A Java Extension With Support for Dimensions

André van Delft
Published in Software - Practice and Experience 29(7), Wiley, 1999

Summary

We present an extension to the Java language with support for physical dimensions and units of measurement. This should reduce programming errors in scientific and technological areas. We discuss various aspects of dimensions and units, and then design principles for support in programming languages. An overview of earlier work shows that some language extensions focused on *units*, whereas we argue that *dimensions* are a better starting point; units can then simply be treated as constants.
Then we present the Java extension, and show how to define and use dimensions and units. The communication between the program and the outer world gets special attention. The programmer can still make dimensional errors there, but we claim the risk is reduced.
It has been simple to build support for this extension into an existing Java compiler. We outline the applied technique.

Introduction

Computing in scientific and technological areas largely deals with manipulating numbers that represent physical dimensions, such as time, mass, and force. A common source of errors is to confuse quantities that are related to different units of measurement. For instance, in a chemical engineering simulation program, a variable representing a number of kilograms of a substance, may be accidentally be taken as a number of grams, a number of liters, or a number of moles. It is possible to extend existing programming languages so that many of such errors are prevented, by providing support for dimensions and units of measurement. We did so for the Java language; our approach seems applicable for other strongly typed languages as well.

Dimensions and Units

A dimension is a class of similar scales, comparable to the notion of type in programming languages. A unit is a specific instance of a dimension; it serves as a reference for measuring quantities of the same dimension. In fact, any nonzero instance of a dimension can serve as a unit, even if it is not constant, such as a person's length.
Some physical dimensions may be considered more basic than others. There are 7 base dimensions for physics defined in the International System of Units (SI): length, mass, time, electric current, thermodynamic temperature, amount of substance, luminous intensity. For each of the SI base dimensions a primary unit exits: meter, kilogram, second, Ampere, Kelvin, mole, Candela. One may also think of different base dimensions, such as money (with unit Dollar, for instance) and amount of storage in computing (with unit byte).
Multiplication and division of dimensions yield other dimensions; some of these dimensions have special names, e.g.,

- length divided by time equals speed
- speed divided by time equals acceleration
- mass times acceleration equals force
- speed times speed is also a dimension; yet we don’t have a special name

Multiplication and division of units yields other units:, e.g.,

- kilogram times meter divided by (second times second) equals Newton
- Meter divided by second (or: meter per second) is also a unit, though we don’t have a special name

It is also possible to multiply or divide a unit by a number; the result can also be used as a unit, as in:

- A meter divided by 100 gives a centimeter
- 2.54 centimeters equals an inch

Multiplication and division are allowed on values of any dimension. Addition, subtraction and comparison only make sense when applied on values with equal dimension. In a programming language with proper support for dimensions and units, we would be allowed to add directly 2.0 meter to 3.0 centimeter, yielding 2.03 meter. Without the support,

we should be cautious not to add a number of meters (2.0) directly to a number of centimeters (3.0); we should apply an explicit conversion instead: 2.0+0.01*3.0.
Some units have a false origin, such as degrees Celsius and degrees Fahrenheit when used for absolute temperatures. E.g., zero degrees Celsius as an absolute temperature means 273 Kelvin, whereas zero Kelvin is the lowest thinkable temperature. An addition of two temperatures expressed in Celsius raises suspicion. E.g., should 0 degrees Celsius plus 0 degrees Celsius equal 0 degrees Celsius or 273 Celsius? Such an addition makes sense in the context of computing the mean of two absolute temperatures, as in (t1+t2)/2. Other appropriate contexts are the addition of two temperature differences, and of a difference and an absolute temperature. It cannot be left to a compiler to decide whether such a context applies. Therefore dedicated support in the programming language for false origins could turn out to be confusing; for sake of simplicity, handling false origins may be left to conversion functions.

Design Principles

There are many ways to add support for dimensions and units to existing programming languages; to judge the usefulness one may consider the general set of design principles by Bentley [1] for programming languages; these principles also apply for extensions with dimension support:

- *Design goals*: extend a widely used programming language with sound support for physical dimensions and units of measurement, without performance penalty.
- *Simplicity*: the extension must be simple, both for the user and for the implementation.
- *Fundamental Abstractions*: it should be possible to define base dimensions with corresponding base units, and derived dimensions. Other units may be expressed as constants in terms of base units.
- *Linguistic structure*: to express derived dimensions it should be permitted to write plain multiplications and divisions of dimensions
- *Yardsticks of language design*: desirable properties are:
  - *Orthogonality*: the notion of dimension is in principle independent of numeric precision (integer, floating point).
  - *Generality*: it should be possible to have generic functions that operate on parameters of any dimension, e.g., to summarize arrays.
  - *Parsimony*: introduce as few new keywords as possible; prevent large changes to the syntax of value expressions.
  - *Completeness*: multiplying and dividing dimensions should always yield other valid dimensions.
  - *Similarity*: dimensions may be multiplied and divided, analogous to numbers.
  - *Extensibility*: rather than this property, we'll strive for the inverse property *Conservation* since we are already dealing with a language extension: existing programs in the base language should also be valid in the extension, except for the use of identifiers that become new keywords.
  - *Openness*: Functions in external libraries that are not aware of dimensions must be accessible.
- *The design process*: experience with an implementation may change the design.
- *Insights from compiler building*: dimensions may possibly have a different name space than other language constructs such as variables and types. For parsing programs the set of valid dimension names should not need to be known in advance.

Earlier work

In an early paper [2] Karr and Loveman make an important remark:
The essential point for us is that units may be carried along in calculations, where they act like "variables", obeying commutativity, associativity, laws of exponents, etc.:
G= 32(feet/sec)/sec = 32 feet/sec^2.
From the point of view of calculating, we can actually regard the "32" as being multiplied by the expression "feet/sec^2".

Later the paper states that *syntactically there need be no distinction between a variable and a unit*. This agrees with our earlier statement that any instance of a dimension can serve as a unit. Then Karr and Loveman suggest expressions such as

```
2*WEEKS + 5*DAYS
```

So units can be incorporated without change in expression syntax. An alternative syntax would allow:

```
2 WEEKS + 5 DAYS
```

This binding by a space adds to syntactic complexity; that should be avoided especially because it is not necessary. It won't take programmers too long to get used to writing an asterisk between numbers and units. Since units can be viewed as variables (usually constants), language support for dimensions should not focus on units. Unfortunately, the remarks by Karr and Loveman have had little impact in most subsequent proposals to extend programming languages with dimension support.

House [3] starts from unit definitions. He confuses units with dimensions in the quote:

```
dim volume: metre**3;
   newton: metre*kg*sec**(-2);
```

House's proposal allows for generic functions such as

```
function SQRT (x: real newdim whatever): real dim whatever**(1/2);
```

This comes at the significant cost of introducing *dimension variables*, a kind of identifiers next to variables and functions; they are in fact not needed to allow for generic functions. House mentiones the issue of access to existing libraries that are not aware of dimensions, but he fails to address it.

Gehani [4] also emphasizes on units, as in his example

```
V: FLOAT {MILES HOUR-1};
A: FLOAT {MILES HOUR-2};
T: FLOAT {HOUR};
S: FLOAT {MILES};
S = V*T + 0.5*A*T*T;
```

Gehani does not give syntax for defining units.

Männer's [5]. offers starting language constructs to define units. He confuses the concepts unit and dimension as witnessed by the quote: *"…CallDistance has the dimension cm"*. His examples suggest that his proposal only allows dimension types to be specified as subrange types, such as

```
TYPE Energy = 0 Mev .. 1e3 MeV;
```

Dreiheller [6] also start from unit definitions; the supporting language constructs are rather complicated. Dimension types may be specified in phrases such as

```
TYPE Energy = REAL [MeV];
```

A better way to define energy would be as a definition in terms of base dimensions length, mass and time.

Baldwin [7] fails to provide syntax and programming examples, so his contribution is unclear.

More recently Wand and O'Keefe [8], Goubault [9] and Kennedy [10] have tackled dimension type *inference*, determining polymorphic dimension types in the absence of programmer-provided type information. This naturally fits in the style of the language ML on which their work is based; it fits less in the style of widely used programming languages. Kennedy [11] presents an ML extension with a sound treatment of units and dimensions. This also treats units and variables as syntactically equal, so expression syntax does not get complicated. However, it does not allow for the specifications of derived dimensions such as speed being equal to length divided by time. It also fails to provide access to existing libraries without dimension support.

The Java extension

Why Java

Java [12] is a quite new, but already widely used programming language that offers the benefits of object-oriented programming with modest complexity. It is as if there is some space left in the language definition for small extensions.

A Java compiler is relative simple, since it produces byte codes that are on a higher level than the usual machine code. Byte codes are collected in so-called class files. These files have an extendible format: specific compilers may include additional information of any desired type in class files using class file attributes. This eases implementations of Java language extensions.
A pragmatic reason to take Java was that we had a compiler available that we had developed for Scriptic, a Java extension with parallel constructs [13].

Dimensions as members of classes and interfaces

Java programs contain classes and interfaces that are grouped in packages. A class is an aggregate of class members: variables and functions; in Java jargon, these are called fields and methods. A class is also a type from which instance objects may be created. An interface is also an aggregate of variables and methods, but the variables must be static (i.e. shared over all instances) and the methods are all abstract (i.e. the headers are specified only, without statement bodies). Moreover, it is not possible to create instance objects directly from interfaces. Classes and interfaces are situated in an inheritance hierarchy.
Introducing dimensions as a new kind of members of classes and interfaces benefits from the inheritance mechanism. Moreover, since the compiler translates classes and interfaces into class files, dimensions as new members may be compiled into class file attributes.

Base dimensions and base units

In our extension, the **dimension** *dimensionName*(*unitName*) construct specifies the base dimensions with associate base units. The parentheses are merely syntax to visualize the association.
Inside an interface named Dimensions in a package named physics, we can have:

```
dimension Length            (meter   );
dimension Mass              (kilogram);
dimension Time              (second  );
dimension ElectricCurrent   (Ampere);
dimension Temperature       (Kelvin  );
dimension AmountOfSubstance (mole    );
dimension LuminousIntensity (Candela);
```

Derived dimensions

The **dimension** *dimensionName* = *dimensionExpression* construct covers the definition of derived dimensions. The *dimensionExpression* has a sequence of other dimension names separated by asterisks and slashes. Rational exponents are not allowed there: these would require unnecessary syntactic complexity; moreover, they would allow for dimension definitions as combinations of base dimensions with non-integer exponents, without physical meaning. The *dimensionExpression* may also start with "1" and then a slash, to deal with such dimensions as Frequency:

```
dimension Speed        = Length / Time;
dimension Acceleration = Speed  / Time;
dimension Impulse      = Mass   * Speed;
dimension Force        = Mass   * Acceleration;
dimension Frequency    = 1 / Time;
```

Numeric types and dimension specifiers

A numeric type may now be followed by a dimension specifier: an alternating sequence of operators (asterisks or slashes) and dimension names. The optional array brackets in the type appear now after the dimension specifier.
So an application program may declare:

```
double * Time           t;
double / Time           f;
float    * Length / Time  v;
double * Time []        tt;
double * Time           ttt[];
```

Alternatives such as Time * **double** are prohibited since those would not add much worth while complicating both parsing by compilers and inspection by humans. A numeric variable declared without dimension expression is said to have an *empty* dimension.
Dimension specifiers are not considered to be part of the numeric type; they are independent extra information.

Method declarations

Like elsewhere, numeric types in method declarations may also be followed by dimension specifiers, as in:

```
abstract double*Speed divide (double*Length, double*Time);
```

The original Java language specification defines: *The signature of a method consists of the name of the method and the number and types of formal parameters to the method.*. This definition does not change; hence a method signature does not contain dimension specifiers of the formal parameters. Instead the dimension specifiers constitute a rather independent *dimension signature* for the method. An unchanged rule from the Java language specification is: *A class may not declare two methods with the same signature, or a compile-time error occurs*. So adding another divide method with different dimensions would cause a *function already defined* error:

```
abstract double*Acceleration divide(double*Speed, double*Time);//error
```

Instance methods may override others in superclasses and interfaces that have equal method signatures. A notable restriction is that the return types are equal. An extra restriction is now that the dimension signatures are equal, as well as the dimension specifiers of the return types.
Java's original resolution rules for method calls involve the following compile-time steps:

1. *Determine Class or Interface to Search*
2. *Determine Method Signature*

2.1 *Find Methods that are Applicable and Accessible*
2.2 *Choose the Most Specific Method*

3. *Is the Chosen Method Appropriate*

Step 2 involves only parameter types, not the return types, and not the dimension specifiers. Step 3 has checks dealing with staticness; it also requires that a void method may only be called in a top level expression. In this step, an extra requirement is now that the dimensions of the formal and actual parameters match.

Expressions

Expressions must be dimensionally consistent. Both sides of assignments, additions and subtractions must have equal dimensions. Multiplications and divisions result in appropriate dimensions:

```
double * Time     t;
double * Length   s;
double * Speed    v;
t = 18.3*second;  // t = 18.3; would be a compile error
s = 26.4*meter;   // s = 26.4; would be a compile error
v = t/s;
```

Derived units

Units that are not defined with the base dimensions may be regarded as normal constant values that are initialized using dimensionally sound expressions. The interface physics.Dimensions may have:

```
final int      * Time        minute = 60 * second;
final int      * Time        hour   = 60 * minute;
final int      * Frequency Hertz  = 1 / second;
final double * Mass        gram   = kilogram / 1000.0;
```

As a matter of taste one could define well established abbreviations and prefixes for units, as in:

```
final int            kilo        = 1000;
final int            mega        = 1000*kilo;
final double         milli       = 1.0 / 1000;
final double         micro       = milli / 1000;
final double         nano        = micro / 1000;
final int * Length  m            = meter;
final int * Length  kilometer  = kilo * meter;
final int * Length  km           = kilometer;
```

String conversion

The Java string concatenation operator '+' automatically converts numbers into strings. These numbers must have empty dimension. So the following would be illegal:

```
double * Time   t = 18.3*second;
double * Length s = 64.2*meter;
System.out.println ("speed: " + (s/t)); // compile error
```

We first need to "divide off" the dimension present in the expression s/t. This can be done by dividing it by a unit of equal dimension, for instance kilometer/hour; the result then represents the number of kilometers per hour:

```
System.out.println ("speed: "+ ((s/t) / (1.0*kilometer/hour)) + " km/h");
```

Allowing for "speed: "+(s/t)+" km/h" would have been error prone; the programmer could wrongly assume that a number of kilometers per hour would appear in the string, instead of a number of meters per second.
In dividing off the dimension the programmer explicitly states the unit that holds for the number being appended to the string.

Generic dimension specifiers

To compute a sum of speeds, we could write:

```
double*Speed summarize(double*Speed speeds[]) {
   double*Speed result=0*meter/second;
   for (int i=0; i<speeds.length; i++) result += speeds[i];
   return result;
}
```

This would work, but we do not want to write such a method for each kind of physical dimension for which we may have to compute sums. Besides, this could easily lead to *method already declared* errors.

We want to specify just one generic summarize method, applicable for all those dimensions; summarize accepts an array of dimensioned double numbers and returns a double scalar value of the same dimension. This is solved by some extra syntax for parameter declarations and return type declarations:

- **double*dimension** p - parameter p is a number of any dimension with double precision
- **dimension**(*valueExpression*) - the dimension of *valueExpression*. The expression must be numeric or a numeric array type. If this construct is used for a formal parameter in a method header, then the expression may refer to other parameters appearing more to the left. When used for a return type or a local variable, references to any parameter are allowed.

So we can write a more general summarize method:

```
double*dimension(v[]) summarize (double*dimension v[]) {
   double*dimension(v[]) result=0;
   for (int i=0; i<v.length; i++) result += v[i];
   return result;
}
```

The variable result has a non-empty dimension; it is initialized with 0. This is allowed since 0 now by definition compatible with any dimensioned number: the zero is *polymorphic*.

Dimension casting

Occasionally we may want to use existing library mehtods that are not aware of dimensions. Suppose this is a library method to use:

```
double [][] multiply (double[][] a1, double[][] a2)
```

This calls for the ability to cast expressions to different dimensions, also applicable to arrays. The syntax is:
(**dimension** *dimensionExpression*) *valueExpression*

Examples:

```
(dimension Speed) 1
(dimension dimension(1)) anArray
```

The latter example also applies the construct **dimension**(*valueExpression*) in the *dimensionExpression*. A shorthand syntax for such combined use is:

(**dimension** (*valueExpression*)) *valueExpression*

This yields:

```
(dimension (1)) anArray
```

The shorthand syntax saves an occurrence of the word **dimension**. The pair of parentheses around the *valueExpression* could be made optional, but that would come at the cost of merging the name spaces for variables and dimensions.
Now we can build a wrapper method for multiply. It has the same methodsignature and a generic dimension signature. The method body casts the parameters in the call to multiply into an empty dimension; the return value is casted back to the product dimension of the parameters:

```
double*dimension(a1[0][0]*a2[0][0])
multiplyD (double*dimension[][] a1, double*dimension[][] a2) {
  return (dimension(a1[0][0]*a2[0][0])) multiply((dimension(1))a1,(dimension(1))a2);
}
```

Dimension casting may lead to misuse. To avoid the need of casting, new library methods should have generic dimension specifiers, like:

```
double*dimension(v[]) summarize (double*dimension v[])
```

Temperatures

The extension does not provide specific support for false origins, such as with degrees Celsius and degrees Fahrenheit when used for absolute temperatures. Extra classes may handle false origins, e.g. a class Temperature in the package physics:

```
package physics;

public class Temperature implements Dimensions {

 public static double*Temperature zeroDegreesCelsius = 273.15*Kelvin;

 public static double toAbsoluteDegreesCelsius (double*Temperature t){
     return t-zeroDegreesCelsius;
 }
 public static double*Temperature fromAbsoluteDegreesCelsius(double t){
    return t + zeroDegreesCelsius;
 }
   ... likewise for Fahrenheit ...
}
```

There is no conflict between the class named Temperature and the dimension named Temperature, since different name spaces are involved.

Square root

The current approach leaves no way to specify a generic square root method. Using an extended syntax one could write with rational exponents:

```
double*dimension(v)^(1/2) sqrt (double*dimension v);
```

Then this method could accept any argument having a dimension that is actually a square of another dimension. However, the cost of additional syntactic complexity may not justify the feature of rational exponents. An alternative approach would allow for:

```
double*dimension sqrt (double*dimension(return*return) v);
```

Here **return** in the expression of the **dimension**(*valueExpression*) construct would refer to the dimension of the method return value. For the time being support for this feature will not be implemented, until it proves to be needed in practice.

Implementation

Dimension descriptors

The implementation of the dimensions system in Java is largely based on *dimension descriptors*, i.e. strings that describe the dimensions of variables, expressions, method parameters and method return types. Dimension descriptors have been inspired on field descriptors and method descriptors that occur in Java class files to describe types of variables, parameters and return values. E.g., the descriptor for

```
Object m(int i, double[] j, java.util.Hashtable)
```

is:

```
(I[DLjava/util/Hashtable;)Ljava/lang/Object;
```

A dimension descriptor for an acceleration in terms of base dimensions in the interface physics.Dimensions is:

```
2D1physics/Dimensions.Length;D-2physics/Dimensions.Time;
```

- the leading "2" says the dimension is composed of two base dimensions
- the D's introduce qualified dimension names together with associated powers; the names are ordered alphabetically for sake of easy processing
- the semicolons delimit the names.

The dimension descriptor of a method is, roughly speaking, a concatenation of the dimension descriptors of its parameters and the return value, as far as these are of numeric type; “0” is taken instead of an empty string for a dimensionless numeric parameter. A parameter with a free dimension gets as dimension descriptor: "V*n*;" with n a sequence number, e.g., "V2;". A reference to a power of the dimension of such a parameter is be "W*power*V *n*;". For example, the two methods

```
double*report (int i, double*Speed speeds[]);
double*dimension(a1[0][0]*a2[0][0])
    multiplyD (double*dimension[][] a1, double*dimension[][] a2);
```

have dimension descriptors:

```
0;2D1physics/Dimensions.Length;D-1physics/Dimensions.Time;0
V1;V2;2W1V1;W1V2;
```

Class file attributes

The implementation requires two new kinds of attributes in class files:

- Dimensions – this attribute is optional for the class or interface. It contains a table with dimensions that have been locally defined. Each entry contains a bit set for the access modifiers (private, public or protected), the name of the dimension, and a string. If the latter string starts with a digit, it denotes a dimension descriptor and the dimension is a compound dimension. Otherwise the dimension is a base dimension and the string is the name of the corresponding unit.
- Dimension – this attribute is optional for each member variable and method. It gives the dimension descriptor.

Compilation and execution

The compiler translates base units into integer constants with value 1. It also processes the new types of class file attributes, so that it is aware of dimensions and dimension signatures in earlier compiled classes and interfaces. The compiler signals dimensional errors verbosely, e.g. when an acceleration is passed as parameter to a method that expects a speed:

Incompatible dimension: physics.Dimensions.Length*physics.Dimensions.Time^-2
formal: physics.Dimensions.Length*physics.Dimensions.Time^-1

The Java virtual machine does not need to know about the dimensions. Programs run without performance penalty, except possibly for some more multiplications with units that are not translated into constants with value 1. A debugger that is not aware of the dimensions system shows dimensioned numbers relative to the units of the base dimensions.

Effort

Using the convenient definition of dimension signatures, the implementation in our existing Java compiler was relatively straightforward and easy: it took about 8 working days, resulting in approximately 2000 lines of code. 275 of these lines are in the parser, or 8% of the total parsing code. Another 275 lines implement essential utility functions, e.g. to compute the product of 2 dimension descriptors (more precisely: the dimension descriptor corresponding to the product of 2 dimensions that are given by their descriptors).

Conclusion

We have presented a simple extension to Java that can prevent typical types of errors involving dimensions and units of measurement. Errors remain possible in areas of input and output, such as string conversion, access to external libraries, interprocess communication and database access. This seems to be inevitable for any language extension supporting dimensions. Proper use of dimension casting and dividing-off units may minimize error-proneness.
The extension has been easy to implement in an existing Java compiler. There does not seem to be a fundamental objection to extend other languages such as C++ likewise.

References

1. J. Bentley. Programming pearls. *Communications of the ACM*, 29(8); 711-721, August 1986.
2. M. Karr, D.B. Loveman III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5): 385-391, May 1978.
3. R.T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4): 366-374, 1983
4. N.H. Gehani. Ada's derived types and units of measure. *Software – Practice and Experience*, 15(6):555-569, June 1985.
5. R. Männer. Strong typing and physical units. *SIGPLAN Notices*, 21(3):11-20, March 1986.
6. A. Dreiheller, M. Moerschbacher, B. Mohr. PHYSCAL – programming Pascal with physical units. *SIGPLAN Notices*, 21(2):114-123, December 1986.
7. G. Baldwin. Implementation of physical units. *SIGPLAN Notices*, 22(8):45-50, August 1987.
8. M. Wand and P. M. O'Keefe. *Automatic dimensional inference*. In J.-L. Lassez and G. Plotkin, editors, Computational Logic: Essays in Honor of Alan Robinson, pages 479-486. MIT Press, 1991.
9. J. Goubault. *Inference d'unites physiques en ML.* In P. Cointe, C. Queinnec, and B. Serpette, editors, Journees Francophones des Langages Applicatifs, Noirmoutier, pages 3-20. INRIA, 1994.
10. A.J. Kennedy. *Dimension Types.* In Proceedings of the 5th European Symposium on Programming, volume 788 of Lecture Notes in Copmuter Science, pages 348-362. Springer-Verlag, 1994.
11. A.J. Kennedy. *Programming Languages and Dimensions*, Ph.D. Thesis, Computer Laboratory, University of Cambridge, 1995. Available as Technical Report No. 391
12. J. Gosling, B. Joy, G. Steele. *The Java Language Specification*. The Java Series, Addison-Wesley
13. A. van Delft. *Scriptic: Parallel Programming in Extended Java*, Proceedings of the WoTUG Parallel Programming and Java Conference, Enschede 1997.