Kotlin

https://kotlinlang.org/docs/classes.html

Expressions

```
1 + 2

"Hello world"

course.getName()
```

Kotlin syntax is based on Java, but with **substantial** improvements.

Symbols and bindings

```
val name: String = "Albert Einstein"

var age: Int = 50

age = age + 1

name = "Programing Languages" ❌
```

Symbols are declared as either:

- val: value
- var: variable

Value bindings cannot change.

Variables can re-bind to a different expression later on.

Symbols and type annotations

```
val name: String = "Albert Einstein"

var age: Int = 50

age = "hello world" ❌
```

We annotate the symbols at declaration with their type signatures.

Type checking ensures that the symbols are bound to instances of their respective type.

Type inference

```
val name = "Albert Einstein" ✅
var age = 50 ✅
age = "hello world" ❌
```

Type annotation can be omitted **if** the compiler can figure out what type the symbols should have.

All symbols must be initialized at declaration (unless *lateinit* is used).

Nullable Types

```
var x: String = "Albert Einstein" ✅
x = null ❌

var y: String? = "Albert Einstein" ✅
y = null ✅
```

Kotlin is NULL-safe. It's possible to ensure that a value is **never** the null value.

By default, all Kotlin types are not nullable.

To allow a value to be nullable, its type must be T?

Null Safety

```
var x: String? = "Albert Einstein" ✅
println(x.get(0)) ✅
x = null ✅
println(x.get(0)) ❌


println(x?.get(0)) ✅
```

If obj is nullable, then we can use safe calling of members and methods of the object to avoid runtime problems.

obj?.method(...) will call the method only if obj is not null.

It is equivalent to:

```
if(obj != null) obj.method()
```

Concrete Classes: Kotlin style

```
class Car {
    val make: String ❌
    var miles: Int ❌
}
```

Kotlin requires initialize of all members at declaration.

Concrete Classes: Kotlin style

```
class Car(make_: String, miles_: Int) {
    val make: String = make_
    var miles: Int = miles_
}
```

Classes can be parameterized. The parameters allow us to construct *different* instances of the class.

Class structure

```
class Name(parameters...) {
```

- *member declaration + initialization*

```
    init {
        code to be executed during construction
    }
    constructor(parameters...): this(...)
    {
        ...
    }
```

- *method declarations*

```
}
```

The **primary** constructor is described by the parameters and the *init* { ... } block.

A class can have secondary constructors specified by *constructor(...)*

Concrete Classes: Kotlin style

```
class Car(make_: String, miles_: Int) {
    val make: String = make_
    var miles: Int = miles_
}
```

Equivalent to

```
class Car(val make: String, var miles: Int)
```

Better written as

```
class Car(
  val make: String,
  var miles: Int
)
```

We can declare and initialize members directly in the parameter.

Concrete Classes: Kotlin style

```
class Car(
  val make: String,
  var miles: Int
) {
  init {
    if(miles > 1E6) {
      println("$make is over 1M miles.")
    }
  }
  constructor(make: String): this(make, 0)

  fun drive(distance: Int): Unit {
    miles = miles + distance
  }

  override fun toString(): String {
    return "$make with $miles miles"
  }
}
```

Let's use some features of the language:

1. Primary constructor with val and val members
2. Secondary constructor that relies on primary constructor for initialization
3. Init block
4. Method that modifies var member
5. Override default string renderer

Using concrete classes to create objects

```
val mycar = Car("Toyota", 130_000)

// driving to Alberta
mycar.drive(3_000)
println(mycar)

val newcar = Car("Subaru")

// drive to Toronto
newcar.drive(50)
println(newcar)
```

Classes are actually treated as functions.

A class returns instances.

Extension by inheritance

- Create a new type (class) T based on an existing type S.
- T is the sub-class of S, and S is the super-class of T.
- Every instance of T is also an instance of S.

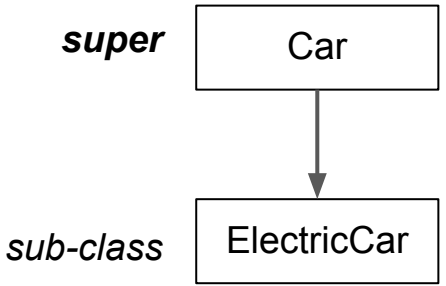
Creating instance of T:

- T has constructor(s).
- Constructors of T **must** rely on constructors of S.

Extension by inheritance

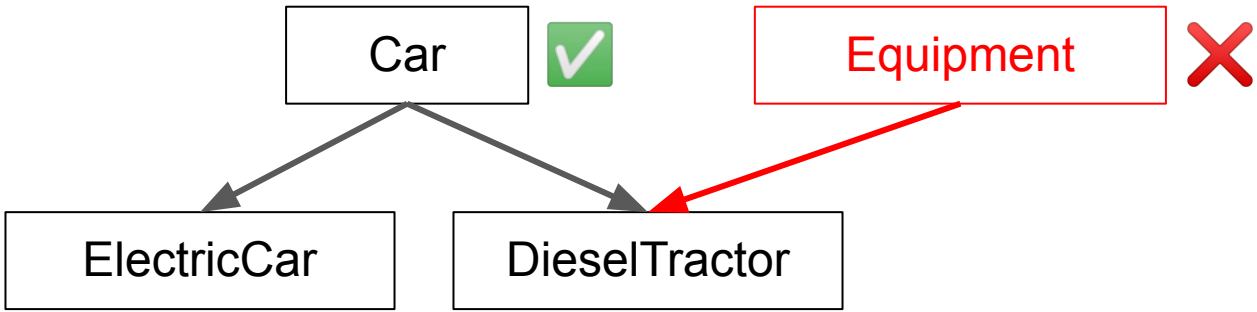
```
class ElectricCar(
  make: String,
  miles: Int,
  val voltage: Float,
): Car(make, miles) {
  fun charge() {
    ...
  }
  override drive(distance: Int) {
    super.drive(distance)
    ...
  }
}
```

Cars can be extended to something more complex.



Extension by inheritance

Super is unique.



Composition

- Create new type T with members of existing types S1, S2, ...
- Usually, instances of T are not instances of Si

Composition

```
class Camper(
  carMake: String,
  carMiles: Int,
  occupancy: Int,
) {
  val car = Car(carMake, carMiles)
  val trailer = Trailer(occupancy)

  fun drive(miles: Int)
    = car.drive(miles)
}
```

Consider a camper: