LAST UPDATED 2019-02-19

User Manual for Physical Scalars and Plotting Tools in Scala

Russell A. Paielli*

An open-source scalar package and associated software tools have been developed in the Scala programming language, including plotting tools based on the free GRACE plotting package. The scalar package represents physical scalars and can help to prevent errors involving physical units in engineering and scientific computation. The scalar package includes a complete implementation of the standard SI metric system of units and many common non-metric units. The design also allows users to easily define a specialized or reduced set of physical units for any particular application or domain. The scalar package can be used in two different modes: one mode provides unit compatibility checking but is slower, and the other mode bypasses the compatibility checks but is much faster and still prevents the most common type of unit error. Switching between the two modes requires no changes in the user's code, making it convenient and usable with no significant performance penalty for even the most computationally intensive applications.

Contents

*Aerospace Engineer, Russ.Paielli@gmail.com.

1 Introduction

Physical units and scalars are fundamental to scientific and engineering calculations and computations. Scientists and engineers learn to add, subtract, multiply, divide, and convert units, and to keep track of them with pencil and paper. When they program a computer, however, they usually drop the explicit units and leave them implicit in the actual numerical calculations, with perhaps a comment to document the units. They do that for two basic reasons: (1) software that automatically tracks and checks units is not widely standardized and well known, and (2) that software can drastically reduce computational speed. The scalar package to be discussed here addresses these problems.

As a consequence of the lack of explicit units in most scientific and engineering software, mistaken units are a common source of error. Perhaps the most common such error involves passing an angle in degrees to a trigonometric function that takes it in radians. Humans tend to think in degrees, but standard trig functions take arguments in radians, and the conversion is often forgotten until its omission is discovered after time-consuming debugging. In one case that this author is aware of, aircraft aerodynamic simulation results over a period of six months were corrupted by such an error. In another case, confusion between seconds and minutes was found in a critical section of research software three years after results from it were published.

In air traffic management (ATM), horizontal distance is usually specified in nautical miles, whereas altitude is specified in feet, hundreds of feet, or thousands of feet. Horizontal speed is usually given in knots, whereas vertical speed is usually given in feet per minute. Such units can easily get confused if comments in the code are the only mechanism for enforcing consistency. Permitted units can be restricted by software coding standards, but draconian restrictions on permitted units can be inelegant and inconvenient, and they can force error-prone conversions on input and output.

The approach taken here to prevent such confusion is to allow the user to select units that are appropriate for the job, then to track those units explicitly in software just as an engineer or scientist would do on paper. A unit definition file is used to define the preferred units and the relevant conversion factors for the particular application or domain. Two such unit definition files are included with the scalar package, one for metric and other common units, and one for traditional aviation units. These files also serve as examples for defining other sets of units.

When run in the Scala interpreter, the scalar package can serve as an interactive calculator that tracks units and checks consistency. No longer must engineers enter only the numbers into a calculator and manipulate the units separately in their heads or on paper. But the larger benefit is in automatically catching unit errors in Scala software.

The scalar package can be used in two different modes: one mode provides unit compatibility checking but is slower, and the other mode bypasses the compatibility checks but is much faster. Switching between the two modes requires no changes to client code. Thus, the full checking mode can be used during development and testing to check unit correctness, then the fast mode can be used for production runs. Although the fast mode does not check for unit incompatibility (e.g., adding a length to a speed), it still prevents the most common type of unit error, which is confusion between units of the same physical type (e.g., radians/degrees, seconds/minutes, etc.).

2 Scalar Package

2.1 Basic Usage

The compiled "jar" (Java archive) files for the scalar package are in the top-level directory with names of the form

```
physical-scalar-x.x.jar.off
physical-scalar-x.x.jar.on
```

If these jar files are not already available, they can be generated with the makeScalarJarFiles script, which is also in he top-level directory. This script invokes sbt (the Scala Build Tool), which must be installed separately. The "on" version provides full unit compatibility checks, and the "off" version bypasses those checks for better computational performance as will be explained in more detail in a later section called "Switching Modes." That section also explains the proper use of these jar files and scripts for managing them.

The Scalar class constructor is not intended to be used directly. Instead, a function called unit, which returns a Scalar object, is used to define units. Moreover, the user need not even call the unit function directly unless a new unit is needed that is not already available in the included unit definition files. The units package in the units.scala file defines a comprehensive set of standard units, including the complete SI metric system and many common non-metric units. To access those units, import the scalar and units package objects as follows:

```
import scalar_._
import units_._
```

Note also that the mathx_ package provides several useful math functions for scalars. The predefined units in the units package are as consistent as possible with standard metric unit abbreviations such as s for seconds and m for meters. Thus, for example, 23 meters per second-squared can be constructed as

```
val accel = 23 * m/sqr(s)
```

The function Scalar.unitList returns a text listing of the currently defined units. As in the standard metric system of units, the base unit for length in the units package is the meter, and several common scaled variations of it are defined:

```
val m = base_unit("m", "meter", "length")

val mm = unit("mm", m/1000, "millimeter", "length")
val um = unit("um", mm/1000, "micrometer", "length")
val cm = unit("cm", m/100, "centimeter", "length")
val km = unit("km", 1000*m, "kilometer", "length")
```

The meter definition in the first line shows the creation of a base unit using the base_unit function. This function takes three String arguments, the last two of which are optional. The first argument, m, is the abbreviated name of the unit, which is used internally as a key in a hash map of the exponents for each base unit, and it is also used for output representation. The second (optional) argument, meter, is the full name of the unit, and the third (optional) argument, length, is the unit type. The remaining four lines above show the creation of derived units based on the base meter unit. This function is the same as the base_unit function except that it takes a Scalar type as the second argument to define the derived unit. Only one base unit should be defined for each

unit type, such as time or length. When inconsistent units are added, subtracted, or compared, as in `4 * m + 3 * s`, an exception will be thrown (unless unit compatibility checks are disabled for efficiency as will be explained later).

In addition to the `units` package, another smaller package called `ATMunits`, designed for air traffic management, is also included. It provides an example of a simplified unit definition file for a particular application or domain without the many unneeded units in the `units` package. The smaller number of units and the lack of single-letter unit names makes it more manageable for large projects. Users are also free to create their own units package for their particular application of domain or to copy the units package and strip out what they don't need, of course.

A Scalar object can be converted to a basic numeric type such as a `Double` if it is dimensionless. The predefined units of `rad` (radians) and `deg` (degrees) in the units package are dimensionless, and the base unit is radians. This convention guarantees that standard trigonometric functions will be passed arguments in terms of radians, as required, and it prevents any other unit from being erroneously passed to a trigonometric function. For example, `sin(30*deg)` returns 0.5 as expected, rather than being converted to `sin(30)`, which would be wrong.

Outputs are printed by default in terms of base units, such as meters for length. For example:

```
scala> val dist = 5.227 * m
dist: Scalar = 5.227 m

scala> println(dist)
5.227 m
```

To show the value in meters, simply divide by meters:

```
scala> println(dist/m)
5.227
```

To show the quantity in a different unit, simple divide by that unit:

```
scala> println(dist/ft)
17.148950131233597
```

To format the number in a different unit, the standard Java `String` format function can be used:

```
scala> println("%2.2f ft".format((dist/ft).toDouble))
17.15 ft
```

Note that `toDouble` is needed to explicitly convert to a `Double` for the format function. To avoid that explicit conversion, a convenient function called `form` is provided in the Scalar companion object:

```
scala> println("%2.2f ft".form(dist/ft)) // conversion to feet
17.15 ft
```

The `form` function can be always be used in place of the standard String format function and is the generally recommended way to format Scalars. No function is provided to extract the numerical coefficient of a scalar independent of the units, because that would depend on the base units chosen, and the output should not depend on the choice of base units. To pass data to a third-party function or application without the units, simply divide by the required unit and convert to a `Double`, as in `(dist/ft).toDouble`.

The convention for printing compound units is to show multiplication with asterisks, division with slashes, and exponents with carats. For example, kilogram-meters/second-squared would be

shown as `kg*m/s^2`. Only one slash is used, and if the denominator has multiplied units they are placed in parentheses, as in `kg/(m*s^2)`. To allow the Scalar class to be bypassed for efficiency (to be discussed later), it has very few public member functions (other than the overloaded arithmetic operators). The reason is that some calls of member functions using standard dot notation do not work on basic numeric types. The `scalar` and `mathx` packages redefine `sqrt`, `atan2`, and other function so they work correctly for both numeric and Scalar argument types.

For convenience, an object called `zero` in the Scalar companion object is defined as the Scalar version of zero if the Scalar class is enabled or the Double version of zero otherwise. This allows the use of normal Scala type inference conventions that work for variables in both cases. For example, the line `val x = zero` is equivalent to `val x: Scalar = 0`. Note also that a scalar with a value of zero effectively has no units and can be added to or subtracted from any unit type without error.

Implicit conversions are provided from `Double`, `Float`, and `Int` to a dimensionless Scalar, so one can write, `val x: Scalar = 1`, for example. However, no implicit conversions are provided to go in the opposite direction, because that would cause problems. To convert a dimensionless Scalar to a Double, one would use the usual `x.toDouble`.

2.2 Absolute Temperatures

Temperature is unique among physical units of measure in that different systems of measurement can have not only differently scaled units but also different absolute references. A difference of a degree Celsius (or Centigrade) is equal to a difference of a degree Kelvin, but the Kelvin and Celsius temperature scales differ by 273.15 degrees Kelvin. The Kelvin and Celsius temperature scales are therefore scaled the same but have a constant offset. The Fahrenheit temperature scale, on the other hand, is scaled differently (by a factor of 9/5) than both Kelvin and Celsius and is also offset from both of them.

The following utility functions handle the constant offsets between temperature scales:

```
def convertTempKtoC(T: Scalar) = T - 273.15 * K // Kelvin to Celsius
def convertTempCtoK(T: Scalar) = T + 273.15 * K // Celsius to Kelvin
def convertTempFtoC(T: Scalar) = T - 32 * degF // Fahrenheit to Celsius
def convertTempCtoF(T: Scalar) = T + 32 * degF // Celsius to Fahrenheit
def convertTempFtoK(T: Scalar) = convertDegCtoK(convertDegFtoC(T))
def convertTempKtoF(T: Scalar) = convertDegCtoF(convertDegKtoC(T))
```

Scaling for input and output should be handled the same way it is handled for other units: by multiplying (for initialization or input) or dividing (for output) by the appropriate unit (`K`, `degC`, or `degF`.) Suppose, for example, you are using the Kelvin scale internally, but you wish to print the temperature in Fahrenheit. You could do that with

```
println(convertTempKtoF(temp)/degF)
```

2.3 Switching Modes

The Scalar class can be used as an interactive units-based calculator in the Scala interpreter shell, and it is also computationally efficient enough for many applications.[1] However, it may be unacceptably slow for many computationally intensive applications. The computational overhead of the Scalar class can make arithmetic operations more than an order of magnitude slower than corresponding operations on the basic numeric types such as `Double`.

The source of that overhead is twofold. First, the character-string manipulation involved with tracking and checking the units obviously takes some time. But just disabling the unit tracking

[1] A short script called "calculator" is provided in the home directory to start a calculator in the Scala REPL, but it may need some simple tweaks to work, depending where you placed the scalar directory.

cannot increase the efficiency to the level of the `Double` type. The Scalar class "boxes" a basic numeric type, which adds substantial overhead in the Java Virtual Machine (JVM) compared to using a basic numeric type directly even if unit checking were disabled within the Scalar class.

Fortunately, a simple method has been devised to eliminate both sources of overhead if performance is or becomes an issue for a particular application. After an application has been tested and its unit consistency verified, the Scalar class can be disabled or bypassed at compile time for production runs. During testing and development, the Scalar class can be enabled, but if the tests themselves are computationally intensive, it needs to be enabled only occasionally to detect the vast majority of unit errors.

Two versions of the source file `Scalar.scala` are provided, one called `Scalar.scalar.on` that implements the Scalar class, and the other called `Scala.scalar.off` that bypasses it. The latter file replaces the unit function with a function of the same name that simply returns the Double 1.0 rather than an instance of the Scalar class, for base units. Thus, the expression `25 * m` would be replaced with `25 * 1.0`, or `25 * km` would be replaced with `25 * 1000.0`. This replacement of the Scalar class eliminates its overhead. It may leave a few unnecessary multiplications and divisions by 1.0, but that should not cause a significant performance penalty unless they occur in high-rate, numerically intensive loops. If they do, the multiplication by the unit should be moved outside the loop if possible.

For convenient switching between modes, two one-line bash scripts called `scalar-on` and `scalar-off` are provided to create symbolic links to the selected source file. The `scalar-on` script contains:

```
ln -sf Scalar.scala.on Scalar.scala
```

and the `scalar-off` script contains:

```
ln -sf Scalar.scala.off Scalar.scala
```

After either of these scripts is run, a complete recompilation should be executed (first deleting all previously compiled bytecode).

A more convenient way to switch modes is to use the `makeScalarJarFiles` script in the scalar home directory. This script creates two jar files, also in home directory, called

```
physical-scalar-x.x.jar.off
physical-scalar-x.x.jar.on
```

These jar files can be placed in the `lib` directory of the client application and selected as needed by making a symbolic link named `physical-scalar.jar` to the desired version. After switching jar files, a complete clean and recompilation may be necessary. Also, the user should make sure that no conflicts exist on the Java class path (the `CLASSPATH` environment variable). (It is prudent to run each project that uses the scalar package using a script that defines a complete class path for that particular project rather than relying on a general class path defined in a shell initialization file such as `.bashrc`). For convenience, a simple script called `selectScalarJarFile` can be written containing the following:

```
#!/usr/bin/env bash
ln -sf $1 physical-scalar.jar
```

In practice, multiplication by units tends to occur mainly on input, and division by units tends to occur mainly on output, so neither typically occur in high-rate loops. Assuming no unit inconsistencies and the observance of a few basic rules, all outputs with the Scalar class disabled should be identical to what they were with it enabled. The main rule is that output of Scalar objects

should be done using the format function rather than println, or they should be made dimensionless for output by dividing by the appropriate unit and converting to a Double, as explained above.

To quantify the speedup resulting from disabling the Scalar class, a basic timing test was done in which scalars were added one hundred million times. With the Scalar class enabled, the time required on a Sun Ultra 24 Linux workstation was approximately 47.4 seconds. When the Scalar class was switched off, the computation time dropped to approximately 1.38 seconds, for a speedup factor of approximately 34. Similar results occurred for multiplication.

Note that most of the benefits of the scalar package are still available even when the Scalar class is bypassed for efficiency. The only thing missing with the Scalar class bypassed is the automatic checking for addition, subtraction, or comparison of scalars with incompatible units. With the Scalar class bypassed, seconds can be added to meters without raising an exception, for example. But with the Scalar class disabled, units are still automatically converted to standard base units, so the user need not worry about base units or conversion factors. Thus, confusion between different units representing the same physical quantity will still be avoided, and those kinds of errors are probably much more common than adding or subtracting incompatible units. Confusion between degrees and radians will still be avoided, for example, as will confusion between seconds and minutes or feet and meters.

If you don't want to be bothered with switching between the enabled and the disabled versions, at least initially, here some simple guidelines. If your application is not computationally intensive, you can simply always use the enabled Scalar class. If your application is computationally intensive, you can simply use the disabled version and not worry about it. As explained above, you will still get most of the error-reducing benefits. If and when you get time to check for unit consistency, you can always enable the Scalar class and run your application. Even if you never do that, you will still be far ahead of using basic numeric types.

2.4 Static vs. Dynamic Checking of Units

When the Scalar class is enabled, the unit compatibility checks are done dynamically (i.e., at run time) rather than statically (by the compiler). Static checking of unit compatibility by the compiler would be desirable, but it may not be possible to do efficiently in Scala.

Jesper Nordenberg has developed an innovative static implementation of physical units in Scala as part of his Metascala project. According to my testing, however, it slows computation by a factor of approximately 27 compared to using basic numeric types such as `Double`. For numerically intensive computation, that is unacceptable. Even though the unit consistency checks are done at compile time, the fact that basic numeric types are "boxed" apparently makes them inefficient on the Java Virtual Machine (JVM).

Another problem with Nordenberg's implementation is that it is based on meta-programming at the level of the Scala type system, which is not Scala programming in the usual sense and is very awkward. That makes customization very difficult. Nordenberg's scalar objects do not show any unit information when printed, for example, and I could not figure out how to show their units.

3 Plotting

3.1 Plot Class

The Plot class in the `tools_` package is an interface to the GRACE plotting software package. GRACE is a free, full-featured plotting program that runs on Linux and other Unix-based operating systems (and has also been ported to Windows). It can be used both interactively and in batch mode. The Plot class makes use of the GRACE 5.x series (not 6.x). A default template file called Default.agr in the `.grace/templates` directory under the user's home directory sets the default parameters for plotting.

The Plot class header is shown below. Each `Plot` is associated with a file, the name of which is the first argument of the constructor. The constructor takes several arguments as the header shows above. The only mandatory argument is the `fileName`, the name of the file that the plot will be stored in. The optional arguments include a title, a subtitle, axis labels, and axis units. The plot is automatically scaled to the specified units. The `xref` argument can be used to offset the x axis by some reference value. This offset feature is particularly useful for times because it allows some significant time to be the zero reference time. The `grid` argument can be set to true to show grid lines for easier reading of values from the plot.

```
class Plot(fileName: Text, title: Text="", subtitle: Text="",
  xlabel: Text="x", ylabel: Text="y", xunit: Scalar=1, yunit: Scalar=1,
  xref: Scalar=0, grid: Bool=false,
```

Functions are also available for setting the title, subtitle, and axes labels after the plot is opened, or for appending to the subtitle after the plot is opened, which is sometimes useful.

The most important methods are the target method, which is used to create a new curve on the plot, and the `Point` and `point` methods for adding a point to a curve. The `close` method must be invoked to close the Plot file before the plot can be viewed.

```
  def target(lineStyle: Text="solid", lineWidth: Real=1, symbol: Text="",
    size: Real=0.5, color: Text="black", legend: Text="", putInBack: Bool=false)
```

The header of the target method shown above has several arguments, all with default values, for specifying the characteristics of the curve, including the `lineStyle`, symbol, and other features. The `lineStyle` defaults to `''solid''`, but can also be `''none''`, `''dash''`, `''dot''`, `''longdash''`, and `''dotdash''`. The symbol type defaults to `''none''`, but it can also be `''circle''`, `''square''`, `''+''`, `''x''`, and `''diamond''`, among other symbols. The color defaults to `''black''` but can also be `''red''`, `''blue''`, `''green''`, and `''gray''`, among other colors.

After a target has been declared, the actual points are added using either the `Point` method or the `point` method. The `Point` method takes the x and y coordinates as arguments (each scalars), and the `point` method takes position and other "2D" types as arguments. For example:

```
val plot = new Plot(fileName, ...)

plot.target()

for (track <- tracks) plot.Point(track.time, track.alt)

plot.close
```

The Plot class automatically sets axes ranges and margins, but methods are also provided for setting them manually if necessary. The Plot class also has a method called `setAxisScalesEqual`

to make the x and y axis scales equal. This method is intended for planview map plots where the map would be distorted if the axes had different scales. It should be called *after* all data points are written to the Plot file. The `legend` function places a legend on the plot at the specified location, using the legend names specified in the `target` function explained above.

The Plot class also has methods for adding lines, circles, ellipses, and text to the plot. The headers for some of these functions follow.

```
  def line(x0: Scalar, y0: Scalar, x1: Scalar, y1: Scalar,
    style: Text="solid", color: Text="black", width: Real=1,
    arrow: Int=0, arrowtype: Int=1, arrowlength: Real=2, label: Text="")

  def circle(xc: Scalar, yc: Scalar, radius: Scalar,
    style: Text="solid", color: Text="black") { // draw a circle

  def text(text: Text, x: Scalar=0, y: Scalar=0, color: Text="black",
    loctype: Text="world", size: Real=1, just: Int=0, outOfFrame: Bool=false)
```

Finally, the Plot class has a `close` function which must be called to properly close the Plot file. Failure to close the file will usually result in an incomplete file (as is usual in Java and Scala).

The Plot class can be used directly or it can also be used as the base class for more specialized case classes for particular types of plots. For example, the header for the `RoutePlot` case class, for flight routes, is shown below. It provides convenient defaults that allow the user to quickly produce a route plot without having to specify all the labels and units each time. Similar classes can also be defined for altitude plots and many other plot types as well.

```
case class RoutePlot(fileName: Text, title: Text="Routes", subtitle: Text="",
    showTRACON: Bool=true, scaleToTRACON: Bool=false)
  extends Plot(fileName, title, subtitle, xlabel="x position / nmi",
    ylabel="y position / nmi", xunit=nmi, yunit=nmi) {
```

Functions can also be defined for plotting. The `plotAltitude` method header is shown below as an example. It takes a `Plot` argument as its first argument, and it plots the altitude profile for the trajectory on that `Plot`. The other arguments all have default values and are therefore optional. The `altitudePlot` method takes a file name as its first argument, then opens a `Plot` and passes it to a call to the `plotAltitude` method. Setting the `close` argument to false leaves the `Plot` (and the underlying file) open for more data to be added if necessary.

```
  def plotAltitude(plot: Plot, symbol: Text="", lineStyle: Text="solid",
    color: Text="black", lineWidth: Real=1)

  def altitudePlot(fileName: Text, close: Bool=true): AltitudePlot
```

3.2 PlotManager Class

The `PlotManager` class is designed to produce and manage sets of plots. It uses the GRACE plotting tool, the LaTeX typesetting program, and the Plot class explained above, to combine multiple plots into a single PDF (Portable Document Format) file. It is useful for managing groups of plots, and it can also be used to produce stacks of traffic plots that can be stepped through to simulate a traffic "movie."

The `PlotManager` constructor requires a `name` argument, and the plots that will be combined into the output file must be in files that start with that name but are unique, as in the following example. The plots will appear in the lexicographic order of the corresponding file names, so if order is important, the file names should constructed to have the desired order.

```
val plotManager = PlotManager(testName)

for (traj <- trajList) { // for each trajectory in a list
  val fileName = s"$testName-${traj.label}.dat"
  traj.altitudePlot(fileName)
  }

plotManager.combinePlots() // create pdf file
```

The `combinePlots` function creates the output PDF file and deletes the intermediate files (.dat, .ps, and .tex). The intermediate files can be saved if necessary (for debugging or other purposes) by setting the `save` argument of the `combinePlots` function to true.

4 Other Tools

The `types_` package object contains several type aliases and convenience functions. Most notably, it contains the following aliases for commonly used types:

```
type Real = Double // better name for Double
type Bool = Boolean // shorter name for Boolean
type Text = String // better name for String

def not(b: Bool) = ! b // clearer form of not!
```

A few other utility classes are also provided in the `tools_` package.

4.1 CommandLine Class

The `CommandLine` class in the tools package parses command-line arguments in a style similar to scala method calls with named arguments. For example, a program could be invoked with the command line

```
myProgram x=123 22 zz=blah 43.5 y=t gg rr=3.445
```

The space-delimited fields can be either assignment-style options (if "=" is used with no spaces) or regular arguments. The regular and assignment-style options can be interwoven arbitrarily. The assignment-style options are accessed by the name on the left side of the assignment (a String), and regular arguments are accessed by an integer index (starting with 1). Methods are provided to read arguments of type Int, Real (Double), Text (String), and Bool (Boolean). Default values can be provided, making the argument optional to specify on the command line.

The user can specify an optional list of valid options if desired, in which case an Exception will be thrown if an invalid option is detected. This catches misspellings on the command line. Alternatively, the method "warnOfUnusedOptions" will accomplish essentially the same thing as using an explicit list of valid options. However, it must be called after all the command-line arguments are read in the client program.

Here is an example of its usage (based on the command line shown above):

```
def main(args: Array[Text]) {

  val options = Set("x", "y", "zz", "rr", "k", "a") // valid options
```

```
val com = CommandLine(args, options) // "options" is optional!

val x = com.Int("x") // read "x" and convert it to an integer
val k = com.Int("k", -33) // -33 is the default value if "k" is not set
val rr = com.Real("rr") // read "rr" and convert it to a Real (Double)
val zz = com.Text("zz") // read "zz" as Text
val a2 = com.Real(2) // regular argument accessed with Int index
val y = com.Bool("y")
val a1 = com.Int(1) // regular argument with no default
val a3 = com.Int(5, 22) // regular argument with default value 22

com.warnOfUnusedOptions // list of valid options not needed!
```

4.2 ConfigReader Class

The `ConfigReader` class reads an optional configuration text file if available and sets parameters accordingly. If the configuration file is not found (or is empty), the program continues uninterrupted. It provides a method to check that all provided parameters are read, which is useful for detecting typos in the configuration file. The configuration text file should contain plain assignment statements (with no val/var keyword), and it can use any units that you might define if you use the Scalar class. For example

```
weightLimit = 200 * kg
```

4.3 PrintWriterx Class

The `PrintWriterx` class extends the standard Java PrintWriter class to add some convenient features. For example, it can take multiple `String` arguments and print them separated by a designated separator, which defaults to a space if not specified.