1.- Definición de Herencia

La herencia es un mecanismo de la Programación Orientada a Objetos, para diseñar dos o más entidades que son diferentes, pero comparten muchas características en común.

El proceso consiste en definir una clase conocida como: clase base, clase padre, superclase o ancestro; que contenga todas las características en común.

Luego defines las clases que heredan las características, denominadas como subclases, clases hijas, clases derivadas o clases descendientes.

2.- Clase Any

La clase Any es la raíz de la jerarquía de clases en Kotlin. Cada clase en el lenguaje derivará de ella si no especificas una superclase.

Creamos la clase ejemplo:

```
class Ejemplo // Hereda de Any
```

Si vas a la declaración de Any.kt desde IntelliJ IDEA, verás que tiene 3 métodos:

equals(): Indica si otro objeto es igual al actual

hashCode(): Retorna el código hash asociado al objeto (es un identificador de 32 bits que se almacena en un Hash en la instancia de la clase)

toString(): Retorna la representación en String del objeto

Por ejemplo, ejecuta los métodos anteriores con dos instancias de Ejemplo:

```
fun main() {
    val ejemplo1 = Ejemplo()
    val ejemplo2 = Ejemplo()
    println(ejemplo1.toString())
    println(ejemplo1.hashCode())
    println(ejemplo1.equals(ejemplo2))
}
```

3.- Sintaxis De Herencia

Añade el modificador **open** para habilitar su capacidad de herencia

Añade dos puntos para expresar en la cabecera, que hereda de otra clase

Añade los paréntesis al final del nombre de la superclase para especificar la llamada a su constructor.

```
open class Ancestro

class Descendiente: Ancestro()
```

3.1.- Constructor Primario En Herencia

Si la superclase tiene constructor primario, debes inicializarlo pasando los parámetros en la llamada de la sintaxis de herencia.

```
open class Ancestro(val propiedad:Boolean)

class Descendiente(propiedad: Boolean) : Ancestro(propiedad)
```

3.2.- Constructor Secundario En Herencia

Si la clase base no tiene constructor primario o deseas realizar la llamada del mismo, desde un constructor secundario de la subclase, entonces usa el constructor junto a la palabra reservada super para la llamada.

```
open class Weapon(val damage: Int, val speed: Double)

class Bow : Weapon {

    constructor(damage: Int, speed: Double) : super(damage, speed)

}
```

4.- Sobrescribir Métodos (Polimorfismo)

Aplicar polimorfismo con la sobrescritura de métodos en Kotlin, requiere habilitar el método con el modificador open. Luego usa el modificador override desde el método que se ha convertido en polifórmico de la subclase.

El polimorfismo es la capacidad de un objeto para ser tratado como un objeto de su tipo padre o de un tipo de interfaz que implementa. Esto permite a los objetos tener diferentes comportamientos basados en su tipo real en tiempo de ejecución.

```
open class Character(val name: String) {
    open fun die() = println("Morir")
}
class Mage(name: String) : Character(name) {
    override fun die() = println("Mago muriendo")
}
fun main() {
    val jaina = Mage("Jaina")
    jaina.die()
}
```

5.- Sobrescribir Propiedades

Similar a la sobrecritura de métodos, sobrescribir una propiedad requiere usar el modificador override sobre la propiedad en la subclase.

Si la propiedad está declarada con val en la clase padre, es posible reescribirla como var en la clase hija. Sin embargo, lo contrario no es posible.

```
open class BaseItem(val name: String) {
    open var quantity = 1
}

class PopularItem(name: String) : BaseItem(name) {
    override var quantity = 6
}

fun main() {
    val notebook = BaseItem("Cuaderno")
    val pencil = PopularItem("Lapicero")
    println("${notebook.name} x ${notebook.quantity}")
    println("${pencil.name} x ${pencil.quantity}")
}
```

6.- Clases abstractas

Son clases de las cuales no se pueden crear objetos (es una clase que no puede ser instanciada y debe ser extendida por otras clases). Una clase abstracta es incompleta o inútil sin algunas subclases concretas (no abstractas), desde las cuales puede instanciar objetos.

Una subclase concreta de una clase abstracta implementa todos los métodos y propiedades definidas en la clase abstracta; de lo contrario, esa subclase también es una clase abstracta.

Si una clase implementa un interfaz, pero no implementa todos los métodos, esa clase también deberá ser abstracta.

```
abstract class Employee (val firstName: String, val lastName: String) {
        abstract fun earnings(): Double
}
```

No todos los miembros tienen que ser abstractos. En otras palabras, podemos tener la implementación predeterminada del método en una clase abstracta.

```
abstract class Employee (val firstName: String, val lastName: String) {
   // ...
   fun fullName(): String {
      return lastName + " " + firstName;
   }
}
```

Ejemplo, la clase Programmer extiende la clase abstracta Employee. En Kotlin usamos un solo carácter de dos puntos (:).

```
class Programmer(firstName: String, lastName: String) : Employee(firstName, lastName)
{
   override fun earnings(): Double {
      // calculate earnings
   }
}
```

7.- Interfaces

Se utilizan para definir simplemente el comportamiento, pero sin entrar al detalle de cómo está implementado. De esa manera queda a elección del programador que implemente la interface, el detalle de implementación de cada uno de los métodos en la clase concreta. En otras palabras, una interfaz es un contrato que las clases que la implementen deben cumplir. La interfaz no especifica el "cómo" ya que no contiene el cuerpo de los métodos, solo el "qué".

Para definir una interfaz en Kotlin usa la palabra reservada **interface** seguido del nombre de la misma.

```
interface StudentRepository {
    fun getById(id: Long): Student
    fun getResultsById(id: Long): List<Result>
}
```

En el código anterior, hemos declarado una interfaz de StudentRepository. Esta interfaz contiene dos métodos abstractos: getById() y getResultsById(). Incluir la palabra clave abstract es redundante en un método de interfaz porque ya son implícitamente abstractos.

Si una clase *implementa* esta interface, tendrá que implementar obligatoriamente todos los métodos definidos en ésta, si no implementa todos los métodos, la clase será **abstracta**.

Una interfaz es inútil si no se implementa clase a partir de él, así que vamos a crear una clase que implementará esta interfaz.

Implementa una interfaz a través del símbolo: para separar el nombre de la clase y la interfaz:

```
class StudentLocalDataSource : StudentRepository {
    override fun getResults(id: Long): List<Result> {
      // do implementation
    }
    override fun getById(id: Long): Student {
                // do implementation
    }
}
```

Una vez escribas la implementación de forma StudentLocalDataSource : StudentRepository, **IntelliJ IDEA** te marcará en rojo la necesidad de sobrescribir los miembros abstractos. Si presionas la bombilla podrás acceder a un menú emergente que te facilita esta tarea a través de la opción **Implement members**

Reglas de interfaces en Kotlin:

Hay que destacar que cada interfaz puede tener varias implementaciones asociadas, y a diferencia de como ocurre con la herencia, **no hay límite en el número de interfaces que una clase puede implementar**.

El modificador override es obligatorio en Kotlin, a diferencia de Java.

No permite la definición de constructores y en las clases abstractas si se permiten.

Junto con los métodos, también podemos declarar propiedades en una interfaz Kotlin.

Un método de interfaz Kotlin puede tener una implementación predeterminada (cuando decimos implementar, queremos decir que puede tener código escrito dentro de la función, no solamente el nombre y los argumentos), se conocen como métodos regulares.

```
interface StudentRepository {
    fun m2() { // Método regular
        print("Método implementado")
    }
}
```

Como se indicó, una interfaz Kotlin puede tener propiedades, pero no puede mantener el estado, es decir no podrán inicializarse con algún valor. (Sin embargo, recuerde que las clases abstractas si pueden mantener el estado). Por lo tanto, la siguiente definición de interfaz con una declaración de propiedad funcionará.

```
interface StudentRepository {
    val propFoo: Boolean // propiedad abstracta
    // ...
}
```

Pero si intentamos agregar algún estado a la interfaz asignando un valor a la propiedad, no funcionará.

```
interface StudentRepository {
    val propFoo: Boolean = true // Error: Property initializers are not allowed in interfaces
    // ..
}
```

Sin embargo, una propiedad de interfaz en Kotlin puede tener métodos get y set (aunque solo este último si la propiedad es una variable). Se conocen como propiedades regulares.

```
interface StudentRepository {
   var propFoo: Boolean
      get() = true
      set(value)  {
        if (value) {
          // do something
        }
      }
   // ...
}
```

Las propiedades y métodos regulares de una interfaz deben ser sobrescritos con el modificador **override** .

```
class StudentLocalDataSource : StudentRepository {
   // ...
   override var propFoo: Boolean
      get() = false
      set(value) {
         if (value) {

         }
      }
}
```

Veamos un caso de cómo resolver conflictos de sobreescritura. Tenemos una clase que implementa múltiples interfaces, las cuales tienen un método con el mismo nombre. Aquí tenemos dos interfaces que tienen un método con la misma nombre funD()

```
interface InterfaceA {
    fun funD() {}
}
interface InterfaceB {
    fun funD() {}
}
```

Vamos a crear una clase que implemente estas dos interfaces. Lo primero a tener en cuenta es que solo deberemos sobrescribir una vez la funcion, ya que ambas se llaman igual.

Si dentro de classA decidiera llamar al método funD() de la clase padre, habría un dilema para el compilador.

```
class classA : InterfaceA, InterfaceB {
    override fun funD() {
        super.funD() // Error: Many supertypes available, please specify the one you mean
in angle brackets, e.g. 'super<Foo>'
    }
}
```

El compilador está confundido acerca de llamar al método super.funD() porque las dos interfaces que implementa la clase tienen la misma firma de método.

Para resolver este problema, puedes proceder a la llamada de la implementación que desees, a través de la referencia super junto al tipo base con paréntesis angulares: **super<Tipo>**

Aquí vamos a llamar al método funD() de InterfaceA.

```
class classA : InterfaceA, InterfaceB {
    override fun funD() {
        super<InterfaceA>.funD()
    }
}
```

Como dijimos anteriormente, una **interfaz** puede tener varias implementaciones.

```
interface Animal {
   var nombre: String
   fun come(comida: String)
   fun peleaCon(contrincante: Animal)
}
```

A continuación, tenemos la clase Perro que implementa el interfaz Animal y además, tiene su propio método adicional llamado ladra().

```
class Perro(override var nombre: String) : Animal{
   override fun come(comida: String) {
      if (comida == "carne") {
         println("Hmmmm, gracias");
      }
   }

   override fun peleaCon(contrincante: Animal) {
       //solo pelea contra otros perros
      if (contrincante is Perro) {
         println("ven aquí que te vas a enterar")
         contrincante.ladra()
      } else {
         println("no me gusta pelear")
      }
  }

   fun ladra() {
      println("Guau guau")
   }
}
```

Vemos como la funcion peleaCon, acepta como parámetro un Animal que, en nuestro caso, podría ser un objeto Perro o Gato. Solo en el caso que el parámetro sea un Perro peleará ¿esto cómo es posible hacerlo? ¿Cómo podemos verificar tipos en Kotlin? Usando lo que en Kotlin se conoce como **smart cast,** los cuales son conocidos también como casting seguros.

En la mayoría de los casos, no es necesario utilizar operadores de conversión explícitos en Kotlin porque el compilador rastrea las comprobaciones **is** y las conversiones explícitas para valores inmutables e inserta conversiones (seguras) automáticamente cuando es necesario.

Si queremos realizar alguna operación para un tipo concreto de objeto, podemos utilizar el operador **is** con una condición para conocer de qué tipo concreto es un objeto, y así el compilador infiere el tipo de dato con el que estamos trabajando y nos da acceso a las propiedades y métodos de dicho tipo, en el ejemplo vemos cómo se puede llamar a la función ladra() que es única para objetos de la clase Perro.

Ahora tenemos una segunda implementación del interfaz Animal, que es la clase Gato, la cual además de implementar el interfaz Animal tiene su propio método adicional llamado maulla().

Vemos además como en la función peleaCon, también podemos utilizar el operador **is** en conjunto con el operador **!** para verificar si un objeto **NO** pertenece a una clase:

```
class Gato(override var nombre: String) : Animal{
    override fun come(comida: String) {
        if (comida == "pescado") {
            println("Hmmmm, gracias");
        }
    }

    override fun peleaCon(contrincante: Animal) {
         //solo pelea contra otros gatos
        if (contrincante !is Gato) {
            println("no soy un gato y no me gusta pelear")
        } else {
            println("ven aquí que te vas a enterar")
            contrincante.maulla()
        }
    }

    fun maulla() {
        println("Miauuuu")
    }
}
```

Con el smart cast estamos haciendo una conversión de tipos implícita segura.

Hay otro tipo de **casting,** llamados **explícitos, que no son seguros** y por tanto podrían lanzar alguna excepción si el casting no fuera posible, por eso son llamados inseguros. Los castings inseguros en Kotlin se realizan con el operador **as**.

Este es un ejemplo de un casting inseguro:

```
fun obtenerCadena(obj: Any):String{

    val cadena = obj as String
    return cadena
}
```

Si llamamos a la función con un entero, por ejemplo: obtenerCadena(1), se producirá un ClassCastException.

Nota: El operador **as?** al igual que el operador **as** nos permite convertir a cualquiera clase que deseemos, pero si la conversión falla en lugar de tirar una excepción nos devolverá null

8.- Para que utilizarlos las clases abstractas y las interfaces

8.1.- Clases abstractas:

Cuando quieres crear una clase base común que proporcione funcionalidad básica para las clases hijas, pero no queremos instanciar objetos de la clase base.
Cuando quieres obligar a las clases hijas a implementar ciertos métodos o propiedades.
Cuando quieres crear una clase que represente un concepto genérico y luego especializarlo en las clases hijas.

8.2.- Interfaces:

Cuando quieres definir un comportamiento que se debe implementar en múltiples clases.
Cuando quieres que una clase implemente más de una interfaz.
Cuando quieres crear un contrato que obligue a las clases a implementar ciertos métodos o propiedades.
La elección entre clases abstractas e interfaces depende de la situación específica y de los requisitos de diseño.

9.- Ejemplos

9.1.- Ejemplo

```
open class Animal {
  open fun makeSound() {
    println("El animal hace un sonido")
  }
}
class Dog: Animal() {
  override fun makeSound() {
    println("El perro ladra")
  }
}
final class Cat: Animal() {
  override fun makeSound() {
    println("El gato maulla")
  }
}
```

La clase "Animal" es marcada como "open" para que se pueda heredar. La clase "Dog" hereda de "Animal" y sobreescribe la función "makeSound". La clase "Cat" también hereda de "Animal" y sobreescribe la función "makeSound", pero se marca como "final" para evitar la herencia adicional.

9.2.- Ejemplo

```
open class Vehicle {
  var speed = 0
  fun increaseSpeed() {
    speed += 10
  }
}
class Car: Vehicle() {
  var color = "Red"
}

final class Bike: Vehicle() {
  var weight = 20
}
```

9.3.- Ejemplo

```
open class Grandfather {
  open var name: String = "Abuelo"
  var age: Int = 80
  open fun talk() {
    println("Hola, soy el abuelo y tengo ${age} años")
  }
}

class Father: Grandfather() {
  var job: String = "Doctor"
  override var name: String = "Padre"
  override fun talk() {
    println("Hola, soy el padre, trabajo como ${job} y tengo ${age} años")
  }
}

class Son: Father() {
  var hobby: String = "Jugar videojuegos"

  override fun talk() {
    println("Hola, soy el hijo, mi hobby es ${hobby} y tengo ${age} años")
  }
}
```

9.4.- Ejemplo polimórfico.

```
open class Animal {
  open fun makeSound() {
    println("Hace sonido de animal")
  }
}
class Dog : Animal() {
  override fun makeSound() {
    println("Guau")
  }
}
class Cat : Animal() {
  override fun makeSound() {
    println("Miau")
  }
}

fun main() {
  val animals = arrayOf(Dog(), Cat(), Animal())
  for (animal in animals) {
    animal.makeSound()
  }}
```

9.5.- Ejemplo(Clase abstracta)

```
abstract class Person {
   abstract val name: String
   abstract val age: Int

   fun sayHello() {
      println("Hello, my name is $name and I am $age years old.")
   }
}

class Student(override val name: String, override val age: Int, val studentId: String) : Person() {
   fun attendClass() {
      println("$name is attending class with student ID $studentId.")
   }
}

class Teacher(override val name: String, override val age: Int, val subject: String) : Person() {
   fun teach() {
      println("$name is teaching $subject.")
   }
}
```

9.6.- Ejemplo (interfaces)

```
interface Person {
    val name: String
    val age: Int

    fun sayHello() {
        println("Hello, my name is $name and I am $age years old.")
    }
}
class Student(override val name: String, override val age: Int, val studentId: String) : Person {
    fun attendClass() {
        println("$name is attending class with student ID $studentId.")
    }

}

class Teacher(override val name: String, override val age: Int, val subject: String) : Person {
    fun teach() {
        println("$name is teaching $subject.")
    }
}
```