**Syntactic Style**

**Naming Convention**

We mostly follow Java's and Scala's standard naming conventions.

* Classes, traits, objects should follow Java class convention, i.e. PascalCase style.
* class ClusterManager

trait Expression

* Packages should follow Java package naming conventions, i.e. all-lowercase ASCII letters.

package com.databricks.resourcemanager

* Methods/functions should be named in camelCase style.
* Constants should be all uppercase letters and be put in a companion object.
* object Configuration {
* val DEFAULT\_PORT = 10000

}

* Enums should be PascalCase.
* Annotations should also follow Java convention, i.e. PascalCase. Note that this differs from Scala's official guide.

final class MyAnnotation extends StaticAnnotation

**Variable Naming Convention**

* Variables should be named in camelCase style, and should have self-evident names.
* val serverPort = 1000

val clientPort = 2000

* It is OK to use one-character variable names in small, localized scope. For example, "i" is commonly used as the loop index for a small loop body (e.g. 10 lines of code). However, do NOT use "l" (as in Larry) as the identifier, because it is very difficult to differentiate "l" from "1", "|", "I".

**Line Length**

* Limit lines to 100 characters.
* The only exceptions are import statements and URLs (although even for those, try to keep them under 100 chars).

**Rule of 30**

"If an element consists of more than 30 subelements, it is highly probable that there is a serious problem" - [Refactoring in Large Software Projects](http://www.amazon.com/Refactoring-Large-Software-Projects-Restructurings/dp/0470858923).

In general:

* A method should contain less than 30 lines of code.
* A class should contain less than 30 methods.

**Spacing and Indentation**

* Use 2-space indentation in general.
* if (true) {
* println("Wow!")

}

* For method declarations, use 4 space indentation for its parameters when they don't fit in a single line. Return types can be either on the same line as the last parameter, or put to next line with 2 space indent.
* def newAPIHadoopFile[K, V, F <: NewInputFormat[K, V]](
* path: String,
* fClass: Class[F],
* kClass: Class[K],
* vClass: Class[V],
* conf: Configuration = hadoopConfiguration): RDD[(K, V)] = {
* // method body
* }
* def newAPIHadoopFile[K, V, F <: NewInputFormat[K, V]](
* path: String,
* fClass: Class[F],
* kClass: Class[K],
* vClass: Class[V],
* conf: Configuration = hadoopConfiguration)
* : RDD[(K, V)] = {
* // method body

}

* For classes whose header doesn't fit in a single line, put the extend on the next line with 2 space indent, and add a blank line after class header.
* class Foo(
* val param1: String, // 4 space indent for parameters
* val param2: String,
* val param3: Array[Byte])
* extends FooInterface // 2 space indent here
* with Logging {
* def firstMethod(): Unit = { ... } // blank line above

}

* Do NOT use vertical alignment. They draw attention to the wrong parts of the code and make the aligned code harder to change in the future.
* // Don't align vertically
* val plus = "+"
* val minus = "-"
* val multiply = "\*"
* // Do the following
* val plus = "+"
* val minus = "-"

val multiply = "\*"

**Blank Lines (Vertical Whitespace)**

* A single blank line appears:
  + Between consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
    - Exception: A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create logical groupings of fields.
  + Within method bodies, as needed to create logical groupings of statements.
  + Optionally before the first member or after the last member of the class (neither encouraged nor discouraged).
* Use one or two blank line(s) to separate class definitions.
* Excessive number of blank lines is discouraged.

**Parentheses**

* Methods should be declared with parentheses, unless they are accessors that have no side-effect (state mutation, I/O operations are considered side-effects).
* class Job {
* // Wrong: killJob changes state. Should have ().
* def killJob: Unit
* // Correct:
* def killJob(): Unit

}

* Callsite should follow method declaration, i.e. if a method is declared with parentheses, call with parentheses. Note that this is not just syntactic. It can affect correctness when apply is defined in the return object.
* class Foo {
* def apply(args: String\*): Int
* }
* class Bar {
* def foo: Foo
* }
* new Bar().foo // This returns a Foo

new Bar().foo() // This returns an Int!

**Curly Braces**

Put curly braces even around one-line conditional or loop statements. The only exception is if you are using if/else as an one-line ternary operator that is also side-effect free.

// Correct:

if (true) {

println("Wow!")

}

// Correct:

if (true) statement1 else statement2

// Correct:

try {

foo()

} catch {

...

}

// Wrong:

if (true)

println("Wow!")

// Wrong:

try foo() catch {

...

}

**Long Literals**

Suffix long literal values with uppercase L. It is often hard to differentiate lowercase l from 1.

val longValue = 5432L // Do this

val longValue = 5432l // Do NOT do this

**Documentation Style**

Use Java docs style instead of Scala docs style.

/\*\* This is a correct one-liner, short description. \*/

/\*\*

\* This is correct multi-line JavaDoc comment. And

\* this is my second line, and if I keep typing, this would be

\* my third line.

\*/

/\*\* In Spark, we don't use the ScalaDoc style so this

\* is not correct.

\*/

**Ordering within a Class**

If a class is long and has many methods, group them logically into different sections, and use comment headers to organize them.

class DataFrame {

///////////////////////////////////////////////////////////////////////////

// DataFrame operations

///////////////////////////////////////////////////////////////////////////

...

///////////////////////////////////////////////////////////////////////////

// RDD operations

///////////////////////////////////////////////////////////////////////////

...

}

Of course, the situation in which a class grows this long is strongly discouraged, and is generally reserved only for building certain public APIs.

**Imports**

* **Avoid using wildcard imports**, unless you are importing more than 6 entities, or implicit methods. Wildcard imports make the code less robust to external changes.
* Always import packages using absolute paths (e.g. scala.util.Random) instead of relative ones (e.g. util.Random).
* In addition, sort imports in the following order:
  + java.\* and javax.\*
  + scala.\*
  + Third-party libraries (org.\*, com.\*, etc)
  + Project classes (com.databricks.\* or org.apache.spark if you are working on Spark)
* Within each group, imports should be sorted in alphabetic ordering.
* You can use IntelliJ's import organizer to handle this automatically, using the following config:
* java
* javax
* \_\_\_\_\_\_\_ blank line \_\_\_\_\_\_\_
* scala
* \_\_\_\_\_\_\_ blank line \_\_\_\_\_\_\_
* all other imports
* \_\_\_\_\_\_\_ blank line \_\_\_\_\_\_\_
* com.databricks // or org.apache.spark if you are working on Spark

**Pattern Matching**

* For method whose entire body is a pattern match expression, put the match on the same line as the method declaration if possible to reduce one level of indentation.
* def test(msg: Message): Unit = msg match {
* case ...

}

* When calling a function with a closure (or partial function), if there is only one case, put the case on the same line as the function invocation.
* list.zipWithIndex.map { case (elem, i) =>
* // ...

}

If there are multiple cases, indent and wrap them.

list.map {

case a: Foo => ...

case b: Bar => ...

}

* If the only goal is to match on the type of the object, do NOT expand fully all the arguments, as it makes refactoring more difficult and the code more error prone.
* case class Pokemon(name: String, weight: Int, hp: Int, attack: Int, defense: Int)
* case class Human(name: String, hp: Int)
* // Do NOT do the following, because
* // 1. When a new field is added to Pokemon, we need to change this pattern matching as well
* // 2. It is easy to mismatch the arguments, especially for the ones that have the same data types
* targets.foreach {
* case target @ Pokemon(\_, \_, hp, \_, defense) =>
* val loss = sys.min(0, myAttack - defense)
* target.copy(hp = hp - loss)
* case target @ Human(\_, hp) =>
* target.copy(hp = hp - myAttack)
* }
* // Do this:
* targets.foreach {
* case target: Pokemon =>
* val loss = sys.min(0, myAttack - target.defense)
* target.copy(hp = target.hp - loss)
* case target: Human =>
* target.copy(hp = target.hp - myAttack)

}

**Infix Methods**

**Avoid infix notation** for methods that aren't symbolic methods (i.e. operator overloading).

// Correct

list.map(func)

string.contains("foo")

// Wrong

list map (func)

string contains "foo"

// But overloaded operators should be invoked in infix style

arrayBuffer += elem

**Anonymous Methods**

**Avoid excessive parentheses and curly braces** for anonymous methods.

// Correct

list.map { item =>

...

}

// Correct

list.map(item => ...)

// Wrong

list.map(item => {

...

})

// Wrong

list.map { item => {

...

}}

// Wrong

list.map({ item => ... })

**Scala Language Features**

**Case Classes and Immutability**

Case classes are regular classes but extended by the compiler to automatically support:

* Public getters for constructor parameters
* Copy constructor
* Pattern matching on constructor parameters
* Automatic toString/hash/equals implementation

Constructor parameters should NOT be mutable for case classes. Instead, use copy constructor. Having mutable case classes can be error prone, e.g. hash maps might place the object in the wrong bucket using the old hash code.

// This is OK

case class Person(name: String, age: Int)

// This is NOT OK

case class Person(name: String, var age: Int)

// To change values, use the copy constructor to create a new instance

val p1 = Person("Peter", 15)

val p2 = p2.copy(age = 16)

**apply Method**

Avoid defining apply methods on classes. These methods tend to make the code less readable, especially for people less familiar with Scala. It is also harder for IDEs (or grep) to trace. In the worst case, it can also affect correctness of the code in surprising ways, as demonstrated in [Parentheses](https://github.com/databricks/scala-style-guide#parentheses).

It is acceptable to define apply methods on companion objects as factory methods. In these cases, the apply method should return the companion class type.

object TreeNode {

// This is OK

def apply(name: String): TreeNode = ...

// This is bad because it does not return a TreeNode

def apply(name: String): String = ...

}

**override Modifier**

Always add override modifier for methods, both for overriding concrete methods and implementing abstract methods. The Scala compiler does not require override for implementing abstract methods. However, we should always add override to make the override obvious, and to avoid accidental non-overrides due to non-matching signatures.

trait Parent {

def hello(data: Map[String, String]): Unit = {

print(data)

}

}

class Child extends Parent {

import scala.collection.Map

// The following method does NOT override Parent.hello,

// because the two Maps have different types.

// If we added "override" modifier, the compiler would've caught it.

def hello(data: Map[String, String]): Unit = {

print("This is supposed to override the parent method, but it is actually not!")

}

}

**Destructuring Binds**

Destructuring bind (sometimes called tuple extraction) is a convenient way to assign two variables in one expression.

val (a, b) = (1, 2)

However, do NOT use them in constructors, especially when a and b need to be marked transient. The Scala compiler generates an extra Tuple2 field that will not be transient for the above example.

class MyClass {

// This will NOT work because the compiler generates a non-transient Tuple2

// that points to both a and b.

@transient private val (a, b) = someFuncThatReturnsTuple2()

}

**Call by Name**

**Avoid using call by name**. Use () => T explicitly.

Background: Scala allows method parameters to be defined by-name, e.g. the following would work:

def print(value: => Int): Unit = {

println(value)

println(value + 1)

}

var a = 0

def inc(): Int = {

a += 1

a

}

print(inc())

in the above code, inc() is passed into print as a closure and is executed (twice) in the print method, rather than being passed in as a value 1. The main problem with call-by-name is that the caller cannot differentiate between call-by-name and call-by-value, and thus cannot know for sure whether the expression will be executed or not (or maybe worse, multiple times). This is especially dangerous for expressions that have side-effect.

**Multiple Parameter Lists**

**Avoid using multiple parameter lists**. They complicate operator overloading, and can confuse programmers less familiar with Scala. For example:

// Avoid this!

case class Person(name: String, age: Int)(secret: String)

One notable exception is the use of a 2nd parameter list for implicits when defining low-level libraries. That said, [implicits should be avoided](https://github.com/databricks/scala-style-guide" \l "implicits)!

**Symbolic Methods (Operator Overloading)**

**Do NOT use symbolic method names**, unless you are defining them for natural arithmetic operations (e.g. +, -, \*, /). Under no other circumstances should they be used. Symbolic method names make it very hard to understand the intent of the methods. Consider the following two examples:

// symbolic method names are hard to understand

channel ! msg

stream1 >>= stream2

// self-evident what is going on

channel.send(msg)

stream1.join(stream2)

**Type Inference**

Scala type inference, especially left-side type inference and closure inference, can make code more concise. That said, there are a few cases where explicit typing should be used:

* **Public methods should be explicitly typed**, otherwise the compiler's inferred type can often surprise you.
* **Implicit methods should be explicitly typed**, otherwise it can crash the Scala compiler with incremental compilation.
* **Variables or closures with non-obvious types should be explicitly typed**. A good litmus test is that explicit types should be used if a code reviewer cannot determine the type in 3 seconds.

**Return Statements**

**Avoid using return in closures**. return is turned into try/catch of scala.runtime.NonLocalReturnControl by the compiler. This can lead to unexpected behaviors. Consider the following example:

def receive(rpc: WebSocketRPC): Option[Response] = {

tableFut.onComplete { table =>

if (table.isFailure) {

return None // Do not do that!

} else { ... }

}

}

the .onComplete method takes the anonymous closure { table => ... } and passes it to a different thread. This closure eventually throws the NonLocalReturnControl exception that is captured **in a different thread** . It has no effect on the poor method being executed here.

However, there are a few cases where return is preferred.

* Use return as a guard to simplify control flow without adding a level of indentation
* def doSomething(obj: Any): Any = {
* if (obj eq null) {
* return null
* }
* // do something ...

}

* Use return to terminate a loop early, rather than constructing status flags
* while (true) {
* if (cond) {
* return
* }

}

**Recursion and Tail Recursion**

**Avoid using recursion**, unless the problem can be naturally framed recursively (e.g. graph traversal, tree traversal).

For methods that are meant to be tail recursive, apply @tailrec annotation to make sure the compiler can check it is tail recursive. (You will be surprised how often seemingly tail recursive code is actually not tail recursive due to the use of closures and functional transformations.)

Most code is easier to reason about with a simple loop and explicit state machines. Expressing it with tail recursions (and accumulators) can make it more verbose and harder to understand. For example, the following imperative code is more readable than the tail recursive version:

// Tail recursive version.

def max(data: Array[Int]): Int = {

@tailrec

def max0(data: Array[Int], pos: Int, max: Int): Int = {

if (pos == data.length) {

max

} else {

max0(data, pos + 1, if (data(pos) > max) data(pos) else max)

}

}

max0(data, 0, Int.MinValue)

}

// Explicit loop version

def max(data: Array[Int]): Int = {

var max = Int.MinValue

for (v <- data) {

if (v > max) {

max = v

}

}

max

}

**Implicits**

**Avoid using implicits**, unless:

* you are building a domain-specific language
* you are using it for implicit type parameters (e.g. ClassTag, TypeTag)
* you are using it private to your own class to reduce verbosity of converting from one type to another (e.g. Scala closure to Java closure)

When implicits are used, we must ensure that another engineer who did not author the code can understand the semantics of the usage without reading the implicit definition itself. Implicits have very complicated resolution rules and make the code base extremely difficult to understand. From Twitter's Effective Scala guide: "If you do find yourself using implicits, always ask yourself if there is a way to achieve the same thing without their help."

If you must use them (e.g. enriching some DSL), do not overload implicit methods, i.e. make sure each implicit method has distinct names, so users can selectively import them.

// Don't do the following, as users cannot selectively import only one of the methods.

object ImplicitHolder {

def toRdd(seq: Seq[Int]): RDD[Int] = ...

def toRdd(seq: Seq[Long]): RDD[Long] = ...

}

// Do the following:

object ImplicitHolder {

def intSeqToRdd(seq: Seq[Int]): RDD[Int] = ...

def longSeqToRdd(seq: Seq[Long]): RDD[Long] = ...

}

**Exception Handling (Try vs try)**

* Do NOT catch Throwable or Exception. Use scala.util.control.NonFatal:
* try {
* ...
* } catch {
* case NonFatal(e) =>
* // handle exception; note that NonFatal does not match InterruptedException
* case e: InterruptedException =>
* // handle InterruptedException

}

This ensures that we do not catch NonLocalReturnControl (as explained in [Return Statements](https://github.com/databricks/scala-style-guide#return-statements)).

* Do NOT use Try in APIs, that is, do NOT return Try in any methods. Instead, prefer explicitly throwing exceptions for abnormal execution and Java style try/catch for exception handling.

Background information: Scala provides monadic error handling (through Try, Success, and Failure) that facilitates chaining of actions. However, we found from our experience that the use of it often leads to more levels of nesting that are harder to read. In addition, it is often unclear what the semantics are for expected errors vs exceptions because those are not encoded in Try. As a result, we discourage the use of Try for error handling. In particular:

As a contrived example:

class UserService {

/\*\* Look up a user's profile in the user database. \*/

def get(userId: Int): Try[User]

}

is better written as

class UserService {

/\*\*

\* Look up a user's profile in the user database.

\* @return None if the user is not found.

\* @throws DatabaseConnectionException when we have trouble connecting to the database/

\*/

@throws(DatabaseConnectionException)

def get(userId: Int): Option[User]

}

The 2nd one makes it very obvious error cases the caller needs to handle.

**Options**

* Use Option when the value can be empty. Compared with null, an Option explicitly states in the API contract that the value can be None.
* When constructing an Option, use Option rather than Some to guard against null values.
* def myMethod1(input: String): Option[String] = Option(transform(input))
* // This is not as robust because transform can return null, and then
* // myMethod2 will return Some(null).

def myMethod2(input: String): Option[String] = Some(transform(input))

* Do not use None to represent exceptions. Instead, throw exceptions explicitly.
* Do not call get directly on an Option, unless you know absolutely for sure the Option has some value.

**Monadic Chaining**

One of Scala's powerful features is monadic chaining. Almost everything (e.g. collections, Option, Future, Try) is a monad and operations on them can be chained together. This is an incredibly powerful concept, but chaining should be used sparingly. In particular:

* Avoid chaining (and/or nesting) more than 3 operations.
* If it takes more than 5 seconds to figure out what the logic is, try hard to think about how you can express the same functionality without using monadic chaining. As a general rule, watch out for flatMaps and folds.
* A chain should almost always be broken after a flatMap (because of the type change).

A chain can often be made more understandable by giving the intermediate result a variable name, by explicitly typing the variable, and by breaking it down into more procedural style. As a contrived example:

class Person(val data: Map[String, String])

val database = Map[String, Person]

// Sometimes the client can store "null" value in the store "address"

// A monadic chaining approach

def getAddress(name: String): Option[String] = {

database.get(name).flatMap { elem =>

elem.data.get("address")

.flatMap(Option.apply) // handle null value

}

}

// A more readable approach, despite much longer

def getAddress(name: String): Option[String] = {

if (!database.contains(name)) {

return None

}

database(name).data.get("address") match {

case Some(null) => None // handle null value

case Some(addr) => Option(addr)

case None => None

}

}

**Concurrency**

**Scala concurrent.Map**

**Prefer java.util.concurrent.ConcurrentHashMap over scala.collection.concurrent.Map**. In particular the getOrElseUpdatemethod in scala.collection.concurrent.Map is not atomic (fixed in Scala 2.11.6, [SI-7943](https://issues.scala-lang.org/browse/SI-7943)). Since all the projects we work on require cross-building for both Scala 2.10 and Scala 2.11, scala.collection.concurrent.Map should be avoided.

**Explicit Synchronization vs Concurrent Collections**

There are 3 recommended ways to make concurrent accesses to shared states safe. **Do NOT mix them** because that could make the program very hard to reason about and lead to deadlocks.

1. java.util.concurrent.ConcurrentHashMap: Use when all states are captured in a map, and high degree of contention is expected.

private[this] val map = new java.util.concurrent.ConcurrentHashMap[String, String]

1. java.util.Collections.synchronizedMap: Use when all states are captured in a map, and contention is not expected but you still want to make code safe. In case of no contention, the JVM JIT compiler is able to remove the synchronization overhead via biased locking.

private[this] val map = java.util.Collections.synchronizedMap(new java.util.HashMap[String, String])

1. Explicit synchronization by synchronizing all critical sections: can used to guard multiple variables. Similar to 2, the JVM JIT compiler can remove the synchronization overhead via biased locking.
2. class Manager {
3. private[this] var count = 0
4. private[this] val map = new java.util.HashMap[String, String]
5. def update(key: String, value: String): Unit = synchronized {
6. map.put(key, value)
7. count += 1
8. }
9. def getCount: Int = synchronized { count }

}

Note that for case 1 and case 2, do not let views or iterators of the collections escape the protected area. This can happen in non-obvious ways, e.g. when returning Map.keySet or Map.values. If views or values are required to pass around, make a copy of the data.

val map = java.util.Collections.synchronizedMap(new java.util.HashMap[String, String])

// This is broken!

def values: Iterable[String] = map.values

// Instead, copy the elements

def values: Iterable[String] = map.synchronized { Seq(map.values: \_\*) }

**Explicit Synchronization vs Atomic Variables vs @volatile**

The java.util.concurrent.atomic package provides primitives for lock-free access to primitive types, such as AtomicBoolean, AtomicInteger, and AtomicReference.

Always prefer Atomic variables over @volatile. They have a strict superset of the functionality and are more visible in code. Atomic variables are implemented using @volatile under the hood.

Prefer Atomic variables over explicit synchronization when: (1) all critical updates for an object are confined to a *single* variable and contention is expected. Atomic variables are lock-free and permit more efficient contention. Or (2) synchronization is clearly expressed as a getAndSet operation. For example:

// good: clearly and efficiently express only-once execution of concurrent code

val initialized = new AtomicBoolean(false)

...

if (!initialized.getAndSet(true)) {

...

}

// poor: less clear what is guarded by synchronization, may unnecessarily synchronize

val initialized = false

...

var wasInitialized = false

synchronized {

wasInitialized = initialized

initialized = true

}

if (!wasInitialized) {

...

}

**Private Fields**

Note that private fields are still accessible by other instances of the same class, so protecting it with this.synchronized (or just synchronized) is not technically sufficient. Make the field private[this] instead.

// The following is still unsafe.

class Foo {

private var count: Int = 0

def inc(): Unit = synchronized { count += 1 }

}

// The following is safe.

class Foo {

private[this] var count: Int = 0

def inc(): Unit = synchronized { count += 1 }

}

**Isolation**

In general, concurrency and synchronization logic should be isolated and contained as much as possible. This effectively means:

* Avoid surfacing the internals of synchronization primitives in APIs, user-facing methods, and callbacks.
* For complex modules, create a small, inner module that captures the concurrency primitives.

**Performance**

For the vast majority of the code you write, performance should not be a concern. However, for performance sensitive code, here are some tips:

**Microbenchmarks**

It is ridiculously hard to write a good microbenchmark because the Scala compiler and the JVM JIT compiler do a lot of magic to the code. More often than not, your microbenchmark code is not measuring the thing you want to measure.

Use [jmh](http://openjdk.java.net/projects/code-tools/jmh/) if you are writing microbenchmark code. Make sure you read through [all the sample microbenchmarks](http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/) so you understand the effect of deadcode elimination, constant folding, and loop unrolling on microbenchmarks.

**Traversal and zipWithIndex**

Use while loops instead of for loops or functional transformations (e.g. map, foreach). For loops and functional transformations are very slow (due to virtual function calls and boxing).

val arr = // array of ints

// zero out even positions

val newArr = list.zipWithIndex.map { case (elem, i) =>

if (i % 2 == 0) 0 else elem

}

// This is a high performance version of the above

val newArr = new Array[Int](arr.length)

var i = 0

val len = newArr.length

while (i < len) {

newArr(i) = if (i % 2 == 0) 0 else arr(i)

i += 1

}

**Option and null**

For performance sensitive code, prefer null over Option, in order to avoid virtual method calls and boxing. Label the nullable fields clearly with Nullable.

class Foo {

@javax.annotation.Nullable

private[this] var nullableField: Bar = \_

}

**Scala Collection Library**

For performance sensitive code, prefer Java collection library over Scala ones, since the Scala collection library often is slower than Java's.

**private[this]**

For performance sensitive code, prefer private[this] over private. private[this] generates a field, rather than creating an accessor method. In our experience, the JVM JIT compiler cannot always inline private field accessor methods, and thus it is safer to use private[this] to ensure no virtual method call for accessing a field.

class MyClass {

private val field1 = ...

private[this] val field2 = ...

def perfSensitiveMethod(): Unit = {

var i = 0

while (i < 1000000) {

field1 // This might invoke a virtual method call

field2 // This is just a field access

i += 1

}

}

}

**Java Interoperability**

This section covers guidelines for building Java compatible APIs. These do not apply if the component you are building does not require interoperability with Java. It is mostly drawn from our experience in developing the Java APIs for Spark.

**Java Features Missing from Scala**

The following Java features are missing from Scala. If you need the following, define them in Java instead. However, be reminded that ScalaDocs are not generated for files defined in Java.

* Static fields
* Static inner classes
* Java enums
* Annotations

**Traits and Abstract Classes**

For interfaces that can be implemented externally, keep in mind the following:

* Traits with default method implementations are not usable in Java. Use abstract classes instead.
* In general, avoid using traits unless you know for sure the interface will not have any default implementation even in its future evolution.

// The default implementation doesn't work in Java

trait Listener {

def onTermination(): Unit = { ... }

}

// Works in Java

abstract class Listener {

def onTermination(): Unit = { ... }

}

**Type Aliases**

Do NOT use type aliases. They are not visible in bytecode (and Java).

**Default Parameter Values**

Do NOT use default parameter values. Overload the method instead.

// Breaks Java interoperability

def sample(ratio: Double, withReplacement: Boolean = false): RDD[T] = { ... }

// The following two work

def sample(ratio: Double, withReplacement: Boolean): RDD[T] = { ... }

def sample(ratio: Double): RDD[T] = sample(ratio, withReplacement = false)

**Multiple Parameter Lists**

Do NOT use multi-parameter lists.

**Varargs**

* Apply @scala.annotation.varargs annotation for a vararg method to be usable in Java. The Scala compiler creates two methods, one for Scala (bytecode parameter is a Seq) and one for Java (bytecode parameter array).
* @scala.annotation.varargs

def select(exprs: Expression\*): DataFrame = { ... }

* Note that abstract vararg methods does NOT work for Java, due to a Scala compiler bug ([SI-1459](https://issues.scala-lang.org/browse/SI-1459), [SI-9013](https://issues.scala-lang.org/browse/SI-9013)).
* Be careful with overloading varargs methods. Overloading a vararg method with another vararg type can break source compatibility.
* class Database {
* @scala.annotation.varargs
* def remove(elems: String\*): Unit = ...
* // Adding this will break source compatibility for no-arg remove() call.
* @scala.annotation.varargs
* def remove(elems: People\*): Unit = ...
* }
* // This won't compile anymore because it is ambiguous

new Database().remove()

Instead, define an explicit first parameter followed by vararg:

class Database {

@scala.annotation.varargs

def remove(elems: String\*): Unit = ...

// The following is OK.

@scala.annotation.varargs

def remove(elem: People, elems: People\*): Unit = ...

}

**Implicits**

Do NOT use implicits, for a class or method. This includes ClassTag, TypeTag.

class JavaFriendlyAPI {

// This is NOT Java friendly, since the method contains an implicit parameter (ClassTag).

def convertTo[T: ClassTag](): T

}

**Companion Objects, Static Methods and Fields**

There are a few things to watch out for when it comes to companion objects and static methods/fields.

* Companion objects are awkward to use in Java (a companion object Foo is a static field MODULE$ of type Foo$ in class Foo$).
* object Foo
* // equivalent to the following Java code
* public class Foo$ {
* Foo$ MODULE$ = // instantiation of the object

}

If the companion object is important to use, create a Java static field in a separate class.

* Unfortunately, there is no way to define a JVM static field in Scala. Create a Java file to define that.
* Methods in companion objects are automatically turned into static methods in the companion class, unless there is a method name conflict. The best (and future-proof) way to guarantee the generation of static methods is to add a test file written in Java that calls the static method.
* class Foo {
* def method2(): Unit = { ... }
* }
* object Foo {
* def method1(): Unit = { ... } // a static method Foo.method1 is created in bytecode
* def method2(): Unit = { ... } // a static method Foo.method2 is NOT created in bytecode
* }
* // FooJavaTest.java (in test/scala/com/databricks/...)
* public class FooJavaTest {
* public static void compileTest() {
* Foo.method1(); // This one should compile fine
* Foo.method2(); // This one should fail because method2 is not generated.
* }

}

* A case object (or even just plain companion object) MyClass is actually not of type MyClass.
* case object MyClass
* // Test.java
* if (MyClass$.MODULE instanceof MyClass) {
* // The above condition is always false

}

To implement the proper type hierarchy, define a companion class, and then extend that in case object:

class MyClass

case object MyClass extends MyClass

**Miscellaneous**

**Prefer nanoTime over currentTimeMillis**

When computing a *duration* or checking for a *timeout*, avoid using System.currentTimeMillis(). Use System.nanoTime()instead, even if you are not interested in sub-millisecond precision.

System.currentTimeMillis() returns current wallclock time and will follow changes to the system clock. Thus, negative wallclock adjustments can cause timeouts to "hang" for a long time (until wallclock time has caught up to its previous value again). This can happen when ntpd does a "step" after the network has been disconnected for some time. The most canonical example is during system bootup when DHCP takes longer than usual. This can lead to failures that are really hard to understand/reproduce. System.nanoTime() is guaranteed to be monotonically increasing irrespective of wallclock changes.

Caveats:

* Never serialize an absolute nanoTime() value or pass it to another system. The absolute value is meaningless and system-specific and resets when the system reboots.
* The absolute nanoTime() value is not guaranteed to be positive (but t2 - t1 is guaranteed to yield the right result)
* nanoTime() rolls over every 292 years. So if your Spark job is going to take a really long time, you may need something else :)

**Prefer URI over URL**

When storing the URL of a service, you should use the URI representation.

The [equality check](http://docs.oracle.com/javase/7/docs/api/java/net/URL.html#equals(java.lang.Object)) of URL actually performs a (blocking) network call to resolve the IP address. The URI class performs field equality and is a superset of URL as to what it can represent.