Typing

Values - Literals that we use in applications that represent data.

- e.g. a string `"hello"`, or a integer `4`

1. What's a Type

A data type is a set of values where you say `v is of type T means v ∈ T`.
If an expression or function is of type `T`, their `result ∈ T`

2. Different Types of Data Types

Primitives

For example in C, Java, and C++ we deal with `int`, `double`, `long`, `char`, etc.

- Just values that get passed around.
- Typically don't have any behavior.

> There are languages that are purely Object Oriented languages such as Ruby and Scala. These languages deal with objects and not primitive types.

Composites

A composite type is a type that is a collection of other types. Typically Objects are composites, as they can be made up of different types.

> Purely Object Oriented languages are made up of composites.

Recursive

For example, an object may have objects which in turn have other objects.

- In the case of a recursive structure, a person is related to another person, and that person may have a relationship with another person.

3. Functional Languages: Lists and List Processing

Functional languages often tend to deal with lists or collections of data much more often.

> There's a special treatment for lists in functional languages. Data is treated immutable entities.

Because the content is immutable, you need to find an efficient way to access lists.
List processing is critical:

- Often functional languages take a list of values and compose a list by taking an element and attaching it to the head of the list.
- Functional languages will operate on a list and often closely do work with the head or the tail of the list.

Consider a list of values in Erlang.

```
main(_) ->
        process([1, 2, 3, 4, 5]).

process([H|[]])
process(L) ->
        io:format("~p-", [L]).
```

Take a look at the syntax on line 6.

```
process([H]|[])
```

We can split a list by its head and the rest of the list. We can also represent the rest of it by a letter `T`.
We can also split and print the list as seen below.

```
main(_) ->
        process([1, 2, 3, 4, 5]).

process([H|[]]) ->
        io:format("~p", [H]).
process([H|T]) ->
        io:format("~p-", [H]),
        io:format("~p-", [T]).
        process(T).
```

Splitting lists is built directly into functional languages such as Erlang. We are then able to process lists much easier.

This shows how a list can be processed into heads and tails. These list operations are provided in these objects, so it provides a different design consideration.

- Designing code in these languages thus is very different than designing code in Object Oriented languages.

4. Static vs Dynamic Typing

Languages can be classified by their typing systems. There are **statically typed languages** and **dynamically typed languages**.

There are noticeable advantages that each typing system has over the other. These are all tools that are great to use in the right case.

Statically Typed Languages

A ***statically typed language*** is a language that will verify the type information of variables, objects, and data used in the program at **compile time**.

Dynamically Typed Languages

A ***dynamically typed language*** is a language that may not even have a compiler. **Type verification is not enforced**, the typing is often somewhat liberal. This is either because it doesn't have a compiler or because the compiler is lenient.

- Rather than ensuring type safety at this time, it works with a language to see if the type of call is supported and thus whether it will fail.

A Statically Typed Example:

```
pubic class Sample {
        public static void main(String[] args){
                int value = 4;
                value = "hello" // this would cause a compilation error.
                System.out.println(value)
        }
}
```

In the code above, a variable declared as an integer obviously cannot be assigned the value of a string.

Yet, this is not to say that static languages like Java don't have inference based type conversions. Let's take look at a quick example:

```
pubic class Sample {
        public static void main(String[] args){
```

```
                double value = 4.0;
                value = 3 //implictly converts the int to a double
                System.out.println(value)
        }
}
```

What happened at compile time:

1. Compiler stored a value of `3.0` into the variable `value`.
    - Compiler accepted the assignment but implicitly converted it to a `double` behind the scenes.

```
value = 1
puts value

value = "hello"
puts value
```

Notice that Ruby did not complain. Ruby is a purely object oriented language so what Ruby did was, it pointed `value` to a `Fixnum` (int) object and then altered this pointer when reassigning `value` to a string.

> A dynamic language allows you to reassign a value to a reference of a different type.

> A static language, when you redefine a reference of an object of a certain will enforce that the new reference is of the same type.

Do NOT assume that a Dynamically typed language is one where you don't specify a type and a static language is one where you do define a type.

That is a wrong assumption. It is not true the the difference is specifying a data type.

- If you look at Java, we did in fact specify the data type. However, static typing does not mean more typing. It simply means that the type information is verified at compile time.

Let's use Scala, a statically typed language, as an example.

```
var value = 1
println(value)
```

```
value = 4
println(value)
```

Notice how we did not specify the data type of the variable anywhere. And Scala runs this code perfectly fine. Why?

> Scala supports **type inference**.

Type inference means we the language is smart enough to infer the type based on the context.

- In our example, Scala sees that we initialize `value` with the value `1`. Thus Scala infers that the data type of our variable is an integer.

Now let's look at this a bit more. What if we assigned `value` to a different data type? What happens?

```
var value = 1
value = "hello"
println(value)
```

We receive a **compilation error** that says we required an Int yet we found a string value.

- The compiler was expecting an integer and is now enforcing it at compiling time.

> The less statically typed a language is the more typing you will have to do and provide more details because the language is not adequate in figuring out what these type details are for you.

5. Which one is better?

A Drawback to Statically Typed Language
Times Where You Need to Lowering of Type Checking
Just because a language is statically typed it does not mean that all the type information is verified perfectly at compile time.

- There are times when you will have to resort to the lowering of type checking even in static typed languages. There may be times when you can't infer a type correctly at compile time.
- Or in the case where you have a very generalized object.

Let's take look at such an example:

```
public class Sample {
        public static void main(String[] args){
```

```
                List scores = new ArrayList();
                scores.add(1);
                scores.add(2);

                int total = 0;
                for(Object e : cores) {
                        total += (Integer) e;
                }
                System.out.println("the total is, " + total);
        }
}
```

In this example, it thinks that scores is returning objects and not integers, and this is because we are using a `List`. So we have to go ahead and cast as you see above in our for loop.

- this issue sits on a massive flaw. Our list can hold objects but not all objects have to be of the same data type. So what happens if we added a double to our List and we keep the `Integer` casting?

```
public class Sample {
        public static void main(String[] args){
                List scores = new ArrayList();
                scores.add(1);
                scores.add(2);
                scores.add(1.0)

                int total = 0;
                for(Object e : cores) {
                        total += (Integer) e;
                }
                System.out.println("the total is, " + total);
        }
}
```

The code above will result in an casting exception error. Because we can't cast a double to an int like this.

You could fix by providing better type safety by declaring that the List as a list of integers. You then will not need to do the casting yourself in the source code. However, the casting will still be done under the hood.

```
public class Sample {
        public static void main(String[] args){
                List<Integer> scores = new ArrayList<Integer>();
```

```
                scores.add(1);
                scores.add(2);

                int total = 0;
                for(int e : cores) {
                        total += e;
                }
                System.out.println("the total is, " + total);
        }
}
```

If we add a double, `scores.add(1.0)` for instance, we will cause a compilation error. However if we pass this list to a function our integer casting will run without a compilation error! BUT, when we run the code, the cast exception error has come back!

```
public class Sample {
        public static void oops(List list) {
                list.add(1.0);
        }
        public static void main(String[] args){
                List<Integer> scores = new ArrayList<Integer>();
                scores.add(1);
                scores.add(2);
                oops(scores);

                int total = 0;
                for(int e : cores) {
                        total += e;
                }
                System.out.println("the total is, " + total);
        }
}
```

This is an example of the static typing in Java can still get by and cause incorrect casting.

Benefits of Static Typing

There are many benefits to Static Typing:

- Built in type safety brought through the compiler's verification.

Benefits of Dynamic Typing

There are many benefits to Dynamic Typing:

- It's easier to write ***metaprogramming*** - Can write a program that can write part of the program at runtime.
- Freer, not as limited by the compiler.

Let's quickly look at Metaprogramming

Java will not allow you to add this function at run time, as it throws a compilation error before you can even run the program.

```
public class Sample {
        public static void main(String[] args){
                String greet = "hello";
                System.out.println(greet);

                greet.shout(); //this gets rejected at compile time.
        }
}
```

Groovy is an example of a language that supports both static typing and dynamic typing. Look at how Groovy lets us run this following code. Despite not being able to identify the method `shout` at compile time.

```
greet = "hello"
println greet

println greet.shout()
```

So let's add a try catch and see what happens

```
greet = "hello"
println greet

try {
        println greet.shout()
} catch(Exception ex){
        println "oops, looks like I can't do that"
}
```

It still compiles and runs! We catch the error and now we know to add the method! It doesn't tell you that you can't do it. It's **far less restricting**. It beckons you to try things out!

```
greet = "hello"
println greet

try {
        println greet.shout()
} catch(Exception ex){
        println "oops, looks like I can't do that"
}

println "Let's add it!"
String.metaClass.shout = {->
        delegate.toUpperCase()
}

println greet.shout()
```

Let's take a quick detour over to an example in Ruby, a dynamic language.

```
class Person
        def work
                puts "working..."
        end
end

same = Person.new
sam.work

sam.playTennis
sam.playFootball
```

We catch a runtime error where we have multiple methods that are not defined. So let's try to fix that by adding a function called `method_missing`.

```
class Person
        def work
                puts "working..."
        end

        def method_missing(name, *args)
                puts "you called #{name}"
        end
end

same = Person.new
```

```
sam.work

sam.playTennis
sam.playFootball
```

Notice now, rather than crashing and complaining we now invoke `method_missing`.

```
class Person
        def work
                puts "working..."
        end

        def method_missing(name, *args)
                activities = ['Tennis', 'Football']

                activity = name.to_s.split('play')[1] # method name starts with
play

                if (activities.include?(acitivity)) # check activity named.
                        puts "I'd like to play #{activity}"
                else
                        puts "Nope, I don't play #{activity}"
                end
        end
end

same = Person.new
sam.work

sam.playTennis
sam.playFootball
sam.playPolitics
```

Notice how we dynamically change the behavior of the code.
Think of a real world example of this. You may need to query a database at runtime and you can either end up being accepted or declined.

> This is exactly what frameworks like Ruby on Rails provides, dynamic capabilities for you based on certain command states.

Note that dynamic typing does not impede on metaprogramming.

So having looked at both static and dynamic languages, we return back to the big question. Which one is better? Well it depends on the given problem on hand.

- You may have an application that needs decent compile time and interface verifications (Static >>> Dynamic)
- You may need to write code which utilized metaprogramming, or a fully dynamic domain specific language (Dynamic >>> Static)
  Whether you should use a static or dynamic language depends on the best way to achieve your goal. These languages are tools, it is a vehicle.
- You cannot be adamant and say only one type of typing is correct.
- We should have the flexibility to pick and choose the language that best fits your needs.

In conclusion they both have pros and cons.

6. Design by Contract vs Design by Capability

The languages that you use affect your software design!

When you code in a statically typed language you often use what is called a design by contract.

- You ask the compiler "Do you support this?" and follow its restrictions at compile time.
- If an object conforms to this interface then you are **guaranteed to know** that this object provide its services.
  Dynamic languages do not typically use design by contract but rather use a **Design by Capability**.
- If an object supports a certain method it's more of a handshaking. You approach your coding like "Oh can I do?" often experimenting as you type.

7. Strong vs Weak Typing

Does your programming language or the run-time that it stands on, ensure that the object that you are dealing with is the proper type when you are exercising it's behavior?

For example, you have an object and using a static typing system you can verify that it is of a certain type. What if the static typing allowed you to cast the object to a certain type.

- In the case of C and C++, you can take a pointer to an object and cast it.

```
Car* pCar = (Car*) pObj;
pCar->drive(); //C++
```

What happens if `pObj` isn't really a `Car` object? You perform a casting that is forcing the conversion of this pointer to a pointer of a different type. (**Coercion**).

In C++, once you get past the static typing of the compiler, at runtime there are no guarantees. The behaviors are unpredictable. **This is an example of a weakly typed language.**

In a **strongly typed language**, the type checking/verification happens at the time the program is running. The type is verified at run-time.

> Java is an example of a **strongly typed language**

```
public class Sample {
        public static void main(String[] args) {
                Object obj1 = "hello";
                Object obj2 = 2;

                use(obj1);
                use(obj2);
        }
        private static void use(Object obj) {
                String str = (String) obj;

                System.out.prinln(str);
        }
}
```

Notice in this example there's a class cast exception on line 12 at run time. Everything compiles fine, but Java is able to find a run-time exception.

> In C++, a weakly typed language, if you aren't given a seg-fault a bug can be hard to find as the program misbehaviors in a subtle manner. Because the language itself is weakly typed.

> People often assume that static typing means strong typing and dynamic typing means weak typing. This is **NOT** true. Strong typing is a type verification at run time it has nothing to do with if verification is done at compile time.

Let's take a look at an example in Groovy (**Strongly Dynamic**)

```
def inst1 = "hello"
String inst2 = "hello"

println inst1
println inst1.class

println inst2
```

```
println inst2.class

inst1 = 1 // assigne new Integer reference to inst1
inst2 = 1 // called toString() obtained String instance and set inst2 to it

println inst1
println inst2
println inst1.class
```

Notice that at first `inst1` pointed to an instance of a string object, then it was reassigned to an integer. The `def` keyword allows for Groovy to use type inference. While on line 2, your string variable was declared as a string and thus its type was enforced on line 11.

> The type is specified, thus showing Groovy's optional typing. However, Groovy slants towards being a dynamically typed language despite the small tinge of static typing permitted within it. As seen in the example below.

```
def inst1 = 1
Integer inst2 = 1

println inst1
println inst1.class
println inst2
println inst2.class

inst1 = "A"
inst2 = "A" // converted A to an Integer under the hood.

println inst1
println inst1.class

println inst2 //printing 65 not an A
println inst2.class // pointing to an object of an Integer
```

Whenever you assign an object to an instance reference you are performing a type conversion behind the scenes.

Because we assigned `inst1` to an integer (`1`), `inst1` is defined to point at an `Integer` so when we reassign it to a string, that string's value is implicitly converted.

```
def inst1 = 1
Integer inst2 = 1

println inst1
```

```
println inst1.class
println inst2
println inst2.class

inst1 = "A"
inst2 = "A" // converted A to an Integer under the hood.

println inst1
println inst1.class

println inst2 //printing 65 not an A
println inst2.class // pointing to an object of an Integer

// Say we assign inst1 to a newly constructed String
inst1 = new StringBuilder("hello")

println inst1
println inst1.class

inst2 = new StringBuilder("hello")
println inst2
```

A type error is caused from line 18 (`inst1 = new StringBuilder("hello")`),
`Cannot cast object 'hello' with class 'java.lang.StringBuilder' to class 'java.lang.Integer'

In other words, Groovy has caught a cast exception because it doesn't know how to cast a `StringBuilder` to an `Integer` so it bailed out. Showing how it is strongly typed.

- A weakly typed language would have allowed such a conversion.