Обобщения

Обобщенные классы и функции

Generics или обобщения представляют технику, посредством которой методы и классы могут использовать объекты, типы которых на момент определения классов и функций неизвестны. Обобщения позволяют определять шаблоны, в которые можно подставлять различные типы.

Какие задачи решают обобщения? Допустим, у нас есть следующий класс

Обобщенные типы

Обобщенные типы (generic types) представляют типы, в которых типы объектов параметризированы. Что это значит? Рассмотрим следующий класс:

```
class Person<T>(val id: T, val name: String)
```

Класс Person использует параметр T. Параметры указываются после имени класса в угловых скобках. Данный параметр будет представлять некоторый тип данных, который на момент определения класса неизвестен.

В первичном конструкторе определяется свойство id, которое представляет идентификатор. Оно представляет тип, который передается через параметр T. На момент определения класса Person мы не знаем, что это будет за тип.

Само название параметра произвольное (если оно не совпадает с ключевыми словами). Но нередко используется T как сокращение от слова type.

При использовании типа Person необходимо его типизировать определенным типом, то есть указать, какой тип будет передаваться через параметр T:

```
fun main() {

    val tom: Person<Int> = Person(367, "Tom")
    val bob: Person<String> = Person("A65", "Bob")

    println("${tom.id} - ${tom.name}")
    println("${bob.id} - ${bob.name}")
}

class Person<T>(val id: T, val name: String)
```

Для типизации объекта после названия типа в угловых скобках указывается конкретный тип:

```
val tom: Person<Int>
```

В данном случае мы говорим, что параметр T фактически будет представлять тип Int. Поэтому в конструктор объекта Person для свойства id необходимо передать числовое значение Int:

```
Person(367, "Tom")
```

Второй объект типизируется типом String, поэтому в конструкторе для свойства id передается строка:

```
val bob: Person<String> = Person("A65", "Bob")
```

Если конструктор использует параметр T, то в принципе мы можем не указывать, каким типом типизируется объект - данный тип будет выводиться из типа параметра конструктора:

```
val tom = Person(367, "Tom")
val bob = Person("A65", "Bob")
```

При этом параметры типа могут широко применяться внутри класса, не только при определении свойств, но и в функциях:

```
fun main() {

    val tom = Person("qwrtf2", "Tom")
    tom.checkId("qwrtf2")   // The same
    tom.checkId("q34tt")    // Different
}

class Person<T>(val id: T, val name: String){

    fun checkId(_id: T){
        if(id == _id){
            println("The same")
        }
        else{
            println("Different")
        }
    }
}
```

Здесь класс Person определяет функцию checkId(), которая проверяет, равен ли id значению параметра _id. При этом параметр _id имеет тип T - то есть он будет представлять тот же тип, что и свойство id.

Стоит отметить, что generic-типы широко используются в Kotlin. Самый показательный пример, который представлен классом - Array. Параметр класса определяет, элементы какого типа массив будет хранить:

```
val people: Array<String> = arrayOf("Tom", "Bob", "Sam")
val numbers: Array<Int> = arrayOf(1, 2, 3, 4)
```

Применение нескольких параметров

Можно одновременно использовать несколько параметров:

```
fun main() {

    var word1: Word<String, String> = Word("one", "один")
    var word2: Word<String, Int> = Word("two", 2)

    println("${word1.source} - ${word1.target}")    // one - один
    println("${word2.source} - ${word2.target}")    // two - 2
}

class Word<K, V>(val source: K, var target: V)
```

В данном случае класс Word применяет два параметра - K и V. При создании объекта Word эти параметры могут представлять один и тот же тип, а могут представлять и разные типы.

Обобщенные функции

Функции, как и классы, могут быть обобщенными.

```
fun main() {

    display("Hello Kotlin")
    display(1234)
    display(true)
}
fun <T> display(obj: T){
    println(obj)
}
```

Функция display() параметризирована параметром T. Параметр также указывается в угловых скобках после слова fun и перед названием функции. Функция принимает один параметр типа T и выводит его значение на консоль. И при использовании функции мы можем передавать в нее данные любых типов.

Другой более практический пример - определим функцию, которая будет возвращать наибольший массив:

```
fun main() {

    val arr1 = getBiggest(arrayOf(1,2,3,4), arrayOf(3, 4, 5, 6, 7, 7))
```

```
    arr1.forEach { item -> print("$item ") }    // 3  4  5  6  7  7

    println()

    val arr2 = getBiggest(arrayOf("Tom", "Sam", "Bob"), arrayOf("Kate", "Alice"))
    arr2.forEach { item -> print("$item ") }    // Tom  Sam  Bob
}

fun <T> getBiggest(args1: Array<T>, args2: Array<T>): Array<T>{
    if(args1.size > args2.size) return args1
    else return  args2
}
```

Здесь функция getBiggest() в качестве параметров принимает два массива. При этом мы точно не значем, объекты какого типа эти массивы будут содержать. Однако оба массива типизированы параметром T, что гарантирует, что оба массива будут хранить объекты одного и того же типа. Внутри функции сравниваем размер массивов с помощью их свойства size и возвращаем наибольший массив.

Ограничения обобщений

Ограничения обобщений (generic constraints) ограничивают набор типов, которые могут передаваться вместо параметра в обобщениях.

Например, мы хотим определить универсальную функцию для сравнения двух объектов и возвращать из функции наибольший объект. На первый взгляд мы можем просто определить обобщенную функцию:

```
fun <T> getBiggest(a: T, b: T): T{
    if(a > b) return a   // ! Ошибка
    else return b
}
```

Но компилятор не скомпилирует эту функцию, потому что вместо параметра типа T могут передаваться самые различные типы, в том числе такие, которые не поддерживают операцию сравнения.

Однако все типы, которые по умолчанию поддерживают эту операцию сравнения, применяют интерфейс Comparable. То есть нам надо, чтобы два параметра представляли один и тот же тип, который реализует тип Comparable. И в этом случае можно определить одну обобщенную функцию, которая будет ограниченна типом Comparable:

```
fun main() {

    val result1 = getBiggest(1, 2)
    println(result1)
    val result2 = getBiggest("Tom", "Sam")
    println(result2)

}
```

```
fun <T: Comparable<T>> getBiggest(a: T, b: T): T{
    return if(a > b) a
    else b
}
```

Ограничение указывается после названия параметра через двоеточие: <T: Comparable> - то есть в данном случае тип T ограничен типом Comparable, иначе говоря должен представлять тип Comparable. Причем тип Comparable сам является обобщенным.

Стоит отметить, что по умолчанию ко всем параметрам типа также применяется ограничение в виде типа Any?. То есть определение параметра типа фактически аналогично определению <T: Any?>

Подобным образом мы можем использовать в качестве ограничений собственные типы. Например, нам надо определить функцию для условной отправки сообщения:

```
fun<T:Message> send(message: T){
    println(message.text)
}

interface Message{
    val text: String
}
class EmailMessage(override val text: String): Message
class SmsMessage(override val text: String): Message
```

Здесь определен интерфейс Message, который имеет одно свойство - text и представляет условное сообщение. И также есть два класса, которые реализуют этот интерфейс: EmailMessage и SmsMessage.

Функция send() использует ограничение <T:Message>, то есть она принимает объект некоторого типа, который должен реализовать интерфейс Message.

Далее мы можем вызвать эту функцию, передав ей соответствующий объект:

```
fun main() {
    val email1 = EmailMessage("Hello Kotlin")
    send(email1)
    val sms1 = SmsMessage("Привет, ты спишь?")
    send(sms1)
}
```

Здесь в обоих вызовах функция send() ожидает объект Message. Однако мы можем указать точный тип, используемый функцией:

Установка нескольких ограничений

В примере выше мы могли передавать в функцию getBiggest() любой объект, который реализует интерфейс Comparable. Но что, если мы хотим, чтобы функция могла сравнивать только числа? Все

числовые типы данных наследуются от базового класса Number. И мы можем задать еще одно ограничение - чтобы сравниваемый объект представлял тип Number:

```
fun <T> getBiggest(a: T, b: T): T where T: Comparable<T>,
                                        T: Number {
    return if(a > b) a
    else b
}
```

Если параметра типа надо установить несколько ограничений, то все они указываются после возвращаемого типа функции после слова where через запятую в форме:

```
параметр_типа: ограничений
```

И в этом случае мы сможем передать в функцию объекты, которые одновременно реализуют интерфейс Comparable и являются наследниками класса Number:

```
fun main() {

    val result1 = getBiggest(1, 2)
    println(result1)    // 2

    val result2 = getBiggest(1.6, -2.8)
    println(result2)    // 1.6

    // val result3 = getBiggest("Tom", "Sam")  // ! Ошибка - String не является
производным от класса Number
    // println(result3)
}
```

Подобным образом мы можем использовать собственные типы в качестве ограничений:

```
fun main() {
    val email1 = EmailMessage("Hello Kotlin")
    send(email1)
    val sms1 = SmsMessage("Привет, ты спишь?")
    send(sms1)
}
fun<t> send(message: T) where T: Message, T: Logger{
    message.log()
}

interface Message{ val text: String }
interface Logger{ fun log() }

class EmailMessage(override val text: String): Message, Logger{
```

```
    override fun log() = println("Email: $text")
}
class SmsMessage(override val text: String): Message, Logger{
    override fun log() = println("SMS: $text")
}
</t>
```

Здесь для функции send() установлено два ограничения: используемый параметр типа T должен представлять одновременно оба интерфейса - Message и Logger.

Ограничения в классах

Классы, как и функции, могут принимать ограничения обощений. Например, установка одного ограничения:

```
class Messenger<T:Message>(){
    fun send(mes: T){
        println(mes.text)
    }
}
```

Установка нескольких ограничений:

```
fun main() {
    val email1 = EmailMessage("Hello Kotlin")
    val outlook = Messenger<EmailMessage>()
    outlook.send(email1)

    val skype = Messenger<SmsMessage>()
    val sms1 = SmsMessage("Привет, ты спишь?")
    skype.send(sms1)
}
class Messenger<T>() where T: Message, T: Logger{
    fun send(mes: T){
        mes.log()
    }
}
interface Message{ val text: String }
interface Logger{ fun log() }

class EmailMessage(override val text: String): Message, Logger{
    override fun log() = println("Email: $text")
}
class SmsMessage(override val text: String): Message, Logger{
    override fun log() = println("SMS: $text")
}
```

Здесь стоит обратить внимание, что поскольку конструктор класса Messenger не принимает параметров типа T, то нам надо явным образом указать, какой именно тип будет использоваться:

```
val outlook = Messenger<EmailMessage>()
```

В качестве альтернативы можно было бы явным образом указать тип переменной:

```
val outlook: Messenger<EmailMessage> = Messenger()
```

Вариантность, ковариантность и контравариантность

Вариантность описывает, как обобщенные типы, типизированные классами из одной иерархии наследования, соотносятся друг с другом.

Инвариантность

Инвариантность предполагает, что, если у нас есть классы Base и Derived, где Derived - производный класс от Base, то класс C не является ни базовым классом для C, ни производным. Например, у нас есть следующие типы:

```
interface Messenger<T: Message>()

open class Message(val text: String)
class EmailMessage(text: String): Message(text)
```

В данном случае мы не можем присвоить объект Messenger переменной типа Messenger и наоборот, они никак между собой не соотносятся, несмотря на то, что EmailMessage наследуется от Message:

```
fun changeMessengerToEmail(obj: Messenger<EmailMessage>){
    val messenger: Messenger<Message> = obj    // ! Ошибка
}
fun changeMessengerToDefault(obj: Messenger<Message>){
    val messenger: Messenger<EmailMessage> = obj      // ! Ошибка
}
```

Мы можем присвоить переменным по умолчанию только объекты их типов:

```
fun changeMessengerToDefault(obj: Messenger<Message>){
    val messenger: Messenger<Message> = obj
}
fun changeMessengerToEmail(obj: Messenger<EmailMessage>){
    val messenger: Messenger<EmailMessage> = obj
}
```

Ковариантость

Ковариантость предполагает, что, если у нас есть классы Base и Derived, где Base - базовый класс для Derived, то класс SomeClass является базовым классом для SomeClass

Для определения обобщенного типа как ковариантного параметр обощения определяется с ключевым словом out:

```
interface Messenger<out T: Message>
open class Message(val text: String)
class EmailMessage(text: String): Message(text)
```

В данном случае интерфейс Messenger является ковариантным, так как его параметр определен со словом out: interface Messenger. И теперь переменной типа Messenger мы можем присвоить значение типа Messenger

```
fun changeMessengerToEmail(obj: Messenger<EmailMessage>){
    val messenger: Messenger<Message> = obj
}
```

Вообще не случайно используется именно слово out. Оно указывает, что обобщенный тип может возвращать из функции значение типа T:

```
fun main() {

    val messenger: Messenger<Message> = EmailMessenger()
    val message = messenger.writeMessage("Hello Kotlin")
    println(message.text)    // Email: Hello Kotlin
}
open class Message(val text: String)
class EmailMessage(text: String): Message(text)

interface Messenger<out T: Message>{
    fun writeMessage(text: String): T
}
class EmailMessenger(): Messenger<EmailMessage>{
    override fun writeMessage(text: String): EmailMessage {
        return EmailMessage("Email: $text")
    }
}
```

В данном случае обобщенный интерфейс Messenger определяет функцию writeMessage() для генерации объекта Message. Класс EmailMessenger применяет интерфейс Messenger и реализует эту функцию. То есть в данном случае тип EmailMessenger по сути представляет тип Messenger.

Поскольку в Messenger параметр T определен с аннотацией out, то мы можем присвоить переменной типа Messenger значение типа EmailMessenger (а по сути значение типа Messenger)

```
val messenger: Messenger<Message> = EmailMessenger()
```

В то же время тип T нельзя использовать в качестве типа входных параметров функции. Например, в следующем случае компилятор известит нас об ошибке:

```
interface Messenger<out T: Message>{
    fun writeMessage(text: String): T
    fun sendMessage(message: T)     // Ошибка - тип T может представлять только
возвращаемый тип
}
```

Контравариантность

Контравариантость предполагает в какой-то степени обратную ситуацию. Контравариантость предполагает, что, если у нас есть классы Base и Derived, где Base - базовый класс для Derived, то объекту SomeClass мы можем присвоить значение SomeClass (при ковариантности, наоборот, - объекту SomeClass можно присвоить значение SomeClass)

Для определения обобщенного типа как контравариантного параметр обобщения определяется с ключевым словом in:

```
interface Messenger<in T: Message>
open class Message(val text: String)
class EmailMessage(text: String): Message(text)
```

В данном случае интерфейс Messenger является контравариантным, так как его параметр определен со словом in: interface Messenger. И теперь переменной типа Messenger мы можем присвоить значение типа Messenger

```
fun changeMessengerToDefault(obj: Messenger<Message>){
    val messenger: Messenger<EmailMessage> = obj
}
```

Применение аннотации in означает, что обобщенный тип может получать значение типа T через параметр функции:

```
fun main() {

    val messenger: Messenger<EmailMessage> = InstantMessenger() //
```

```
InstantMessenger - это Messenger<Message>

    val message = EmailMessage("Hi Kotlin")
    messenger.sendMessage(message)
}
open class Message(val text: String)
class EmailMessage(text: String): Message(text)

interface Messenger<in T: Message>{
    //fun writeMessage(text: String): T
    fun sendMessage(message: T)
}

class InstantMessenger(): Messenger<Message>{
    override fun sendMessage(message: Message){
        println("Send message: ${message.text}")
    }
}
```

В данном случае обобщенный интерфейс Messenger определяет функцию sendMessage(), которая принимает объект Message в качестве параметра. Класс InstantMessenger применяет интерфейс Messenger и реализует эту функцию. То есть в данном случае тип InstantMessenger по сути представляет тип Messenger.

Поскольку в интерфейсе Messenger параметр T определен с аннотацией in, то мы можем присвоить переменной типа Messenger значение типа InstantMessenger (то есть значение типа Messenger)

```
val messenger: Messenger<EmailMessage> = InstantMessenger()
```

В то же время тип T нельзя использовать в качестве типа результата функции. Например, в следующем случае компилятор известит нас об ошибке:

```
interface Messenger<in T: Message>{
    fun writeMessage(text: String): T   // Ошибка - тип T может представлять
только параметр функции
    fun sendMessage(message: T)
}
```