types.
[57] You can also use an access modifier keyword on the primary constructor of a class. Put it after the type name and type parameters, if any, and before the argument list, as in this example: **class Restricted[+A] private (name: String) {…}**
[58] Java, in contrast, makes protected members visible throughout the enclosing package (not only for the childs types as Scala).

- Scoped Private and Protected Visibility: Scala allows you to fine-tune the scope of visibility with the scoped private and protected visibility declarations. Note that using private or protected in a scoped declaration is interchangeable, as they behave identically[59]:
    - The most restrictive visibility is **private[this]**, next **private[T]** visibility for type **T**, next **private[P]** for package **P**.

4.8 Object equality

In Scala **==** (or its inverse **!=**) always compare value equality, both primitive and reference types:

- In Java, you can use **==** to compare both primitive and reference types:
    - On primitive types, compares value equality, as in Scala.
    - On reference types, compares reference equality, which means the two variables point to the same object on the JVM's heap.
- Scala provides a facility for comparing reference equality, as well, under the name **eq**. However, **eq** and its opposite, **ne**, only apply to objects that directly map to Java objects.
- Examples:
    - Same object type:
        - **1 = = 2** (false)
        - **1 != 2** (true)
        - **List(1, 2, 3) == List(1, 2, 3)** (true)
        - **List(1, 2, 3) == List(4, 5, 6)** (false)
        - **("he"+"llo") = = "hello"** (true)
    - Different object type:
        - **1 == 1.0** (true)
        - **List(1, 2, 3) == "hello"** (false)
    - Against **null** (or things that might be **null**):
        - **List(1, 2, 3) == null** (false)

The Scala **= =** behavior is accomplished with a very simple rule: first check the left side for **null**, and if it is not **null**, call the **equals** method. Since **equals** is an infix method, the precise comparison you get depends on the type of the left-hand argument[60]:

- This kind of comparison will yield **true** on different objects, so long as:
    - their contents are the same,
    - their equals method is written to be based on contents.

[59] While either choice is fine, it is more common to see **private**[X] rather than **protected**[X].
[60] Because the method is invoked on the object at the left-hand.

5 Working with functions

Scala offers several ways to define functions that are not present in Java. Besides methods, which are functions that are members of some object, there are also functions nested within functions, function literals, and function values.

5.1 *First-class functions*

5.1.1 What is a function?

Function object

A function in Scala is a complete object (a function object)[61]. In Scala, a function object is simply an object that has one or more **apply** methods. So if we have:

```
object Square {
        def apply(in: double) = in * in
        def apply(in: float) = in * in
        def apply(in: int) = in * in
}
```

The **Square** object is a function object that computes the square of a number, and can do it for **Double**, **Float**, and **Int** types. To use the function object is also quite simple:

```
val xSquared = Square(x)
```

The above syntaxis is because Scala treats **<object>(<args>)** as syntactic sugar[62] for **<object>.apply(<args>)**.

FunctionN traits

Defining a function through the creation of an object with an **apply** method is Ok, but there is a better way: creating an object mixed with some of the Scala traits **scala.Function0**, **scala.Function2**[63], etc:

```
object Square extends Function1[double,double] {
        def apply(in: double) = in * in
}
```

Making your function objects derive from the appropriate **FunctionN** trait is generally a good idea, but it has one restriction: only one **apply** method can be defined. But despite this restriction, you gain in generality:

[61] It's a functor , i.e. an object that acts as a function (that you can use like a function).

[62] Syntactic sugar is added syntax to make certain constructs easier or more natural to specify (a shorthand syntax). The step in which the compiler replaces these constructs by their more verbose equivalents is called desugaring. In the example above the syntactic sugar is **Square(x)** and the desugaring is **Square.apply(x)**.

[63] **FunctionN** receives **N+1** parameters, the **N** first parameters are the functions arguments and the last parameter is the function type. Then the trait **scala.Function1** has two arguments, this trait represents the functions with a single argument.

- Mixing the function object with the trait **FunctionN** allows you to pass the object to a method which accepts an arbitrary function:
    - For example if you have defined the following method: **def someMethod(f: Function1[double,double]) = {val r = f(12); println(r)}**, you are allowed to invoke it with **Square**: **someMethod(Square)**.

Due to the functions of a single argument are fairly common, Scala provides the trait **Function** as an alias for the trait **Function1**[64], then the the above definition of **Square** is equivalent to:

```
object Square extends Function[double,double] {
        def apply(in: double) = in * in
}
```

Another alternative notation for **scala.FunctionN** is the **=>** declaration. For example:

```
object Square extends double => double {
        def apply(in: double) = in * in
}

def someMethod(f: double => double) = {val r = f(12); println(r)}
```

5.1.2 Function literal

Scala has first-class functions: which means you can express functions with no name (anonymous) in function literal syntax[65], i.e., **(x: Int) => x + 1**, and that functions can be represented by objects, which are called function values[66] (aka function objects). For example:

```
class test {
        def m1(x:Int) = x+3
        val f1 = (x:Int) => x+3
}
```

When the class test is compiled[67], Scala creates two files: **test.class** and **test$$anonfun$1.class**[68]. That strange extra class file is the anonymous class[69] for the function object that Scala created in response to the function expression assigned to **f1**.

[64] The **Function** trait is defined in the **scala.Predef** object.
[65] That follows the lambda-calculus: **λx.x+1** . The parameters are at the left hand of the **=>** (**x:Int**), and the function body is at the right hand (**x+1**).
[66] The distinction between function literals and values is that function literals exist in the source code, whereas function values exist as objects at runtime. The distinction is much like that between classes (source code) and objects (runtime).
[67] With scalac.
[68] You can run javap on the function class **test$$anonfun$1** in order to see the java code generated.
[69] Which extends the trait **scala.Function1**, because the function has a single parameter.

Function values are objects, so you can store them in variables. These variables are functions, too, so you can invoke them using the usual parentheses function-call notation:

```
var increase = (x: Int) => x + 1
increase(10)
```
[70]

If you want to have more than one statement in the function literal, surround its body by curly braces and put one statement per line, thus forming a block:

```
increase = (x: Int) => {
                        println("We")
                        println("are")
                        println("here!")
                        x + 1
                  }
```

5.1.3 Short forms of function literals

Scala provides a number of ways to leave out redundant information and write function literals more briefly:

- Leave off the parameter types[71]. You can simply start by writing a function literal without the argument type, and, if the compiler gets confused, add in the type:
  - **someNumbers.filter((x) => x > 0)**
- Leave out parentheses around a parameter whose type is inferred:
  - **someNumbers.filter(x => x > 0)**

5.1.4 Repeated parameters

Scala allows you to indicate that the last parameter to a function may be repeated. This allows clients to pass variable length argument lists to the function:

- **def echo(args: String*) = for (arg <- args) println(arg)**

To denote a repeated parameter, place an **asterisk** after the type of the parameter. Inside the function, the type of the repeated parameter is an **Array** of the declared type of the parameter[72].

[70] Because **increase**, in this example, is a **var**, you can reassign it a different function value later on: **increase = (x: Int) => x + 9999**.

[71] This is called target typing, because the targeted usage of an expression, in this case an argument to **someNumbers.filter()** , is allowed to influence the typing of that expression—in this case to determine the type of the x parameter.

[72] Thus, the type of **args** inside the **echo** function, which is declared as type **String*** is actually **Array[String]**.

Nevertheless, if you have an array of the appropriate type, and attempt to pass it as a repeated parameter, you'll get a compiler error. To accomplish this, you'll need to append the array argument with a colon (**:**) and an _*:

- **val arr = Array("What's", "up", "doc?")**
  echo(arr: _*)[73]

5.1.5 Placeholder syntax

To make a function literal even more concise, you can use underscores (_) as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal:

- For example, instead of using x => x > 0 , it's possible use _ > 0, because **x** appears only one time within the body (**x** >**0**) of the function literal:
  - Then, you can use the function literal as allways: **someNumbers.filter(_ > 0)**

Sometimes when you use underscores as placeholders for parameters, the compiler might not have enough information to infer missing parameter types:

- For example: **val f** = _ + _ , will not compile because the Scala compiler can't infer the parameter types:
  - In these cases you can specify the types using a colon:
    - **val f = (_: Int) + (_: Int)**[74]
  - Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly.

5.1.6 Partially applied functions

You can also replace an entire parameter list with an underscore[75] (_):

- For example, given the following function: **def sum(a: Int, b: Int, c: Int) = a + b + c**
  - A partially applied function[76] is an expression in which you don't supply all of the arguments needed by the function:
    - Defining a partially applied function of **sum**: **val a = sum _**
      - Next, you can call **a** as follow: **a(1,2,3)**
    - Defining another partially function of **sum**: **val b = sum(1, _: Int, 3)**
      - Next, you can call b as follow: **b(2)**

If you are writing a partially applied function expression in which you leave off all parameters, such as **sum** _, you can express it more concisely by leaving off the underscore if a function is required at that point in the code:

[73] This notation tells the compiler to pass each element of **arr** as its own argument to **echo**, rather than all of it as a single argument.
[74] It definition is equivalent to **val f = (x:Int,y:Int) => x + y** .
[75] Remember that you need to leave a space between the function name and the underscore.
[76] There is a trait **scala.PartialFunction**.

- Instead of **someNumbers.foreach(println _)** you can use **someNumbers.foreach(println)** because in the **foreach** a function is required as parameter.

5.1.7 Closures

The body of a function literal may refer to variables defined elsewhere, not only to variables defined as a parameters of the function literal:

- For example in: **(x: Int) => x + y**, the variable **y** isn't a parameter of the function literal, as it is the variable **x**.
- From the point of view of the function literal:
  - The **x** variable is a bound variable.
  - The **y** variable is a free variable.

If you try using this function literal by itself, without **y** defined in its scope, the compiler will complain, there must be a **y** in the scope of the function literal, as in the following example:

- **var y = 1; val addY = (x: Int) => x + y**
  - For example, **addY(10)** will return **11**.

A function literal with no free variables, such as **(x: Int) => x + 1**, is called a closed term. But any function literal with free variables, such as **(x: Int) => x + y**, is an open term.

- The function value (the object) that's created at runtime from an open term is called a closure[77]:
  - The resulting function value will contain references (a pointer) to the captured free variables:
    - If the variable to which the free variable references changes, the function value will change: **y=2; addY(10)** will return 12.
    - The same is true in the opposite direction. Changes made by a closure to a captured variable are visible outside the closure.
  - What if a closure accesses some variable that has several different copies as the program runs? For example, what if a closure uses a local variable of some function, and the function is invoked many times? Which instance of that variable gets used at each access?:
    - The instance used is the one that was active at the time the closure was created. The Scala compiler rearranges things in cases like this so that the captured parameter lives out on the heap, instead of the stack, and thus can outlive the method call that created it:
      - For example: **def Increaser(y: Int) = (x: Int) => x + y**. Each time this function is called will create a new

[77] The name arises from the act of "closing" the function literal by "capturing" the bindings of its free variables.

closure. Each closure will access the **y** variable that was active when the closure was created:

- For example: **val inc9999 = Increaser(9999); inc9999(10)** will return **10009**.

5.2 *Function vs. methods*

In Scala functions and methods are not the same thing:

- Function: is a functional object, i.e. an object that has one or more **apply** methods.
- Method: a method is a member of some object, and is always defined with the keyword **def**:
  - The syntax is: **def funtion_name(par1:tipo1, ..,parN:tipoN): result_type = { .. }**:
    - The result_type is optional[78], unless it isn't obvious or the method is recursive.
    - If the method consists of just one statement, you can optionally leave off the curly braces:
      - **def max(x: Int, y: Int) = if (x > y) x else y**
    - A result_type **Unit** indicates the method 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.

A method can be type-parameterized, but an anonymous function can not:

- For example: **def m2[T](x:T) = x.toString.substring(0,4)**
- However, if you are willing to define an explicit class for your function, then you can type-parameterize it similarly:

  -
    ```
    class myfunc[T] extends Function1[T,String] {
            def apply(x:T) = x.toString.substring(0,4)
    }
    val f = new myfunc[String]
    ```

In the absence of any explicit **return** statement, a Scala function/method returns the last value computed by the function/method:

- The recommended style for functions/methods is in fact to avoid having explicit, and especially multiple, return statements[79]. Instead, think of each function/method as an expression that yields one value, which is returned:

[78] The Scala compiler can make type inference.
[79] On the other hand, design choices depend on the design context, and Scala makes it easy to write methods that have multiple, explicit **returns**.

- This philosophy will encourage you to make function/methods quite small, to factor larger function/methods into multiple smaller ones.

It's possible to make an eta expansion, i.e. pass a method as a function object, the way is a partial application of the method[80] , which yields a function object:

-
  ```
  def method1(f: double => double)
  def method2 (in:Int) = in*in
  method1(method2 _) // Here method1 is receiving a partial application of method2
  ```

One important characteristic of function/methods parameters in Scala is that they are **vals**, not **vars**.

5.3 Local functions

An important design principle of the functional programming style: programs should be decomposed into many small functions (helper functions) that each do a well-defined task[81].

One problem with the approach before is that all the helper function names can pollute the program namespace. Once functions are packaged in reusable classes and objects, it's desirable to hide the helper functions from clients of a class. In Java, the main tool for this purpose is the **private** method. Scala offers an additional approach: you can define functions inside other functions. Just like local variables, such local functions are visible only in their enclosing block:

```
import scala.io.Source
object LongLines {

    def processFile(filename: String, width: Int) {

        def processLine(line: String) {
            if (line.length > width)
            print(filename +": "+ line)
        }

        val source = Source.fromFile(filename)

        for (line <- source.getLines)
            processLine(line)
    }
}
```

The function **processLine**[82] is a local function of the function **processFile**. The function **processLine** can access the parameters of their enclosing function (**processFile**).

[80] The method partial application so serves as a means of getting a function from a method.

[81] Individual functions are often quite small. The advantage of this style is that it gives a programmer many building blocks that can be flexibly composed to do more difficult things. Each building block should be simple enough to be understood individually.

[82] It serves as a helper function of the function **processFile**.

6 Built-in Control Structures

The only built-in control structures are **if**, **while**, **for**, **try**, **match**, and *function calls*. The reason Scala has so few is it has function literals that can be used to implement custom control structures.

One thing you will notice is that almost all of Scala's control structures result in some value[83]. This is the approach taken by functional languages, in which programs are viewed as computing a value, thus the components of a program should also compute values.

6.1 If expressions

Imperative style:

```
var filename = "default.txt"
if (!args.isEmpty)
        filename = args(0)
```

Functional style:

```
val filename =
        if (!args.isEmpty) args(0)
        else "default.txt"
```

The functional code has two advantages:

- The code uses **val** instead of **var**. It tells readers of the code that the variable will never change, saving them from scanning all code in the variable's scope to see if it ever changes.
- A second advantage to using a val instead of a var is that it better supports equational reasoning. The introduced variable is equal to the expression that computes it, assuming that expression has no side effects. Thus, any time you are about to write the variable name, you could instead write the expression.

6.2 While loops

The **while** and **do-while**[84] constructs are called "loops," not expressions, because they don't result in an interesting value, because the type of the result is **Unit**.

while example:

```
var a = 0
while (a < 0) {
```

[83] Programmers can use these result values to simplify their code, as for example, without this facility, the programmer must create temporary variables just to hold results that are calculated inside a control structure.
[84] It works like the **while** loop except that it tests the condition after the loop body instead of before.

```
        a += 1
    }
```

do-while example:

```
    var line = ""
    do {
        line = readLine()
        println("Read: "+ line)
    } while (line != "")
```

Other construct that results **Unit**[85], which is relevant here, is *assignment* to **vars**, then you need to take care using the assignment in a **while** condition:

```
    var line = ""
    while ((line = readLine()) != "") // This doesn't work!
    println("Read: "+ line)
```

The above condition **((line = readLine()) != "")** will return allways **true**, because **(line = readLine())** returns the unit value **()**, that is different from **""**.

6.3 For expressions

The Scala's **for** expression is a Swiss army knife of iteration. The simple uses enable common tasks such as iterating through a sequence. More advanced expressions can iterate over multiple collections of different kinds, can filter out elements based on arbitrary conditions, and can produce new collections.

Iteration through collections

```
    val filesHere = (new java.io.File(".")).listFiles
    for (file <- filesHere)
        println(file)
```
[86]

The expression **file <- filesHere** , which is called a generator, iterate through the elements of filesHere (that is an **Array[File]**). In each iteration, a new **val** named **file** is initialized with an element value.

The **for** expression syntax works for any kind of collection, not just arrays[87]. One convenient special case is the **Range** type. You can create Ranges using syntax like **1 to 5** or **1 until 5**:

The following iterate from **1 to 5**:

[85] The *unit value* and is written **()**.
[86] Because **File**'s **toString** method yields the name of the file or directory, the names of all the files and directories in the current directory (**"."**)will be printed.
[87] To be precise, the expression to the right of the **<-** symbol in a for expression can be any type that has certain methods, in this case **foreach**, with appropriate signatures.

```
for (i  <- 1 to 5)
        println("Iteration "+ i)
```

The following iterate from **1 to 4**:

```
for (i  <-  1 until 5)
        println("Iteration "+ i)
```

Filtering

Sometimes you do not want to iterate through a collection in its entirety. You want to filter it down to some subset. You can do this with a for expression by adding a filter: an **if** clause inside the for's parentheses:

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere if file.getName.endsWith(".scala"))
        println(file)
```

The filter here is: **if file.getName.endsWith(".scala"))**, which filter the collection to a subset that only contain those files in the current directory whose names end with **".scala"**.

You can include more filters if you want. Just keep adding **if** clauses[88]:

```
for (
      file <- filesHere
      if file.isFile;
      if file.getName.endsWith(".scala")
) println(file)
```

The filters here are: **if file.isFile** and **if file.getName.endsWith(".scala"))**, which filter the collection to a subset that only contain those files (and not directories) in the current directory whose names end with **".scala"**.

Nested iteration

If you add multiple **<-** clauses, you will get nested "loops."

```
def fileLines(file: java.io.File) =
        scala.io.Source.fromFile(file).getLines.toList

def grep(pattern: String, dir:String)  {
        val filesHere = (new java.io.File(dir)).listFiles
        for (
                file <- filesHere
                if file.getName.endsWith(".scala");
                line <- fileLines(file)
                if line.trim.matches(pattern)
```

[88] Separated by semicolons (**;**).

```
          ) println(file +": "+ line.trim)
    }
```

The outer loop iterates through **filesHere** for any file that ends with **.scala**, and the inner loop iterates through **fileLines(file)** for any line that matches the **pattern**.

Mid-stream variable bindings

Note that the previous code repeats the expression **line.trim**. This is a non-trivial computation, so you might want to only compute it once. You can do this by binding the result to a new variable using an equals sign (**=**). The bound variable is introduced and used just like a **val**.

```
    def grep(pattern: String, dir String) =
          for {[89]
                val filesHere = (new java.io.File(dir)).listFiles
                file <- filesHere
                if file.getName.endsWith(".scala")
                line <- fileLines(file)
                trimmed = line.trim
                if trimmed.matches(pattern)
          } println(file +": "+ trimmed)
```

In the example the variable **trimmed** is the bound variable.

Producing a new collection

It's possible generate a value to remember the iterated values. To do so, you prefix the body of the **for** expression by the keyword **yield**.

The following example defines the function **scalaFiles**, which identifies the **.scala** files and stores them in an **Array[File]**:

```
    def scalaFiles =
          for {
                file <- filesHere
                if file.getName.endsWith(".scala")
          } yield file[90]
```

The following example generates the **val** variable **forLineLengths**, which has the type **Array[Int]**:

[89] If you prefer, you can use curly braces instead of parentheses to surround the generators and filters. One advantage to using curly braces is that you can leave off some of the semicolons that are needed when you use parentheses.
[90] In each **for** iteration the **yield** is returning the **val** variable **file** and storing it in an **Array[File]**.

```
val forLineLengths =
        for {
                file <- filesHere
                if file.getName.endsWith(".scala")
                line <- fileLines(file)
                trimmed = line.trim
                if trimmed.matches(".*for.*")
        } yield trimmed.length
```

6.4 *Exception handling with try expressions*

Throwing exceptions

In order to throw an exception, you create an exception object, and then you throw it with the **throw** keyword:

```
throw new IllegalArgumentException
```

in Scala, **throw** is an expression that has the result type **Nothing**, and then you can use a **throw** as an expression[91]:

```
val half =
        if (n % 2 == 0)
                n / 2
        else
                throw new RuntimeException("n must be even")
```

One branch of the **if** computes a value, while the other throws an exception and computes **Nothing**. The type of the whole **if** expression is then the type of that branch which does compute something, because if the exception is throwed, the variable **half** never is assigned[92].

Catching exceptions

In Scala the exceptions are catched using pattern matching:

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
try {
        val f = new FileReader("input.txt")
        // Use and close file
} catch {
        case ex: FileNotFoundException => // Handle missing file
        case ex: IOException => // Handle other I/O error
```

[91] An expression is something that returns a value, then it's possible to assign an expression to a variable.
[92] Because of this, it is safe to treat the **throw** as an expression and is useful in cases like the example.

```
}
```

The **try** body is executed, and if it throws an exception, each **catch** clause is tried in turn. In this example:

- If the exception is of type FileNotFoundException, the first clause will execute.
- If it is of type IOException, the second clause will execute.
- If the exception is of neither type, the try-catch will terminate and the exception will propagate further.

The finally clause

You can wrap an expression with a finally clause if you want to cause some code to execute no matter how the expression terminates.

```
import java.io.FileReader
val file = new FileReader("input.txt")
try {
        // Use the file
} finally {
        file.close() // Be sure to close the file
}
```

Here you might want to be sure an open file gets closed even if a method exits by throwing an exception.

Yielding a value

The **try-catch-finally** results in a value, which is:

- The result of the **try** clause if no exception is thrown.
- The result of the relevant **catch** clause if an exception is thrown and caught.
- No result, if an exception is thrown but not caught.

The value computed in the **finally** clause, if there is one, is dropped[93].

The following example defines the function **urlFor**, which return a new **URL** object with the **path** passed, or an default **URL** if an **MalformedURLException** exception is caught:

```
import java.net.URL
import java.net.MalformedURLException
def urlFor(path: String) =
        try {
                new URL(path)
        } catch {
```

[93] Usually **finally** clauses do some kind of clean up such as closing a file; they should not normally change the value computed in the main body or a **catch** clause of the try, and is best to avoid returning values from **finally** clauses.

```
                    case e: MalformedURLException =>
                    new URL("http://www.scalalang.org")
            }
```

6.5 Match expressions

Scala's **match** expression lets you select from a number of alternatives, just like **switch** statements in other languages.

```
    val firstArg = if (!args.isEmpty) args(0) else ""
    val friend =
      firstArg match {
              case "salt" => "pepper"
              case "chips" => "salsa"
              case "eggs" => "bacon"
              case _ => "huh?"
      }
   println(friend)
```

If **firstArg** is the string **"salt"**, it returns **"pepper"** (which is stored in the variable **friend**), while if it is the string **"chips"**, it returns **"salsa"**, and so on. The default case is specified with an underscore[94].

The main differences with the Java switch:

- The variable used[95], could be any kind of constant, as well as other things not just the integer-type and enum constants.
- There are no **breaks** at the end of each alternative. Instead the **break** is implicit, and there is no fall through from one alternative to the next.
- The **match** expressions result in a value.

6.6 Living without break and continue

Scala doesn't have the **break** and **continue** control structures.

[94] The underscore (_), is a wildcard symbol frequently used in Scala as a placeholder for a completely unknown value.
[95] The variable **firstArg** in the example.

7 Control Abstraction

Control abstraction is a user or library function[96] that looks to programmers like a native language mechanism[97].

7.1 Currying

To understand how to make control abstractions that feel more like language extensions, you first need to understand the functional programming technique called currying.

The currying process transforms a function of two parameters into a function of one parameter which returns a function of one parameter which itself returns the result:

- The following shows a regular, non-curried function:
    - **def addNc(x:Int, y:Int) = x + y**
        - **addNc(1,2)** returns **3**
- The following shows a similar function that's curried:
    - **def addC(x:Int) = (y:Int) => x + y**[98]
        - **addC(1)(2)** returns **3**[99]

It's possible shorten the curried definition with the following Scala syntactic sugar:

- **def addC(x:Int)(y:Int) = x + y**
    - **addC(1)(2)** returns **3**

If you call the curried function with less parameters (using the placeholder notation), you get a partially applied function (that is the currying closure), like this:

- **val onePlus = addC(1)_**
    - **onePlus(2)** returns **3**.

7.2 By-name parameters

Suppose you want to implement an assertion construct called myAssert. The myAssert function will take a function value as input and consult a flag to decide what to do. If the flag is set, myAssert will invoke the passed function and verify that it returns true. If the flag is turned off, myAssert will quietly do nothing at all:

```
var assertionsEnabled = true
```

[96] As the **exists** method from the Scala's **Collection** API. It is a special-purpose looping construct provided by the Scala library rather than being built into the Scala language like **while** or **for**.
[97] Feel like native language support.
[98] In fact, the function **addC** returns a new function, the function literal **(y:Int) => x + y**, which is a closure, because has the free variable **x** (the **addC**'s parameter).
[99] **val second = addC(1);second(2)** returns **3**, because **second** is the closure that has the free variable **x** bound to **1**.

```
def myAssert(predicate: () => Boolean) =
        if (assertionsEnabled && !predicate())
                throw new AssertionError
```

The definition is fine, but using it is a little bit awkward: myAssert(() => 5 > 3), you would really prefer to leave out the ( ) and => symbol in the function literal and write the code like this: myAssert(5 > 3)[100]. In order to obtain this behavior, Scala offers a syntactic sugar[101] for parameterless functions[102]:

```
var assertionsEnabled = true
def myAssert(predicate: => Boolean) =
        if (assertionsEnabled && !predicate)
                throw new AssertionError
```

Now, it's possible to call **myAssert** with the syntax: myAssert(5 > 3). The result is that now using **myAssert** looks exactly like using a built-in control structure.

But there is one more benefit, in addition to look like a built-in control structure. This extra benefit, for which is named a by-name parameter[103], is the following:

- Because the type of **myAssert**'s **predicate** parameter is **=> Boolean**[104], the expression inside the parentheses in myAssert(5 > 3) is not evaluated before the call to **myAssert**. Instead a function value will be created whose apply method will evaluate **5 > 3** each time **predicate** gets used inside of the **myAssert** body:
    - As a function value is created, it support closures, then the following expression is valid:
        - x=1;myAssert(x>3), then **x>3** will generate a closure with **x** as its free variable that is bound to the value **1**.

7.3 Writing new control structures

Using currying and by-name parameters it's possible to define new control structures that looks a native control structure. The following example implements the **while** control structure:

```
def whileLoop(cond: () => Boolean, body: () => Unit): Unit =
        if (cond () ) {
                body
                whileLoop(cond,body)
```

[100] That it won't work, because the missing () and =>.
[101] To leave out the *empty parentheses* of the parameterless function when it's defined as a parameter of a high-order function. This syntactic sugar (only for parameterless functions) is named a by-name parameter, really Scala is using the trait **scala.ByNameFunction**.
[102] A function defined as () => AnyType is a parameterless function, because the function don't receive any parameter.
[103] In pass by value, the called method gets a copy of whatever the caller passed in. In pass by reference, the called method gets a reference to the caller's value. In pass by name, the argument of the called method is evaluated only when the called method does anything to the argument, i.e. argument evaluation happens every time the argument gets mentioned, and only when the argument gets mentioned.
[104] It's a function literal.

```
        }
```

A call to the above function will look as follows: **var x =1; whileLoop(() => x < 10, () => {print(x);x+=1})**, where **cond** is **() => x > 1**, and **body** is **() => { println(x);x+=1}**.

Using the by-name parameter syntactic sugar, the first **whileLoop** definition can be rewritten as:

```
def whileLoop(cond: => Boolean , body: => Unit): Unit =
        if (cond) {
                body
                whileLoop(cond,body)
        }
```

Now, the call to the above function will look as follows: **var x=1; whileLoop(x<10, { println(x);x+=1})**[105]. But this call still looks as a function call and not as a native control structure, then using currying, the second whileLoop definition can be rewritten as:

```
def whileLoop(cond: => Boolean )(body: => Unit): Unit =
        if (cond) {
                body; whileLoop(cond)(body)
        }
```

Now, the call to the above function will look as follows: **var x=1; whileLoop(x<10) { println(x);x+=1}**, which now, looks a native contol structure.

[105] Take note of the following: The expression **x<10** will generate a closure with **x** its free variable, and **{println(x);x+=1}** will also generate a closure with **x** its free variable, both closures referencing the same variable **x** (defined with **var x=1**). As the second closure is modifying the **x** variable, then, after the call **whileLoop(x<10, { println(x);x+=1})**, the variable **x** will have the value **10** (because the closure modifies the variable that is referencing).

8 Working with I/O

8.1 Reading from a file

The following is a script that reads from a file (the name is passed as an argument at command line), and prints its lines:

```
import scala.io.Source

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

Script description:

- The expression **Source.fromFile(args(0))** attempts to open the specified file and returns a **Source** object.
- Over the **Source** object you call **getLines**, which returns an **Iterator[String]**.[106]

[106] Which provides one line on each iteration, including the end-of-line character.

9 Useful Scala classes

9.1 Basic types

Type	Range
Byte	8-bit signed two's complement integer (-27 to 27 - 1, inclusive)
Short	16-bit signed two's complement integer (-215 to 215 - 1, inclusive)
Int	32-bit signed two's complement integer (-231 to 231 - 1, inclusive)
Long	64-bit signed two's complement integer (-263 to 263 - 1, inclusive)
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false
Char	16-bit unsigned Unicode character (0 to 216 - 1, inclusive)
String	a sequence of Chars

Scala basic types[107]

All the types shown in the table are members of the **scala** package, with the exception of the **String** type which resides in package **java.lang**.[108]

9.1.1 Literals

A literal is a way to write a constant value directly in code.

Integer literals

Rules:

- Integer literals for the types **Int**, **Long**, **Short**, and **Byte** come in three forms: decimal, hexadecimal (the number begins with a 0x or 0X), and octal (the number begins with a 0):
  - Decimal: **val num = 31**
  - Hexa: **val num = 0x5**
- If an integer literal ends in an **L** or **l**, it is a **Long**, otherwise it is an **Int**:
  - Long: **val num = 35L**

[107] Collectively, types **Byte**, **Short**, **Int**, **Long**, and **Char** are called **integral types**. The integral types plus **Float** and **Double** are called **numeric types**.

[108] Given that all the members of package **scala** and **java.lang** are automatically imported into every Scala source file, you can just use the simple names (i.e., names like Boolean, Char, or String) everywhere.

- If an **Int** literal is assigned to a variable of type **Short** or **Byte**, the literal is treated as if it were a **Short** or **Byte** type so long as the literal value is within the valid range for that type:
    - Short: **val num: Short = 367**

Floating point literals

Rules:

- Floating point literals are made up of decimal digits, optionally containing a decimal point, and optionally followed by an **E** or **e** and an exponent. If a floating-point literal ends in an **F** or **f**, it is a **Float**, otherwise it is a **Double**:
    - Double: **val num = 1.2345**
    - Float: **val num = 1.2345F**
    - Double: **val num = 123E45**[109]

Characters literals

Rules:

- Character literals are composed of any Unicode character between single quotes[110]:
    - Character **A**: **val c = 'A'**
- Also, you can provide an octal or hex number for the character unicode preceded by a backslash:
    - Character **A** with the unicode in octal[111]: **val c = '\101'**
    - Character **A** with the unicode in hexa[112]: **val c = '\u0041'**

String literals

Rules:

- A string literal is composed of characters surrounded by double quotes:
    - String **Hello**: **val s = "hello"**
- The syntax of the characters within the quotes is the same as with character literals:
    - String \"': **val s = "\\\"\""**
- Raw strings: because the syntax used before is awkward for strings that contain a lot of escape sequences or strings that span multiple lines, Scala includes a special syntax for *raw strings* You start and end a *raw string* with three double quotation marks in a row ("""):
    - Example: **println("""Welcome to Ultamix 3000.**
      Type "HELP" for help.""")

[109] 123E45 = 123* 10^{45}.
[110] There are special character literal scape sequences, as: **'\n'**, **'\b'**, **'\t'**, '\\', '\'', '\"', and others.
[111] The octal number must be between **'\0'** and **'\377'**.
[112] Four hex digits and preceded by a **\u**.

Symbols literals

Rules:

- A symbol literal is written **'ident**, where ident can be any alphanumeric identifier[113]:
  - Example: **val s = 'aa001bb**[114]
- Symbols are interned, if you write the same symbol literal twice, both expressions will refer to the exact same Symbol object[115].
- So what's a Symbol?[116]: it's like a dynamically created instance of a String; all Symbols with the same value are equivalent[117].

Boolean literals

Rules:

- The Boolean type has two literals, **true** and **false**:
  - Example: **val t = true**

9.1.2 Operators are methods

Scala provides a rich set of operators for its basic types. These operators are just a nice syntax for ordinary method calls:

- For example **1 + 2** really means: **(1).+(2)**

Type of operators:

- Infix operators: the method to invoke sits between the object and the parameter or parametersyou wish to pass to the method.
  - In Scala any method can be used as an infix operator: for example **s.indexOf('o', 5)**, can be used in operator notation as: **s indexOf('o', 5)**. If the method has only one argument, then can be used without parenthesis, for example example **s.indexOf('o')**, can be used in operator notation as: s indexOf 'o'.
- Prefix operators: you put the method name before the object on which you are invoking the method.
  - Prefix operators are a shorthand way of invoking methods: in this case, the name of the method has **unary_** prepended to the operator character:

[113] Then you can't have spaces in a symbol literal.
[114] Internally Scala is calling the factoy method **Symbol**, then is calling **Symbol("aa001bb")**. Over the variable **s** it's possible invoke the method **name**: **s.name** (it will return the **String** **aa001bb**).
[115] Symbols are interned strings. This means, for example, that equality comparison works correctly: **'abcd == 'abcd** will return true, while **"abcd" == "abcd"** might not (depending on JVM's whims).
[116] See: http://alblue.blogspot.com/2007/12/scala-introduction-to-scala-case.html .
[117] A **Symbol** is a short, convenient way of passing in data that's **matchable** and also **unique**. It's quite often used with **tuples** in languages like Erlang or Lisp as a poor man's data type or structured record. Thus, tuples such as **('Card,'Hearts, 1)** might be ways of representing structured data in such languages. One advantage of using data structures like this is that they're easy to **stream** (they're all tuples) and the Symbols remain the same after streaming across a network.

        - For example, Scala will transform the expression **-2** into the method invocation (2).unary_- .
    - The only identifiers that can be used as prefix operators are +, -,!, and ~. Thus, if you define a method named **unary_!**, you could invoke that method on a value or variable of the appropriate type using prefix operator notation, such as **!p**. But if you define a method named **unary_***, you wouldn't be able to use prefix operator notation, because * isn't one of the four identifiers that can be used as prefix operators. You could invoke the method normally, as in **p.unary_***, but not ***p** .
- Postfix operators: you put the method after the object.
    - In the case of a method that requires no arguments, you can alternatively leave off the dot and use postfix operator notation:
        - For example for **s.toLowerCase** you can use s toLowerCase.

9.1.3 Operations

Arithmetic

You can invoke arithmetic methods via infix operator notation for addition (+), subtraction (-), multiplication (*), division (/), and remainder (**%**), on any numeric type.

The numeric types also offer unary prefix operators + (method **unary_+**) and **-** (method **unary_-**),which allow you to indicate a literal number is positive or negative:

- If you don't specify a unary + or , **-** a literal number is interpreted as positive.
- The unary **-** can also be used to negate a variable:
    - Example: **val neg = 2; val neg2 = -neg**

Relational

You can compare numeric types with relational methods greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=), which yield a **Boolean** result:

- Example: **'a' >= 'A'**

Logical

The logical methods, logical-and (**&&**) and logical-or (||), take **Boolean** operands in infix notation and yield a Boolean result. In addition, you can use the unary **!** operator (the **unary_!** method) to invert a **Boolean** value.

The logical-and and logical-or operations are short-circuited as in Java: expressions built from these operators are only evaluated as far as needed to determine the result. In other words, the right-hand side of logical-and and logical-or expressions won't be evaluated if the left-hand side determines the result[118].

Bitwise

Scala enables you to perform operations on individual bits of integer types with several bitwise methods:

- bitwise-and **&**: **1 & 2** = (0001) & (0010) = (0&0) (0&0) (0&1) (1&0) = 0000 = 0
- bitwise-or | : 1 | 2 = (0001) | (0010) = (0|0) (0|0) (0|1) (1|0) = 0011 = 3
- bitwise-xor ˆ : 1 ˆ 3 = (0001) ˆ (0011) = (0ˆ0) (0ˆ0) (0ˆ1) (1ˆ1) = 0010 = 2
- The unary bitwise complement operator ~ (the method **unary_~**): inverts each bit in its operand: ~1 = ~0001 = 11111111111111111111111111111110 = -2
- shift left << , shift right >> and unsigned shift right >>>

9.1.4 Operator precedence and associativity

Operator precedence

Operator precedence determines which parts of an expression are evaluated before the other parts:

- Given that Scala doesn't have operators, just methods, Scala decides precedence based on the first character[119] of the methods used in operator notation:
  - For example **a +ii b *ii c** will be evaluated **(a).+ii((b).*ii(c))** , because the first character of the method **+ii** is + and the first character of the method ***ii** is *.
  - The one exception to the above rule, is the following: if an operator ends in an equals character (=), and the operator is not one of the comparison operators <=, >=, ==, or =, then the precedence of the operator is the same as that of simple assignment (=):
    - For example **x = y + 1** will be evaluated **(x).*=((y).+(1))** , because *= is classified as an assignment operator whose precedence is lower than +.

Operator associativity

When multiple operators of the same precedence appear side by side in an expression, the associativity of the operators determines the way operators are grouped. The associativity of an operator in Scala is determined by its last character:

- Any method that ends in the **:** character is invoked on its right operand, passing in the left operand:
  - For example **a:::b** will be evaluated **(b).:::(a)**

[118] How short-circuiting can work given operators are just methods?. Normally, all arguments are evaluated before entering a method, so how can a method avoid evaluating its second argument? The answer is that all Scala methods have a facility for delaying the evaluation of their arguments, or even declining to evaluate them at all. The facility is called by-name parameters.
[119] The list shows the precedence given to the first character of a method in decreasing order of precedence: * / % , + - , : , = ! , < > , & , ˆ , | , **all letters**, **all assignment operators** .

- Methods that end in any other character are invoked on their left operand, passing in the right operand:
  - For example **a*b** will be evaluated **(a).*(b)**

Operands evaluation order

No matter what associativity an operator has, however, its operands are always evaluated left to right:

- So if **b** is an expression that is not just a simple reference to an immutable value, then **a ::: b** is more precisely treated as the following block:
  - **{ val x = a; b.:::(x) }**[120]

9.1.5 Rich wrappers

You can invoke many more methods[121] on Scala's basic types than were described in the previous sections. These methods are available via implicit conversions, for each basic type described in this chapter, there is also a rich wrapper that provides several additional methods.

Type	Rich wrapper
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean
Char	scala.runtime.RichChar
String	scala.runtime.RichString

9.2 Collections

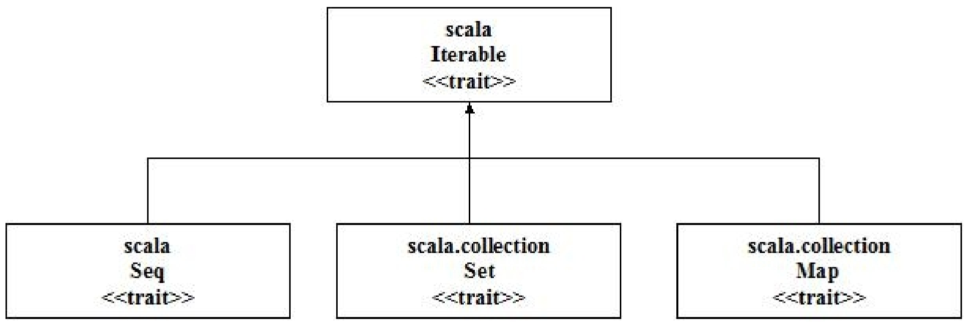
The main collection trait is **Iterable**, which is the supertrait of both mutable and immutable variations of sequences (Seqs), sets, and maps. **Iterable** is so named because it represents collection objects that can produce an **Iterator** via a method named **elements**.

Iterable provides dozens of useful concrete methods, all implemented in terms of the **Iterator** returned by elements, which is the sole abstract method in trait **Iterable**. Among the methods defined in **Iterable** are many higher-order methods[122], some example are **map**, **flatMap**, **filter**, **exists**, and **find**.

[120] As **a** is the left operand, it's evaluated before **b** and then the result of this evaluation is passed as an operand to **b**'s **:::** method.

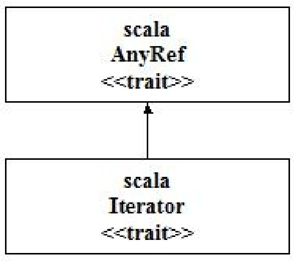
[121] For example **0 max 5**, that returns **5**, **(1.0/0).isInfinity** that returns **true**, **"robert" drop 2** that returns **"bert"**, etc.

The difference between **Iterable** and **Iterator** is that trait **Iterable** represents types that can be iterated over (i.e., collection types), whereas trait **Iterator** is the mechanism used to perform an iteration.



9.2.1 Iterator

The iterators provide the mechanism[123] to iterate over the members of a collection (once a time).



Although an **Iterable** can be iterated over multiple times, an **Iterator** <u>can be used just once</u>. Once you've iterated through a collection with an **Iterator**, you can't reuse it. If you need to iterate through the same collection again, you'll need to call **elements** on that collection to obtain a new **Iterator**.

An **Iterator** has many of the same methods as **Iterable**, including the higher-order ones. The many concrete methods provided by **Iterator** are implemented in terms of two abstract methods, **next** and **hasNext**.

9.2.2 Sequences

A sequence is a ordered collection, because the elements are ordered, you can ask for the first element, second element, and so on.

[122] These higher-order methods provide concise ways to iterate through collections for specific purposes, such as **map**, that transform each element and produce a new collection.
[123] Although most of the **Iterable** implementations you are likely to encounter will represent collections of a finite size, **Iterable** can also be used to represent infinite collections. The **Iterator** returned from an infinite collection could, for example, calculate and return the next digit of **π** each time its next method was invoked.

9.2.2.1 Lists

Scala **scala.List**[124], differs from Java's **java.util.List** type in that Scala Lists are always immutable. More generally, Scala's List is designed to enable a functional style of programming.

Creating and initializing a list:

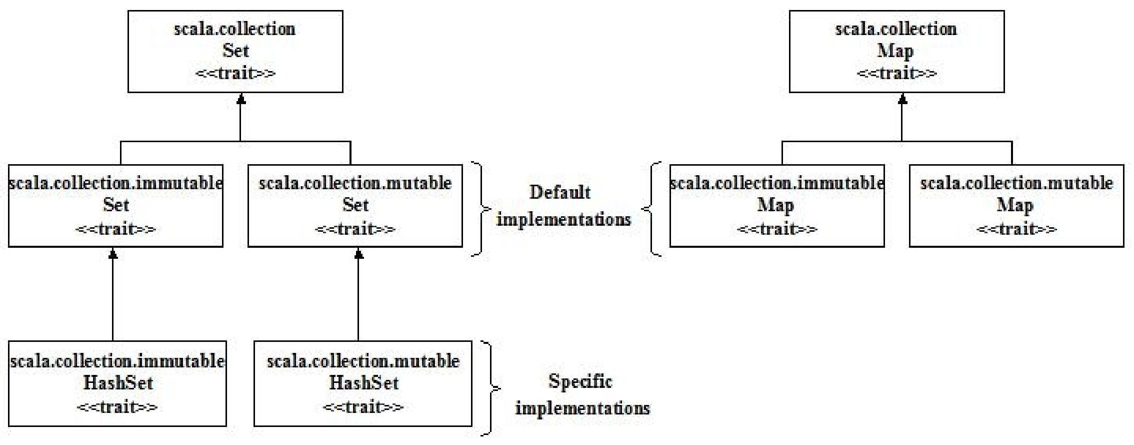
- **val li = List(1, 2, 3)**[125].
- **val li = 1:: 2 :: 3 :: Nil**[126].

A general Scala rule is:

- When a method is used in operator notation, such as **a * b**, the method is invoked on the left hand operator: **a.*(b)**
- If the method name ends in a colon (**:**), such as **1 :: li**, the method is invoked on the right operand: **li.::(1)**[127].

9.2.3 Sets and maps

Arrays are always mutable, whereas lists are always immutable. For sets[128] and maps[129], Scala provides mutable and immutable alternatives.


scala.collection Set <<trait>>
scala.collection Map <<trait>>
scala.collection.immutable Set <<trait>>
scala.collection.mutable Set <<trait>>
Default implementations
scala.collection.immutable Map <<trait>>
scala.collection.mutable Map <<trait>>
scala.collection.immutable HashSet <<trait>>
scala.collection.mutable HashSet <<trait>>
Specific implementations

For default, the sets and maps created are immutables:

- **var jetSet = Set("Boeing", "Airbus")**

[124] A Scala **List** is a sequence of objects that share the same type.
[125] Here Scala uses the factory **List.apply()** on the **scala.List** companion object.
[126] **Nil** is shorthand to specify an empty list. The reason **Nil** is needed at the end is that **::** is defined on class **List**. If you try to just say **1 :: 2 :: 3**, it won't compile because 3 is an **Int**, which doesn't have a **::** method.
[127] The operator **::** , which is named **cons**, prepends a new element to the beginning of an existing list, and return a new list (no list operator modifies the original list, because the lists are immutables).
[128] A collection that contains no duplicate elements.
[129] An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

    - It creates a immutable set with 2 elements[130]
- **jetSet += "Pluna"**
    - It creates and returns a new set with the new element added[131]
- **println(jetSet.contains("Cessna"))**
    - It prints true if the set contains "Cessna", false otherwise

In general the default set or map implementations are enough, but if is needed an explicit feature (an specific implementation), it's needed to import the necessary class:

- **import scala.collection.immutable.HashSet**
  val hashSet[132] = HashSet("Tomatoes", "Chilies")

When it's necessary use a mutable set or map, it's necessary to import it:

- **import scala.collection.mutable.Map**
  val treasureMap = Map[Int, String]()[133]
- **treasureMap += (1 ->"Go to island.")**[134]
    - It associates the key **1** with the value **"Go to the island."**.
    - **A general Scala rule is**: it is possible to invoke the method **->** on any object in a Scala program, and will return a two-element tuple containing the key and value:
        - Next, this tuple is passed to the += method of the map object to which treasureMap refers.
- **println(treasureMap(2))**
    - It prints the value associated with the key **2**.

9.3 Tuples

Another useful container object is the tuple. Like lists, tuples are immutable, but unlike lists, tuples can contain different types of elements.

Tuples are very useful, for example, if you need to return multiple objects from a method.

Creating and initializing a tuple:

- **val t = (1, 'a', "hello")**.[135]

Accesing to the tuple elements[136]:

[130] Shorthand for the method **apply** on a Set companion object.
[131] A mutable set will add the element to itself.
[132] A collection that contains no duplicate elements and offers constant time performance for the basic operations (add, remove, contains and size) because is based in a hash table.
[133] The explicit type parameterization, **[Int, String]**, is required because without any values passed to the factory method, the compiler is unable to infer the map's type parameters.
[134] Here is really invoking the method: **(1).->("Go to island.")**.
[135] Although conceptually you could create tuples of any length, currently the Scala library only defines them up to **Tuple22**.
[136] Why you can't access the elements of a tuple like the elements of a list, for example, with "t(0)"?. The reason is that a list's apply method always returns the same type, but each element of a tuple may be a

- **println(t._1); println(t._2); println(t._3)**[137]

The actual type of a tuple depends on the number of elements it contains and the types of those elements. Thus, the type of **(1, "Hello")** is **Tuple2[Int, String]**.

different type: _1 can have one result type, _2 another, and so on. These _N numbers are one-based, instead of zero-based, because starting with 1 is a tradition set by other languages with statically typed tuples, such as Haskell and ML.

[137] The semi-colon (**;**) is optional in Scala (because Scala can make semicolon inference), but when there are more than one statement at the same line it is obligatory.

10 Scripts

A script is just a sequence of statements in a file that will be executed sequentially.

Command line arguments to a Scala script are available via a Scala array named **args.**

In order to write a test scropt, create and open a file named *test.scala*, and write the following lines:

```
var i = 0
for (arg[138] <- args) {
        if (i != 0)
                print(" ")
        print(arg)
        i += 1
}
```

Save the file and execute in the command line: **scala**[139] ***test.scala***[140] **It is a test**[141].

A script must end in a result expression (unlike classes that end in a definition).

[138] Here the variable **arg** is a **val** variable (immutable), then can't be reassigned inside the body of the **for** expression. 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.
[139] The command **scala** is used to call the Scala interpreter.
[140] The name of the script (**test.scala**).
[141] **It is a test** are the command line arguments (they are 4) to the *test.scala* script, the arguments are available to the script through the Scala array **args**.

11 Scala applications

11.1 A Scala application

To run a Scala program, you must supply the name of a standalone singleton object with a main method that takes one parameter, an **Array[String]**, and has a result type of **Unit**. Any singleton object with a main method of the proper signature can be used as the entry point into an application:

```
import ChecksumAccumulator.calculate[142]
object Summer {
        def main(args: Array[String]) {
                for (arg <args)
                println(arg +": "+ calculate(arg))
        }
}
```

In order to execute an application are need the following two steps:

- Compile the classes and singletons:
  - Put the code of the class or singleton in a file[143]
  - Execute: **scalac**[144] **ChecksumAccumulator.scala**[145] **Summer.scala**[146]
- Execute the **scala** interpreter with the name of a standalone object containing a **main** method of the proper signature:
  - **scala Summer It is an argument**

11.2 The application trait

Scala provides a trait, **scala.Application**:

```
import ChecksumAccumulator.calculate
object FallWinterSpringSummer extends Application {
        for (season <- List("fall", "winter", "spring"))
```

[142] Import the member **calculate** method defined in the **ChecksumAccumulator** object.
[143] Unlike Java isn't necessary that the file has the same name.
[144] The Scala compiler is **scalac**. This compiles your source files to Java class files, but there may be a perceptible delay before the compilation finishes. The reason is that every time the compiler starts up, it spends time scanning the contents of **jar** files and doing other initial work before it even looks at the fresh source files you submit to it. For this reason, the Scala distribution also includes a Scala compiler daemon called **fsc** (for fast Scala compiler). You use it like this: **fsc ChecksumAccumulator.scala Summer.scala**. The first time you run **fsc**, it will create a local server daemon attached to a port on your computer. It will then send the list of files to compile to the daemon via the port, and the daemon will compile the files. The next time you run **fsc**, the daemon will already be running, so **fsc** will simply send the file list to the daemon, which will immediately compile the files. In order to stop the **fsc** daemon, you can do so with **fsc -shutdown**.
[145] The file with the code of the class **ChecksumAccumulator**.
[146] The file with the code of the singleton object **Summer**.

```
                    println(season +": "+ calculate(season))
      }
```

Then instead of writing a **main** method, you place the code you would have put in the **main** method directly between the curly braces of the singleton object. Next, you can compile and run this application just like any other.

Using the **Application** trait has some shortcomings:

- You can't use this trait if you need to access command-line arguments, because the **args** array isn't available.
- Because of some restrictions in the JVM threading model, you need an explicit main method if your program is multi-threaded.
- Some implementations of the JVM do not optimize the initialization code of an object which is executed by the **Application** trait.

You should use the **Application** trait only when your program is relatively simple and single-threaded.

12 Imperative style vs. Functional style

When you write in assembly language, you are telling the machine how to resolve the problem directing its steps one by one: load this register with this value, test the value, branch if some condition was met, and so on. Contrast this with writing formula functions in Excel. In Excel, we describe the way to solve some problem using the formula functions and cell addresses, and it's up to Excel to determine how the cells need to be recalculated and the order for the recalculation.
Directing the machine's steps is termed imperative coding. Imperative coding describes, as we saw earlier, the *how*. Writing functions describing the *what*, the goal to be achieved, and allowing the computer to figure out the *how*, is termed functional programming.

The imperative style[147], is the style you normally use with languages like Java, C++, and C, where you give one imperative command at a time, iterate with loops, and often mutate state shared between different functions[148].

Scala enables you to program imperatively or functionaly.The reason Scala encourages a functional style[149], is that the functional style can help you write more understandable (clearer and more concise), and less error-prone code. Another benefit of this approach is that it can help make your programs easier to test[150].

Example the imperative vs. functional style in order to filter the odds number from a list:

Java imperative style:

```
int[] x = {1,2,3,4,5};
ArrayList<Integer> res = new ArrayList<Integer>();
for (int v : x) {
        if (v % 2 == 1) res.add(new Integer(v));
}
```

Scala functional style:

```
def isOdd(x: Int) = x % 2 == 1
List(1,2,3,4,5).filter(isOdd)
```

There are two pieces of logic that we are concerned with: what operation is being performed and the formula for that logic. In the Java code, the logic gets lost in the boilerplate.

12.1 Prefer vals and functions without side effects[151]

[147] It emphasizes changes in state.
[148] A functional language disables mutable data.
[149] It treats computation as the evaluation of mathematical functions and avoids state.
and mutable data
[150] Because the function returns only values, you could test it simply by checking its result: **assert(function_result == value_desired)**.
[151] That said, bear in mind that neither vars nor side effects are inherently evil. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.

If code contains any **vars**, it is probably in an imperative style. If the code contains no **vars** at all—i.e., it contains only **vals**—it is probably in a functional style.

Imperative style:

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

Functional style:

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

You can go even further, though. The refactored printArgs method is not purely functional, because it has **side effects**[152]: printing to the standard output stream. A more functional approach would be to define a method that formats the passed **args** for printing, but just returns the formatted string:

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```
[153]

Now you're really functional: no **side effects** or **vars** in sight. Of course, this function doesn't actually print anything, but you can easily pass its result to **println** to accomplish that: **println(formatArgs(args))**.

A way to express methods with side effects is to leave off the result type and the equals sign, and enclose the body of the method in curly braces[154]. In this form, the method looks like a procedure, a method that is executed only for its side effects:

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

[152] The telltale sign of a function with **side effects** is that its result type is **Unit**. If a function isn't returning any interesting value, which is what a result type of **Unit** means, the only way that function can make a difference in the world is through some kind of side effect. A side effect is generally defined as mutating state somewhere external to the function or performing an I/O action.
[153] The **mkString** method, which you can call on any iterable collection, returns a string consisting of the result of calling **toString** on each element, separated by the passed string: for example the list **li** with the elements **1**,**2** and **3**, for the call **li.mkString('+')**, will return the string **"1+2+3"**.
[154] Take care, because if you use these notation, the function result type will be definitely **Unit**. If for example the last result of the function is a **String**, the **String** will be converted to **Unit** (because the Scala compiler can convert any type to **Unit**), and its value lost.

12.2 Prefer immutable objects (aka. functional objects[155])

One of the big ideas of the functional style of programming is that methods should not have side effects. Applying this functional philosophy to the world of objects means making objects immutable[156].

Immutable objects offer several advantages over mutable objects:

- Immutable objects are often easier to reason about than mutable ones, because they do not have complex state spaces that change over time.
- You can pass immutable objects around quite freely, whereas you may need to make defensive copies of mutable objects before passing them to other code.
- There is no way for two threads concurrently accessing an immutable to corrupt its state once it has been properly constructed, because no thread can change the state of an immutable.
- Immutable objects make safe hashtable keys. If a mutable object is mutated after it is placed into a HashSet, for example, that object may not be found the next time you look into the HashSet.

The main disadvantage of immutable objects is that:

- They sometimes require that a large object graph be copied where otherwise an update could be done in place[157].

12.3 Using the built-in control structures in a functional way

Try to use the result values that return the control structures to avoid using temporal variables:

Imperative style:

```
var filename = "default.txt"
if (!args.isEmpty)
        filename = args(0)
```

Functional style:

```
val filename =
        if (!args.isEmpty) args(0)
        else "default.txt"
```

Because the loops[158] results in no value, it is often left out of pure functional languages (which likely uses recursion). Such languages have expressions, not loops. In general,

[155] Functional objects = objects that do not have any mutable state.
[156] Scala's **List** is designed to enable a functional style of programming because are always immutable.
[157] In some cases this can be awkward to express and might also cause a performance bottleneck. As a result, it is not uncommon for libraries to provide mutable alternatives to immutable classes. For example, class **StringBuilder** is a mutable alternative to the immutable **String**.
[158] Scala includes the **while** loop nonetheless, because sometimes an imperative solution can be more readable, especially to programmers with a predominantly imperative background.

we recommend you challenge **while** loops in your code in the same way you challenge vars. If there isn't a good justification for a particular **while** or **do-while** loop, try to find a way to do the same thing without it:

Imperative style[159]:

```
def gcdLoop(x: Long, y: Long): Long = {
        var a = x
        var b = y
        while (a != 0) {
                val temp = a
                a = b % a
                b = temp
        }
        b
}
```

Functional style:

```
def gcd(x: Long, y: Long): Long =
        if (y == 0) x else gcd(y, x % y)
```

12.4 Using tail recursion

Here's an example of a function that approximates a value by repeatedly improving a guess until it is good enough:

Imperative style:

```
def approximateLoop(initialGuess: Double): Double = {
        var guess = initialGuess
        while (!isGoodEnough(guess))
                guess = improve(guess)
        guess
}
```

Functional style:

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

Which of the two versions of approximate is preferable? In terms of brevity and **var** avoidance, the functional version wins. But is the imperative approach perhaps more efficient?: in the general case yes, but if the functional version implements tail recursion[160] they have almost exactly the same efficiency, because the Scala compiler is able to apply optimization techniques.

[159] Calculating greatest common divisor.
[160] Functions like **approximate**, which call themselves as their last action, are called tail recursive.

Recursion without tail recursion

```
def  incAll(xs: List[Int]): List[Int] = xs match {
        case List() => List()
        case x :: xs1 => x + 1 :: incAll(xs1)
}
```

The function **incAll** is not tail recursive. Note that the recursive call to **incAll** occurs inside of the **::** operation. On today's virtual machines this means that you cannot apply **incAll** to lists of much more than about 30,000 to 50,000 elements.

In order to write a version of **incAll** that can work on lists of arbitrary size (as much as heap-capacity allows), it is necessary to use the imperative style (with a mutable structure and a **for** instead of recursion):

```
import scala.collection.mutable.ListBuffer

def  incAll(xs: List[Int]): List[Int] = {
        val buf = new ListBuffer[Int]
        for (x <-xs) buf += x + 1
        buf.toList
}
```

Limits of tail recursion

The use of tail recursion in Scala is fairly limited, because the JVM instruction set makes implementing more advanced forms of tail recursion very difficult. Scala only optimizes directly recursive calls back to the same function making the call[161].

12.5 Using Function literals when is possible

Refactoring the followin code to convert it to functional style:

Imperative style:

```
var maxWidth = 0
for (line <- lines)
        maxWidth = maxWidth.max(widthOfLength(line))
```

Functional style:

```
val longestLine = lines[162].reduceLeft[163]((a, b) => if (a.length > b.length) a else b)
val maxWidth = widthOfLength(longestLine)
```

[161] As in the example: if the function is **approximate**, the recursive call must be to **approximate**.

[162] The variable **lines** is a **List[String]**.

[163] The **reduceLeft** method applies the passed function to the first two elements in **lines**, then applies it to the result of the first application and the next element in lines, and so on.

Many Scala libraries give you opportunities to use function literals. For example, a **foreach** method is available for all collections. It takes a function as an argument and invokes that function on each of its elements. Here is how it can be used to print out all of the elements of a list:

```
val someNumbers = List(11,10,5,0, 5, 10)
someNumbers.foreach((x: Int) => println(x))
```

As another example, collection types also have a **filter** method. This method selects those elements of a collection that pass a test the user supplies. That test is supplied using a function:

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

This function maps positive integers to true and all others to false.

12.6 Using high-order functions to simplify code

12.6.1 Reducing code duplication

Using higher-order functions[164] in order to condense and simplify code:

```
object FileMatcher {
    private def filesHere = (new java.io.File(".")).listFiles
    private def filesMatching(matcher: String => Boolean) =
        for (file <-filesHere; if matcher(file.getName))
            yield file

    def filesEnding(query: String) =
        filesMatching(_.endsWith(query))

    def filesContaining(query: String) =
        filesMatching(_.contains(query))

    def filesRegex(query: String) =
        filesMatching(_.matches(query))
}
```

The function **filesMatching** is a high-order function, because has the function **matcher** as a parameter. The expressions **_.endsWith(query)**, **_.contains(query)** and **_.matches(query)** are *function literals*, which at runtime becomes *closures* because they have a *bound variable*[165] and a *free variable*[166]:

[164] Functions that take functions as parameters.
[165] The function literal represents the function **matches** which is passed to the function **filesMatching**. This function **matches** is invoked inside of **filesMatching** with the **String** parameter **file.getName**, this parameter is the bound variable of the function literal (represented with the underscore).
[166] The parameter **query**.

- The use of a high-order function is helping to reduce code in the previous API implementation[167], because otherwise it would have been necessary to repeat the **filesMatching** code, once for each **file.getName**'s method[168] to use.

12.6.2 Simplifying client code

Another important use of higher-order functions is to put them in an API itself to make client code more concise.

As an example the **exists** method that determines whether a passed value is contained in a **Collection**. If **exists** didn't exist, and you wanted to write a **containsOdd** method, to test whether a list contains odd numbers, you might write it like this:

```
    def containsOdd(nums: List[Int]): Boolean = {
        var exists = false
        for (num <- nums)
            if (num % 2 == 1)
                exists = true
        exists
    }
```

Using **exists**, you could write this instead:

```
    def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

Using **exists** through Scala's **Collections** API, the client code is reduced (simplified).

12.7 Using lists as the main data structure

List are not always performant!!

[167] Reduces code duplication in the implementation of the object **FileMatcher**.
[168] The methods **endsWith**, **contains** and **matches**.

13 Miscellaneous

1. Comments, there are two type of comments:
   - One line comment: //
   - One or more line comment: /* */

2. Semicolon inference, in a Scala program, a semicolon at the end of a statement is usually optional
   - You can type one if you want but you don't have to if the statement appears by itself on a single line.
   - A semicolon is required if you write multiple statements on a single line.
   - Rules for semicolon inference, line ending is treated as a semicolon unless one of the following conditions is true:
     - The line in question ends in a word that would not be legal as the end of a statement, such as a period or an infix operator.
     - The next line begins with a word that cannot start a statement.
     - The line ends while inside parentheses (...) or brackets [...], because these cannot contain multiple statements anyway.

3. Implementing a cache, in general, you would likely implment a cache only if you encountered a performance problem that the cache solves, and might use a weak map, such as **WeakHashMap** in **scala.collection.jcl**, so that entries in the cache could be garbage collected if memory becomes scarce.

4. Scala implicit imports, are implicitly imported members of packages **java.lang** and **scala**, as well as the members of a singleton object named **Predef**[169], into every Scala source file.

5. Scala file classes names, one difference between Scala and Java is that whereas Java requires you to put a public class in a file named equal to the class[170], in Scala, you can name **.scala** files anything you want, no matter what Scala classes or code you put in them. In general in the case of non-scripts, however, it is recommended style to name files equal to the classes they contain as is done in Java.

6. Leave off empty parentheses in methods calls, in Scala, you can leave off empty parentheses on method calls. The convention is that you include parentheses if the method has side effects, such as println(), but you can leave them off if the method has no side effects.

7. Implementing the loan pattern technique, with control-abstraction functions:

[169] **Predef**, which resides in package **scala**, contains many useful methods. For example, when you say **println** in a Scala source file, you're actually invoking println **Predef.println**. When you say **assert**, you're invoking **Predef.assert**.

[170] For example, you'd put class **SpeedRacer** in file **SpeedRacer.java**.

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) {
        val writer = new PrintWriter(file)
        try {
                op(writer)
        } finally {
                writer.close()
        }
}
```

The control abstraction implemented with the function **withPrintWriter**, loans to the function **op** the resource (in the example a **File**). When the function **op** completes, it signals that it no longer needs the "borrowed" resource. The resource is then closed in a **finally** block, to ensure it is indeed closed, regardless of whether the function completes by returning normally or throwing an exception.

8. **Unit** value, when something return a **Unit**, it returns that is called the *unit value* and is written **()**:
    - If for example the procedure **green**[171] is defined as: **def greet() { println("hi") }**, the comparision **greet() == ()** will return **true**.

9. Converting the Iterator returned by the Source object to a list:
    - Example: **val lines = Source.fromFile(args(0)).getLines.toList**

10. Eliminating leading spaces in a raw string, put a pipe character (|) at the front of each line, and then call **stripMargin** on the whole string:
    - Example:
    ```
    println("""|Welcome to Ultamix 3000.
               |Type "HELP" for help.""".stripMargin)
    ```

[171] Because no equals sign precedes its body, **greet** is defined to be a procedure with a result type of **Unit**.

14 Internet Resources

Online Scala book:
http://programming-scala.labs.oreilly.com

Function vs. Methods:
http://jim-mcbeath.blogspot.com/2009/05/scala-functions-vs-methods.html

Parsers Combinators:
http://www.codecommit.com/blog/

Traits vs. Aspects:
http://blog.objectmentor.com/articles/2008/09/27/traits-vs-aspects-in-scala

Lift AMQP with RabbitMQ and Scala - Tutorial and Screencast:
http://blog.getintheloop.eu/tags/lift

Why it's important concurrency (and new paradigms as Erlang/Scala with actors):
http://www.gotw.ca/publications/concurrency-ddj.htm

Scala actors, Scala OTP and Active Objects:
http://jonasboner.com/2008/12/11/real-world-scala-fault-tolerant-concurrent-asynchronous-components.html

Function memoization
http://michid.wordpress.com/
Monads:
http://debasishg.blogspot.com/2008/03/monads-another-way-to-abstract.html
http://james-iry.blogspot.com/2007/09/monads-are-elephants-part-1.html

Continuations (for non-blocking operations):
http://www.scala-lang.org/node/2096

Applicative functors (the most convenient model to deal with parsers)
http://lucdup.blogspot.com/2008/01/tmp_2968.html
http://dibblego.wordpress.com/2008/06/20/applicative-functors-in-scala/

OSGi:
http://scala-blogs.org/search/label/lift

Qi4j (Composite Oriented Programming):
http://www.qi4j.org/160.html

Lift (una implemetación the un Twitter like: ESME):
http://incubator.apache.org/esme/

15 To analyze

Using currying:

Currying allows you to use a method that takes multiple arguments lists

The advantage of currying is that if you call a function with less parameters, you get a partially applied function.

Using import:

import can be used inside any code block, and the **import** will be active only in the scope of that code block.

Better function 's explanation:

http://creativekarma.com/ee.php/weblog/comments/scala_function_objects_from_a_java_perspective/

http://www.nabble.com/val-vs.-def-td23628720.html

http://jim-mcbeath.blogspot.com/2009/05/scala-functions-vs-methods.html

Class parameters:

Which are the differences between class parameters and class fields?

- Although class parameters n and d are in scope in the code of your add method, you can only access their value on the object on which add was invoked. Thus, when you say n or d in add's implementation, the compiler is happy to provide you with the values for these class parameters. But it won't let you say that.n or that.d, because that does not refer to the Rational object on which add was invoked
- A parametric field is a field defined as a class parameter.

A way to implement **exec**:

```
def exec(cmd: String)(func: (String)=>Unit): Unit = {
        val rt = Runtime.getRuntime
        val p = rt.exec(cmd)
        val src =scala.io.Source.fromInputStream(p.getInputStream)
        src.getLines.foreach(func)
}
```