Applikationsudvikling: CS101

Inheritance & Access modification

NIFR 2024

Agenda

Inheritance and object access modification

- Classes & Objects
- Concept of inheritance
- Inheritance in Kotlin
  - Open classes
- Overriding methods
- Properties and access modification

What is a class?

What is a class?

Object oriented programming

- Classes are *blueprints* of objects
- Objects are collections of data (properties) & functions (methods)
- Objects are instantiated by a constructor
- Objects have their own state and place in memory

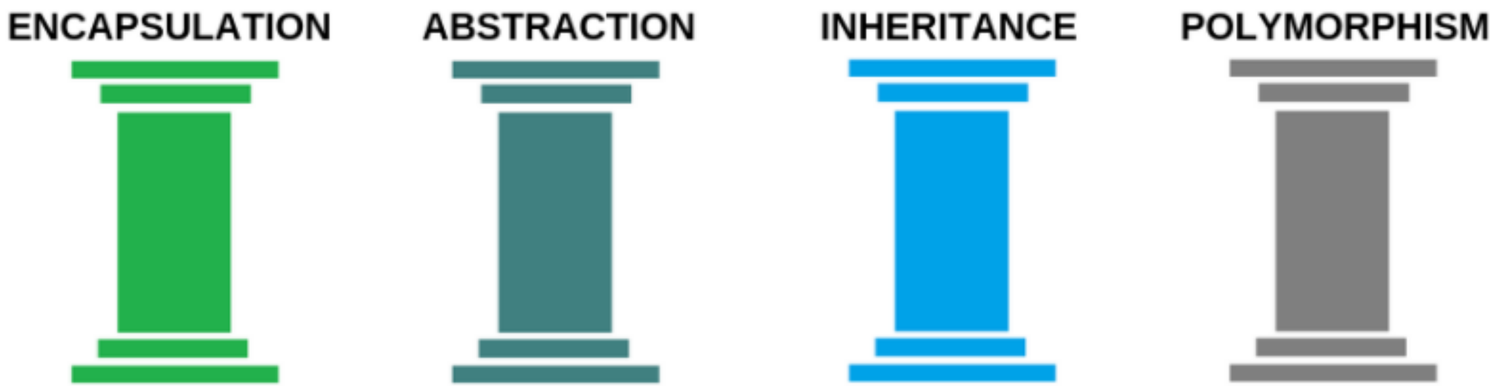
```
class Song(title: String, artist: String, lengthInMs: Int) {
    val lengthInMS: Int = lengthInMs;
    val lengthInMinutes : Int get() {
        return lengthInMS / 1000
    }
}

fun main() {
    val hella = Song("Hella", "Ukendt Kunstner", 4000);
    val happy = Song("Happy", "Pharell", 4250);
    val night = Song("Night", "Benga", 4500);

}
```

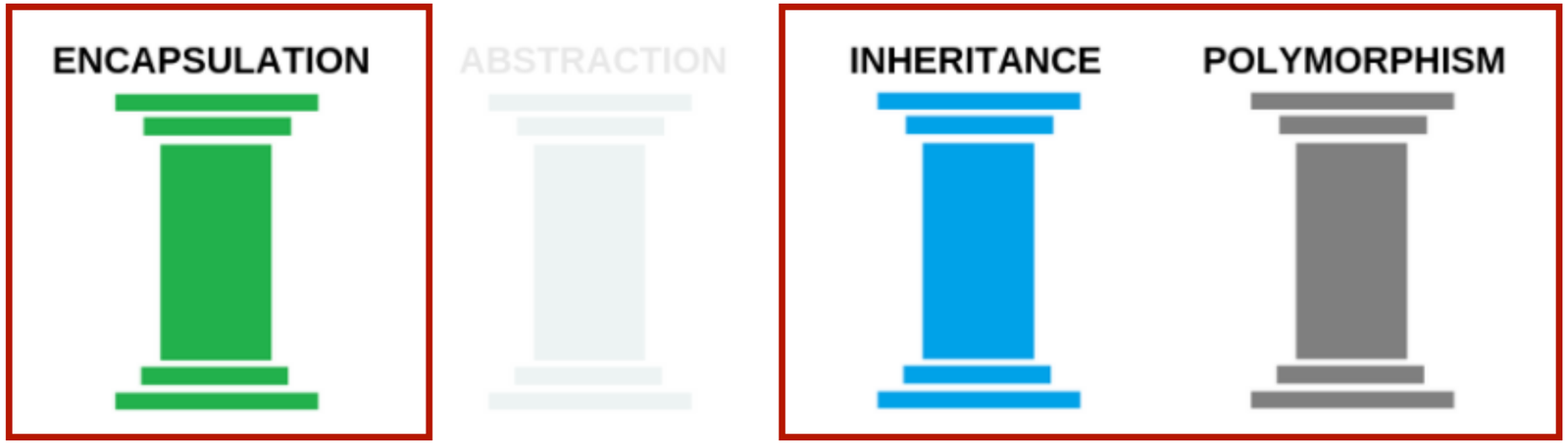
Object oriented programming

4 pillars of OOP

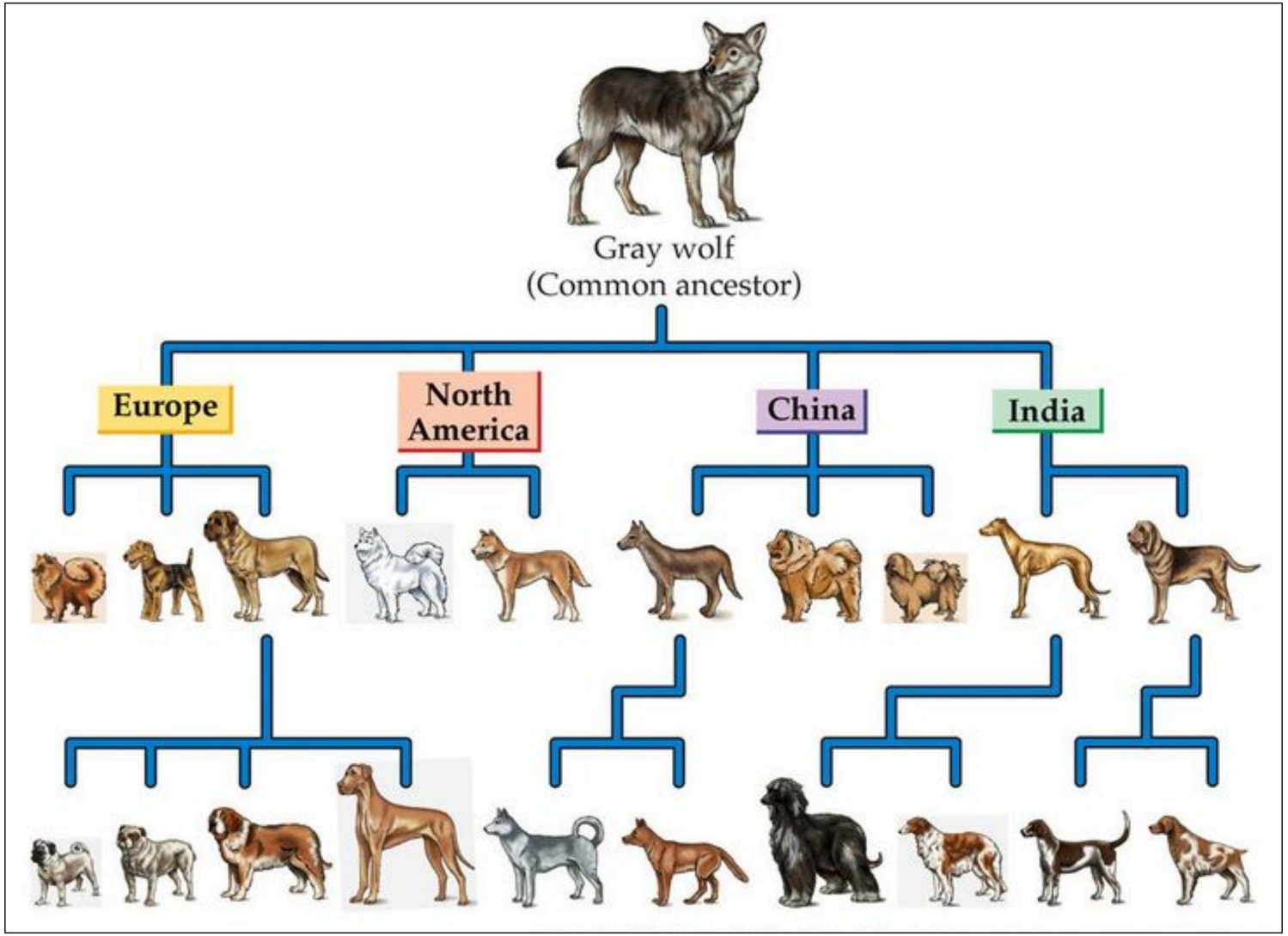


Object oriented programming

4 pillars of OOP



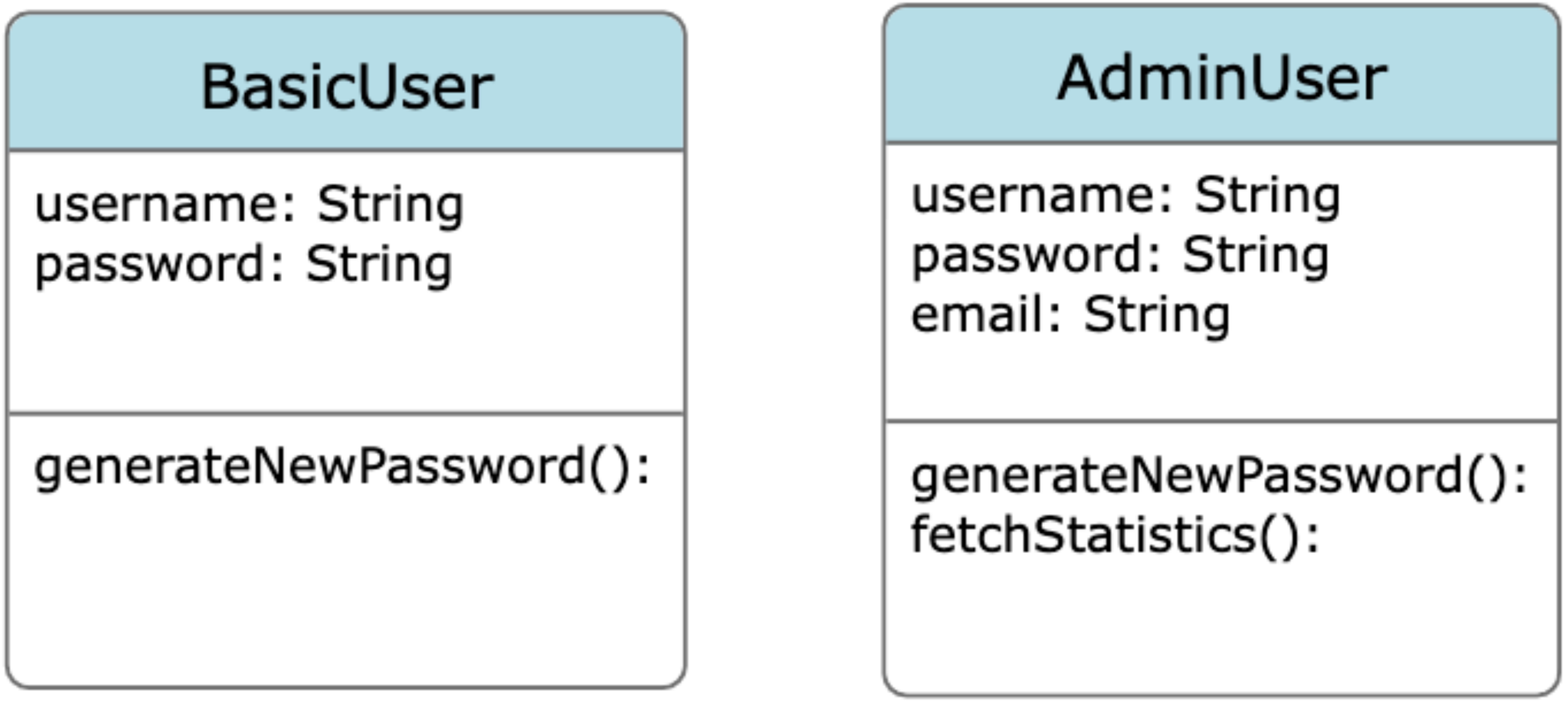
INHERITANCE



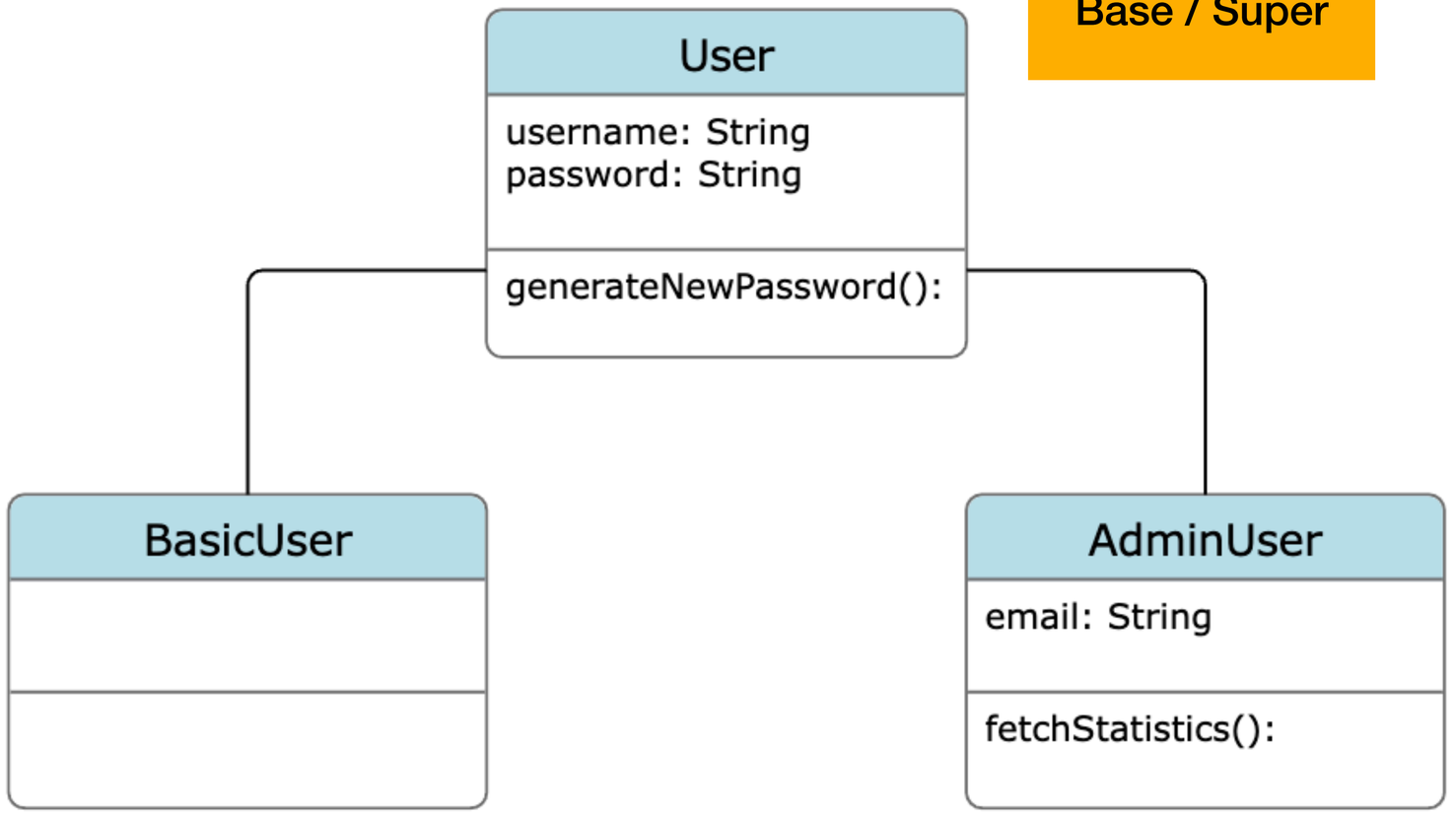
What problem is inheritance solving?

D.R.Y

Don’t Repeat Yourself



Problem: Duplicated code with same intention

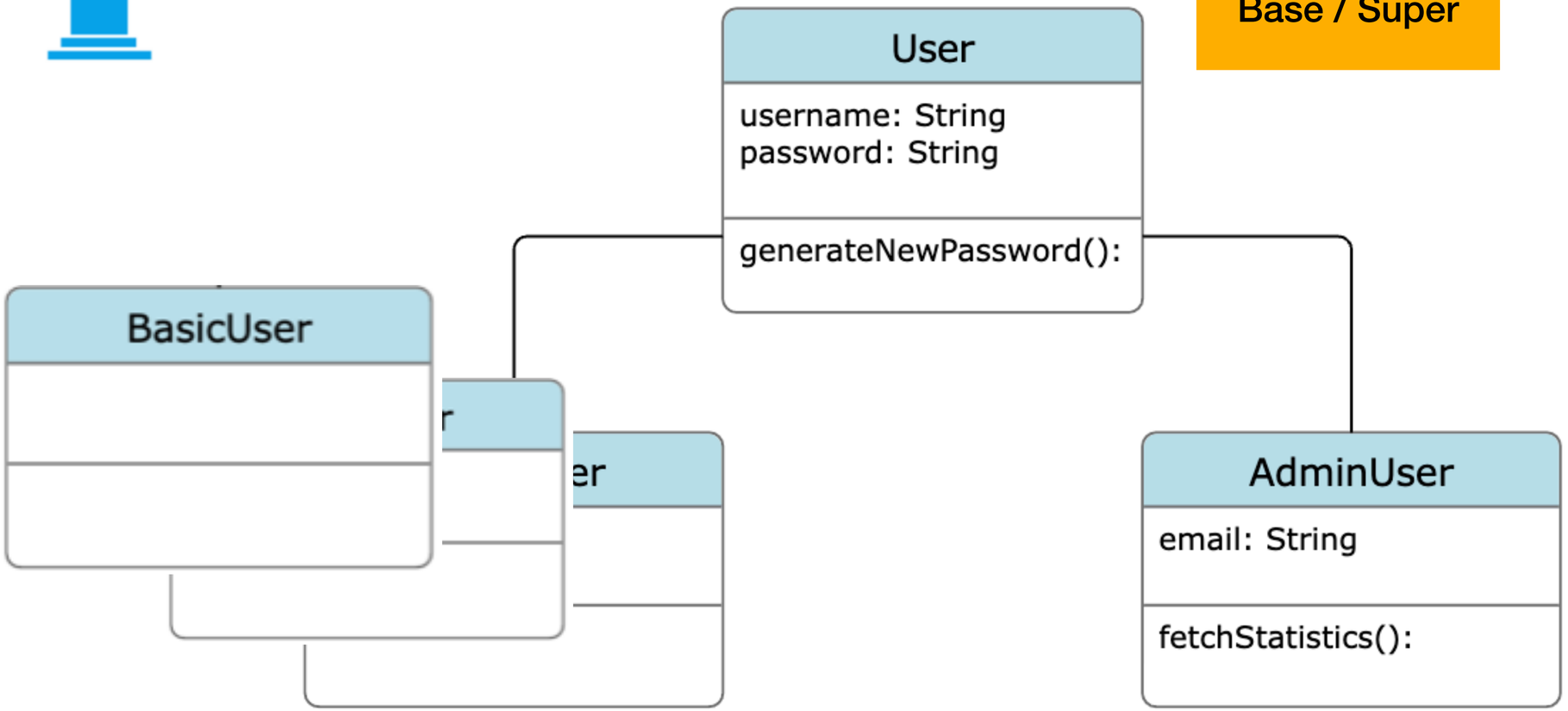


Base / Super

Derived / child / sub

Solution: Inheritance

Base / Super



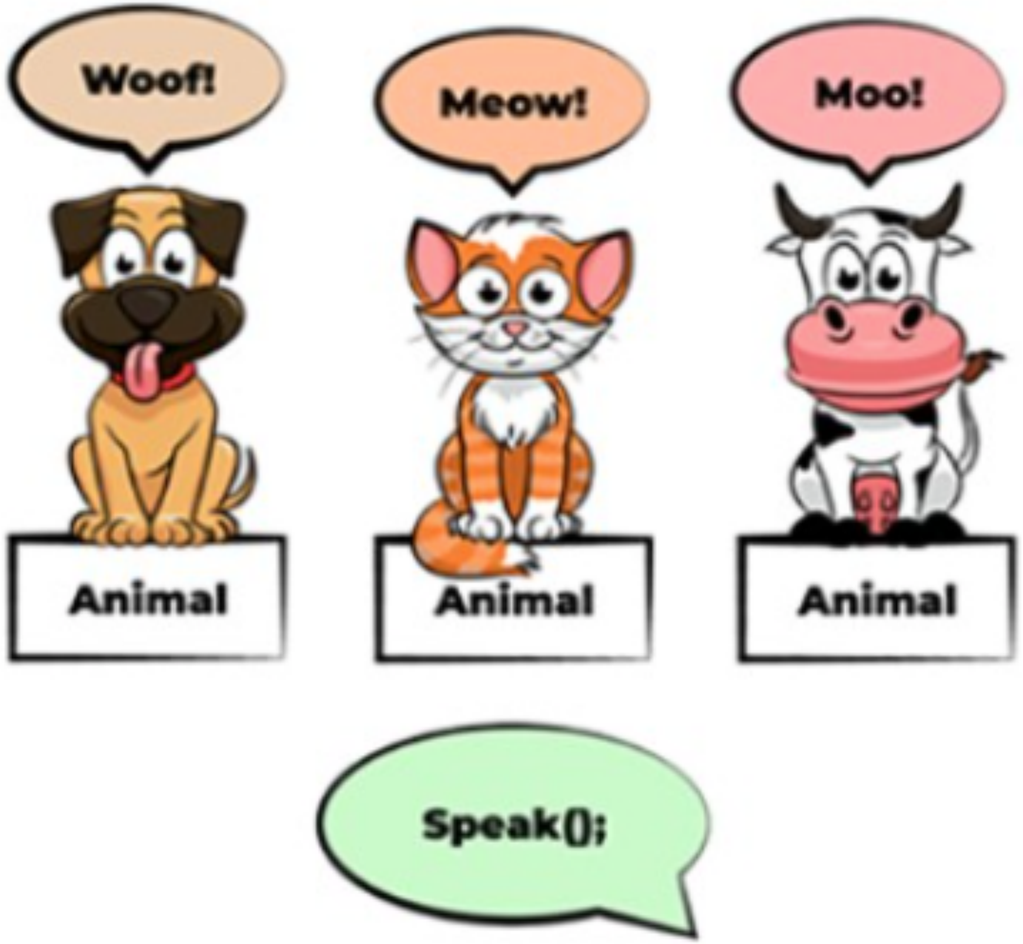
Derived / child / sub

Scalability

Example: User Inheritance

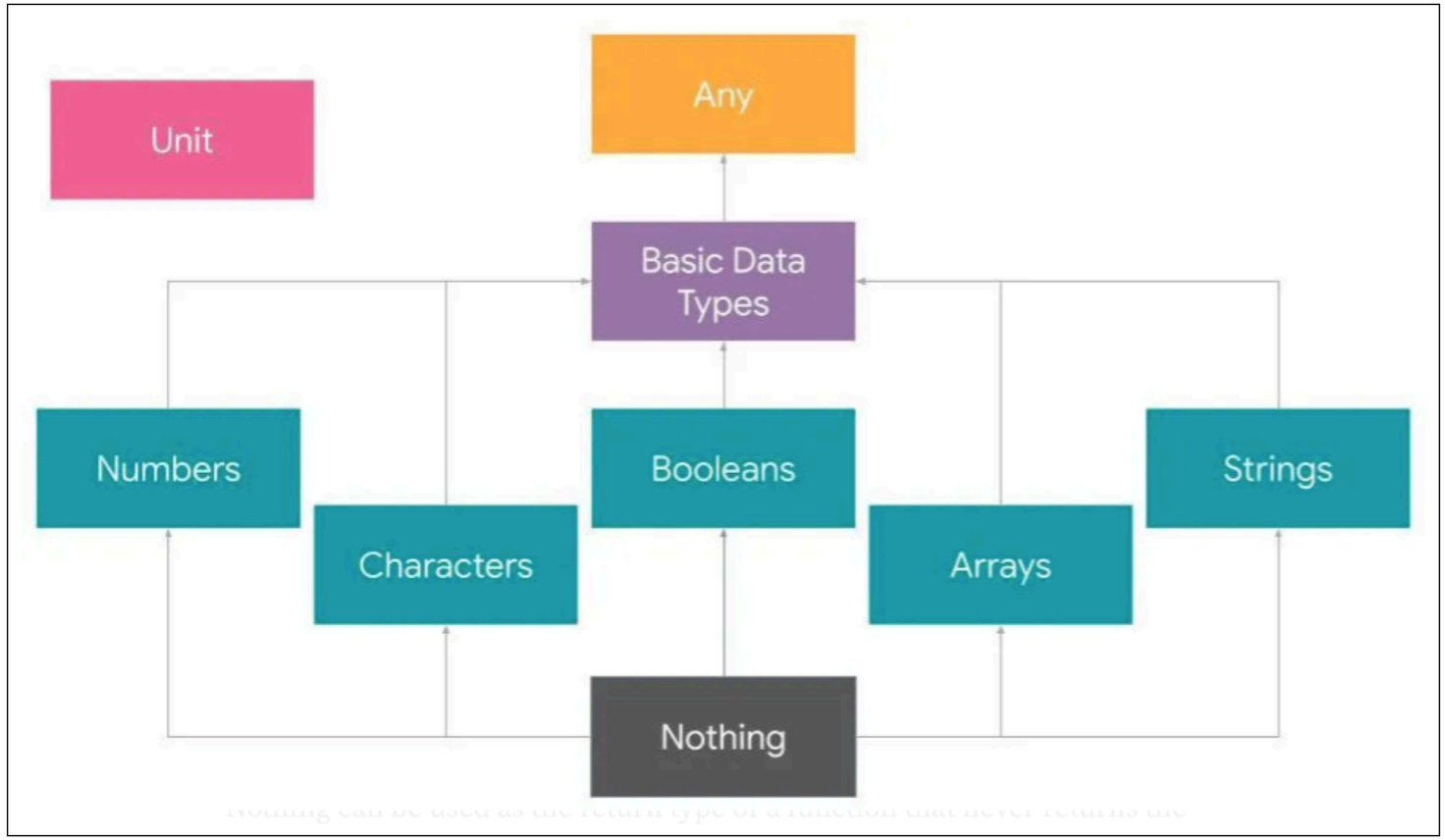
Overriding functions

INHERITANCE



Every class is “Any” class

4 pillars of OOP



INHERITANCE

Overriding the toString function

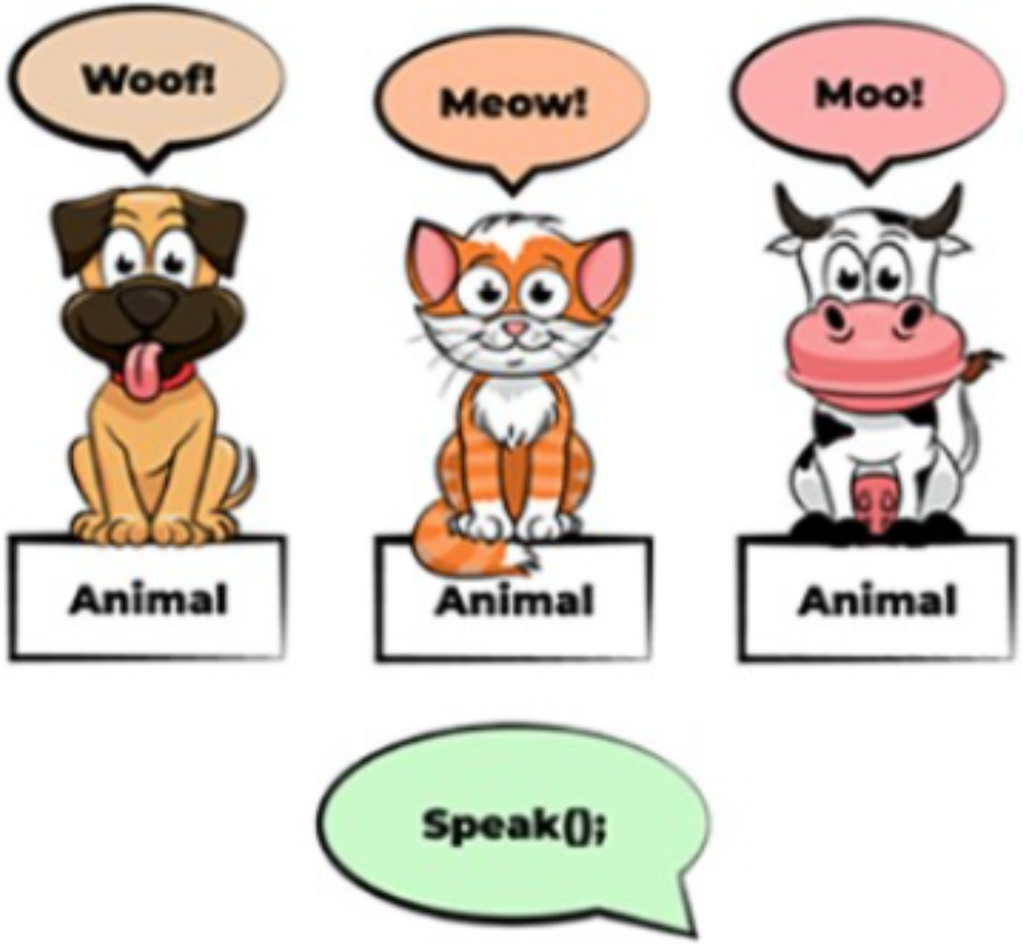
Common use case

- How does one represents an object as a string?
- That differs from object to object
- To accommodate this we can override the toString function
- In this way - objects are represented as strings when they are printed
- Example: Song object

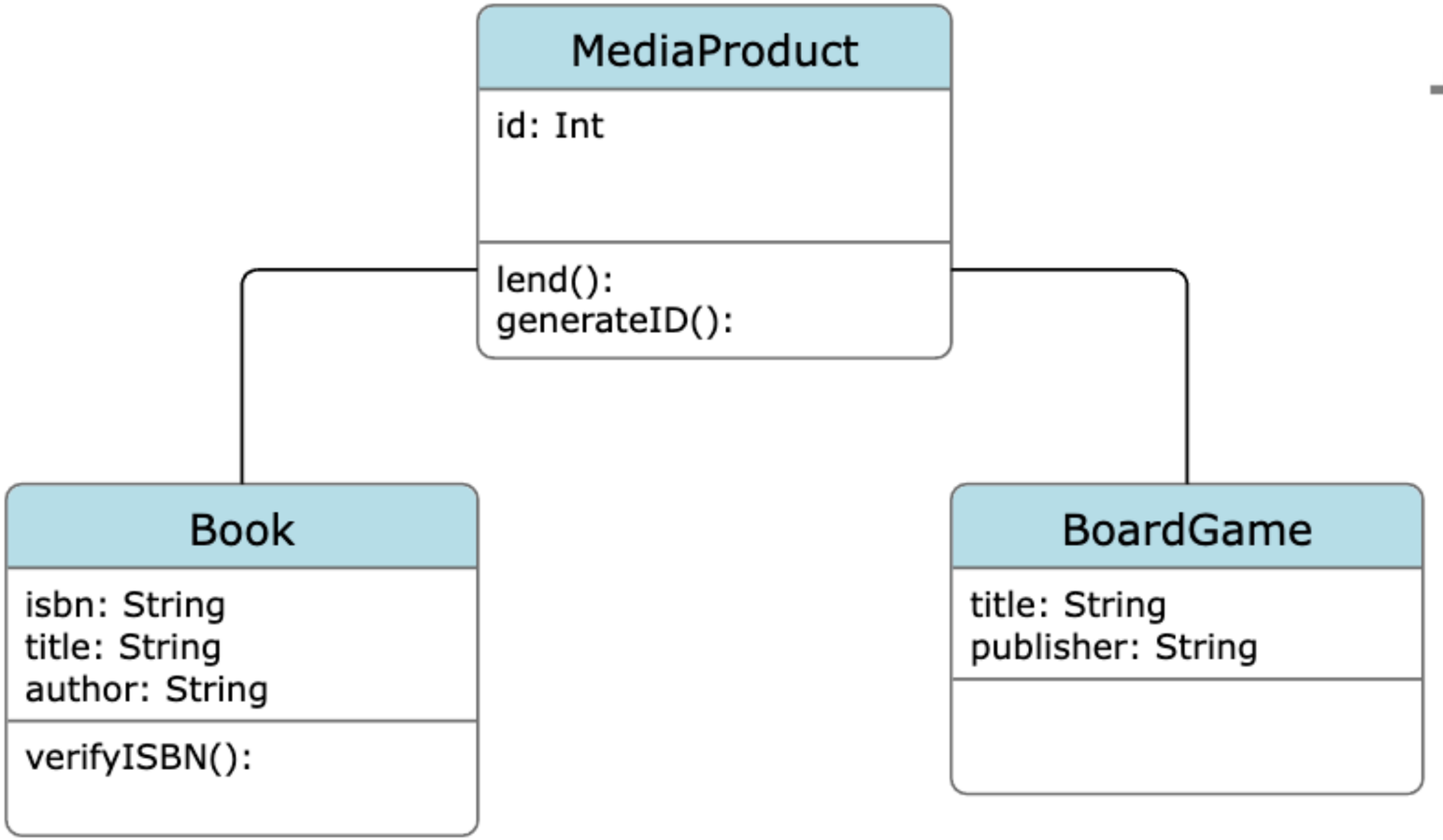
Polymorphism: Flere former

Example: IdentifyUser

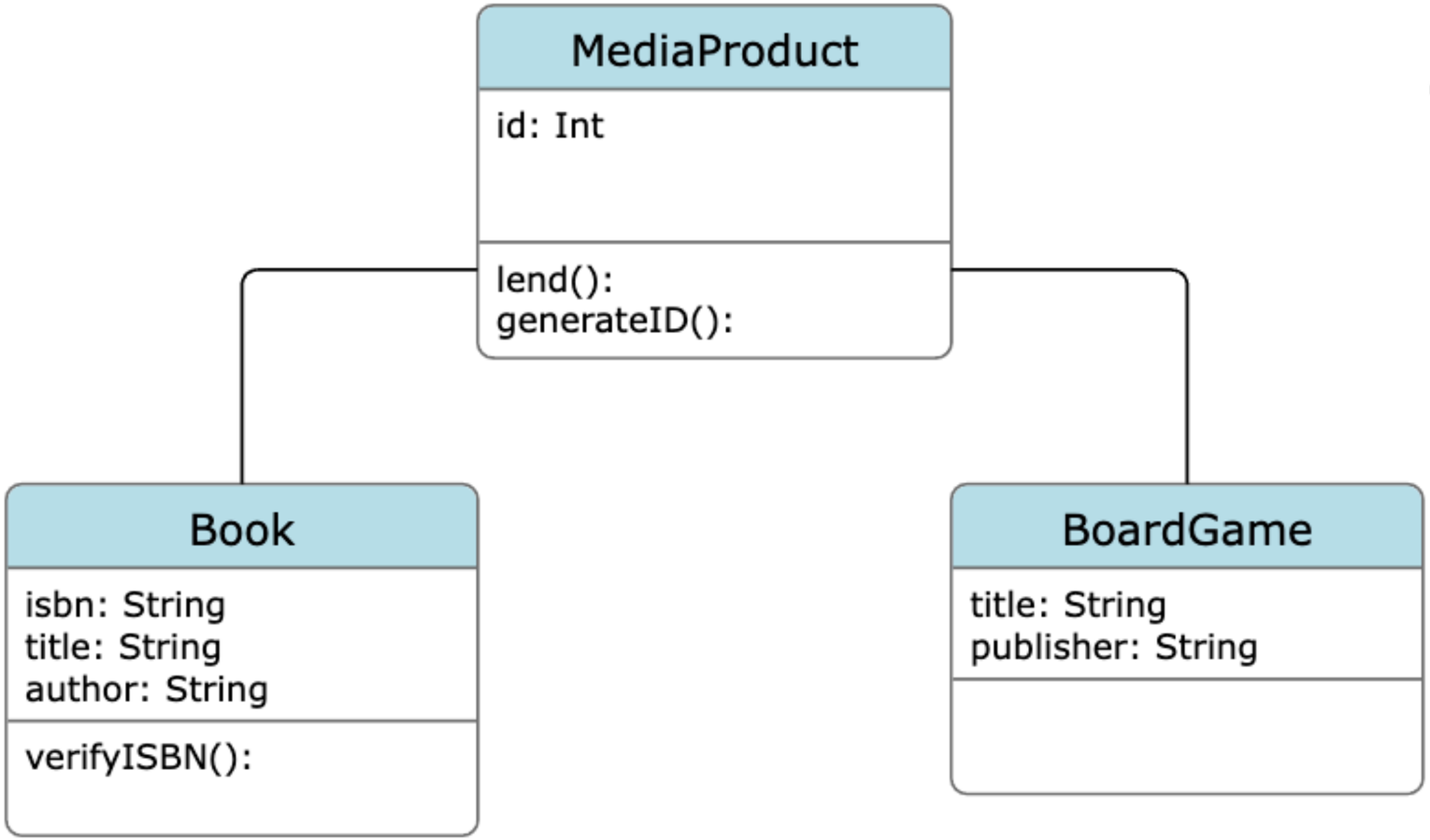
INHERITANCE



POLYMORPHISM



POLYMORPHISM



Because of polymorphism - in a list of mediaproducts - the generateID( ) & lend( ) functions can be called

Exercise A: Inheritance

Encapsulation
Fields, properties, getters & setters


93.20


x1000r/min
ODO
41 km
km/h

Encapsulation

Motivation

- Only required properties are exposed to client code
- Only required functions are exposed to client code
- Makes objects simpler to use for client code / developers
- Makes objects maintain an internal consistency

```
fun main(args: Array<String>) {
    val list1 = listOf("a", "p", "p", "l", "e")
    val listSize = list1.size
    print(listSize)
}
```

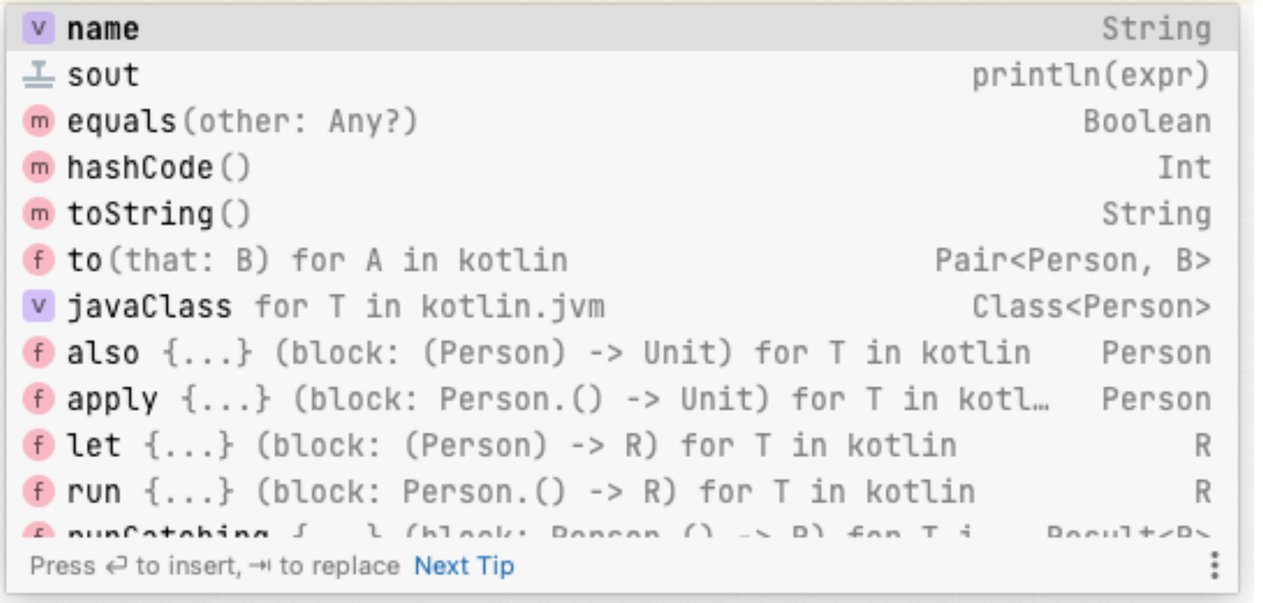
No outside access to the size property

Access modifiers in practice

Public / Private

```
class Person (cpr:String, name:String){
    private val cpr:String = cpr;
    public val name:String = name;
}
```

```
fun main() {
    val nicklas: Person = Person( cpr: "150690-0000",  name: "Nicklas");
    nicklas.
}
```



Name is public CPR is not

Public Private

Modifier	Description
public	visible everywhere
private	visible inside the same class only
internal	visible inside the same module
protected	visible inside the same class and its subclasses

https://www.geeksforgeeks.org/kotlin-visibility-modifiers/

Properties

Access modification

- Objects properties are the values that define the object
  - E.g. name, email, length etc.
- If they are not defined, they are unavailable outside of the object
- Access can be controller by getter/setter methods
- Properties can be mutable or unmutable

Public by default

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

Backing fields

Getters / setters

- If we want more control over getting / setting values
- A higher degree of encapsulation
- We will need to use backing fields

```
class Salary (salary : Int){
    // Private property
    private var _salary: Double = salary

    // Getter and setter for the salary property
    var salary: Double
        get() = _salary
        set(value) {
            if (value >= 0) {
                _salary = value
            } else {
                println("Salary cannot be negative. Setting salary to 0.")
                _salary = 0.0
            }
        }
}
```

Backing field

Property

Example: Person

Exercises B: access modification