Coordinate(x=0, y=0, z=0)
Coordinate(x=110, y=120, z=130)

Inheritance

In this section, we will explain and demonstrate what inheritance is. Simply put inheritance is the process where one class acquires the properties (functions and fields) of another.

The class which inherits the properties of others is known as “subclass” (“derived class”, “child class” and so on) and the class whose properties are inherited is known as “superclass” (“base class”, ”parent class”).

All classes at the top of the hierarchy are inheriting “Any” superclass. From ”Any” class we inherit the following functions (functions):

- “equals”
- “hashCode”
- “toString”

“equals” function

Indicates whether some other object is "equal to" this one. Implementations must fulfill the following requirements:

- Reflexive: for any non-null value x, x.equals(x) should return true.
- Symmetric: for any non-null values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- Transitive: for any non-null values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- Consistent: for any non-null values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- Never equal to null: for any non-null value x, x.equals(null) should return false.

“hashCode” function

Returns a hash code value for the object. The general contract of hashCode is:
- Whenever it is invoked on the same object more than once, the hashCode function must consistently return the same integer, provided no information used in equals comparisons on the object is modified.

- If two objects are equal according to the equals() function, then calling the hashCode function on each of the two objects must produce the same integer result.

“toString” function

Returns a string representation of the object.

We will demonstrate inheritance with simple examples. From book code examples open “Inheritance.kt”:

```
// Class 'Human' Inherits  by default: 'Any':
open class Human

// Class 'Indian' inherits Human:
class Indian : Human()

// Class with non-empty constructor:
open class Vehicle(type: String)

// Class 'Truck' inherits 'Vehicle' and it's constructor:
class Truck(type: String) : Vehicle(type)

// Class 'Train' inherits class 'Vehicle'
// but it has empty constructor.
// Value is passed to the parent constructor:
class Train : Vehicle("Civil")

// Another way to inherit class"
class Bus : Vehicle {

    // Constructor goes here:
    constructor(type: String) : super(type)

    init {

        // Your special initialization code
        // ...
    }
}
```

Note:

Class can't be inherited until it is defined as "open". By default, all Kotlin classes are closed for inheritance. Also, it is possible to inherit only one superclass. On the other hand, multiple interfaces can be implemented. We will explain this in upcoming sections of this book.

Overriding

If a subclass provides the specific implementation of the function that has been declared by its parent class, this is called function (function) overriding. Function overriding is used to provide the specific implementation of a function that is already provided by its superclass.

Note:

The function must have the same name as in the parent class and it must have the same parameter as in the parent class

In Kotlin to override function in the class we must define it as "open" in our "super" (parent) class. Let's have a look at an example that illustrates this. From book code examples open "Overriding.kt":

```
open class Engine(protected val model: String) {

    open fun turnOn() = println("$model: Turning on")

    open fun turnOff() = println("$model: Turning off")
}

class CarEngine(model: String) : Engine(model) {

    override fun turnOn() {

        super.turnOn()
        println("$model: Car is starting")
    }

    override fun turnOff() {

        super.turnOff()
        println("$model: Car is stopping")
    }
```

```
}

class CustomEngine(model: String) : Engine(model) {

    // We override just 'turnOn' function for this class:
    override fun turnOn() {

        // We don't want super class business logic,
        // so we do not call 'super':
        println("$model: Car is starting")
    }
}
```

Let’s play with these classes:

```
val carEngine = CarEngine("Fiat")
carEngine.turnOn()
carEngine.turnOff()

val customEngine = CustomEngine("Rocket")
customEngine.turnOn()
customEngine.turnOff()
```

Executing an example program will produce the following output:

```
Fiat: Turning on
Fiat: Car is starting
Fiat: Turning off
Fiat: Car is stopping
Rocket: Car is starting
Rocket: Turning off
```

Note:

A member function that is marked with ”override” is itself “open”. If you want to prohibit re-overriding, use the “final” modifier:

```
open class TruckEngine(model: String) : Engine(model) {

    final override fun turnOn() {

        // Executing something before 'super' business logic:
```

```
        println("$model: Preparing")
        super.turnOn()
        println("$model: Engine is running")
    }

    final override fun turnOff() {

        println("$model: Preparing")
        super.turnOff()
        println("$model: Engine has stopped")
    }
}

class Scania : TruckEngine("Scania"){

    // We cannot override 'turnOn' and 'turnOff' functions
    // since they are marked with 'final override'
}
```

As you can see from these simple examples, overriding is one of the most important features of object-oriented development. Overriding increases the usability of our code and gives us greater flexibility in development.

Object-oriented features

In this section, we will continue our journey through the world of object-oriented development. We will demonstrate some very important Kotlin features such as:

- Data classes
- Basics of abstraction
- Companion object
- Interfaces.

Data classes

Kotlin has a rich set of idioms to offer to developers. One of such is ”data classes”. “Data classes” are classes whose main purpose is to hold the data. In such types of classes, some standard functionalities are provided.