Seeking Grace: A new object-oriented language for novices

Andrew P. Black
Portland State University
black@cs.pdx.edu

Kim B. Bruce
Pomona College, CA
kim@cs.pomona.edu

Michael Homer
Victoria University of Wellington
mwh@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Amy Ruskin
Pomona College, CA
asr02010@pomona.edu

Richard Yannow
Pomona College, CA
rmy02010@mymail.pomona.edu

G is Grace, the Flaming Star is the Torch of Reason. Those who possess this knowledge are indeed Illuminati.
Adam Weishaupt

Grace is the absence of everything that indicates pain or difficulty, hesitation or incongruity.
William Hazlitt

ABSTRACT

We are designing a new object-oriented language, Grace, whose target audience is novices. Grace is intended to integrate proven new ideas in programming languages into a conceptually simple language that allows the instructor and students to focus on the essential complexities of programming, while supporting a variety of approaches to teaching.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: computer science education

General Terms

Languages, Design

Keywords

Grace

1. INTRODUCTION

In the SPLASH conference in 2010, Black, Bruce, and Noble presented a "design manifesto" for a new educational programming language. Two and a half years later we are ready to present our design for a new object-oriented programming language, Grace[1], designed for novices.

1.1 Why a new language?

While many instructors prefer to teach object-oriented programming in their introductory courses, no existing language is the obvious choice at this time. Novices need conceptually simple languages with minimal inessential complexity to better illustrate the key ideas in object-oriented languages. Another difficulty is that the languages most used today were originally developed 20 years ago or more and may not provide good support for more modern language features like closures, generics, and concurrency.

Instructors preferring statically typed languages have tried C++ and Java, but have not always found success. C++ is a large and complicated language, and novices are often overwhelmed by the complexity. While Java began as a simpler language than C++, it has added features over the years, not always gracefully, and too often requires syntax that cannot be explained easily to novices.

Instructors preferring dynamically typed languages have gravitated to Python. Python, however, has some anomalies as an object-oriented language. Methods must be written with an explicit self parameter that disappears when invoked, and there is no facility to support information hiding, a key element for reusability in object-oriented languages.

There are other interesting object-oriented languages that are candidates for use in introductory courses, like Smalltalk and Eiffel, but have not gained widespread traction in contemporary courses, so we do not discuss them further here.

Our primary interest is in the use of an object-oriented programming language for general purpose computing. Thus, while we appreciate the advantages of instructional languages like KarelJRobot and Alice that introduced simplified or restricted languages based on more general languages, we do not wish to restrict our students to programs over a narrow domain.

Much has been written about the choice of languages for introductory computer science courses. A good survey can be found in section 3.3 of Pears et al [11].

[1]The name of the language was chosen to honor the late Admiral Grace Hopper and to represent our goals for the language.

1.2 History of Pedagogical Languages

There is a long history of the use of pedagogical languages in Computer Science. The languages BASIC [8] and Pascal [7] were extremely successful from the 1960's through the 80's, but have since mainly fallen out of favor. Other interesting pedagogical languages like Turing [5] were proposed, but never gained widespread acceptance. A particularly interesting object-oriented pedagogical language was Blue[10]. Blue had the misfortune of being developed about the same time as Java, and seemed to have difficulty receiving attention due to the excitement at the use of Java with the web.

One of the main criticisms of pedagogical programming languages is that they were not immediately usable by students for internships or other projects. Both Basic and Pascal overcame that barrier to some degree by being used outside of education. For example, the original Macintosh operating system was written in an object-oriented extension of Pascal.

This is an important criticism with the use of pedagogical languages, and institutions for whom employment as a professional programmer after a first course or two is an important goal will not be interested in such a language.

We believe, however, that there are significant advantages to learning computer science with a simple and flexible language. A student wishing to become a commercial pilot does not start learning by flying a jumbo jet. On the one hand, it is just too complicated. On the other hand, a plane of that size and complexity hides some important features from the pilot because they are too difficult to be handled in real time by a human (e.g., in a "fly-by-wire" control system). As a result, student pilots start in smaller aircraft.

A student wishing to be a professional programmer should similarly find it helpful to start with a smaller language whose features have been designed to be semantically simple.

2. GOALS OF GRACE

In this section we lay out high level goals for Grace along with some specific objectives to reach those goals.

A simple statement of our goals is the following:

- We wish to integrate proven new ideas in programming languages into a simple object-oriented language designed for teaching novices.
- The language features must represent the key concepts cleanly in an easy-to-explain fashion.
- The language design must allow students to focus on the essential, rather than accidental, complexities of programming and modeling.

Let's elaborate on that last point. There are essential difficulties in programming that students must learn. This includes an understanding of and ability to apply concepts like conditional execution, iteration, parameter passing, recursion, encapsulation and information hiding. It is essential to understand these notions to be able to be a good programmer (in any language). On the other hand (to pick on Java), there is no good reason to subject novices to "public static void main(String[] arr)" early in a first course or to have them obsess over which lines should end with semicolons. These are accidental complexities that arise from a particular language design. An important goal in the design of Grace is to eliminate as many of these accidental complexities as possible so that students can focus their attention on the essential complexities of the key concepts of computing.

Some of our key objectives in achieving these goals are the following:

- Programs to accomplish simple tasks should be simple, with little or no syntactic overhead.
- Novice programmers should have access to an IDE that is specifically designed to support novices.
- Language concepts should have a simple semantic model.
- The language should support a variety of approaches to teaching including:
  - Different approaches to objects: objects-first, objects-late, and functional first (for a limited time)
  - The language should support a variety of approaches to typing: statically typed (like Java), dynamically typed (like Python), and gradually typed.
- The language should support an easy transition to other languages.

We invite the reader to assess how well we have met these goals after our description of the language.

3. A BRIEF OVERVIEW OF GRACE

The description of Grace in this section is necessarily brief because of the space restrictions. See the language website at gracelang.org for more details and examples.

3.1 Objects and Classes

Grace classes and objects consist of definitions, variable declarations, and methods, as well as any code needed to perform initialization.

```
class aCat.named(n) {
  def name = n
  var livesLeft := 9
  method meow { print "Meow" }
  print "A cat named {n} has been created"
}

var theFirstCat := aCat.named "Timothy"
theFirstCat.meow
```

The class aCat has a constructor named that takes a single parameter n. When the constructor is executed in the next to last line of the program segment above, the string "Timothy" is associated with the constant name, the initial value 9 is assocated with variable livesLeft, the parameterless method meow is defined, and the string "A cat named Timothy has been created" will be printed. (String literals support interpolation, using a syntax like that of Ruby, which results in inserting into the string the results of sending asString to objects enclosed in braces.)

In Grace the code normally contained in a class constructor is simply included with the declarations of the class and is executed when objects are constructed.

Variables are assigned values with ":=", as in Algol 60, Pascal, and Eiffel. Using an operator different from = makes it clearer the assignment is different from a claim of identity (unlike definitions).

Parameterless methods like meow are written without extraneous parentheses (). Parentheses may be dropped from the actual arguments to single-parameter methods if the arguments are delimited by double quotes or by braces (see below).

In many novice programs, only a single object is constructed from a class. For simplicity, Grace allows programmers to define such objects directly as an object literal. Thus

```
def theSecondCat = object {
    def name = ''Timothy''
    var livesLeft := 9
    method meow { print "Meow" }
    print "A cat named {name} has been created"
}
```

results in a value operationally equivalent to theFirstCat.

Everything in the language is an object, including numbers and booleans. The type Number contains rational representations of exact numbers and inexact representations approximated to 64-bit precision.

Grace does not have a built-in nil or null value. Uninitialized variables have a special uninitialized value. Accessing this value is an error. See the pattern matching section for alternatives to nil.

3.2 Types & Information hiding

The examples so far have not mentioned types, but types are supported by the language, as are information hiding annotations. Our intention is to support a variety of approaches using types, so types may be completely omitted, completely specified, or anything in-between. For example, we expect that some instructors will wish to begin using Grace as dynamically typed and only gradually add types as the semester progresses. Dynamic checks are inserted where necessary to guarantee that Grace is type-safe.

For example we can define

```
type Cat = {
    name −> String
    livesLeft −> Number
    meow −> Done
    decrementLivesBy(n:Number) −> Done
}

class anotherCat.named(n) −> Cat {
    def name: String is public, readable = n
    var livesLeft : Number is public, readable := 9
    var reserveLives : Number = 1
    method meow −> Done is public { print "Meow" }
}
```

The definition of the type Cat shows that it has four public methods, though only two of these are defined directly in the class anotherCat. The other two, name and livesLeft, are generated implicitly by the accessibility annotations.

Classes may not be used as types in Grace. Classes specify how to construct an object and provide the code that determines the behavior of an object. Types support abstraction by providing specifications of the methods that may be requested of an object without providing the details of the implementation.

The default accessibility for variables and definitions is *private*, which means they are only accessible inside an object's definition. (Notice this restriction is by object, not class! One object generated from a class does not have access to the privates of a different object from the same class). The default accessibility for methods is *confidential*, which means it is accessible only within the object or class expression or within its subclass expressions. In particular, this means it may only be accessed by calls of the form self.m or super.m. (Identifier self is used to refer to the object executing the code, corresponding to this in Java and C++.)

Methods can be made accessible to other objects by annotating them to be public. Definitions can be annotated as readable, which is equivalent to confidential, or as public, readable. More variations are possible with instance variables, which can be annotated as readable and/or writable

If a definition is annotated as public then a method of the same name is generated. Variables declared to be readable or writable result in the implicit generation of appropriate methods. These methods can then be overridden in subclasses or subobjects

The top level declarations in a module are treated as methods of an implicit object with the same name as the module. Thus,

```
print "hello world"
```

is a perfectly fine stand-alone program. Similarly variables, definitions, and methods can appear at the top level.

As in most other object-oriented languages, the receiver self can be omitted. In the above code, self has been dropped from the request to execute method print.

Grace has adopted multi-part method names (sometimes call mixfix) from Smalltalk. Thus

```
myVector.at(i)set(newValue)
```

represents a request to myVector to execute the at()set() method with actual parameters i and newValue.

3.3 Blocks

Blocks, representing anonymous functions, may be assigned to variables or passed to parameters in Grace. A block is written between braces, and its body contains code whose execution is deferred. A block may have arguments, which are separated from the body by − >. A simple example is the following block which doubles its argument {x:Number −> 2∗x}. The body is evaluated with an apply method:

```
{x:Number −> 2∗x}.apply(5)
```

Control structures in Grace are methods that take blocks as parameters. Thus, the if()then()else() method takes as parameters a boolean-valued expression and two blocks. E.g.

```
if (graceLanguage.isGreat) then {
        print ''Adopt it now''
} else {
        print ''Come up with a better design''
}
```

where the portions in braces are parameterless block parameters. As noted earlier, parentheses may be dropped around a method argument if the argument is delimited by either double quotes or braces.

Two other examples are:

```
var sum := 0
myList.forEach{e −> sum := sum + e}
```

and

```
method sqrt(n) {
    var sqrt := 0
    while {sqrt∗sqrt <= n} do {sqrt := sqrt + 1}
    sqrt−1
}
```

Notice how the forEach method provides the same facilities as an iterator except in this case forEach takes a block as a parameter, whereas an iterator would have to be used as part of a standard for loop.

A method returns the value of the last expression evaluated, which in the method above is sqrt−1. An explicit **return** statement can be used to specify an immediate return, but is not necessary for normal terminations of methods.

The last example illustrates that while loops take boolean-valued blocks as their first parameters. This is because the boolean expression is (potentially) executed more than once.

The boolean expression used in the if ()then() else () method is executed when the method is evaluated, so it used a parenthesized boolean expression rather than braces. On the other hand, because the code provided for the then and else parameters are evaluated only if needed, they must be blocks.

While this is different from most existing languages, it provides opportunities for instructors to discuss optional (as in then or else clauses) or repeated execution of code, and how that differs from code that is executed exactly when it is encountered in program text.

An important aspect of having blocks as "first-class" elements of the language is that library writers, faculty, and students can add new methods to data structures that provide high-level operations to those data structures. For example the forEach method illustrated above might be implemented in a Vector class by the following code

```
method forEach(action:Block)−>Void is public {
     for (1.. size ) do {i−>
          action . apply(at( i ))}
}
```

where size is the length of the vector and method at(i) returns the element with index i.

3.4 Pattern matching

Grace provides support for pattern matching of the sort found in functional languages like ML and Haskell, and like the object-oriented language Scala. While a more complete description of these facilities can be found in [6], we present a brief sketch here.[2]

The programmer can use pattern matching on literal values (like the switch statements in the C family of languages) or on types. A simple example is:

```
match (exp)
    case{n: Number −> "Number {n}"}
    case{s: String −> "String {s}"}
```

This expression returns a string of the form "Number n" if the value of exp is a number n, and "String s" if the value is string s.

Pattern matches can also be used to extract information from objects. For example, we might want to write

[2]Interestingly, the pattern matching facilities are mostly definable from the core features of Grace, and hence can be supported via libraries.

```
match(aList)
    case {(emptyList) −> print "empty list"}
    case { List<Number>(hd,tl) −>
                print "a list with head: {hd}, tail: {
                      tl}"
```

This destructuring match requires only that the type has an extract method of appropriate type:

```
type List<T> = {
    ...
    extract −> Tuple<Number,List<Number>>
}
```

To promote support for exhaustive checking of pattern matches, Grace supports tagless variant types of the form T|U. Value of that type are objects that conform either to T or to U.

Variant types along with singleton types make it straightforward to program replacements for null. For example the object emptyList above, which normally would support fewer methods than List, would have a singleton type written singleton (emptyList). We could then declare

```
type OptList<T> = List<T> | singleton(emptyList)
...
method subseq −> OptList<T> {...}
```

and pattern matching code could handle empty versus non-empty results differently.

3.5 Inheritance

Grace supports inheritance for both objects and classes. For example we can extend our cat example as follows:

```
class aHipCat.named(n) {
   inherits aCat.named(n)
   def vibe = "Cool"
   method meow is overriding { print "Meow, man!" }
}
```

The **inherits** clause contains an expression generating an object, typically the invocation of a class constructor, but it can also be an object.

4. SAMPLE GRACE CODE

We have now written substantial Grace code, including a Grace compiler written in Grace. We have also written a first pass at a collections library for Grace, as well as typical assignments for a data structures course. This can be found at http://www.cs.pomona.edu/~kim/GraceCollections/grace_page.html.

5. THE STATE OF GRACE

Most of the details of the language design have now been worked out, though we expect refinements to be made as we develop further experience with the language.

We now have a nearly complete implementation of the Grace language, called minigrace. It can generate either JavaScript that can be run in a web browser or C code that can run on any platform. See gracelang.org.

We are convinced that the success of a language for novices depends heavily on having a supportive interactive development environment (IDE). Rather than designing such an environment from scratch, we have looked into modifying BlueJ or DrRacket (formerly DrScheme) to support Grace.

We currently have a partial implementation of Grace inside of DrRacket [3]. We intend to complete that implementation and to create language levels for Grace inside of DrRacket. We anticipate having a directed graph of such levels rather than simply a linear progression so that we can better support multiple approaches to teaching Grace in the first few weeks.

Taking advantage of the BlueJ implementation would apparently require an implementation of Grace on the JVM. While we would eventually like to build such an implementation, it is currently on a back burner.

5.1 Plans

We will create libraries for supporting parallelism and concurrency in Grace. We hope to have at least one standard library for shared-memory and one for message-passing (distributed) parallel and concurrent programming. Because we do not know which language features for parallelism are likely to be most successful, we would like to provide libraries that can be used with different mechanisms.

Because we have not yet had a solid development system, we have not yet been able to class-test Grace. We have had good experience in having a student research assistant writing libraries for data structures, which we find encouraging. Moreover, we plan on using Grace in small classes of novices in the summer and fall of 2013. Based on the results of this test we will likely be making minor tweaks to the language. Because our goal is to provide a clear and straightforward model to students, we will respond to any difficulties with the language that they encounter.

We intend to develop at least two different approaches to teaching with Grace, one based on the text by Felleisen et al [2] that takes a functional first approach, and one based on the text by Bruce et al [1], which takes an objects-first approach. We will make the lecture notes and examples from these courses freely available on-line.

6. EVALUATING GRACE

While it is difficult to evaluate Grace without direct experience with teaching with it, we can provide a first evaluation based on criteria that others have presented on language choices for novices. For the purposes of this paper we adopt the checklist proposed by Michael Kölling [9]. In the rest of this section we provide an initial evaluation of Grace according to the 11 criteria of Kölling (explained in detail in the referenced paper).

Clean concepts.

We have endeavored to provide clean uncluttered features with a straight-forward intuitive semantics. Students can begin by defining objects directly before being introduced to classes, providing them with a concrete syntax for the values they will be manipulating.

We have separated classes from types in Grace. Classes are used to create objects, types are used to specify the methods that an object will respond to, without making a commitment to any particular implementation.

Pure object-orientation.

Everything in Grace is an object, including values like numbers or booleans and constructs like blocks. As a result, there is no need for the complications of values versus references. While Grace supports functional constructs, they are obtained in an object-oriented way.

Safety.

Grace is strongly typed, with type errors caught at either compile time or run time. Because it is gradually typed, a student may write programs with no type annotations, resulting in dynamic checks, or complete annotations, resulting in static checking, or anything between (resulting in dynamic checks ensuring run-time values meet the type specification). These will be specified by annotations.

Grace also checks for uninitialized variables, and emits an error message if a user accesses an uninitialized variable.

High level.

Grace is high-level in several ways. Of course Grace uses automatically managed memory, eliminating the possibility of some of the most frustrating errors for students.

More importantly, blocks are first-class in Grace: they may be assigned to variables or passed as parameters. This provides the student (and teacher) with the ability to write methods that accept blocks as parameters to simplify operating on elements of data structures. A good example would be a method that would traverse a binary tree, applying a single-parameter block to each element of the tree. Blocks with parameters are simple notations for anonymous functions.

Simple object/execution model.

All objects are on the heap, and all computation takes place by requesting a method to be executed by an object.

Readable syntax.

We have tried hard to support this, with minor variations from common syntax. All declarations are associated with keywords: **def** for definition, **var** for variable, and **method** for methods. These are the same words that we use to describe the concepts in class, so the code corresponds closely to the way we talk about the concepts.

Because we expect students to transition to languages that use C-style syntax, we have adopted features that are close to those languages, though normally with a twist. For example, we use braces to delimit blocks, but those blocks must also be consistently indented. Because the rules for ";" can often be confusing (e.g., no ";" before "{"), we found a syntax that allows us to omit them.

The names of identifiers in class features come before their types, making them more visible. It is common for students to look for the identifier x, not for all integers, so x appears to the left of its type. We distinguish between "=" for definitions and ":=" (following the tradition in Algol-like languages) because they are different concepts. One is an identity, while the other temporarily associates a value with an identifier. Other differences of syntax from C-style syntax were generally made to increase readability.

No redundancy.

This is difficult to verify, but we have worked to build Grace from a small number of primitive constructs. Because blocks are first class, we can build any number of looping constructs. That might be considered redundant, but we feel it is important to be able to build the most appropriate looping structures for a given data structure.

The one construct that likely looks most redundant is the pattern-matching facility. After all, most languages don't have such a construct. Most languages do provide a construct to check the type of run-time values, however, and pattern matching provides similar facilities, though in a type-safe way. Moreover, we find pattern matching very powerful, especially in data types where we want to handle some values (e.g., empty lists) differently from others. Finally, the pattern matching facilities are really defined in Grace from other more primitive features of the language, so they can be considered a feature added by libraries rather than a primitive.

Small.

The core of Grace is quite small, as many of the constructs are defined (from other primitive operations in Grace) in libraries. Because Grace is intended for novices, it does not provide many facilities for programming in the large, as would be necessary in an industrial-strength language.[3] Thus all of Grace should be easily learnable by a student in one to one and a half semesters.

Easy transition.

The basic concepts supported by Grace are available in other object-oriented languages, though often obscured by complicated syntax. Even blocks, not currently a part of Java, will be added in the next revision (under the name of lambda expressions), though undoubtedly with a more complex syntax. Under the name of "anonymous functions", they are already included in C# 3.0 [4].

Correctness assurance.

We do not currently have annotations for pre- and post-conditions, but we anticipate that we will be able to provide them via libraries.

Environment.

We are well aware of the importance of a good IDE for Grace. Based on the preliminary work, we anticipate having Grace support in the DrRacket environment in the near future. While we hope to provide support for Grace in BlueJ, that may take longer. At a more primitive level, we do have a Grace mode in emacs.

We believe that Grace stacks up very well against languages like Java, C#, Python, and Eiffel with respect to these criteria. We are well aware that a positive reception by students to the language is crucial to the success of the language. Thus class testing will be essential to the further development of the language.

7. CONCLUSION

While this project is still in relatively early stages, we feel that it is now far enough along to present to potential adoptors. We are eager to have feedback on the language design to this point, and are happy to have others pitch in with library designs, work on development environments, improvements in the implementation, and in testing the language on novices.

More information on the project, including information on current prototype compilers is available at gracelang.org.

[3]Though there is a relatively simple module facility for importing features from libraries.

8. REFERENCES

[1] K. Bruce, A. Danyluk, and T. Murtagh. *Java: An Eventful Approach.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[2] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to design programs: an introduction to programming and computing.* MIT Press, Cambridge, MA, USA, 2001.

[3] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[4] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *C# Programming Language.* Addison-Wesley Professional, 4th edition, 2010.

[5] R. C. Holt and J. R. Cordy. The Turing programming language. *Commun. ACM*, 31(12):1410–1423, Dec. 1988.

[6] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *Dynamic Languages Symposium*, 2012.

[7] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer, 1975.

[8] J. G. Kemeny and T. E. Kurtz. *Back to Basic; The History, Corruption, and Future of the Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[9] M. Kölling. The problem of teaching object-oriented programming, part i: Languages. *JOOP*, 11(8):8–15, 1999.

[10] M. Kölling and J. Rosenberg. Blue – a language for teaching object-oriented programming. In *Proceedings of the 27th SIGCSE technical symposium on Computer science education*, SIGCSE '96, pages 190–194, New York, NY, USA, 1996. ACM.

[11] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.