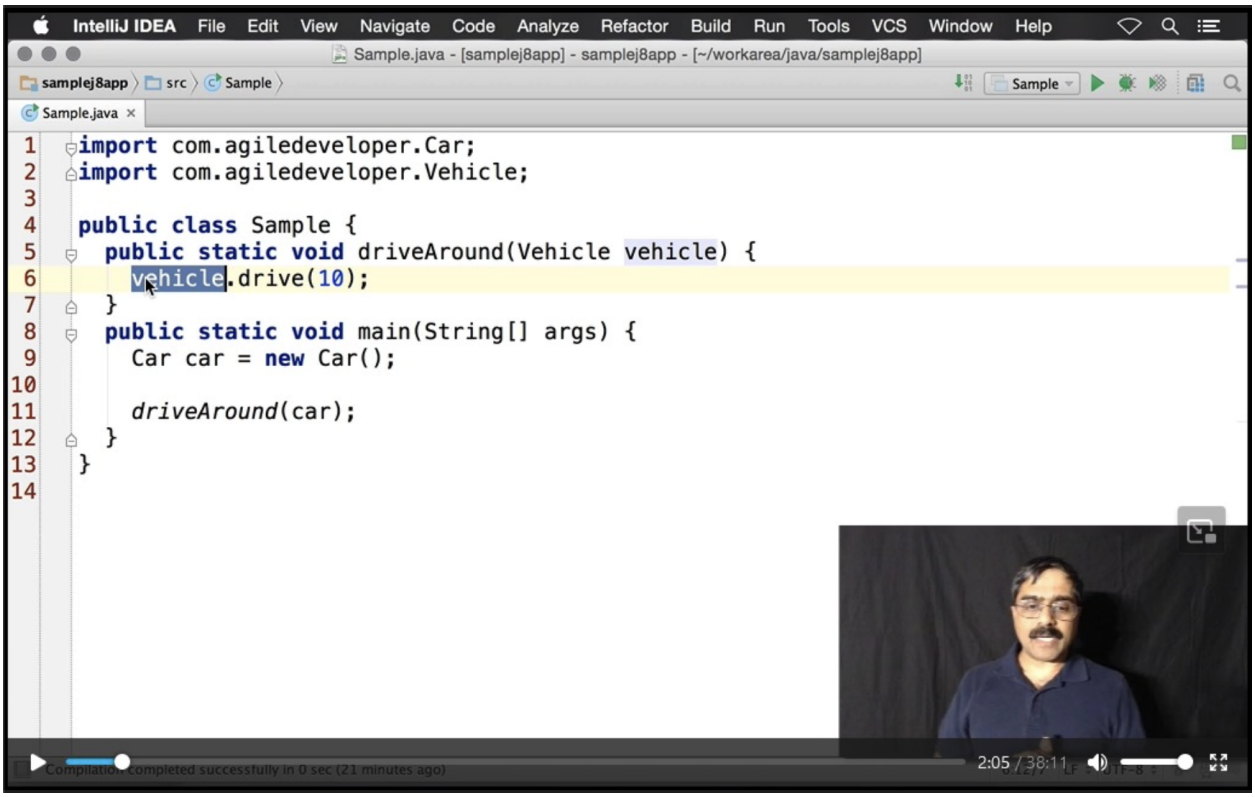
Flavours of Polymorphism

- Without polymorphism, not a whole lot of OOP is left.
- With Encapsulation and Polymorphism, we can create very powerful extensible systems.
- Encapsulations gives the ability to preserve the invariants and also helps us to evolve interfaces and implementations separately. Combined with polymorphism, we get the real benefits of extensibility.
- Inheritance is the weakest link among OOP concepts.

What's Polymorphism?

When you invoke a method on an object, the method is not based on the type of the reference at the compile time, but on the type of the object at the runtime.

In the following image, `driveAround` method is calling the `drive` method of the `Vehicle` class. The type of the reference is `Vehicle`, but we do not know what the type of the class is going to be. `driveAround` could be called by different objects, such as `Car` or `Truck` or `Boat`. We do not know what `drive` method will be called at runtime — that's Polymorphism.



What's the big deal?

Polymorphism provides extensibility.

Considering the above example, you can create new classes that implement/extend `Vehicle` class and the `driveAround` method can be reused.

Inheritance

Inheritance doesn't lead to Polymorphism, or Polymorphism doesn't require Inheritance.

Languages like Java, C++, C#, we rely on inheritance quite heavily.

Static vs. Dynamic

Java is a statically typed language. In statically typed languages, often we use inheritance as a gateway to arrive at polymorphism.

However, in dynamically typed languages, this isn't the case. Eg. Ruby cares about whether the method is available in the instance, not the type of the instance.

Considering the above example, the `driveAround` method would work with both `Vehicle` and `Car` even if `Car` *does not* inherit from `Vehicle`, simply because both have the `drive` method.

`new` is Polymorphic in Ruby, but not Polymorphic in C++, C#, or Java.

JavaScript has prototypal inheritance, not regular inheritance.

Polymorphism doesn't care about inheritance. It just calls the method based on the object at the runtime.

- *Static/Dynamic Typing* is about **when** type information is acquired (Either at compile time or at runtime)
- *Strong/Weak Typing* is about **how strictly** types are distinguished (e.g. whether the language tries to do an implicit conversion from strings to numbers).

Source: https://stackoverflow.com/questions/2351190/static-dynamic-vs-strong-weak

87

Static typing vs dynamic typing:

Static typing is when your type checking occurs at compile time. You must define a type for your variables inside of your code and any operations you perform on your data would be checked by the compiler.

Dynamic typing is when your type checking occurs at runtime. Instead of errors coming up when you compile your code you will get runtime errors if you try performing operations on incompatible types. However, you will get the benefit of having more versatile functions as they can be written once for multiple data types.

Strong typing vs weak typing:

When you have strong typing, you will only be allowed operations on the data by direct manipulation of the objects of that data type.

Weak typing allows you to operate on data without considering its type. Some language do this through pointers. Other languages will convert one of your types to the other before performing the operations.

The links I included have a bit more detailed (and probably clearer) explanations.

Source: https://stackoverflow.com/questions/11889602/difference-between-strong-vs-static-typing-and-weak-vs-dynamic-typing

Good resource for Static/Dynamic & Weak/Strong: https://medium.com/@cpave3/understanding-types-static-vs-dynamic-strong-vs-weak-88a4e1f0ed5f

Weak vs. Strong

In the case of weakly typed languages, it's garbage in, garbage out; You don't know what type with which you're dealing.

In the case of strongly typed languages, type checking happens at runtime. So, calling a method on the wrong type in Java, you'd get a compilation error, however, if you passed that by conversion, you'd get a `ClassCastException` at runtime. That's why Java is statically and strongly typed language.

Ruby is dynamically & strongly typed. The runtime will tell you if you called a method that doesn't exist.

JS & Perl are weakly typed.

- In Java, the parameter decision is made at the compile time. The function decision is made at the runtime. (Ref. video ~32 mins)

Multimethods

The parameter decision is made at compile time, while the function is decided at runtime based on the object. Eg. `func(2)` may be converted to `func(2.0)` at compile time, because the base class may have the method with the double value as the parameter. If a class does not have a method that takes a double value, then the method in the base class will be used at runtime.

- Closure, Groovy have this. Java doesn't have this.
- Polymorphism: Pick the method based on the type of the target object at runtime.
- Multimethods: Pick the method based on the type of both the target object and the types of the parameters at runtime.