Класс данных – data class

Нередко в программах требуются объекты, предназначенные во многом для хранения данных. Например, для книг надо описывать их автора, название, год издания. В более старых языках программирования, таких как Паскаль и Си, для подобных целей существует такой тип данных как "запись". В более современных языках обычно для этих целей используют обычные классы, в которых методов может и не быть. Вот как мог бы выглядеть подобный класс и его объекты на языке Kotlin:

```
class Book(val title: String,
           val author: String,
           val year: Int)
```

```
fun main() {
    val a = Book("ABC", "Tom", 2015)
    val b = Book("River", "Sam", 2007)
}
```

С объектами таких классов данных часто выполняются стандартные действия. Например, вывод значений свойств объекта, создание другого объекта с почти такими же значениями свойств, как у существующего. Поэтому Kotlin идет дальше и вводит в язык особый вариант класса – класс данных. Его объявление начинается со слова `data`.

```
data class Book(val title: String,
           val author: String,
           val year: Int)
```

Разница между этим вариантом и предыдущем в том, что к таким классам компилятор добавляет методы `toString()`, `equals()` и `hashCode()`, которые переопределяют эти методы, наследуемые по умолчанию всеми классами от `Any`. В результате в дата-классах эти функции-члены работают по-другому, они

адаптированы под задачи, которые выполняют дата-классы. Также компилятор добавляет несколько других функций-членов, например, `copy()`.

Метод `toString()` data-класса создает строку, содержащую перечень свойств и их значений.

Не забываем, что функция `println()` сама вызывает `toString()`. Конечно, мы можем переопределить метод в дата-классе, если нам не нравится его реализация по-умолчанию.

```
data class Book(val title: String,
                val author: String,
                val year: Int) {

    override fun toString(): String {
        return "$title, $author, $year"
    }
}
```

При этом мы переопределяем не тот `toString()`, который будет добавлен компилятором в связи с модификатором **`data`**. Мы переопределяем `toString()` класса **`Any`**. Поэтому если реализация метода `toString()` будет выглядеть как ниже, то это возврат к тому, что делает **`Any`**, несмотря на то, что класс **`data`**.

```
override fun toString(): String {
    return super.toString()
}
```

Функция-член `equals()` (перегружает оператор **`==`**), которую добавляет компилятор к data-классам, сравнивает поля и на этом основании выносит суждение о том, равны ли объекты.

```
fun main() {
    val a = Book("ABC", "Tom", 2015)
    val c = Book("ABC", "Tom", 2015)
    val b = Book("River", "Sam", 2007)
    println(a == c) // true
```

```
    println(a == b) // false
}
```

Если бы класс *Book* был объявлен без модификатора `data`, то результат обоих сравнений был бы `false`, потому что переменные *a* и *c* указывают на разные объекты. То есть сравнивались бы ссылки на объекты, а не значения полей объекта.

Функция `copy()` дата-класса, позволяет не просто создавать копию объекта, также на ходу изменять данные при необходимости:

```
fun main() {
    val a = Book("ABC", "Tom", 2015)
    val b = a.copy()
    val c = a.copy(title="River", year=2007)
    println(a)
    println(b)
    println(c)
}
```

Результат выполнения:

```
Book(title=ABC, author=Tom, year=2015)
Book(title=ABC, author=Tom, year=2015)
Book(title=River, author=Tom, year=2007)
```

Мультидекларация – это "распаковка" объекта таким образом, что значения его свойств присваиваются сразу нескольким переменным. В случае data-класса это выглядит так:

```
fun main() {
    val book1 = Book("ABC", "Tom", 2015)
    val (a, b, c) = book1 // мультидекларация!
    println(a) // ABC
    println(b) // Tom
    println(c) // 2015
}
```

Чтобы подобное было возможно, компилятор добавляет в дата-класс функции, перегружающие операцию мультидекларации. Обычные классы по-умолчанию не поддерживают такую распаковку, но программист может добавить эту возможность в любой класс. Так бы выглядел обычный класс, но с поддержкой мультидекларации:

```
class Book(val title: String,
           val author: String,
           val year: Int) {
    operator fun component1() = title
    operator fun component2() = author
    operator fun component3() = year
}
```

Операция мультидекларации также часто используется в цикле `for`. Если имеется список книг, можно легко пройтись по их свойствам:

```
fun main() {
    val books = ArrayList<Book>()
    books.add(Book("ABC", "Tom", 2015))
    books.add(Book("What", "Tom", 2016))
    books.add(Book("River", "Sam", 2007))

    for ((t, a, y) in books) {
        println("$y, $t")
    }
}
```

Результат:

```
2015, ABC
2016, What
2007, River
```

Все вышеперечисленные функции по-умолчанию обрабатывают только свойства перечисленные в первичном конструкторе. Однако класс данных может содержать

и другие.

```
data class Book(val title: String,
            val author: String,
            val year: Int) {
    val pages: Int = 0
}
```

В данном случае поле *pages* будет игнорироваться как при строковом представлении объекта, сравнении объектов, копировании, так и в мультидекларации.

В Котлин есть встроенные дата-классы – `Triple` и `Pair`, предназначенные для создания объектов с тремя или двумя свойствами. Тип свойств может быть любым.

```
fun main() {
    val t = Triple(1, "First", "Один")
    val t2 = t.copy(third = "Первый")
}
```

Объекты класса `Pair` нередко используются при обработке коллекций в цикле `for`.

PDF-версия курса с дополнительными уроками