Glossary

Terms

GLOSSARY FROM THE DEFINITIVE BOOK ON SCALA, PROGRAMMING IN SCALA.

Look up a term

algebraic data type

A type defined by providing several alternatives, each of which comes with its own constr decompose the type through pattern matching. The concept is found in specification langu languages. Algebraic data types can be emulated in Scala with case classes.

alternative

A branch of a match expression. It has the form “`case` *pattern* => *expression*.” Another n

annotation

An annotation appears in source code and is attached to some part of the syntax. Annotat you can use them to effectively add an extension to Scala.

anonymous class

An anonymous class is a synthetic subclass generated by the Scala compiler from a new name is followed by curly braces. The curly braces contains the body of the anonymous s However, if the name following new refers to a trait or class that contains abstract membe inside the curly braces that define the body of the anonymous subclass.

anonymous function

Another name for function literal.

apply

You can apply a method, function, or closure to arguments, which means you invoke it on

argument

When a function is invoked, an argument is passed for each parameter of that function. Tl refers to the argument. The argument is the object passed at invocation time. In addition, line) arguments that show up in the `Array[String]` passed to main methods of singletor

assign

You can assign an object to a variable. Afterwards, the variable will refer to the object.

auxiliary constructor

Extra constructors defined inside the curly braces of the class definition, which look like n with no result type.

block

One or more expressions and declarations surrounded by curly braces. When the block ev declarations are processed in order, and then the block returns the value of the last expres commonly used as the bodies of functions, for expressions, `while` loops, and any other number of statements together. More formally, a block is an encapsulation construct for w and a result value. The curly braces in which you define a class or object do not, therefore methods (which are defined inside those curly braces) are visible from the out- side. Such

bound variable

A bound variable of an expression is a variable that’s both used and defined inside the exp
function literal expression `(x: Int) => (x, y)`, both variables `x` and `y` are used, but o
defined in the expression as an `Int` and the sole argument to the function described by t

by-name parameter

A parameter that is marked with a `=>` in front of the parameter type, e.g., `(x: => Int)`.
name parameter is evaluated not before the method is invoked, but each time the paramet
method. If a parameter is not by-name, it is by-value.

by-value parameter

A parameter that is not marked with a `=>` in front of the parameter type, e.g., `(x: Int)`.
value parameter is evaluated before the method is invoked. By-value parameters contrast

class

Defined with the `class` keyword, a *class* may either be abstract or concrete, and may be
when instantiated. In `new Array[String](2)`, the class being instantiated is `Array` and
`Array[String]`. A class that takes type parameters is called a *type constructor*. A type
as in: the class of type `Array[String]` is `Array`.

closure

A function object that captures free variables, and is said to be “closed” over the variables

companion class

A class that shares the same name with a singleton object defined in the same source file
companion class.

companion object

A singleton object that shares the same name with a class defined in the same source file
have access to each other’s private members. In addition, any implicit conversions define
scope anywhere the class is used.

contravariant

A *contravariant* annotation can be applied to a type parameter of a class or trait by putting
parameter. The class or trait then subtypes contravariantly with—in the opposite direction
For example, `Function1` is contravariant in its first type parameter, and so `Function1[`
`Function1[String, Any]`.

covariant

A *covariant* annotation can be applied to a type parameter of a class or trait by putting a p
parameter. The class or trait then subtypes covariantly with—in the same direction as—th
example, `List` is covariant in its type parameter, so `List[String]` is a subtype of `Lis`

currying

A way to write functions with multiple parameter lists. For instance `def f(x: Int)(y: I`
parameter lists. A curried function is applied by passing several arguments lists, as in: `f`
to write a *partial application* of a curried function, such as `f(3)`.

declare

You can *declare* an abstract field, method, or type, which gives an entity a name but not a
difference between declarations and definitions is that definitions establish an implementa
declarations do not.

define

To *define* something in a Scala program is to give it a name and an implementation. You c
objects, fields, methods, local functions, local variables, *etc*. Because definitions always i
abstract members are declared not defined.

static type
structural type
subclass
subtrait
subtype
superclass
supertrait
supertype
synthetic class
tail recursive
target typing
template
trait
type
type constraint
type constructor
type parameter
type signature
uniform access principle
unreachable
unreferenced
value
value type

direct subclass

A class is a *direct subclass* of its direct superclass.

direct superclass

The class from which a class or trait is immediately derived, the nearest class above it in
`Parent` is mentioned in a class `Child`'s optional extends clause, then `Parent` is the di
mentioned in `Child`'s extends clause, the trait's direct superclass is the `Child`'s direct s
clause, then `AnyRef` is the direct superclass of `Child`. If a class's direct superclass tak
class `Child` extends `Parent[String]`, the direct superclass of `Child` is still `Parent`,
hand, `Parent[String]` would be the direct supertype of `Child`. See supertype for more
between class and type.

equality

When used without qualification, *equality* is the relation between values expressed by `==`

existential type

An existential type includes references to type variables that are unknown. For example,
an existential type. It is an array of `T`, where `T` is some completely unknown type. All th
exists at all. This assumption is weak, but it means at least that an `Array[T] forSome`
not a banana.

expression

Any bit of Scala code that yields a result. You can also say that an expression *evaluates*

filter

Documentation API Learn Quickref Contribute SIPs/SLIPs Search in documentati

filter expression

A *filter expression* is the boolean expression following an `if` in a for expression. In `for(`
`0)` ,the filter expression is "`i % 2 == 0`".

first-class function

Scala supports *first-class functions*, which means you can express functions in function li
`1`, and that functions can be represented by objects, which are called function values.

for comprehension

A *for comprehension* is a type of for expression that creates a new collection. For each ite
the yield clause defines an element of the new collection. For example, `for (i <- (0 u`
`yield (i, j)` returns the collection `Vector((0,2), (0,3), (1,2), (1,3))`.

for expression

A *for expression* is either a for loop, which iterates over one or more collections, or a for c
collection from the elements of one or more collections. A `for` expression is built up of g
and (in the case of for comprehensions) a yield clause.

for loop

A *for loop* is a type of for expression that loops over one or more collections. Since `for` l
produce side-effects. For example, `for (i <- 0 until 100) println(i)` prints the nu

free variable

A *free variable* of an expression is a variable that's used inside the expression but not def
instance, in the function literal expression `(x: Int) => (x, y)`, both variables `x` and `y`
variable, because it is not defined inside the expression.

function

A *function* can be invoked with a list of arguments to produce a result. A function has a pa
type. Functions that are members of a class, trait, or singleton object are called methods.

functions are called local functions. Functions with the result type of `Unit` are called proc
source code are called function literals. At run time, function literals are instantiated into o

function literal

A function with no name in Scala source code, specified with function literal syntax. For e
`y`.

function value

A function object that can be invoked just like any other function. A function value's class
traits (e.g., `Function0`, `Function1`) from package `scala`, and is usually expressed in s
syntax. A function value is "invoked" when its apply method is called. A function value tha
closure.

functional style

The *functional style* of programming emphasizes functions and evaluation results and dee
operations occur. The style is characterized by passing function values into looping metho
no side effects. It is the dominant paradigm of languages such as Haskell and Erlang, anc

generator

A generator defines a named val and assigns to it a series of values in a for expression. F
`10)`, the generator is "`i <- 1 to 10`". The value to the right of the `<-` is the generator e

generator expression

A generator expression generates a series of values in a for expression. For example, in
expression is "`1 to 10`".

A class that takes type parameters. For example, because `scala.List` takes a type par
class.

generic trait

A trait that takes type parameters. For example, because trait `scala.collection.Set` ta
trait.

guard

See filter.

helper function

A function whose purpose is to provide a service to one or more other functions nearby. H
implemented as local functions.

helper method

A helper function that's a member of a class. Helper methods are often private.

immutable

An object is *immutable* if its value cannot be changed after it is created in any way visible
be immutable.

imperative style

The *imperative style* of programming emphasizes careful sequencing of operations so tha
order. The style is characterized by iteration with loops, mutating data in place, and metho
dominant paradigm of languages such as C, C++, C# and Java, and contrasts with the fur

initialize

When a variable is defined in Scala source code, you must initialize it with an object.

instance

An *instance*, or class instance, is an object, a concept that exists only at run time.

instantiate

To *instantiate* a class is to make a new object from the class, an action that happens only

invariant

Invariant is used in two ways. It can mean a property that always holds true when a data s it is an invariant of a sorted binary tree that each node is ordered before its right subnode, also sometimes used as a synonym for nonvariant: “class `Array` is invariant in its type p

invoke

You can *invoke* a method, function, or closure *on* arguments, meaning its body will be exe

JVM

The *JVM* is the Java Virtual Machine, or runtime, that hosts a running Scala program.

literal

`1`, `"One"`, and `(x: Int) => x + 1` are examples of *literals*. A literal is a shorthand way shorthand exactly mirrors the structure of the created object.

local function

A *local function* is a `def` defined inside a block. To contrast, a `def` defined as a member is called a method.

local variable

A *local variable* is a `val` or `var` defined inside a block. Although similar to local variables

A *member* is any named element of the template of a class, trait, or singleton object. A m name of its owner, a dot, and its simple name. For example, top-level fields and methods that class. A trait defined inside a class is a member of its enclosing class. A type define a member of that class. A class is a member of the package in which is it defined. By con function is not a member of its surrounding block.

message

Actors communicate with each other by sending each other *messages*. Sending a messa receiver is doing. The receiver can wait until it has finished its current activity and its inva

meta-programming

Meta-programming software is software whose input is itself software. Compilers are meta `scaladoc`. Meta-programming software is required in order to do anything with an annotat

method

A *method* is a function that is a member of some class, trait, or singleton object.

mixin

Mixin is what a trait is called when it is being used in a mixin composition. In other words, but in “`new Cat extends AnyRef with Hat`,” `Hat` can be called a mixin. When used as example, you can *mix* traits _in_to classes or other traits.

mixin composition

The process of mixing traits into classes or other traits. *Mixin composition* differs from tra the type of the super reference is not known at the point the trait is defined, but rather is d mixed into a class or other trait.

modifier

A keyword that qualifies a class, trait, field, or method definition in some way. For exampl that a class, trait, field, or method being defined is private.

multiple definitions

The same expression can be assigned in *multiple definitions* if you use the syntax `val v`

nonvariant

A type parameter of a class or trait is by default *nonvariant*. The class or trait then does n changes. For example, because class `Array` is nonvariant in its type parameter, `Array[` supertype of `Array[Any]`.

operation

In Scala, every *operation* is a method call. Methods may be invoked in *operator notation*, : notation, `+` is an *operator*.

parameter

Functions may take zero to many *parameters*. Each parameter has a name and a type. Tl and arguments is that arguments refer to the actual objects passed when a function is inv that refer to those passed arguments.

parameterless function

A function that takes no parameters, which is de- fined without any empty parentheses. Ir may not supply parentheses. This supports the uniform access principle, which enables tl without requiring a change to client code.

parameterless method

A *parameterless method* is a parameterless function that is a member of a class, trait, or

partially applied function

A function that's used in an expression and that misses some of its arguments. For instar `Int => Int`, then `f` and `f(1)` are *partially applied functions*.

path-dependent type

A type like `swiss.cow.Food`. The `swiss.cow` part is a path that forms a reference to an sensitive to the path you use to access it. The types `swiss.cow.Food` and `fish.Food`,

pattern

In a `match` expression alternative, a *pattern* follows each `case` keyword and precedes ei symbol.

pattern guard

In a `match` expression alternative, a *pattern guard* can follow a pattern. For example, in " the pattern guard is "`if x % 2 == 0`". A case with a pattern guard will only be selected if guard yields true.

predicate

A *predicate* is a function with a `Boolean` result type.

primary constructor

The main constructor of a class, which invokes a superclass constructor, if necessary, ini executes any top-level code defined between the curly braces of the class. Fields are initi passed to the superclass constructor, except for any that are not used in the body of the c away.

procedure

A *procedure* is a function with result type of `Unit`, which is therefore executed solely for i

reassignable

A variable may or may not be *reassignable*. A `var` is reassignable while a `val` is not.

recursive

A function is *recursive* if it calls itself. If the only place the function calls itself is the last e
function is tail recursive.

reference

A *reference* is the Java abstraction of a pointer, which uniquely identifies an object that re
type variables hold references to objects, because reference types (instances of `AnyRef`)
that reside on the JVM's heap. Value type variables, by contrast, may sometimes hold a r
and sometimes not (when the object is being represented as a primitive value). Speaking g
an object. The term "refers" is more abstract than "holds a reference." If a variable of type
as a primitive Java `int` value, then that variable still refers to the `Int` object, but no refe

reference equality

Reference equality means that two references identify the very same Java object. Referer
reference types only, by calling `eq` in `AnyRef`. (In Java programs, reference equality can
reference types.)

reference type

A *reference type* is a subclass of `AnyRef`. Instances of reference types always reside on

referential transparency

A property of functions that are independent of temporal context and have no side effects.
of a referentially transparent function can be replaced by its result without changing the pr

A variable in a running Scala program always *refers* to some object. Even if that variable i
refers to the `Null` object. At runtime, an object may be implemented by a Java object or
allows programmers to think at a higher level of abstraction about their code as they imagi

refinement type

A type formed by supplying a base type a number of members inside curly braces. The m
types that are present in the base type. For example, the type of "animal that eats grass"
`Grass }`.

result

An expression in a Scala program yields a *result*. The result of every expression in Scala

result type

A method's *result type* is the type of the value that results from calling the method. (In Ja
type.)

return

A function in a Scala program `returns` a value. You can call this value the result of the f
function *results in* the value. The result of every function in Scala is an object.

runtime

The Java Virtual Machine, or JVM, that hosts a running Scala program. Runtime encompa
defined by the Java Virtual Machine Specification, and the runtime libraries of the Java AP
phrase at run time (with a space between run and time) means when the program is runnin

runtime type

The type of an object at run time. To contrast, a static type is the type of an expression at
are simply bare classes with no type parameters. For example, the runtime type of `"Hi"`
`(x: Int) => x + 1` is `Function1`. Runtime types can be tested with `isInstanceOf`.

script

A file containing top level definitions and statements, which can be run directly with `scal` script must end in an expression, not a definition.

selector

The value being matched on in a `match` expression. For example, in “`s match { case`

self type

A *self type* of a trait is the assumed type of `this`, the receiver, to be used within the trait the trait must ensure that its type conforms to the trait’s self type. The most common use class into several traits (as described in Chapter 29 of Programming in Scala.

semi-structured data

XML data is semi-structured. It is more structured than a flat binary file or text file, but it c programming language’s data structures.

serialization

You can *serialize* an object into a byte stream which can then be saved to files or transmit *deserialize* the byte stream, even on different computer, and obtain an object that is the sa

shadow

A new declaration of a local variable *shadows* one of the same name in an enclosing scop

signature

Signature is short for type signature.

shares its name with a class, and is defined in the same source file as that class, is that is its companion class. A singleton object that doesn’t have a companion class is a stand

standalone object

A singleton object that has no companion class.

statement

An expression, definition, or import, *i.e.*, things that can go into a template or a block in S

static type

See type.

structural type

A refinement type where the refinements are for members not in the base type. For examp structural type, because the base type is `AnyRef`, and `AnyRef` does not have a member

subclass

A class is a *subclass* of all of its superclasses and supertraits.

subtrait

A trait is a *subtrait* of all of its supertraits.

subtype

The Scala compiler will allow any of a type’s *subtypes* to be used as a substitute whereve and traits that take no type parameters, the subtype relationship mirrors the subclass relat is a subclass of abstract class `Animal`, and neither takes type parameters, type `Cat` is if trait `Apple` is a subtrait of trait `Fruit`, and neither takes type parameters, type `Apple` classes and traits that take type parameters, however, variance comes into play. For exar is declared to be covariant in its lone type parameter (i.e., `List` is declared `List[+A]`), `List[Animal]`, and `List[Apple]` a subtype of `List[Fruit]`. These subtype relationsh each of these types is `List`. By contrast, because `Set` is not declared to be covariant ir declared `Set[A]` with no plus sign), `Set[Cat]` is not a subtype of `Set[Animal]`. A subt contracts of its supertypes, so that the Liskov Substitution Principle applies, but the comp level of type checking.

superclass

A class's *superclasses* include its direct superclass, its direct superclass's direct supercla `Any`.

supertrait

A class's or trait's *supertraits*, if any, include all traits directly mixed into the class or trait supertraits of those traits.

supertype

A type is a *supertype* of all of its subtypes.

synthetic class

A synthetic class is generated automatically by the compiler rather than being written by h

tail recursive

A function is *tail recursive* if the only place the function calls itself is the last operation of

target typing

Target typing is a form of type inference that takes into account the type that's expected. for example, the Scala compiler infers type of `x` to be the element type of `nums`, becaus function on each element of `nums`.

A *template* is the body of a class, trait, or singleton object definition. It defines the type si the class, trait, or object.

trait

A *trait*, which is defined with the `trait` keyword, is like an abstract class that cannot tak "mixed into" classes or other traits via the process known as mixin composition. When a t trait, it is called a mixin. A trait may be parameterized with one or more types. When para constructs a type. For example, `Set` is a trait that takes a single type parameter, wherea said to be "the trait of" type `Set[Int]`.

type

Every variable and expression in a Scala program has a *type* that is known at compile tim values to which a variable can refer, or an expression can produce, at run time. A variable referred to as a *static type* if necessary to differentiate it from an object's runtime type. In static type. Type is distinct from class because a class that takes type parameters can c `List` is a class, but not a type. `List[T]` is a type with a free type parameter. `List[Int` types (called ground types because they have no free type parameters). A type can have class of type `List[Int]` is `List`. The trait of type `Set[String]` is `Set`.

type constraint

Some annotations are *type constraints*, meaning that they add additional limits, or constra includes. For example, `@positive` could be a type constraint on the type `Int`, limiting th those that are positive. Type constraints are not checked by the standard Scala compiler, extra tool or by a compiler plugin.

type constructor

A class or trait that takes type parameters.

type parameter

A parameter to a generic class or generic method that must be filled in by a type. For exa "`class List[T] { . . .` ", and method `identity`, a member of object `Predef`, is de `x`”. The `T` in both cases is a type parameter.

type signature

A method’s *type signature* comprises its name, the number, order, and types of its parame
type signature of a class, trait, or singleton object comprises its name, the type signatures
constructors, and its declared inheritance and mixin relations.

uniform access principle

The *uniform access principle* states that variables and parameterless functions should be
Scala supports this principle by not allowing parentheses to be placed at call sites of para
parameterless function definition can be changed to a `val`, or *vice versa*, without affectin

unreachable

At the Scala level, objects can become *unreachable*, at which point the memory they occu
runtime. Unreachable does not necessarily mean unreferenced. Reference types (instance
objects that reside on the JVM’s heap. When an instance of a reference type becomes un
unreferenced, and is available for garbage collection. Value types (instances of `AnyVal`) a
type values and as instances of Java wrapper types (such as `java.lang.Integer`), whic
instances can be boxed (converted from a primitive value to a wrapper object) and unboxe
to a primitive value) throughout the lifetime of the variables that refer to them. If a value ty
a wrapper object on the JVM’s heap becomes unreachable, it indeed becomes unreferenc
collection. But if a value type currently represented as a primitive value becomes unreach
unreferenced, because it does not exist as an object on the JVM’s heap at that point in tir
memory occupied by unreachable objects, but if an Int, for example, is implemented at ru
occupies some memory in the stack frame of an executing method, then the memory for t

unreferenced

See unreachable.

value

The result of any computation or expression in Scala is a *value*, and in Scala, every value
essentially means the image of an object in memory (on the JVM’s heap or stack).

value type

A *value type* is any subclass of `AnyVal`, such as `Int`, `Double`, or `Unit`. This term has
source code. At runtime, instances of value types that correspond to Java primitive types
primitive type values or instances of wrapper types, such as `java.lang.Integer`. Over t
the runtime may transform it back and forth be- tween primitive and wrapper types (*i.e.*, to

variable

A named entity that refers to an object. A variable is either a `val` or a `var`. Both `val`s a
defined, but only `var`s can be later reassigned to refer to a different object.

variance

A type parameter of a class or trait can be marked with a *variance* annotation, either cova
variance annotations indicate how subtyping works for a generic class or trait. For exampl
covariant in its type parameter, and thus `List[String]` is a subtype of `List[Any]`. By
annotation, type parameters are nonvariant.

yield

An expression can *yield* a result. The `yield` keyword designates the result of a for comp

Current
Nightly

Guides & Overviews
Tutorials
Scala Style Guide

Glossary
Cheatsheets

Source Code
Contributors Guide
Suggestions

Documentation API Learn Quickref Contribute SIPs/SLIPs Search in documentati