Advanced SW Engineering: Dynamically Typed Languages

Uwe Zdun
Software Architecture Research Group
Faculty of Computer Science
University of Vienna
http://cs.univie.ac.at/swa

Definition: Dynamically typed language

Language that does not enforce or check type-safety at compile-time, deferring such checks until run-time

From Dynamically typed languages, Laurence Tratt, Advances in Computing, vol. 77, pages 149-184, July 2009.

Definition: Dynamically typed language

Language that does not enforce or check type-safety at compile-time, deferring such checks until run-time

- Traditional, **simplified**, definition
- While factually true, **this definition leaves out what makes dynamically typed languages interesting**
- For example that they **lower development costs** and provide **flexibility**

Some Historical Remarks

- They trace their roots back to the **earliest days of high-level programming languages**
  - i.e. in the 1950's via Lisp
- Many important techniques have been **pioneered** in dynamically typed languages
  - Examples: lexical scoping, Just-In-Time (JIT) compilation
- From the mainstream's perspective, dynamically typed languages have **finally come of age**

Some Historical Remarks

- Great **variation within those languages** which are typically classified as dynamically typed languages
- Nevertheless all such languages **share a great deal in common**

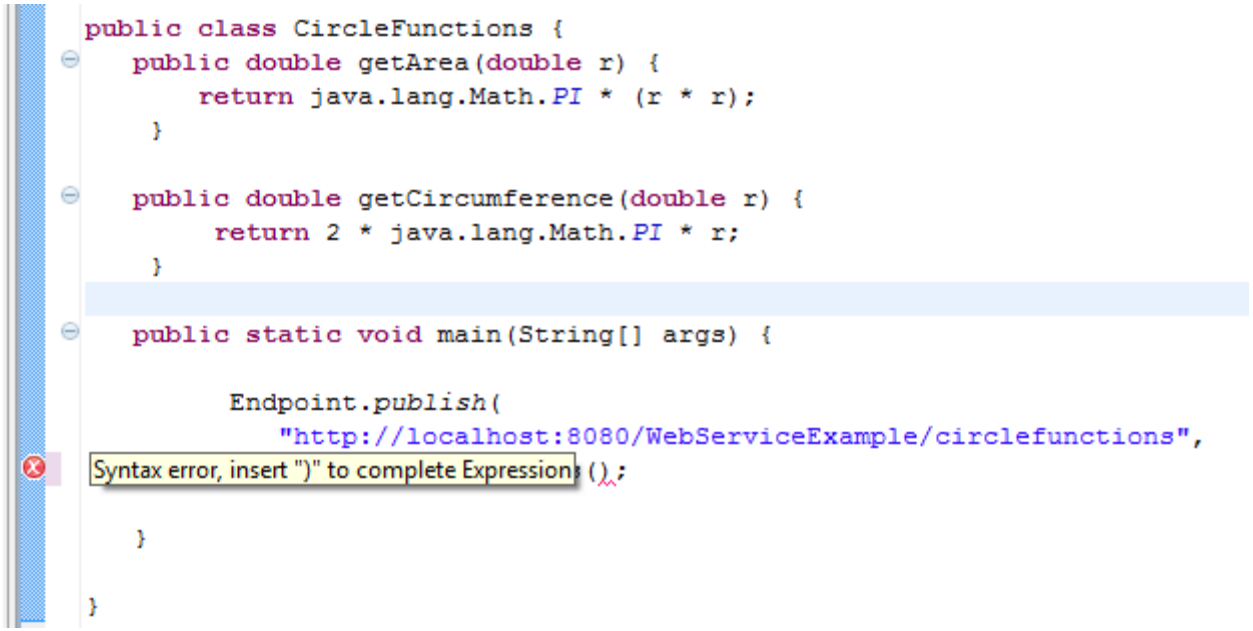
What is a type?

- At an abstract level, a type **is a constraint which defines the set of valid values which conform to it**
- Types are typically organized into **hierarchies**
- In programming languages, types are typically used to both **classify values**, and to determine the **valid operations** for a given type
- Many built-in programming language types represent **finite sets**

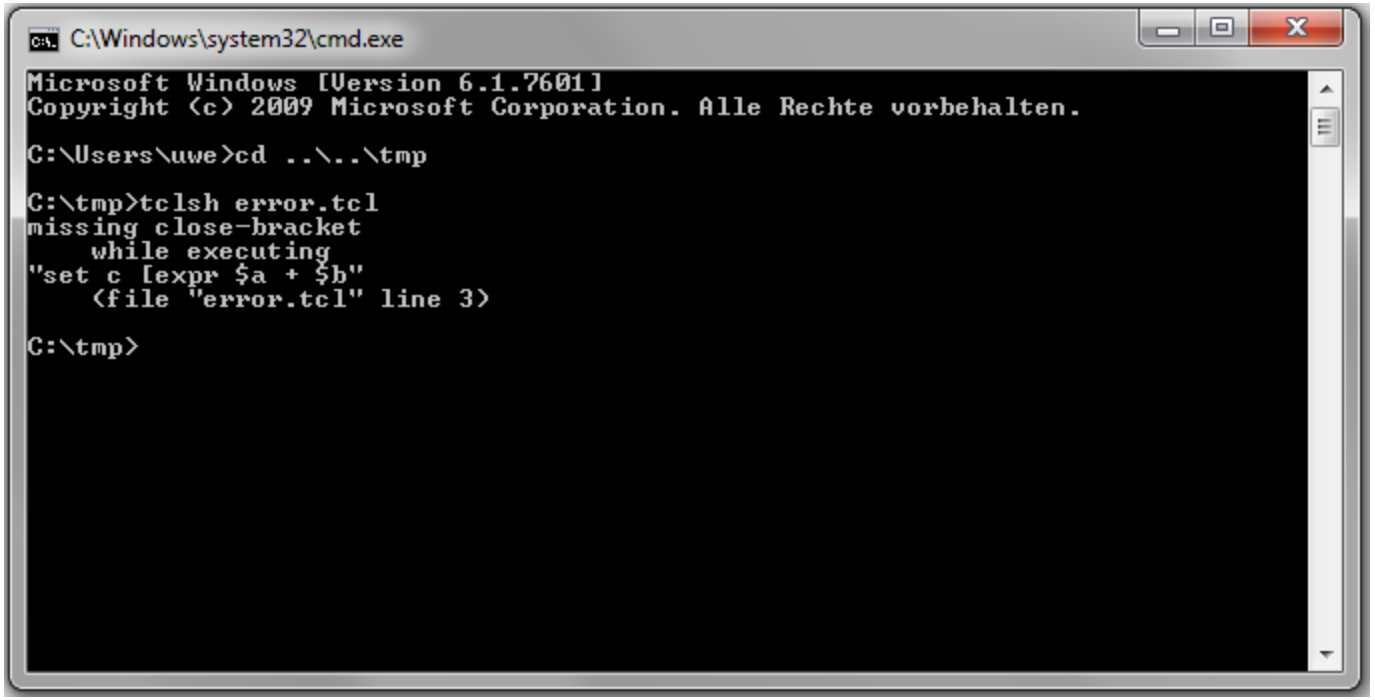
Compile-time vs Runtime (Errors)

- Statically typed languages typically have clearly **distinct compile-time and runtime phases**
- In dynamically typed languages running a file either **both compiles and executes** or **interprets** it
- **Compile-time errors** are those which are determined by analyzing program code without executing it
- **Runtime errors** are those that occur during program execution

Compile-time Errors (in the IDE)



Runtime Error (in the Shell)



Static Typing

What happens in this piece of Java code?

```
int i = 3;
String s = "4";
int x = i + s;
```

This is the essence of static typing: code which violates a type's definition is invalid and is not compiled

Implicit Type Declarations

What happens in this piece of Haskell code?

```
let
    i = 3
    s = "4"
in
    i + s
```

Some statically typed languages can automatically infer the correct type of many expressions, requiring explicit declarations only when automatic inference by the compiler fails

Dynamic Typing

This code in Python yields a runtime error

```
i = 3
s = "4"
x = i + s
```

1. Note that no types are declared
2. Error: TypeError: cannot concatenate 'str' and 'int' objects

Dynamic Typing

Corrected code in Python

```
i = 3
s = "4"
x = i + int(s)
```

Implict Type Conversion

This code in Perl code sets $x to the value of 7

```
$i = 3;
$s = "4";
$x = $i + $s;
```

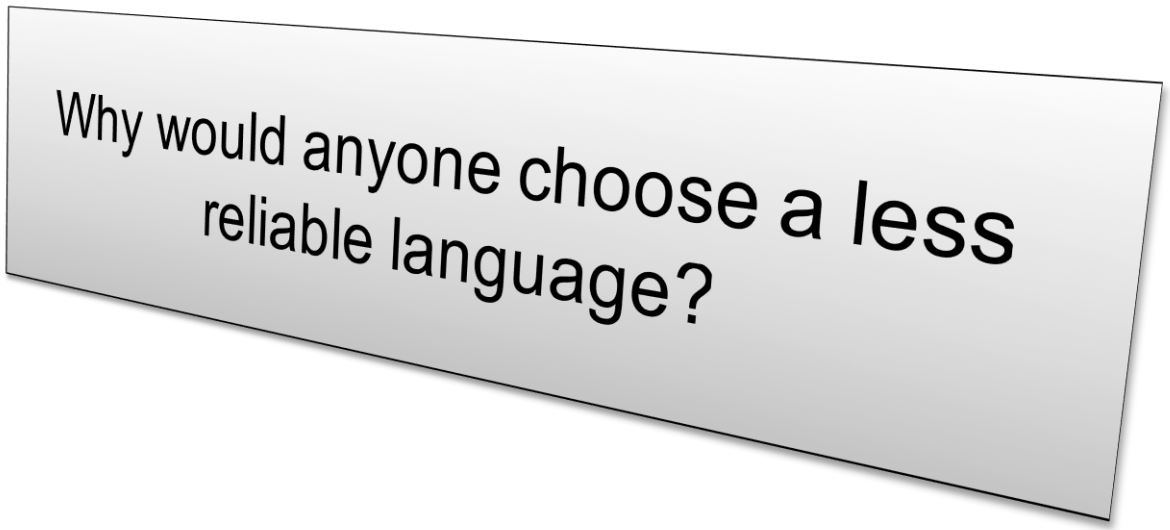
- In many languages – both statically and dynamically typed – a number of implicit type conversions (also known as ‘coercions’) are defined
- In a given context, values of an ‘incorrect’ type are automatically converted into the ‘correct’ type

Overview Typing in Different Languages

	C	Haskell	Java	Perl	Python	Ruby	Tcl
Compile-time type checking	X	X	X	-	-	-	-
Run-time type checking	-	-	X	X	X	X	X
Implicit typing	-	X	-	n/a	n/a	n/a	n/a
Run-time type errors	-	-	X	X	X	X	X
Implicit type conversions	X	-	X	X	-	X	X

Advantages of Static Typing

- Each error detected at compile-time prevents a run-time error
- Types are a form of documentation / comment
- Types enable many forms of optimization



Reasons for not using static typing

- Static types are often inexpressive
  - Just consider this simplistic error in Java
  - It can not be detected by the compiler because int represents numbers including zero

```
int x = 2;
int y = 0;
int z = x / y;
```

Reasons for not using static typing

- Type system complexity
  - From a pragmatic point of view, relatively small increases in the expressivity of static type systems cause a disproportionately large increase in complexity
- Dealing with changes
  - It is often desirable to change small sections at a time and see the effect of that change
  - This can be difficult if the system as a whole must always be type correct
- Runtime dynamicity
  - Software is increasingly required to inspect and alter its behavior at runtime
  - Traditionally most static languages do not offer meaningful reflection, and those that do (e.g. Java), do support only very limited changes

Characteristics

- Simplicity

```
# Hello World in Tcl
puts "Hello world!"
```

```
// Hello World in Java
class HelloWorldApplication {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Characteristics

- Built-in Data types
  - Example: Associative arrays

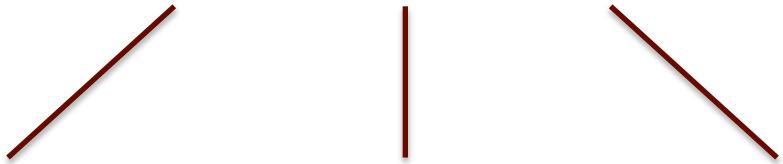
```
set commits(Mo)  0
set commits(Tu)  2
set commits(We)  1
puts $commits(Mo)
```

```
set color(0,0) blue
set color(0,1) red
set color(1,0) brown
set color(1,1) blue
puts $color(0,1)
```

Characteristics

- Automatic memory management
  - Manual memory management wastes programmer resources
    - Consuming perhaps around 30% – 40% of a programmer's time
  - Is a significant source of bugs
- Meta-programming
  - Meta-programming is the querying, manipulation, or creation of one program by another
  - Often a program will perform such actions upon itself

Popular Meta-Programming Technique: Reflection



Introspection

the ability of a program to examine itself

Self-modification

the ability of a program to alter its structure

Intercession

the ability of a program to alter its behavior

Reflection Example in Frag

```
# Preparation
Object create Class1
Object create Class2 -superclasses Class1
Class2 method test {} {
      return "a test"
}
Object create Class3
Class2 create cl2Obj



# Introspection of the superclasses of a class
puts "Superclasses of Class2: [Class2 getSuperclasses]"
```

Reflection Example in Frag

```
# Self-modification: Changing the superclass of Class2
Class2 superclasses Class3
puts "Superclasses of Class2: [Class2 getSuperclasses]"


# Intercession: Changing a method implementation on Class2
# at runtime
puts "Result of method call: [cl2Obj test]"
Class2 method test {} {
      return "a modified test"
}
puts "Result of method call: [cl2Obj test]"
```

Reflection Example in Frag: Output

```
Superclasses of Class2: Class1
Superclasses of Class2: Class3
Result of method call: a test
Result of method call: a modified test
```

Characteristics

- Homoiconic Languages: Evaluate data provided in the language as code

```
set var {$greeting}
set command {puts}
set greeting "Hello World"
eval "$command $var"
```

Characteristics

- Alternative to treat code as data: Closures

```
myMap = ["China": 1 , "India" : 2, "USA" : 3]

result = 0
myMap.keySet().each( { result += myMap[it] } )
println result
```

A closure is an anonymous chunk of code that may take arguments, return a value, and reference and use variables declared in its surrounding scope

Characteristics

- Enabling unanticipated reuse
  - In the style of the UNIX shell

```
find . -name "*.c" | grep -i dynamic | wc -l
```

- Enabling runtime updates through
  - Reflection
  - Running despite possible runtime type errors

Refactoring Comparison

- Refactoring is the act of applying small, behaviour-preserving transformations, to a system

Big advantage of statically typed languages for **small, tightly defined, and automatable refactorings**

Advantage of dynamically typed languages for larger, typically project specific, non-automatable refactorings

Disadvantages of Dynamic Typing

- Performance
- Debugging Support
- IDE Support, such as code completion
- Types as documentation

Relevant Literature and Sources

- Dynamically typed languages, Laurence Tratt, Advances in Computing, vol. 77, pages 149-184, July 2009. http://tratt.net/laurie/research/publications/html/tratt__dynamically_typed_languages/

Many thanks for your attention!

Uwe Zdun

Software Architecture Research Group
Faculty of Computer Science
University of Vienna
http://cs.univie.ac.at/swa