به مانند کلاس های abstract ما مفهومی به نام interface داریم که مسلما کاتلین پشتیبانیش میکنه.

اول شروع کنیم و یک کلاس interface به نام CustomerRepository بسازیم

class Customer

interface CustomerRepository {  
 fun getById**(**id: Int**)** : Customer

}  
  
class SQLCustomerRepository:CustomerRepository{  
 override fun getById**(**id: Int**)**: Customer {  
 TODO**(**"not implemented"**)** }  
}

همونطور که میبینین یک کلاس SQLCustomerRepository هم ساختیم که میاد از این interface استفاده میکنه و چون از یک interface استفاده میکنه باید توابع های اونو هم implement کنیم.

حالا بیاین برگردیم به CustomerRepository و یک چند خط کد اضافه کنیم بهش

class Customer

interface CustomerRepository {  
 fun store**(**obj:Customer**)**{  
 //implement code to store  
 }  
 fun getById**(**id: Int**)** : Customer  
}

class SQLCustomerRepository:CustomerRepository{  
 override fun getById**(**id: Int**)**: Customer {  
 TODO**(**"not implemented"**)** }  
}

که درواقع اینجا اصلا مهم نیست که چه کدی توی بدنه store زده شده، مهم اینه که این تابع بدنه داره،اونم توی interface! و این با مفهوم interface توی زبان های برنامه نویسی دیگه متفاوته. به عنوان مثال توی C# یک اینترفیس نباید هیچگونه بدنه داشته باشه، اگه بخواین همچین کاری بکنین شما باید از کلاسی استفاده کنین به نام abstract که قبلا دربارش صحبت کردیم. ولی خب توی کاتلین interface ها میتونن پیاده سازی دلخواه خودشونو داشته باشن.خب این برامون سوال ایجاد میکنه که دقیقا interface چه تفاوتی با abstract داره ؟ یکی از تفاوت های کلیدی این دو این است که شما نمی تونید توی interface یک state رو نگه دارید! در حالتی که توی abstract میتونید.به این معنی که شما میتونید مثل abstract به بعضی از متدهاتون بدنه بدین ولی نمیتونید به خصیصه هاتون مقدار بدین!

من نمیتونم کلاسی داشته باشم که از دوتا کلاس پایه ای ارث بری کنه در حالی که میتونم کلاسی داشته باشم که از کلاسی ارث بری کنه که اون کلاس از چند interface استفاده میکنه و این یکی دیگه از اهمیت های کاتلینه.

همینطور که گفتم interface ها نمیتونن حالت داشته باشند ولی میتونن خصیصه داشته باشند، که این خصیصه ها abstract باشند.

به عنوان مثال اگه فرض کنین من بیام و یک متغیر به فرم زیر تعریف کنم و بهش مقدار بدم

class Customer

interface CustomerRepository {  
 val isEmpty: Boolean = true

fun store**(**obj:Customer**)**{  
 //implement code to store  
 }  
 fun getById**(**id: Int**)** : Customer  
}

با اروری مواجه میشم که میگه نمیشه به خصیصه های interface مقداردهی کرد.البته چیزی که ممکنه اینه که از getter و setter استفاده کنیم!

class Customer

interface CustomerRepository {  
 val isEmpty: Boolean  
 get**()** = true

fun store**(**obj:Customer**)**{  
 //implement code to store  
 }  
 fun getById**(**id: Int**)** : Customer  
}

توی اینترفیس ها یادگرفتیم که میتونیم توابع رو override کنیم، درواقع خصیصه هارو هم میتونیم override کنیم

class Customer

interface CustomerRepository {  
 val isEmpty: Boolean  
 get**()** = true  
 fun store**(**obj:Customer**)**{  
 //implement code to store  
 }  
 fun getById**(**id: Int**)** : Customer  
}  
  
class SQLCustomerRepository:CustomerRepository{  
 override fun getById**(**id: Int**)**: Customer {  
 TODO**(**"not implemented"**)** }  
  
 override val isEmpty: Boolean  
 get**()** = false  
}

خب ما خصیصه هایی داخل interface داشتیم و تونستیم اونارو override کنیم، درمورد عضوهای توابع interface که implement کردیمشون چی؟ ایا میشه اونا رو هم override کرد؟ درواقع اره،میتونیم.

class Customer

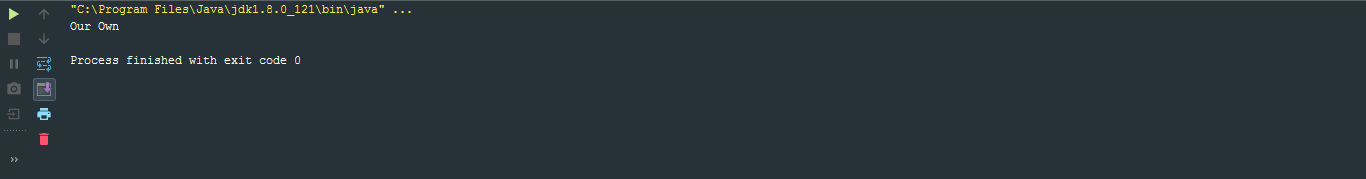
interface CustomerRepository {  
 val isEmpty: Boolean  
 get**()** = true  
 fun store**(**obj:Customer**)**{  
 //implement code to store  
 }  
 fun getById**(**id: Int**)** : Customer  
}  
  
class SQLCustomerRepository:CustomerRepository{  
 override fun getById**(**id: Int**)**: Customer {  
 return Customer**()** }  
  
 override fun store**(**obj: Customer**)** {  
 //my own implementation  
 }  
  
 override val isEmpty: Boolean  
 get**()** = false  
}

اینجا همینطور که میبینین با این که store قبلا پیاده سازی شده بود ولی میتونیم اونو override کنیم و پیاده سازی خودمون رو انجام بدیم.

خب از این موضوع بگذریم، بیاین کد زیر رو نگاه کنین:

interface Interface1 {  
 fun funA**()**{  
 println**(**"fun A from Interface 1"**)** }  
}  
  
interface Interface2 {  
 fun funA**()**{  
 println**(**"fun A from Interface 2"**)** }  
}  
  
class Class1And2 : Interface1,Interface2{  
 override fun funA**()** {  
 println**(**"Our Own"**)** }  
}  
  
fun main**(**args: Array<String>**)** {  
 val c = Class1And2**()** c.funA**()**}

دو اینترفیس داریم که دو 2 تابع هم نام رو داخل خودشون implement کردن و یک کلاس داریم که هردو اینترفیس رو implement میکنه و دوباره همون تابع با همون اسم رو override میکنه. حالا اگه داخل main اون تابع رو صدا بزنیم نتیجه چی میشه ؟



خب مسلما چون داخل کلاس Class1And2 دوباره متد funA رو پیاده سازی کردیم، همین یکی اجرا خواهد شد. اگه یادتون باشه ما میتونستیم از کلیدواژه super استفاده کنیم، ولی خب حالا اگه بیایم و اینجوری بنویسیم

interface Interface1 {  
 fun funA**()**{  
 println**(**"fun A from Interface 1"**)** }  
}  
  
interface Interface2 {  
 fun funA**()**{  
 println**(**"fun A from Interface 2"**)** }  
}  
  
class Class1And2 : Interface1,Interface2{  
 override fun funA**()** {  
 super.funA**()** }  
}

fun main**(**args: Array<String>**)** {  
 val c = Class1And2**()** c.funA**()**}

کامپایلر میمونه و ارور میده! چراکه نمیدونه منظور کدوم funA هه،آیا میخوای از متد Interface1 استفاده کنی یا از Interface2 ؟ اگر مثلا یکی شون پیاده سازی نشده بود خب مشکلی نبود و کامپایلر میفهمید منظور کدومه ولی حالا که هردو پیاده سازی شدن چی؟

خب این مشکل با استفاده از “<>” قابل حله.مثلا فرض کنین من میخوام Interface2 رو تغییر بدم.کافیه کدم رو اینجوری بنویسم

interface Interface1 {  
 fun funA**()**{  
 println**(**"fun A from Interface 1"**)** }  
}  
  
interface Interface2 {  
 fun funA**()**{  
 println**(**"fun A from Interface 2"**)** }  
}

class Class1And2 : Interface1,Interface2{  
 override fun funA**()** {  
 super<Interface2>.funA**()** }  
}

fun main**(**args: Array<String>**)** {  
 val c = Class1And2**()** c.funA**()**}

و خب اینجوری نشون میدم که میخوام کدوم یکی رو تغییر بدم.