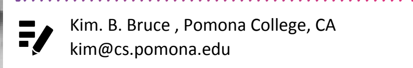
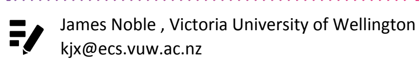
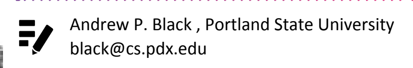
Grace: an open-source, object-(programming language education



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



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



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

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

Although object-oriented programming is widely taught in introductory computer science courses, no existing object-oriented programming language is the obvious choice for a teaching language. While Java was the *de facto* standard throughout the 2000's, in the last few years a range of newer languages such as Python, Ruby, Scala, C#, F#, Processing, and JavaScript have begun to make their way into classrooms — and so to research labs, offices, and, eventually, large software systems.

During ECOOP 2010, a group of language researchers and educators concluded that the time was ripe for an effort to design a language focussed on teaching. A "design manifesto" was presented at SPLASH 2010, in which we attempted to lay out design principles for a suitable a language. Since then three of us (Black, Bruce and Noble) have been meeting weekly to pursue the design of the language, which we have named "Grace", in honor of Admiral Grace Hopper, and in the hope that the name would serve as an admonition not to settle for less-than-graceful solutions.

GRACE IN A NUTSHELL

Grace is an imperative object-oriented language with block structure, single dispatch, and many familiar features. Our design choices have been guided by the desire to make Grace look as familiar as possible to programmers who know contemporary object-oriented languages such as Java, C#, Ruby, Scala, and Python. We have also been motivated by the need to give instructors and text-book authors the freedom to choose their own teaching sequence. Thus, in Grace it is possible to start teaching using types, to introduce types later, or not to use types at all. It is also possible to begin with objects, or with classes, or with functions. Importantly,

oriented

for

instructors can move from one approach to another while staying within the same language.

The traditional first program in a new language is "Hello, World". In Grace, we kept this program as simple as we possibly could:

```
print "Hello, World"
```

We think that "Hello, World" needs to be especially simple, for a number of reasons. Not only is "Hello World" the first program many experienced programmers write in a new language, it is also the first programmer beginners will write in any language. We want those first programs to be easy because programming is hard enough as it is: especially for novices, we don't want the programming language to get in the way of teaching the fundamental ideas. The less syntax that is required, the less syntax there is for novices to get wrong. Our experiences teaching Java, where it is necessary to have the whole class chant incantations like `"pubic static void main(String arg[])"` before the students could even print "Hello" — and then having to explain the resulting error messages — have convinced us that avoiding such accidental complexity is important.

OBJECTS AND CLASSES

Grace can be regarded as either a class-based or an object-based language, with single inheritance. A Grace class is an object with a single factory method that returns a new object:

```
class aCat.named(n : String) {
        def name = n
        method meow { print "Meow" }

}
```

Here the class is called `aCat` and the factory method is called `named`. We can create an instance of that class — a new cat object — and store it in a variable:

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

After executing this code sequence, `theFirstCat` is bound to an object with two attributes: a constant field `(name)`, and a method `meow`. The method request `theFirstCat.name` answers the string object `"Timothy"` and `theFirstCat.meow` has the effect of printing *Meow.*

An object can also be constructed using an object literal — a particular form of Grace expression that creates a new object when it is executed. For example:

```
var theSecondCat := object {
        def name = "Timothy"
        method meow { print "Meow" }

}
```

This code binds the variable `theSecondCat` to a newly created object, which happens to be operationally equivalent to `theFirstCat.`

FIELDS AND VARIABLES

Mutable and immutable bindings are distinguished by keyword: `var` defines a name with a variable binding, which can be changed using the `:=` operator, whereas `def` defines a constant binding, initialized using `=`, as shown here.

```
var currentWord := "hello"
def world = "world"

currentWord := "new"
```

The keywords `var` and `def` are used to declare both local bindings and fields inside objects. Like Java — but unlike JavaScript — fields and methods cannot be added to an object after it is created. A field that is declared with `def` is constant. Each constant field declaration creates an accessor method

on the object. Declaring a field with var creates two accessor methods, one for fetching the currently bound object and one for changing it. So, after the declaration

```
def car = object {
        def numberOfSeats = 4
        var speed: Number := 0.

}
```

the object car will have three methods called `numberOfSeats`, `speed`, and `speed:=()`. When we use `()` in the name of a method, it indicates the need to supply arguments. So, the last method might be used by writing `car.speed := 30.`

REQUESTING METHODS

Grace method names may consist of multiple parts (“mix-fix notation”) as in Smalltalk or Objective-C. Separate lists of arguments are interleaved between the parts of the name, allowing them to be clearly labeled with their purpose. Thus we might define on `Number` objects

```
method between (l:Number) and (u:Number)
{
        return (l < self) && (self < u)

}
```

The syntax of a method request is similar to that used in Java, C++, and many other object-oriented languages: `obj.meth(arg1, arg2)`, but extended to allow the name of the method to have multiple parts. We could request the above method `between()and()` on 7 by writing

```
7.between(5)and(9)
```

Single arguments that are literals do not require parentheses, so alternatively we could write

```
7.between 5 and 9
```

Following many other languages, the receiver `self` can be omitted. We have already seen several messages requested of an omitted receiver; for example, `print “Meow”` is short for `self.print “Meow”.`

BLOCKS AND CONTROL STRUCTURES

Like Ruby, C#, and Scala, Grace includes first-class blocks (lambda expressions). A block is written between braces and contains some piece of code for deferred execution. A block may have arguments, which are separated from the code by `->`, so the successor function is `{x -> 1+x}.` A block can refer to names bound in its surrounding lexical scope, and returns the value of the last-evaluated expression in its body.

Control structures are designed to look familiar to users of other languages. However, as in Smalltalk and Self, control structures in Grace are just methods that take blocks as arguments.

```
if (x > 5) then {
        print “Greater than five”
} else {
        print “Too small”
}

for (node.children) do {
        child -> process(child)

}
```

Notice that the use of braces and parentheses is not arbitrary: parenthesized expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may thus be evaluated zero, one, or many times. A `return` statement inside a block terminates the *method* that lexically encloses the block, so it is possible to program *quick exits* from a method by returning from the *then* block of an `if()then()` or the *do* block of a `while()do()`.

TYPES

Types and classes are strictly separated in Grace. A Grace class is not a type, nor does a Grace class or object implicitly define a type. When programmers need types they must define them explicitly. We hope this separation will help us teach the concepts of types independently from classes. To this end, Grace supports both statically and dynamically typed code: omitted types of local variables and constants are inferred (as e.g. in Scala or C#), but omitted argument types are treated as the predefined type `Dynamic`. Messages requested on `type Dynamic` will be checked dynamically.

Grace types are structural: they describe properties of objects. A type is a set of method requests; a type declaration gives a name to a type.

```
type Vehicle = {
        numberOfSeats -> Number
        speed -> Number
        speed:=(n : Number) -> Nothing

}
```

An object has a type if it has the appropriate methods, and if

the signatures of those methods conform to the signatures in the type. No inheritance relationships or implements declarations are necessary. The `car` object defined above has the `Vehicle` type, but also has the smaller type `{ speed -> Number}`.

Within dynamically typed code, types need not be mentioned at all, and so the introduction of the concept of type can be delayed until late in the teaching sequence. When instructors do introduce types, they may do so in the language they are already using, as opposed to, for example, starting teaching in Python and then transitioning to Java. A static type checker will support instructors who wish to require that all student programs be fully typed.

HOW CAN YOU CONTRIBUTE?

The Grace Project maintains a website at http://gracelang.org, including the project diary (as a blog), the evolving language specification, an early prototype implementation, and other documents and papers about Grace. We are actively interested in comments and feedback about the language design, and as the project goes on, about APIs, libraries, and implementations. We would really appreciate programmers trying out prototypes as they are released, and ultimately testing the specification by building alternative implementations.

References

AP Black, KB Bruce, M Homer, J Noble. Grace: the absence of (inessential) difficulty. Accessible from gracelang.org/documents, April 2012.

AP Black, KB Bruce, J Noble. The Grace Programming Language Draft Specification Version 0.353. Accessible from gracelang.org/documents. April 2012.

Award-Winning Vendor of Developer Productivity Tools

Tools Matter™

jetbrains.com

AARHUS
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2012
Conference: Oct 1 - 3
Training: Sept. 30, Oct 4 - 5
goto; conference