حالا فرض کنید من یک اینتفریس دارم به نام EmployeeRepository و این کد هارو براش زدم

class Employee

interface EmployeeRepository{  
 fun store**(**onj: Employee**)**{  
  
 }  
 fun getById**(**id : Int**)** : Employee  
}

میبینین که ما درواقع یک سری کد رو تکرار کردیم. کلاس CustomerRepository (که جلسه قبل نوشتیم) و کلاس EmployeeRepository مون، دوتابع تکراری store و getById دارن و تنها تفاوت بین این دو تایپی که توابعش بر میگردونن است ، یکی Employee و یکی دیگه Customer.حالا اگه شما با برنامه نویسی اشنا باشین میدونین که چجوری بتونیم به مقصود دلخواهمون برسیم و اون هم استفاده از Generic هاست. بذارین شیوه کارمون رو توضیح بدم، ما یک سری کلاس های Generics تولید میکنیم که قراره یک سری پارامتر به نامType Parameter رو بگیرن و محصولی رو تولید کنن.خب پس بیاین ببینیم چجوری این کار انجام میشه.

interface Repository<T> {  
 fun getById**(**id:Int**)** : T  
  
 fun getAll**()** : List<T>  
}

یک اینترفیس ساختیم به نام Repository که یک Type Parameter میگیره به نام T و دو تابع هم داریم که یکی اون پارامتر رو با یک شماره برمیگردونه و یکی دیگه هم لیست کل پارامتر هارو برمیگردونه.

نحوه استفاده ازش هم راحته، کافیه یک کلاس بسازیم به نام CustomerGenericRepository به فرم زیر

class Customer

interface Repository<T> {  
 fun getById**(**id:Int**)** : T  
  
 fun getAll**()** : List<T>  
}

class CustomerGenericRepository<T> : Repository<T> {  
 override fun getById**(**id: Int**)**: T {  
 TODO**(**"not implemented"**)** }  
  
 override fun getAll**()**: List<T> {  
 TODO**(**"not implemented"**)** }  
}

fun main**(**args: Array<String>**)** {  
 val customerRepo = CustomerGenericRepository<Customer>**()**}

حالا شما تقریبا متوجه شدین که Customerهه اول نام CustomerGenericRepository درواقع اضافه است! چون لازم نیست که لغت Customer رو اولش بیاریم. تنها کاری که باید انجام بدیم اینه که یک کلاس بسازیم به نام GenericRepository و بعدش Employee و Customer رو بهش پاس بدیم!خب پس میریم و Customer رو از اولش بر میداریم.

class Customer

class Employee

interface Repository<T> {  
 fun getById**(**id:Int**)** : T  
  
 fun getAll**()** : List<T>  
}

class GenericRepository<T> : Repository<T> {  
 override fun getById**(**id: Int**)**: T {  
 TODO**(**"not implemented"**)** }  
  
 override fun getAll**()**: List<T> {  
 TODO**(**"not implemented"**)** }  
}

fun main**(**args: Array<String>**)** {  
 val customerRepo = GenericRepository<Customer>**()** val employeeRepo = GenericRepository<Employee>**()**}

همچنین ما میتونیم Generic هارو برای توابع هم استفاده کنیم

interface Repo{  
 fun <T> getById**(**id : Int**)** : T  
 fun <T> getAll**()** : List<T>  
}  
  
class MyRepo : Repo {  
 override fun <T> getById**(**id: Int**)**: T {  
 TODO**(**"not implemented"**)** }  
  
 override fun <T> getAll**()**: List<T> {  
 TODO**(**"not implemented"**)** }  
}

البته زمانی که کل کلاس از یک تایپ استفاده میکنن، خب بهتره که کل کلاس رو Type Parameter بدیم تا این که تک تک توابع رو، ولی زمانی که توابع داخل کلاس، هر کدوم Type Parameter های متفاوت میگیرن اون موقع این کار منطقیه.

خلاصه بخش 6:

1. به صورت پیش فرض تموم کلاس ها final هستند، در هنگام ارث بری نیاز به کلید واژه open داریم
2. کلاس های abstract به مانند interface هستند با این تفاوت که اجازه میدن حالت رو نگه داریم
3. کاتلین هم همچنین اجازه میده تا کلاس و متد های Generic داشته باشیم و البته در دوره Advance بیشتر به این مورد ها میپردازیم.