A Perfect Programming Language

Sven Strittmatter

December 11, 2016

Abstract

This paper describes a perfect programming language. We will look at drawbacks of current programming languages. Then try to describe a programming language without these drawbacks.

Contents

1 Motivation

What is my motivation behind this: Describe a new perfect programming language? As of time of writing this I have over a decade of experience in programming stuff. I also have experience in various languages: C++, PHP, JavaScript, Ruby, Perl, Java. Also I saw a lot of languages. After some disappointment about the languages I used I started to look around what others do: Go, Python, Erlang, Prolog, Lisp, ML, OCaml, Scala etc.

More and more I saw different languages and their concepts I recognised that most of them have some drawbacks and I started to think about a programming language without any of these. I'm not sure if it is possible to make such a language, but I think it is worth to think about it.

1.1 Drawbacks and Fails

In this section I'll collect all the drawbacks and fails already existing languages have. These are not drawbacks only because I say. All of them have some common sense in the in the community of software craftmanship, clean code developers or others working hard on better software. So the points mentioned here are quite common sense for all well exercised developers. In a later section I will describe "disappointing" things which are merely based on my opinion.

1.1.1 Null - The Billion Dollar Fail

In a lot of programming languages exists a concept of `null`. This is sometimes called a *null reference* or *null pointer*. Most would have seen such a thing somewhere in their career and have a vague idea what it is: Some not initialized value. Let's have a look what Wikipedia[1] says what it is:

> In computing, a null pointer has a value reserved for indicating that the pointer does not refer to a valid object. Programs routinely use null pointers to represent conditions such as the end of a list of unknown length or the failure to perform some action; this use of null pointers can be compared to nullable types and to the Nothing value in an option type.
>
> A null pointer should not be confused with an uninitialized pointer: A null pointer is guaranteed to compare unequal to any pointer that points to a valid object. However, depending on the language and implementation, an

uninitialized pointer may not have any such guarantee. It might compare equal to other, valid pointers; or it might compare equal to null pointers. It might do both at different times.

The first thing to recognize here is that different languages implement this concept differently. We do not want to dig to deep into these implementation details but concentrate of the perspective of a user of such languages: The problem then arises is the so called *null dereferencing*. From the same Wikipedia[1] article:

> Because a null pointer does not point to a meaningful object, an attempt to dereference (ie. access the data stored at that memory location) a null pointer usually (but not always) causes a run-time error or immediate program crash.

To repeat the crucial part: "…causes a run-time error or immediate program crash". To be clear: the problem is not a concept for a not ininitalized or not present value, rather than that dereferencing such will result in program crashes.

Lets see some brief examples of such null dereferences in various common languages:

C/C++

```
SomeType *obj = nullptr;
obj->methodCall(); // Crashs the program.
```

Java

```
SomeType obj = null;
name.methodCall(); // Crashs the program.
```

C#

```
SomeType obj = null;
obj.MethodCall(); // Crashs the program.
```

JavaScript

```
var obj = null;
obj.methodCall(); // Crashs the program.

obj = {methodCall: null};
obj.methodCall(); // Crashs the program.
```

PHP

```
$obj = null;
obj->methodCall(); // Crashs the program.
```

Python

```
obj = None
obj.methodCall() # Crashs the program.
```

Ruby

```
obj = nil
obj.method_call # Crashs the program.
```

Perl

```
my $obj;
$obj->method_call; # Crashs the program.
```

All these examples above will compile/translate without any error, but will crash tremendously at runtime. Some might say this is not the problem and as a developer you have to deal with this issue. You have to write your program either that you do not produce such null values or you deal with them correctly. But this is easier said than done. To paraphrase Murphys Law: If there is the posibillity to make something wrong, then someone will do it wrong at some point in time. So as a result developers tend to clutter up code with null checks (called defensive programming). Which will make the code harder to read and reason. As Kent[2] stated in his paper *Dynamic Error Remediation: A Case Study with Null Pointer Exceptions*:

> One insideous bug is the null pointer exception, which by its null nature is hard for programmers to fix. These bugs indicate that nothing was found in memory where something should have been, giving programmers very little to work with to fix the bug besides a stack trace. Null pointer exceptions sometimes show themselves only with certain inputs, making these bugs difficult to find before deployment. . . . , 1-2% of developer code is devoted to identifying null objects.

Also an important argument is the economic damage done by this kind of bugs. Kent[2] cites the NIST that along with developer time and money spent in error checking annually cost of $59.5 are estimated. From the origin source it is not clear that these costs are only came from only this kind of bug or from all kinds. What the inventor Tony Hoare[3] said about null[4]:

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Whatever the real economic amount is, the fact that this kind of bug is completely avoidable by removing null pointers is obvious. To be clear: There is no problem with the concept that a value may or may not be present. Some languages (as described later) has implemented this without null pointers. The problem with null pointers is that everything everywhere may be one and either you put lot of checking code into your software or you risk crashes. In statically typed language this also subverts the type system because every type may be null but the statically checking at compile time can't inform you about this problem. As mentioned above the problem will hit reality first at runtime. There are lot of more problems which will be introduced by null. A list of some is described by Paul Draper in his blog post *The worst mistake of computer science*[5].

Of course there are techniques to circumvent null pointer dereferences in such languages without cluttering up the code with lot of null checks and boiler plate code. You can void using null at all. Use empty strings or zero instead of null. Or use *Null Object Pattern* is a special kind of object of your type which represent an absent value. But these are *workarounds* and introduce the big caveat that a software developer need to know about. Remember yourself starting software development as a novice: You learned ab out that null thing in your language of choice and of course you used it. Why else should it be part of the language, unless to use it?

Now the interesting question is: *How to deal with values which may or may not be present without introducing null pointers?* Let's look at some languages which try this:

Objective-C: Despite the fact that *Objective-C* has null due to its inheritance to C (because it is build on top of C). It has an interesting concept called *nil*. It is nearly the same as the C null (instead of `(void *)0` it is `(id)0`). The big difference is that you can send method calls to *nil* without getting an error. The call is simply ignored and returns *nil* as result. So consecutive calls will not fail. This obviates defensive programming as mentioned above. For example a if-expression like

```
    if (name != nil && [name isEqualToString:@"Hello, World!"]) { ... }
```

can be simplified to

```
    if ([name isEqualToString:@"Hello, World!"]) { ... }
```

So it is an opt-in procedure to ask if a value is `nil`: In the special case you do not want it you can react on it, but by default the program does not crash. But *Objective-C* has some drawbacks in the field of not initialized references: First it has the C null as mentioned above. This introduces the risk that it is used (if someone didn't know better). Second it is not possible to store `nil` in collection types and so there is a special container type singleton for that: `NSNull`. Also there is a distinction between unitialized object references (`nil`) and uninitialozed class references (`Nil`). This makes the whole thing complicated and error prone if someone is not familiar with the whole concept.

Go TODO
Swift TODO
Haskell TODO
F# TODO
Erlang TODO

1.1.2 Checked Exceptions

Words from Andre Heilsberg (C#)

1.1.3 Exceptions at All

They are like goto (Javaslang)

1.2 Disappointing

In this section I describe things which are very disappointing, e.g. typing boiler plate code, parens, semicolons and such. In contrast to the section before these points are merely opinion based. But I try to show why we should think about it.

1.2.1 Annoying Braces

Braces for conditions: if (condition) vs. if condition.

1.2.2 Unnecessary Semicolon

Write text.

1.2.3 Too Many Special Character Methods or Operators

Looking at Perl or Scala as a beginner you will notice a lot of strange characters in the code. Some examples from Perl[7]:

- Spaceship operator: `<=>`
- Eskimo Greeting operator: `}{`
- Goatse operator: `=()=`
- Turtle operator: `@{[]}`
- Inchworm operator: `~~`
- Inchworm-On-A-Stick Operator: `~-`
- Spacestation Operator: `-+-`
- Venus operator: `0+`

Perl is an ancient language, but modern languages also have this problem. Here some examples from Scala[8]:

- Upper, lower and view bounds: `<: >: <%`
- Vararg expansion: `_*`
- Many different meanings: `_`

I don't want to blame Perl or Scala in particular here. Other languages does this also: Haskell, Ruby etc. Especially functional languages tend to strange characters to name functions, methods or operators. In my opinion this violates the rule that source code should be least astonishing. Of course for a professional developer this may be time saving to type three characters instead of a long function name. This argument counted decades ago when the developers only had dumb text editors. Today we all have super power driven editors or IDEs with intellisense and auto completion. So there is no reason to confuse developers without years of experience with strange characters.

1.2.4 Missing Tools

A lot of languages lack of appropriate tooling and it is necessary to build them first before you get productive. A perfect language must come with all necessary tools. What are necessary tools?

All languages have at least a compiler or interpreter. And mostly that's all you get. In my opinion there must be more tools for:

- source code formatting
- create API docs and documentation
- run tests with coverage reports
- dependency management

2 What Ideas Are Out There?

In this section ideas and concepts of already existing languages will be described.

2.1 Kotlin

From a blog post from Peter Sommerhoff[6]:

2.1.1 Data Classes

Simple POJOs are declared simpler. Instead of writing boiler plate like:

```
class Book {
    private String title;
    private Author author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }
}
```

You can write:

```
data class Book(var title: String, var author: Author) {
    // ...
}
```

2.1.2 Smart Casts

Instead of:

```
if (node instanceof Leaf) {
    return ((Leaf) node).symbol;
}
```

less verbose:

```
if (node is Leaf) {
    return node.symbol;
}
```

2.1.3 Type Inference

Write text.

2.1.4 Functional Programming

Write text.

2.1.5 Default Arguments

Write text.

2.1.6 Named Arguments

Write text.

2.1.7 Final Classes

Next, Kotlin also supports the principle to either design for inheritance or prohibit it because in Kotlin, you have to explicitly declare a class as open in order to inherit from it. That way, you have to remember to allow inheritance instead of having to remember to disallow it.[6]

3 Ideas

- `unless foo` statt `if !foo`
- type (int, float) overflows generate exceptions
- compiler checks for right versioning in manifest
- compiler does not allow shot names of identifiers
- `finally` to execute something always at least in functions

3.1 Source

A source file `MyModule.ct` may look like:

```
var integer foo = 23
var integer bar // is 0 by default
var integer baz = foo + bar

const snafu = "Hello !"

function doIt(Integer i) {
    ...
}

function Integer doWhat(Integer i, Integer j) {
    return i + j

    finally {
     // Something which is done before function returns, e.g. clean resources.
    }
}

function Integer multiFinallies() {
    File f = new File("/foo");
    f.open();

    finally {
        f.close()
    }

    File d = new File("/tmp");
    d.open()

    finally {
```

```
    d.close()
    }

    return 42
}

var result = doWhat(foo, 42)
```

3.2 Manifest and Versioning

A module descriptor `MyModule.mf` may look like:

```
version 1.0.0

export TypeName
export TypeName
export TypeName

Integer main(List<String> args) {
  return 0
}
```

4 Specification

4.1 Tooling

A perfect language need tools. Some of them are obvious:

- Interpreter: An interpreter is useful in the first steps of language development, but also useful later to provide endusers a REPL (read eval print loop) tool for playing around with the language without ramp up a whole project and build infrastructure.
- Compiler: Of course there must be a compiler to create byte code for a virtual machine.
- Virtual Machine: For interpreting byte code a virtual machine is necessary. This is implemented as register based VM.

The not so obvious tools are a level above just compiling source code.

- Formatter: Most languages say not much about how to format it. Therefore almost mostly formatting is a very opinion based thing. This leads to endless discussions in projects involving more than one programmer. Thats the reason why this language provides a formatter with a default which is highly recommented to use.

- Tester: There is tool which executes tests automatically. Also there is a built in API for unit testing.
- Builder: This is a tool which compiles and links a complete module. It also runs the Tester to verify the tests. Also it manages dependencies of the built module.

4.2 Module Packaging and Visibility

A module is a directory containing all source code. It contains one and only one manifest file which declares basic attributes for the module:

- `group`: This declares the group name of the module.
- `artifct`: This declares the artifact name of the module.
- `version`: This declares the version of the module.
- `namespace`: This declares the namespace base of all types in the module.

Also the manifest declares the module dependencies. Modules are referenced by an import and a module coordinate.

An example of a manifest file `moduledir/Manifest.mf`:

```
group       de.weltraumschaf
artifact     example
version     1.0.0
namespace   de.weltraumschaf.example
import      org.caythe:core:1.0.0
import      org.caythe:testing:1.0.0
```

All types declared in the root of the module directory are referenced by the package name given by the namespace declaration. All sub directories are packages. For example a class `Foo` in the file `moduledir/Foo.ct` will be referenced by `use de.weltraumschaf.example.Foo`. A type is always declared in a single file and a single file can only declare one type. So the File `Foo.ct` declares the type `Foo`. In consequence it is not necessary to type the name of the type again in side the file and renaming the file result in renaming the type.

From outside modules only types with explicit `export` are visible. So that other modules which import `de.weltraumschaf:example:1.0.0` may see the type `Foo` it must be declared exported.

Also each method which should be accessible from outside must be declared exported. Methods or types declared `public` are only visible everywhere inside the module. Everything declared `package` is only visible in its directory and all sub directories. So a type or method in the root module directory declaring `package` has the same effect as declaring it `public`. By default methods and types are private.

```
export // Exports the type
public // Marks the type public

export method callableFromOtherModules() { ... }
public method callableFromWholeModule() { ... }
package method callableFromSameDirAndSubDirs() { ... }
// private is the default so there is no keyword for that.
method callableOnlyFromSelf() { ... }
```

:

Describe the followed items.

- Types
  - integer: 0 by default
  - float: 0.0 by default
  - boolean: false by default
  - string: empty by default
- Scopes
  - Lexical Scopes
  - Variables are only writable in current scope. What about members of classes?

References

[1] Wikipedia *Null pointer*, https://en.wikipedia.org/wiki/Null_pointer

[2] Stephen W. Kent *Dynamic Error Remediation: A Case Study with Null Pointer Exceptions* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.2544&rep=rep1&type=pdf

[3] Wikipedia, *Tony Hoare*, https://en.wikipedia.org/wiki/Tony_Hoare

[4] Tony Hoare, *Speaking at a conference in 2009*, http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[5] Paul Draper *The worst mistake of computer science* https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/

[6] Peter Sommerhoff, *Kotlin for Java Developers: 10 Features You Will Love About Kotlin* https://www.javacodegeeks.com/2016/05/kotlin-java-developers-10-features-will-love-kotlin.html

[7] Peteris Krumin, *Secret Perl Operators* http://www.catonmat.net/blog/secret-perl-operators/

[8] Daniel C. Sobral, *What do all of Scala's symbolic operators mean?*, http://stackoverflow.com/questions/7888944/what-do-all-of-scalas-symbolic-operators-mean