\*برای تمرین این قسمت بهتر است از IntelliJ IDEA استفاده شود\*

خب بیاین در داخل کاتلین با جاوا صحبت کنیم. توجه داشتین که یک کلاس CustomerJava داشتیم که در واقع یک Java Bean بود.خب حالا بیاین توی یک کلاس کاتلین از این کلاس جاوا یک متغیر بسازیم و بعد به یک خصیصه از اون کلاس جاوا دسترسی داشته باشم

fun main(args: Array<String>) {  
 val customer = CustomerJava()  
 *println*(customer.*email*)  
}

اگه دقت کنین برای استفاده از email نیازی نیست که تابع getEmail رو صدا بزنیم! تنها کاری که لازمه اینه که نام اون متغیر رو صدا بزنیم و کاتلین خودش حدس میزنه که به چی نیاز داریم. البته به این معنی نیست که تابع getEmail رو نداریم و در واقع اگه بخوایم میتونیم از این تابع استفاده کنیم ولی خود کاتلین این اجازه رو میده که نیازی به صدا کردن تابع برای خوندن مقدارش نداشته باشیم. این ویژگی مسلما برای set هم هست و هر موقع بخوایم مقدار جدیدی رو بدیم کافیه همین روش رو به کار ببریم.

درمورد ارث بری هم روند کارمون سادست. بیایی یک کلاس به نام PersonJava در کد جاوامون ایجاد کنیم:

public class PersonJava {  
 private int id;  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
}

و حالا بیاین از این ارثبری کنیم:

class PersonKotlin : PersonJava()

به این راحتی از یک کلاس جاوا ارثبری کردیم. این قضیه حتی برای Interface ها هم برقراره

public interface CustomerRepository {  
 CustomerJava getById(int id);  
 List<CustomerJava> getAll();  
}

و حالا میتونم داخل کاتلین از این Interface استفاده کنیم

class KotlinCustomerRepo : CustomerRepository{  
 override fun getAll(): MutableList<CustomerJava> {  
 *TODO*("not implemented")  
 }  
  
 override fun getById(id: Int): CustomerJava {  
 *TODO*("not implemented")  
 }  
}

چیزی که در جاوا زیاد داریم در واقع "َAbstract متد های تنها" هستند که در واقع متد هایی هستند که در داخل کلاس اینترفیس ساخته میشن و یک نمونه ساده میتونه اینترفیس Runnable باشه.

میتونیم به مانند بقیه اینترفیس ها اونو در یک کلاس کاتلین استفاده کنیم، مثلا کد زیر:

class RunnableKotlin : Runnable{  
 override fun run() {  
 *TODO*("not implemented")  
 }  
}

ولی خب این یک راه حل خیلی ضایع برای این کاره، چرا که درواقع داریم یک کلاس رو که تنها یک تابع داره رو برای این موضوع به کار میبریم. میتونیم به مانند جاوا 1.8 به یک روش دیگه ازش استفاده کنیم:

val runnable = *Runnable* **{** *println*("Invoking") **}**

در واقع اجازه داده شده که تا بلاک کد رو پاس بدیم بدون اینکه در گیر استفاده از یک اینترفیس بشیم