What is the difference between statically typed and dynamically typed languages?

Statically typed languages

A language is statically typed if the type of a variable is known at compile time. For some languages this means that you as the programmer must specify what type each variable is; other languages (e.g.: Java, C, C++) offer some form of *type inference*, the capability of the type system to deduce the type of a variable (e.g.: OCaml, Haskell, Scala, Kotlin).

The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.

Examples: C, C++, Java, Rust, Go, Scala

Dynamically typed languages

A language is dynamically typed if the type is associated with run-time values, and not named variables/fields/etc. This means that you as a programmer can write a little quicker because you do not have to specify types every time (unless using a statically-typed language with *type inference*).

Examples: Perl, Ruby, Python, PHP, JavaScript, Erlang

Most scripting languages have this feature as there is no compiler to do static type-checking anyway, but you may find yourself searching for a bug that is due to the interpreter misinterpreting the type of a variable. Luckily, scripts tend to be small so bugs have not so many places to hide.

Most dynamically typed languages do allow you to provide type information, but do not require it. One language that is currently being developed, Rascal, takes a hybrid approach allowing dynamic typing within functions but enforcing static typing for the function signature.

Compiled vs. Interpreted

"When source code is translated"

- **Source Code**: Original code (usually typed by a human into a computer)
- **Translation**: Converting source code into something a computer can read (i.e., machine code)
- **Run-Time**: Period when program is executing commands (after compilation, if compiled)
- **Compiled Language**: Code translated before run-time
- **Interpreted Language**: Code translated on the fly, during execution

Typing

"When types are checked"

5 + '3' is an example of a type error in *strongly typed* languages such as Go and Python, because they don't allow for "type coercion" -> the ability for a value to change type in certain contexts such as merging two types. *Weakly typed* languages, such as JavaScript, won't throw a type error (results in '53').

- **Static**: Types checked before run-time
- **Dynamic**: Types checked on the fly, during execution

Performance

A compiled language will have better performance at run-time if it's statically typed (vs. dynamically); knowledge of types allows for machine code optimization.

Statically typed languages have better performance at run-time intrinsically due to not needing to check types dynamically while executing (it checks before running).

Similarly, compiled languages are faster at run time as the code has already been translated instead of needing to "interpret"/translate it on the fly.

Note that both compiled and statically typed languages will have a delay before running for translation and type-checking, respectively.

More Differences

Static typing catches errors early, instead of finding them during execution (especially useful for long programs). It's more "strict" in that it won't allow for type errors anywhere in your program and often prevents variables from changing types, which further defends against unintended errors.

```
num = 2
num = '3' // ERROR
```

Dynamic typing is more flexible, which some appreciate. It typically allows for variables to change types, which can result in unexpected errors.

Static typed languages (compiler resolves method calls and compile references):

- usually better performance
- faster compile error feedback
- better IDE support
- not suited for working with undefined data formats
- harder to start a development when model is not defined when
- longer compilation time
- in many cases requires to write more code

Dynamic typed languages (decisions taken in running program):

- lower performance
- faster development
- some bugs might be detected only later in run-time
- good for undefined data formats (meta programming)

Scripting Language vs Programming Language

A scripting language is a programming language that doesn't require a compilation process. For instance, when you run a C program you may have to compile it and then run but when you run JavaScript, there is no need to compile it. So, we can say JavaScript is a form of scripting language.

The principal difference between a programming language and a scripting language is their execution process. Programming language uses a compiler to convert into machine language from the middle and high-level programming language.

1. Meaning

A programming language is a formal language that includes a set of commands that delivers specific results when fed into a system.

A scripting language supports scripts written exclusively for computer programs. Scripts help maintain a particular run time environment to automate the execution of specific functions.

2. Interpretation

Programming languages are assembled into a more compact design. They don't need to be translated by any other application or language.

Scripting languages are written in one format and translated within another program. For example, JavaScript has to be incorporated with HTML and will be further interpreted by internet explorers. So, while programming languages can run independently, scripting languages run within programs.

3. Design

Scripting languages are designed specifically to make coding simpler and faster. Whereas programming languages are used for full-fledged coding and software development.

4. Advancement

Programming languages usually require many lines of code for a single function. However, a scripting language allows for faster coding as you only need to write a few lines to perform a particular function. Scripting languages prefer smaller chunks of code.

5. Category

Programming languages are divided into five categories, which are as follows:

- First Generation
- Second Generation
- Third Generation
- Fourth Generation

- Fifth Generation

Scripting languages are divided into the following categories:

- Client-side scripting language
- Server-side scripting language

6. Hosting and conversion

Scripting languages demand line-by-line conversion, whereas programming languages allow one-shot conversion since they often use a compiler. Also, scripting languages require a host, unlike programming languages that are self-executable.

7. Language

C++, C#, Java, Basic, Pascal, and COBOL are a few examples of programming languages.

JavaScript, PHP, Python, Ruby, Rexx, etc., are some of the examples of scripting language.

8. Speed

Compiled programs generally run faster than interpreted programs. This is because compilers analyze and read the code all at once. In a scripting language, an interpreter analyzes and reads the code line by line, and every time it detects errors, it addresses them one at a time.9.

9. Structure

Programming languages work independently and are self-executable. They do not depend on other hosts and platforms, whereas scripting language requires a host and the structure generally runs in small chunks. Programming language creates .exe files.

Other differences:

- Scripting languages are relatively easier to write, learn, and master, whereas programming languages often come with a steep learning curve.

- Scripting languages are translated and cannot be converted into an executable file, whereas programming languages are generally compiled and created to executable the file.
- Scripting languages can combine existing modules or components, while programming languages are used to build applications from scratch.

Programming Paradigms

A **programming paradigm** is a style, or "way," of programming.

Some languages make it easy to write in some paradigms but not others.

*A paradigm is a way of **doing** something (like programming), not a concrete thing (like a language). Now, it's true that if a programming language L happens to make a particular programming paradigm P easy to express, then we often say "L is a P language" (e.g. "Haskell is a functional programming language") but that does not mean there is any such thing as a "functional language paradigm".*

Some Common Paradigms

You should know these:

- **Imperative**: Programming with an explicit sequence of commands that update state.
- **Declarative**: Programming by specifying the result you want, not how to get it.
- **Structured**: Programming with clean, goto-free, nested control structures.
- **Procedural**: Imperative programming with procedure calls.
- **Functional** (Applicative): Programming with function calls that avoid any global state.
- **Function-Level** (Combinator): Programming with no variables at all.

- **Object-Oriented**: Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
    - **Class-based**: Objects get state and behavior based on membership in a class.
    - **Prototype-based**: Objects get behavior from a prototype object.

- **Event-Driven**: Programming with emitters and listeners of asynchronous actions.
- **Flow-Driven**: Programming processes communicating with each other over predefined channels.
- **Logic** (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.
- **Constraint**: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
- **Aspect-Oriented**: Programming cross-cutting concerns applied transparently.
- **Reflective**: Programming by manipulating the program elements themselves.
- **Array**: Programming with powerful array operators that usually make loops unnecessary.

Paradigms are **not meant to be mutually exclusive**; a single program can feature multiple paradigms!

Imperative Programming

Control flow in **imperative programming** is *explicit*: commands show *how* the computation takes place, step by step. Each step affects the global **state** of the computation.

```
result = []
    i = 0
start:
    numPeople = length(people)
    if i >= numPeople goto finished
    p = people[i]
    nameLength = length(p.name)
    if nameLength <= 5 goto nextOne
    upperName = toUpper(p.name)
    addToList(result, upperName)
nextOne:
    i = i + 1
    goto start
finished:
    return sort(result)
```

Structured Programming

Structured programming is a kind of imperative programming where control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

```
result = [];
for i = 0; i < length(people); i++ {
    p = people[i];
    if length(p.name)) > 5 {
        addToList(result, toUpper(p.name));
    }
}
return sort(result);
```

Object Oriented Programming

OOP is based on the sending of **messages** to objects. Objects respond to messages by performing operations, generally called **methods**. Messages can have arguments. A society of objects, each with their own local memory and own set of operations has a different feel than the monolithic processor and single shared memory feel of non object oriented languages.

```
result:= List new.
people each: [:p |
  p name length greaterThan: 5 ifTrue: [result add (p
name upper)]
]
result sort.
```

Many popular languages that call themselves OO languages (e.g., Java, C++), really just take some elements of OOP and mix them in to imperative-looking code. In the following, we can see that length and toUpper are methods rather than top-level functions, but the for and if are back to being control structures:

```
result = []
for p in people {
    if p.name.length > 5 {
        result.add(p.name.toUpper);
    }
}
return result.sort;
```

Declarative Programming

Control flow in **declarative programming** is *implicit*: the programmer states only *what* the result should look like, **not** how to obtain it.

```
select upper(name)
from people
where length(name) > 5
order by name
^result
```

No loops, no assignments, etc. Whatever engine that interprets this code is just supposed go get the desired information, and can use whatever approach it wants. (The logic and constraint paradigms are generally declarative as well.)

Functional Programming

In **functional programming**, control flow is expressed by combining function calls, rather than by assigning values to variables:

```
sort(
  fix(λf. λp.
    if(equals(p, emptylist),
      emptylist,
      if(greater(length(name(head(p))), 5),
        append(to_upper(name(head(p))), f(tail(p))),
        f(tail(people)))))(people))
```

For now, be thankful there's usually syntactic sugar:

```
let
    fun uppercasedLongNames [] = []
      | uppercasedLongNames (p :: ps) =
          if length(name p) > 5 then (to_upper(name
p))::(uppercasedLongNames ps)
          else (uppercasedLongNames ps)
in
    sort(uppercasedLongNames(people))
```

the real power of this paradigm comes from passing functions to functions (and returning functions from functions).

```
sort(
    filter(λs. length s > 5,
        map(λp. to_upper(name p),
            people)))
```

Logic and Constraint Programming

Logic programming and **constraint programming** are two paradigms in which programs are built by setting up relations that specify **facts** and inference **rules**, and asking whether or not something is true (i.e. specifying a **goal**.) Unification and backtracking to find solutions (i.e.. satisfy goals) takes place automatically.

Languages that emphasize this paradigm: Prolog, GHC, Parlog, Vulcan, Polka, Mercury, Fnil.

Languages and Paradigms

One of the characteristics of a language is its support for particular programming paradigms. For example, Smalltalk has direct support for programming in the object-oriented way, so it might be called an object-oriented language. OCaml, Lisp, Scheme, and JavaScript programs tend to make heavy use of passing functions around so they are called "functional languages" despite having variables and many imperative constructs.

There are two very important observations here:

- Very few languages implement a paradigm 100%. When they do, they are **pure**. It is incredibly rare to have a "pure OOP" language or a "pure functional" language. A lot of languages have a few escapes; for example in OCaml, you will program with functions 90% or more of the time, but if you need state, you can get it. Another example: very few languages implement OOP the way Alan Kay envisioned it.
- A lot of languages will facilitate programming in one or more paradigms. In Scala you can do imperative, object-oriented, and functional programming quite easily. If a language is *purposely* designed to allow programming in many paradigms is called a **multi-paradigm language**.