Interfaces and Traits

Java	Scala

`interface Bar {`
`  double fiddle(int n);`
`  int triple(int n);`
`}`
	`trait Bar {`
	`  def fiddle(n: Int): Double`
	`  def triple(n: Int) = 3 * n`
	`}`
	
	`// Traits may contain complete functions`
	`// If result type is obvious, no need to declare it`

Operators

Java	Scala
`! ~ * / % + - << >> >>> < > <= >= == != & | ^ && ||`	`! ~ * / % + - << >> >>> < > <= >= == != & | ^ && ||`
`= +  -= *= /= %= <<= >>= >>>= &= ^= !=`	`= +  -= *= /= %= <<= >>= >>>= &= ^= !=`
`c ? x : y`	`if c then x else y`
`++  --`	`// Deliberately omitted from Scala`

Statements and Expressions

Strictly speaking, Scala does not have statements, only expressions. However, many of the following Scala expressions return `()`, the "unit" value. In this table I use "statement" to indicate that `()` is returned.

Java	Scala
`{ statements }`	`{ expressions }` `// value is last expression evaluated`

`if (condition) statement`
`else if (condition) statement`
`else statement`
	`if (condition) expression`
	`else if (condition) expression`
	`else expression// value is last expression evaluated`

`while (condition) statement`	`while (condition) statement`
`do { statements } while (condition)`	`do { statements } while (condition)`

`for (initialization; test; increment) statement`	`for (generators/guards) statement`
	`// generators are variable <- sequence`
	`// for sequence use list, array, min to max, min until max`
	`// guards are if condition`
	`// must begin with a generator`

`continue`
`break`
	`// No immediate Scala equivalent`
	`// Can be implemented (slowly) with Exceptions`
	`// Consider using a filter instead`

`// No Java equivalent`	`expression match {`
	`  case pattern1 => expression1`
	`  case pattern2 if condition => expression2`
	`  ...`
	`  case patternN => expressionN`
	`}`
	`/* Patterns can be literal values, variables, underscores,`
	`sequences, tuples, options, typed patterns, name of a`
	`case class, or regular expressions */`

`return;`	`return value // must supply a value` `// If used, function must declare return type`

`try { statements }`
`  catch (ExceptionType variable) { statements }`
`  ...`
`  catch (ExceptionType variable) { statements }`
`  finally { statements }`
`}`
	`try { expressions }`
	`catch {`
	`  case name: Exception => { expressions }`
	`  ...`
	`  case name: Exception => { expressions }`
	`} finally { statements }`
	`// Consider having the expressions return an Option type`

Method/Function Definitions

Java	Scala

`returnType methodName(type arg, ..., type arg) {...}`
`// Methods must be declared at top level within a class`
	`def functionName(arg: Type, ..., arg: Type): returnType = {...}`
	`/* returnType may be omitted if function is not recursive`
	`and does not contain return statements */`

`void methodName(type arg, ..., type arg) {...}`	`def functionName(arg: Type, ..., arg: Type): returnType {...}` `// Note the absence of =`
`returnType methodName(type arg, ..., type... arg) {...}` `// Last argument is received as an array`	`def functionName(arg: Type, ..., arg: Type*): returnType = {...}` `// Last argument is received as a Seq`
`// Java does not have default arguments` `// Both Java and Scala may have overloaded methods`	`def functionName(..., arg: Type=value): returnType = {...}` `// Rightmost arguments may have default values`
`// Java does not have named arguments`	`// You can call with named arguments, as` `// functionName(name=value, ...)`

Types and Type Declarations

Java	Scala
These are primitives: `double`, `float`, `byte`, `char`, `short`, `int`, `long`, `boolean`	These are objects (superclass `AnyVal`): `Double`, `Float`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Boolean`
ObjectType<*ContentType*, *ContentType*, ...>	*ObjectType*[*ContentType*, *ContentType*, ...]
// Use interface in java.util.function	(*type*, ..., *type*) => *returnType*
// No equivalent	type *name* = *type* // Gives a name to a type
final int MAX = 100; int count = 0;	val max = 100 -- vals are immutable var count = 0 var count: Int = 0 -- ok to explicitly declare the type
int count;	// No equivalent, variables must have a value
String[] languages = {"C", "C++", "Java", "Scala"};	var languages = Array("C", "C++", "Java", "Scala")

import java.util.LinkedList;
...
LinkedList list = new LinkedList();
list.add("C");
list.add("C++");
list.add("Java");
list.add("Scala");
...
// Approximately 70 methods defined on lists
	var list = List("C", "C++", "Java", "Scala")
	
	// Approximately 170 methods defined on lists

import java.util.HashMap;
...
HashMap<String, Int> map = new HashMap<>;
map.put("Dick", 8);
map.put("Jane", 6);
...
age = map.get("Dick");
	var map = Map("Dick" -> 8, "Jane" -> 6)
	
	...
	
	age = map("Dick")

// Java does not have tuples	("Mary", 12)
null	Scala has `null` only so it can interact with Java Otherwise use type `Option[`*type*`]` with values `Some(`*value*`)` or `None`
void // only as a method return type	() // The "unit" value
(x, y) -> (x + y) / 2	(x, y) => (x + y) / 2

Concise Java to Scala

Packages and Imports

Java	Scala
`package name; // must be first thing in file`	`package name // Can go anywhere`
`import package.class;`	`import package.class,import package.object`
`import package.class.*;`	`import package.class._`
`import static package.class;`	`// All Scala imports are static`
`// No Java equivalent`	`import package.{class, object} // Import selected items`
`// No Java equivalent`	`import package.{class => name} // Import and rename`
`// No Java equivalent`	`import package.{class => _} // Import all except`

Classes, Constructors, Setters and Getters

Java

```
class Foo extends Bar implements Baz {
    private int n;
    private double x;
    private String s;

    public Foo(int n, double x, String s) {
        this.n = n;
        this.x = x;
        this.s = s;
    }

    public Foo(int n) {
        this(2 * n, 0.0, "abc")
    }

    public int getN() { return n; }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
}
```

Scala

```
class Foo(val n: Int, var x: Double, s: String)
        extends Bar with Baz {

    // The above defines the class and saves the
    // arguments as instance variables.

    def this(n: Int) {
        this(2 * n, 0.0, "abc")
    }

    // To create an instance of Foo,
    // say foo = new Foo(1, 2.0, "abc")

    // n is val, so it has a getter
    // To call the getter for foo, say foo.n

    // x is var, so it has both a getter and a setter
    // To set foo.x to 3.5, say foo.x = 3.5

    // s is neither val nor var, so it has no getter
    // and no setter
}
```