Standard Library

The Scala Standard Library

The Scala standard library consists of the package `scala` with a number of classes and modules. Some of these classes are described in the following.

Class hierarchy of Scala

Root Classes

The root of this hierarchy is formed by class `Any`. Every class in a Scala execution environment inherits directly or indirectly from this class. Class `Any` has two direct subclasses: `AnyRef` and `AnyVal`.

The subclass `AnyRef` represents all values which are represented as objects in the underlying host system. Classes written in other languages inherit from `scala.AnyRef`.

The predefined subclasses of class `AnyVal` describe values which are not implemented as objects in the underlying host system.

User-defined Scala classes which do not explicitly inherit from `AnyVal` inherit directly or indirectly from `AnyRef`. They can not inherit from both `AnyRef` and `AnyVal`.

Classes `AnyRef` and `AnyVal` are required to provide only the members declared in class `Any`, but implementations may add host-specific methods to these classes (for instance, an implementation may identify class `AnyRef` with its own root class for objects).

The signatures of these root classes are described by the following definitions.

```
package scala
/** The universal root class */
abstract class Any {

  /** Defined equality; abstract here */
  def equals(that: Any): Boolean
  /** Semantic equality between values */
  final def == (that: Any): Boolean  =
    if (null eq this) null eq that else this equals that

  /** Semantic inequality between values */
  final def != (that: Any): Boolean  =  !(this == that)

  /** Hash code; abstract here */
  def hashCode: Int = $\ldots$

  /** Textual representation; abstract here */
  def toString: String = $\ldots$

  /** Type test; needs to be inlined to work as given */
  def isInstanceOf[a]: Boolean

  /** Type cast; needs to be inlined to work as given */ */
  def asInstanceOf[A]: A = this match {
    case x: A => x
    case _ => if (this eq null) this
              else throw new ClassCastException()
  }
}

/** The root class of all value types */
final class AnyVal extends Any

/** The root class of all reference types */
class AnyRef extends Any {
  def equals(that: Any): Boolean      = this eq that
  final def eq(that: AnyRef): Boolean = $\ldots$ // reference equality
  final def ne(that: AnyRef): Boolean = !(this eq that)

  def hashCode: Int = $\ldots$     // hashCode computed from allocation address
  def toString: String  = $\ldots$ // toString computed from hashCode and class name

  def synchronized[T](body: => T): T // execute `body` in while locking `this`.
}
```

The type test `$x$.isInstanceOf[$T$]` is equivalent to a typed pattern match

```
$x$ match {
  case _: $T'$ => true
  case _ => false
}
```

where the type T' is the same as T except if T is of the form D or $D[tps]$

where $D$ is a type member of some outer class $C$. In this case T' is `$C$#$D$` (or `$C$#$D[tps]$`, respectively), whereas $T$ itself would expand to `$C$.this.$D[tps]$`. In other words, an `isInstanceOf` test does not check that types have the same enclosing instance.

The test `$x$.asInstanceOf[$T$]` is treated specially if $T$ is a numeric value type. In this case the cast will be translated to an application of a conversion method `x.to$T$`. For non-numeric values $x$ the operation will raise a `ClassCastException`.

Value Classes

Value classes are classes whose instances are not represented as objects by the underlying host system. All value classes inherit from class `AnyVal`. Scala implementations need to provide the value classes `Unit`, `Boolean`, `Double`, `Float`, `Long`, `Int`, `Char`, `Short`, and `Byte` (but are free to provide others as well). The signatures of these classes are defined in the following.

Numeric Value Types

Classes `Double`, `Float`, `Long`, `Int`, `Char`, `Short`, and `Byte` are together called *numeric value types*. Classes `Byte`, `Short`, or `Char` are called *subrange types*. Subrange types, as well as `Int` and `Long` are called *integer types*, whereas `Float` and `Double` are called *floating point types*.

Numeric value types are ranked in the following partial order:

```
Byte - Short
             \
               Int - Long - Float - Double
             /
        Char
```

`Byte` and `Short` are the lowest-ranked types in this order, whereas `Double` is the highest-ranked. Ranking does *not* imply a conformance relationship; for instance `Int` is not a subtype of `Long`. However, object `Predef` defines views from every numeric value type to all higher-ranked numeric value types. Therefore, lower-ranked types are implicitly converted to higher-ranked types when required by the context.

Given two numeric value types S and $T$, the *operation type* of S and $T$ is defined as follows: If both S and $T$ are subrange types then the operation type of S and $T$ is `Int`. Otherwise the operation type of S and $T$ is the larger of the two types wrt ranking. Given two numeric values v and w the operation type of v and w is the operation type of their run-time types.

Any numeric value type $T$ supports the following methods.

- Comparison methods for equals (`==`), not-equals (`!=`), less-than (`<`), greater-than (`>`), less-than-or-equals (`<=`), greater-than-or-equals (`>=`), which each exist in 7 overloaded alternatives. Each alternative takes a parameter of some numeric value type. Its result type is type `Boolean`. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given comparison operation of that type.
- Arithmetic methods addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`), which each exist in 7 overloaded alternatives. Each alternative takes a parameter of some numeric value type U. Its result type is the operation type of T and U. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given arithmetic operation of that type.
- Parameterless arithmetic methods identity (`+`) and negation (`-`), with result type T. The first of these returns the receiver unchanged, whereas the second returns its negation.
- Conversion methods `toByte`, `toShort`, `toChar`, `toInt`, `toLong`, `toFloat`, `toDouble` which convert the receiver object to the target type, using the rules of Java's numeric type cast operation. The conversion might truncate the numeric value (as when going from `Long` to `Int` or from `Int` to `Byte`) or it might lose precision (as when going from `Double` to `Float` or when converting between `Long` and `Float`).

Integer numeric value types support in addition the following operations:

- Bit manipulation methods bitwise-and (`&`), bitwise-or {`|`}, and bitwise-exclusive-or (`^`), which each exist in 5 overloaded alternatives. Each alternative takes a parameter of some integer numeric value type. Its result type is the operation type of T and U. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given bitwise operation of that type.

- A parameterless bit-negation method (`~`). Its result type is the receiver type T or `Int`, whichever is larger. The operation is evaluated by converting the receiver to the result type and negating every bit in its value.

- Bit-shift methods left-shift (`<<`), arithmetic right-shift (`>>`), and unsigned right-shift (`>>>`). Each of these methods has two overloaded alternatives, which take a parameter n of type `Int`, respectively `Long`. The result type of the operation is the receiver type T, or `Int`, whichever is larger. The operation is evaluated by converting the receiver to the result type and performing the specified shift by n bits.

Numeric value types also implement operations `equals`, `hashCode`, and `toString` from class `Any`.

The `equals` method tests whether the argument is a numeric value type. If this is true, it will perform the `==` operation which is appropriate for that type. That is, the `equals` method of a numeric value type can be thought of being defined as follows:

```
def equals(other: Any): Boolean = other match {
  case that: Byte   => this == that
  case that: Short  => this == that
  case that: Char   => this == that
  case that: Int    => this == that
  case that: Long   => this == that
  case that: Float  => this == that
  case that: Double => this == that
  case _ => false
}
```

The `hashCode` method returns an integer hashcode that maps equal numeric values to equal results. It is guaranteed to be the identity for for type `Int` and for all subrange types.

The `toString` method displays its receiver as an integer or floating point number.

Example

This is the signature of the numeric value type `Int`:

```
package scala
abstract sealed class Int extends AnyVal {
  def == (that: Double): Boolean  // double equality
  def == (that: Float): Boolean   // float equality
  def == (that: Long): Boolean    // long equality
  def == (that: Int): Boolean     // int equality
  def == (that: Short): Boolean   // int equality
  def == (that: Byte): Boolean    // int equality
  def == (that: Char): Boolean    // int equality
  /* analogous for !=, <, >, <=, >= */

  def + (that: Double): Double    // double addition
  def + (that: Float): Double     // float addition
  def + (that: Long): Long        // long addition
  def + (that: Int): Int          // int addition
  def + (that: Short): Int        // int addition
  def + (that: Byte): Int         // int addition
  def + (that: Char): Int         // int addition
  /* analogous for -, *, /, % */

  def & (that: Long): Long        // long bitwise and
  def & (that: Int): Int          // int bitwise and
  def & (that: Short): Int        // int bitwise and
  def & (that: Byte): Int         // int bitwise and
  def & (that: Char): Int         // int bitwise and
  /* analogous for |, ^ */

  def << (cnt: Int): Int           // int left shift
  def << (cnt: Long): Int          // long left shift
  /* analogous for >>, >>> */

  def unary_+ : Int                // int identity
  def unary_- : Int                // int negation
  def unary_~ : Int                // int bitwise negation

  def toByte: Byte                 // convert to Byte
  def toShort: Short               // convert to Short
  def toChar: Char                 // convert to Char
  def toInt: Int                   // convert to Int
  def toLong: Long                 // convert to Long
  def toFloat: Float               // convert to Float
  def toDouble: Double             // convert to Double
}
```

Class Boolean

Class Boolean has only two values: true and false. It implements operations as given in the following class definition.

```
package scala
abstract sealed class Boolean extends AnyVal {
  def && (p: => Boolean): Boolean = // boolean and
    if (this) p else false
  def || (p: => Boolean): Boolean = // boolean or
    if (this) true else p
  def &  (x: Boolean): Boolean =    // boolean strict and
    if (this) x else false
  def |  (x: Boolean): Boolean =    // boolean strict or
    if (this) true else x
  def == (x: Boolean): Boolean =    // boolean equality
    if (this) x else x.unary_!
  def != (x: Boolean): Boolean =    // boolean inequality
    if (this) x.unary_! else x
  def unary_!: Boolean =            // boolean negation
    if (this) false else true
}
```

The class also implements operations equals, hashCode, and toString from class Any.

The equals method returns true if the argument is the same boolean value as the receiver, false otherwise. The hashCode method returns a fixed, implementation-specific hash-code when invoked on `true`, and a different, fixed, implementation-specific hash-code when invoked on `false`. The `toString` method returns the receiver converted to a string, i.e. either `"true"` or `"false"`.

Class `Unit`

Class `Unit` has only one value: `()`. It implements only the three methods `equals`, `hashCode`, and `toString` from class `Any`.

The `equals` method returns `true` if the argument is the unit value `()`, `false` otherwise. The `hashCode` method returns a fixed, implementation-specific hash-code, The `toString` method returns `"()"`.

Standard Reference Classes

This section presents some standard Scala reference classes which are treated in a special way by the Scala compiler – either Scala provides syntactic sugar for them, or the Scala compiler generates special code for their operations. Other classes in the standard Scala library are documented in the Scala library documentation by HTML pages.

Class `String`

Scala's `String` class is usually derived from the standard String class of the underlying host system (and may be identified with it). For Scala clients the class is taken to support in each case a method

```
def + (that: Any): String
```

which concatenates its left operand with the textual representation of its right operand.

The `Tuple` classes

Scala defines tuple classes `Tuple$n$` for $n = 2, \ldots, 22$. These are defined as follows.

```
package scala
case class Tuple$n$[+T_1, ..., +T_n](_1: T_1, ..., _$n$: T_$n$) {
  def toString = "(" ++ _1 ++ "," ++ $\ldots$ ++ "," ++ _$n$ ++ ")"
}
```

The `Function` Classes

Scala defines function classes `Function$n$` for $n = 1, \ldots, 22$. These are defined as follows.

```
package scala
trait Function$n$[-T_1, ..., -T_$n$, +R] {
  def apply(x_1: T_1, ..., x_$n$: T_$n$): R
  def toString = "<function>"
}
```

The `PartialFunction` subclass of `Function1` represents functions that (indirectly) specify their domain. Use the `isDefined` method to query whether the partial function is defined for a given input (i.e., whether the input is part of the function's domain).

```
class PartialFunction[-A, +B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

The implicitly imported `Predef` object defines the name `Function` as an alias of `Function1`.

Class `Array`

All operations on arrays desugar to the corresponding operations of the underlying platform. Therefore, the following class definition is given for informational purposes only:

```
final class Array[T](_length: Int)
extends java.io.Serializable with java.lang.Cloneable {
  def length: Int = $\ldots$
  def apply(i: Int): T = $\ldots$
  def update(i: Int, x: T): Unit = $\ldots$
  override def clone(): Array[T] = $\ldots$
}
```

If T is not a type parameter or abstract type, the type `Array[T]` is represented as the array type `|T|[]` in the underlying host system, where `|T|` is the erasure of `T`. If T is a type parameter or abstract type, a different representation might be used (it is `Object` on the Java platform).

Operations `length` returns the length of the array, `apply` means subscripting, and `update` means element update.

Because of the syntactic sugar for `apply` and `update` operations, we have the following correspondences between Scala and Java code for operations on an array `xs`:

Scala	*Java*
`xs.length`	`xs.length`
`xs(i)`	`xs[i]`
`xs(i) = e`	`xs[i] = e`

Two implicit conversions exist in `Predef` that are frequently applied to arrays: a conversion to `scala.collection.mutable.ArrayOps` and a conversion to `scala.collection.mutable.ArraySeq` (a subtype of `scala.collection.Seq`).

Both types make many of the standard operations found in the Scala collections API available. The conversion to `ArrayOps` is temporary, as all operations defined on `ArrayOps` return a value of type `Array`, while the conversion to `ArraySeq` is permanent as all operations return a value of type `ArraySeq`. The conversion to `ArrayOps` takes priority over the conversion to `ArraySeq`.

Because of the tension between parametrized types in Scala and the ad-hoc implementation of arrays in the host-languages, some subtle points need to be taken into account when dealing with arrays. These are explained in the following.

Variance Unlike arrays in Java, arrays in Scala are *not* co-variant; That is, $S <: T$ does not imply `Array[$S$] $<:$ Array[$T$]` in Scala. However, it is possible to cast an array of S to an array of T if such a cast is permitted in the host environment.

For instance `Array[String]` does not conform to `Array[Object]`, even though `String` conforms to `Object`. However, it is possible to cast an expression of type `Array[String]` to `Array[Object]`, and this cast will succeed without raising a `ClassCastException`. Example:

```scala
val xs = new Array[String](2)
// val ys: Array[Object] = xs   // **** error: incompatible types
val ys: Array[Object] = xs.asInstanceOf[Array[Object]] // OK
```

The instantiation of an array with a polymorphic element type T requires information about type T at runtime. This information is synthesized by adding a context bound of `scala.reflect.ClassTag` to type T. An example is the following implementation of method `mkArray`, which creates an array of an arbitrary type T, given a sequence of T's which defines its elements:

```scala
import reflect.ClassTag
def mkArray[T : ClassTag](elems: Seq[T]): Array[T] = {
  val result = new Array[T](elems.length)
  var i = 0
  for (elem <- elems) {
    result(i) = elem
    i += 1
  }
  result
}
```

If type T is a type for which the host platform offers a specialized array representation, this representation is used.

Example

On the Java Virtual Machine, an invocation of `mkArray(List(1,2,3))` will return a primitive array of `int`s, written as `int[]` in Java.

Companion object `Array`'s companion object provides various factory methods for the instantiation of single- and multi-dimensional arrays, an extractor method `unapplySeq` which enables pattern matching over arrays and additional utility methods:

```
package scala
object Array {
  /** copies array elements from `src` to `dest`. */
  def copy(src: AnyRef, srcPos: Int,
           dest: AnyRef, destPos: Int, length: Int): Unit = $\ldots$

  /** Returns an array of length 0 */
  def empty[T: ClassTag]: Array[T] =

  /** Create an array with given elements. */
  def apply[T: ClassTag](xs: T*): Array[T] = $\ldots$

  /** Creates array with given dimensions */
  def ofDim[T: ClassTag](n1: Int): Array[T] = $\ldots$
  /** Creates a 2-dimensional array */
  def ofDim[T: ClassTag](n1: Int, n2: Int): Array[Array[T]] = $\ldots$
  $\ldots$

  /** Concatenate all argument arrays into a single array. */
  def concat[T: ClassTag](xss: Array[T]*): Array[T] = $\ldots$

  /** Returns an array that contains the results of some element computation a number
    * of times. */
  def fill[T: ClassTag](n: Int)(elem: => T): Array[T] = $\ldots$
  /** Returns a two-dimensional array that contains the results of some element
    * computation a number of times. */
  def fill[T: ClassTag](n1: Int, n2: Int)(elem: => T): Array[Array[T]] = $\ldots$
  $\ldots$
  /** Returns an array containing values of a given function over a range of integer
    * values starting from 0. */
  def tabulate[T: ClassTag](n: Int)(f: Int => T): Array[T] = $\ldots$
  /** Returns a two-dimensional array containing values of a given function
    * over ranges of integer values starting from `0`. */
  def tabulate[T: ClassTag](n1: Int, n2: Int)(f: (Int, Int) => T): Array[Array[T]] = $\ldots
  $\ldots$

  /** Returns an array containing a sequence of increasing integers in a range. */
  def range(start: Int, end: Int): Array[Int] = $\ldots$
  /** Returns an array containing equally spaced values in some integer interval. */
  def range(start: Int, end: Int, step: Int): Array[Int] = $\ldots$

  /** Returns an array containing repeated applications of a function to a start value. */
  def iterate[T: ClassTag](start: T, len: Int)(f: T => T): Array[T] = $\ldots$

  /** Enables pattern matching over arrays */
  def unapplySeq[A](x: Array[A]): Option[IndexedSeq[A]] = Some(x)
}
```

Class Node

```
package scala.xml

trait Node {

  /** the label of this node */
  def label: String

  /** attribute axis */
  def attribute: Map[String, String]

  /** child axis (all children of this node) */
  def child: Seq[Node]

  /** descendant axis (all descendants of this node) */
  def descendant: Seq[Node] = child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  /** descendant axis (all descendants of this node) */
  def descendant_or_self: Seq[Node] = this::child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }
  override def equals(x: Any): Boolean = x match {
    case that:Node =>
      that.label == this.label &&
        that.attribute.sameElements(this.attribute) &&
          that.child.sameElements(this.child)
    case _ => false
  }

 /** XPath style projection function. Returns all children of this node
  *  that are labeled with 'that'. The document order is preserved.
  */
    def \(that: Symbol): NodeSeq = {
      new NodeSeq({
        that.name match {
          case "_" => child.toList
          case _ =>
            var res:List[Node] = Nil
            for (x <- child.elements if x.label == that.name) {
              res = x::res
            }
            res.reverse
        }
      })
    }

 /** XPath style projection function. Returns all nodes labeled with the
  *  name 'that' from the 'descendant_or_self' axis. Document order is preserved.
  */
  def \\(that: Symbol): NodeSeq = {
    new NodeSeq(
      that.name match {
        case "_" => this.descendant_or_self
        case _ => this.descendant_or_self.asInstanceOf[List[Node]].
        filter(x => x.label == that.name)
      })
  }

  /** hashcode for this XML node */
  override def hashCode =
    Utility.hashCode(label, attribute.toList.hashCode, child)

  /** string representation of this node */
  override def toString = Utility.toXML(this)

}
```

The `Predef` Object

The `Predef` object defines standard functions and type aliases for Scala programs. It is implicitly imported, as described in the chapter on name binding, so that all its defined members are available without qualification. Its definition for the JVM environment conforms to the following signature:

```
package scala
object Predef {

  // classOf ---------------------------------------------------------

  /** Returns the runtime representation of a class type. */
  def classOf[T]: Class[T] = null
   // this is a dummy, classOf is handled by compiler.

  // valueOf -----------------------------------------------------------

  /** Retrieve the single value of a type with a unique inhabitant. */
  @inline def valueOf[T](implicit vt: ValueOf[T]): T {} = vt.value
   // instances of the ValueOf type class are provided by the compiler.

  // Standard type aliases ---------------------------------------------

  type String    = java.lang.String
  type Class[T]  = java.lang.Class[T]

  // Miscellaneous -----------------------------------------------------

  type Function[-A, +B] = Function1[A, B]

  type Map[A, +B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]

  val Map = collection.immutable.Map
  val Set = collection.immutable.Set

  // Manifest types, companions, and incantations for summoning ---------

  type ClassManifest[T] = scala.reflect.ClassManifest[T]
  type Manifest[T]      = scala.reflect.Manifest[T]
  type OptManifest[T]   = scala.reflect.OptManifest[T]
  val ClassManifest     = scala.reflect.ClassManifest
  val Manifest          = scala.reflect.Manifest
  val NoManifest        = scala.reflect.NoManifest
```

```scala
def manifest[T](implicit m: Manifest[T])           = m
def classManifest[T](implicit m: ClassManifest[T]) = m
def optManifest[T](implicit m: OptManifest[T])     = m

// Minor variations on identity functions ------------------------------
def identity[A](x: A): A         = x
def implicitly[T](implicit e: T) = e    // for summoning implicit values from the nether w
@inline def locally[T](x: T): T  = x    // to communicate intent and avoid unmoored statem

// Asserts, Preconditions, Postconditions ------------------------------

def assert(assertion: Boolean) {
  if (!assertion)
    throw new java.lang.AssertionError("assertion failed")
}

def assert(assertion: Boolean, message: => Any) {
  if (!assertion)
    throw new java.lang.AssertionError("assertion failed: " + message)
}

def assume(assumption: Boolean) {
  if (!assumption)
    throw new IllegalArgumentException("assumption failed")
}

def assume(assumption: Boolean, message: => Any) {
  if (!assumption)
    throw new IllegalArgumentException(message.toString)
}

def require(requirement: Boolean) {
  if (!requirement)
    throw new IllegalArgumentException("requirement failed")
}

def require(requirement: Boolean, message: => Any) {
  if (!requirement)
    throw new IllegalArgumentException("requirement failed: "+ message)
}
// Printing and reading ----------------------------------------------

def print(x: Any) = Console.print(x)
def println() = Console.println()
def println(x: Any) = Console.println(x)
```

```
  def printf(text: String, xs: Any*) = Console.printf(text.format(xs: _*))

  // Implicit conversions ------------------------------------------------

  ...
}
```

Predefined Implicit Definitions

The `Predef` object also contains a number of implicit definitions, which are available by default (because `Predef` is implicitly imported). Implicit definitions come in two priorities. High-priority implicits are defined in the `Predef` class itself whereas low priority implicits are defined in a class inherited by `Predef`. The rules of static overloading resolution stipulate that, all other things being equal, implicit resolution prefers high-priority implicits over low-priority ones.

The available low-priority implicits include definitions falling into the following categories.

1. For every primitive type, a wrapper that takes values of that type to instances of a `runtime.Rich*` class. For instance, values of type `Int` can be implicitly converted to instances of class `runtime.RichInt`.
2. For every array type with elements of primitive type, a wrapper that takes the arrays of that type to instances of a `ArraySeq` class. For instance, values of type `Array[Float]` can be implicitly converted to instances of class `ArraySeq[Float]`. There are also generic array wrappers that take elements of type `Array[T]` for arbitrary `T` to `ArraySeq`s.
3. An implicit conversion from `String` to `WrappedString`.

The available high-priority implicits include definitions falling into the following categories.

- An implicit wrapper that adds `ensuring` methods with the following overloaded variants to type `Any`.

  ```
  def ensuring(cond: Boolean): A = { assert(cond); x }
  def ensuring(cond: Boolean, msg: Any): A = { assert(cond, msg); x }
  def ensuring(cond: A => Boolean): A = { assert(cond(x)); x }
  def ensuring(cond: A => Boolean, msg: Any): A = { assert(cond(x), msg); x }
  ```

- An implicit wrapper that adds a `->` method with the following implementation to type `Any`.

  ```
  def -> [B](y: B): (A, B) = (x, y)
  ```

- For every array type with elements of primitive type, a wrapper that takes the arrays of that type to instances of a `runtime.ArrayOps` class. For instance, values of type `Array[Float]` can be implicitly converted to instances of class `runtime.ArrayOps[Float]`. There are also generic array wrappers that take elements of type `Array[T]` for arbitrary `T` to `ArrayOps`s.

- An implicit wrapper that adds `+` and `formatted` method with the following implementations to type `Any`.

  ```
  def +(other: String) = String.valueOf(self) + other
  def formatted(fmtstr: String): String = fmtstr format self
  ```

- Numeric primitive conversions that implement the transitive closure of the following mappings:

  ```
  Byte  -> Short
  Short -> Int
  Char  -> Int
  Int   -> Long
  Long  -> Float
  Float -> Double
  ```

- Boxing and unboxing conversions between primitive types and their boxed versions:

  ```
  Byte    <-> java.lang.Byte
  Short   <-> java.lang.Short
  Char    <-> java.lang.Character
  Int     <-> java.lang.Integer
  Long    <-> java.lang.Long
  Float   <-> java.lang.Float
  Double  <-> java.lang.Double
  Boolean <-> java.lang.Boolean
  ```

- An implicit definition that generates instances of type `T <:< T`, for any type `T`. Here, `<:<` is a class defined as follows.

  ```
  sealed abstract class <:<[-From, +To] extends (From => To)
  ```

  Implicit parameters of `<:<` types are typically used to implement type constraints.