A Polyglot Day:
Learning from language paradigms

Benson Joeris
Kathleen Dollard
@kathleendollard
What if you understood all the paradigms that define modern languages?

*You could program better in your base language,*

*choose the best tool for the job,*

*integrate ideas from other languages and*

*design systems that cross boundaries from legacy to the future.*

Awesome!

But, who has time to do the hard work of learning a bunch of languages from the inside out to be a true polyglot programmer?
When You Write Your Essays in Programming Languages

When You Write Your Essays in Programming Languages

JAVA

I'M TWO PAGES IN AND I STILL HAVE NO IDEA WHAT YOU'RE SAYING.

When You Write Your Essays in Programming Languages

When You Write Your Essays in Programming Languages

When You Write Your Essays in Programming Languages

C++

I asked for one copy, not four hundred.

When You Write Your Essays in Programming Languages

I don't have permission to read this.

When You Write Your Essays in Programming Languages

LATEX

YOUR PAPER MAKES NO GODDAMN SENSE.
BUT IT'S THE MOST BEAUTIFUL THING
I HAVE EVER LAY EYES ON.

When You Write Your Essays in Programming Languages
Benson Joeris
Theoretical mathematician who loves languages

- Post-doctoral researcher, University of Waterloo
- Expert in algorithms, functional programming and type systems who’s worked and experimented with languages across all of today’s paradigms and thinks about problems of dependability, performance and scale from a perspective of core principles
- Pluralsight
  - Haskell Fundamentals
Kathleen Dollard
Passionate pragmatic programmer

• Decades in industry
• Director of Engineering, ROI Code, Denver
  • Love leading teams developing business applications
• Microsoft MVP since 1998
• Teacher, speaker, writer
• Video
  • Pluralsight
    • Event Tracing for Windows (ETW) in .NET
    • .NET Puzzles
    • What’s New in .NET 4.5
    • Visual Studio 2015 Analyzers
    • Understanding Metaprogramming
• WintellectNOW
  • C# 6
Schedule

• Section 1
  • Hello to the planets in 4 languages

• Section 2
  • Language paradigms

• Section 3
  • Advanced paradigms

• Section 4
  • Pulling together languages, libraries and tools
Goals

• Understanding how languages are different and most have value
  • And the realization of how many characteristics are converging
• What can you learn from each language you can take back to your core language
• How to choose the best language for a problem/project
Section 1: Basics

History

Hello to the planets in each language
Language History

• FORTRAN (1954)
  • John Backus, IBM, early high level general purpose language

• BASIC (1964)
  • Kemeny and Kurtz, Dartmouth, Beginner's All-purpose Symbolic Instruction Code

• C (1972)
  • C: Dennis Ritchie developed for Unix based on B (Martin Richards, Ken Thompson)

• C++ (1983)
  • C++: Bjarne Stroustrup set out to enhance the C language with Simula-like features

• Haskell (1990)
  • Built by committee to consolidate the existing functional languages for language research

• Python (1994 (v1))
  • Guido van Rossum over 6 years initially as a hobby. Declared Benevolent Dictator for Life

• Javascript (1995)
  • Brendan Eich originally developed it in 10 days, while working for Netscape

• C# (2000)
  • Anders Hejlsberg and a team at Microsoft resources built a language for widespread use

• F# (2005)
  • Don Syme, Microsoft Research, functional programming in the .NET Framework

From Wikipedia.org
• **C#**, Java, Visual Basic (strict on), Objective C
  • Object oriented, strongly typed, functional features added

• **Python**
  • Object oriented, dynamically typed, functional features added

• **Haskell**
  • Non-object oriented, strongly typed (generally inferred), functional

• **JavaScript**
  • Unstructured objects, dynamically typed, functional features

• **F#** (based on ML/Caml), Scala
  • Crossover language
  • OO features, strongly typed, designed as functional language

• **C++**
  • Object oriented (multiple inheritance), strongly typed, functional features, pointers
- Agda, Idris
  - Dependently typed
- Bash, PowerShell, Batch (DOS), Groovy, Lua
- Lisp, Clojure, Scheme
- TypeScript, CoffeeScript, Dart, TCL
- Erlang
  - Message/actor
- Ruby
- Perl
- Go, Eiffel, Kotlin, Logo, Nemerle, Processing, Rust, Scratch, Swift, Wolfram Language
- PHP
- PowerBuilder (4GL)
- ProLog
DSL

- MatLab, Mathematica, Magma, Sage, Octave
- R
- Geurkin/Specflow/Cucumber
- LaTeX, XML, XSLT, HTML, PostScript
- SQL variations
Others

• Hope
• Joy
• Io
• Kaleidoscope
• Linoleum
• Pizza
• Snowball
• Squeak
• Squirrel
What you do with this knowledge

• Find a better fit for particular problems
• Apply functional techniques to imperative, object-oriented languages
• Apply strongly typed ideas to dynamic languages
• See alternate methods of extension
Type systems

1954

1985
IEEE Floating Point Standard

2016
Type systems

• A type system determines if data is compatible with an operation
  • What kind of data can be assigned to a variable, or passed to a function

• Functions have types, as well as their arguments/return values
  • “A function that takes an integer argument and returns an integer” is a type

• Types also help the compiler generate efficient machine code
  • But we will focus on how types help programmers, rather than how they help the compiler
When you want a Type System but all you have is Javascript

• https://twitter.com/old_sound/status/727427245409746944/photo/1
Hello Planets
Filtered aggregation in each language
Purity

• Not pure if side effects
  • Any change to anything in the universe
  • Cannot write a file or console

• Not pure if answer changes when it’s called multiple times
  • The return value should be able to replace the function call if the interpreter desires
  • Cannot access to the current date or time
  • Cannot access to random
  • Cannot read a file or console
  • Cannot set or read a mutable variable

• Haskell found solutions – like returning a command to output
BREAK
Summary of Hello Planets

• Avoid being terrorized by syntax
• Dynamic vs. static typing
• Functions as first class citizens
• Loops and conditionals vs maps and filtering
• Ability to use similar approaches in different languages
Crazy constructs or Functions as First Class Citizens

Thanks to Sam Artioli for code
What does this code do?

```javascript
var cats = require('./cats.js')(app);
```
There are only three hard problems in computer programming: naming and off by one errors

• Variation of a CS joke
If you had to write the Agile manifesto again, would you change something on it?

I wouldn’t call it “Agile”. The word is too attractive. People who aren’t agile will say they are.

I took a shot at this with “Extreme”, which also has pluses and minuses. Maybe “Monastic”? “Selfless”? “Servent”? Or just “Oregon”.

• Kent Beck, https://www.quora.com/session/Kent-Beck/1?srid=CNsv&share=5e108a1e
Section 2: Language paradigms

Paradigm overview
Language paradigms

- Imperative
- Procedural
- Structured
- Object Oriented
- Functional
- Declarative

- Typing, equality and truth
- Static/dynamic typing
- Type inference
- Pattern matching/deconstruction
- Eager/lazy evaluation
- Purity vs. side effects
- Dependence on ecosystem
- Human readability
- Resilience to change
Functional

• Functional features
  • Functions are a first class citizen
  • C#, Java, Javascript, Python

• Full functional
  • Set of supporting features: type matching,
  • Imperative and object oriented coding are limited or absent
  • Haskell,

• Functional first
  • Set of supporting features: type matching,
  • Imperative and object oriented coding are supported
  • F#,
Object Oriented

• None
  • Some functional languages
  • Haskell

• Classes supported and some inheritance
  • Javascript, Python?

• Full support for OO patterns
  • Overrides?
  • C#
Dynamic and Static Typing

• Dynamic typing
  • Multiple types can be assigned to same variable
  • Shape of objects can be expanded
  • Duck typing?
  • Javascript, Python

• Static typing
  • Variables are strictly of type
  • C#
  • If a dynamic type is available, a variable is of that type

• Linted
  • A design time tool, not the compiler enforces type checking
Tax Calculator
LUNCH
Summary of Tax Calculator

• Big Picture
  • Fundamentally different approaches possible with different paradigms

• Details
  • Pattern matching
  • Classes and objects
    • Ability to add attributes in Python and Javascript
  • Public/private visibility
    • Python underscores
    • Javascript nested function scope
  • Interfaces
  • Function signatures
    • loose arguments (args, kwargs)
"Perl – The only language that looks the same before and after RSA encryption."

- Keith Bostic (quoted on Twitter)
Section 3: Advanced paradigms

Function purity, side effects, and the real world
Async
Make me a pretty program that

• Displays a bunch of balls moving around
  • Different colors
  • Offset from each other
  • Change what’s happening with keyboard input
    • Speed
    • Number or balls
    • Pause

• Be sure it gives me a headache if I watch it too long
Graphical program in Elm (dancing balls with keyboard input)

*Elm is a derivative of Haskell*
Make me a pretty program that

• Displays a bunch of balls moving around
  • Different colors
  • Offset from each other
  • Change what’s happening with keyboard input
    • Speed
    • Number or balls
    • Pause

• Be sure it gives me a headache if I watch it too long

• Technically… this is harmonic interference
  • It’s used as the loading animation for Primata (some keys are available for beers)
  • This actually happens with balls on strings of differing lengths
BREAK
Section 4: Pulling it all together

Fitting languages to your problems
Multiple paradigms, implementations/platforms, language evolution and editors
Choosing a paradigm for your problem

• A project often involves different problems requiring different paradigms
• Languages support multiple paradigms
• Choosing a paradigm
  • Functional
    • Compute values from data
  • Imperative
    • Handling state
  • Object-oriented
    • Hierarchies of specialization and inheritance
Choosing a paradigm for your problem

- Complex calculations
  - Functional
- Simulations and games
  - Imperative/Object Oriented
- Basic CRUD
  - Anything, depends on library/ORM (maybe not pure functional)
- Big data
  - Libraries (SciPi)
- Basic web app
  - Depends on library
- Front end web
  - Many languages transpile to Javascript
- Front end native
  - Library dependent – C#, C++, Java?
Choosing a language for a project

• Library support
• Language strengths
  • Relevant paradigms
  • Resilience to change
• Multiple languages possible
  • Added complexity
  • Performance consideration
F# DEMO
Editors, PEP 8 and Linters
Transpilers

ECMAScript

TypeScript

CoffeeScript

BABEL

ES5

ES6
function User(id, firstName, lastName) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
}

User.prototype = {
    getId: function() {
        return this.id;
    },
    getFirstName: function() {
        return this.firstName;
    },
    getLastName: function() {
        return this.lastName;
    },
    setFirstName: function(firstName) {
        this.firstName = firstName;
    },
    setLastName: function(lastName) {
        this.lastName = lastName;
    }
};
class User {
    constructor(id, firstName, lastName) {
        this.id = id
        this.firstName = firstName
        this.lastName = lastName
    }
    getId() {
        return this.id
    }
    getFirstName() {
        return this.firstName
    }
    setFirstName(firstName) {
        this.firstName = firstName
    }
    getLastname() {
        return this.lastName
    }
    setLastName(lastName) {
        this.lastName = lastName
    }
}
class User {
    private id: number;
    private firstName: string;
    private lastName: string;

    constructor(id: number, firstName: string, lastName: string) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
    getId() {
        return this.id;
    }
    getFirstName(): string {
        return this.firstName;
    }
    setFirstName(firstName: string) {
        this.firstName = firstName;
    }
    getLastName(): string {
        return this.lastName;
    }
    setLastName(lastName: string): void {
        this.lastName = lastName;
    }
}
class User
    constructor (@id, @firstName, @lastName) ->

    getId: ->
        @id

    getFirstName: ->
        @firstName

    setFirstName: (@firstName) ->

    getLastName: ->
        @lastName

    setLastName: (@lastName) ->

Neil Green
http://tinyurl.com/NeilGreenES6Compare
ECMA 6
Zen of Python

• Beautiful is better than ugly.
• Explicit is better than implicit.
• Simple is better than complex.
• Complex is better than complicated.
• Flat is better than nested.
• Sparse is better than dense.
• Readability counts.
• Special cases aren't special enough to break the rules.
• Although practicality beats purity.
• Errors should never pass silently.
• Unless explicitly silenced.

https://www.python.org/dev/peps/pep-0020/
Zen of Python

• In the face of ambiguity, refuse the temptation to guess.
• There should be one— and preferably only one — obvious way to do it.
• Although that way may not be obvious at first unless you're Dutch.
• Now is better than never.
• Although never is often better than right now.
• If the implementation is hard to explain, it's a bad idea.
• If the implementation is easy to explain, it may be a good idea.
• Namespaces are one honking great idea—let's do more of those!

https://www.python.org/dev/peps/pep-0020/
THANK YOU!

Kathleen Dollard
kathleen@mvps.org
@kathleendollard on Twitter

Benson Joeris
bjoeris@uwaterloo.ca
@bjoeris on Twitter