**Глава 6: Интерфейсы, лямбда-выражения и внутренние классы**

1. Интерфейс:

Интерфейс в Java не является классом. Он представляет собой ряд требований, предъявляемых к классу, который должен соответствовать интерфейсу. Все методы интерфейса автоматически считаются открытыми, поэтому, объявляя метод в интерфейсе, указывать модификатор доступа public необязательно. Что интерфейсы не могут предоставить. В частности, у них отсутствуют поля экземпляра. Предоставлять поля экземпляра и оперирующие ими методы — обязанность классов, реализующих соответствующий интерфейс. Таким образом, интерфейс можно рассматривать как абстрактный класс, лишенный всяких полей экземпляра. Но у этих двух понятий имеется существенное отличие, которое будет подробнее проанализировано далее в главе.

Для того чтобы класс реализовал интерфейс, нужно выполнить следующие действия:

1. Объявить, что класс реализует интерфейс.

2. Определить в классе все методы, указанные в интерфейсе.

2) Резонный вопрос , в чем отличие интерфейса от абстрактного класса ?

Если вы помните содержание раздела главы 5, посвященного абстрактным классам, то у вас могут возникнуть следующие резонные вопросы: зачем разработчики языка Java создали механизм интерфейсов и почему бы не сделать интерфейс Comparable абстрактным классом, например, так, как показано ниже?

abstract class Comparable

{

public abstract int compareTo(Object other);

}

В этом случае рассматриваемый здесь класс Employee мог бы просто расширять абстрактный класс и реализовывать метод compareTo () следующим образом:

class Employee extends Comparable

{

public int compareTo(Object other) { ... }

}

К сожалению, это породило бы массу проблем, связанных с использованием абстрактного базового класса для выражения обобщенного свойства. Ведь каждый класс может расширять только один класс. Допустим, класс Employee уже является подклассом какого-нибудь другого класса, скажем, Person. Это означает, что он уже не может расширять еще один класс следующим образом:

class Employee extends Person, Comparable

Но в то же время каждый класс может реализовывать сколько угодно интерфейсов, как показано ниже.

class Employee extends Person implements Conparable

3)Методы по умолчанию в интерфейсах:

Для создания метода по умолчанию в интерфейсе, мы должны использовать ключевое слово default. Вообразим, что в представленном выше интерфейсе метод log(String str) является методом по умолчанию. Теперь, когда какой-либо класс будет реализовывать интерфейс Interface1, не является обязательным обеспечить реализацию методов по умолчанию (в нашем случае — это метод log(String str)).

Мы знаем, что Java не позволяет нам [наследоваться от нескольких классов](https://javadevblog.com/nasledovanie-v-java-prosto-i-na-primerah.html), потому что это приведет к [ромбовидной проблеме](https://javadevblog.com/mnozhestvennoe-nasledovanie-v-java-i-kompozitsiya-vs-nasledovaniya.html), где компилятор не может решить, какой метод суперкласса использовать. Теперь же с появлением методов по умолчанию, эта проблема возникнет и для интерфейсов! Все дело в том, что если класс реализует как Interface1, так и Interface2 и не реализовывает общий метод по умолчанию, то компилятор не может решить что выбрать.

Наследование нескольких интерфейсов является неотъемлемой частью Java, поэтому нам нужно следить за тем, чтобы эта проблема не возникала и в интерфейсах.

4) Коротко о главном. Методы по умолчанию в интерфейсах:

·Методы по умолчанию помогаю реализовывать интерфейсы без страха нарушить работу других классов.

·Методы по умолчанию в [Java 8](https://javadevblog.com/obzor-vozmozhnostej-java-8-funktsional-ny-e-interfejsy-lyambda-vy-razheniya-stream-i-time-api.html" \t "_blank) позволяют избежать создания служебных классов, так как все необходимые методы могут быть представлены в самих интерфейсах.

·Методы по умолчанию дают свободу классам выбрать метод, который нужно переопределить.

·Одной из основных причин внедрения методов по умолчанию является возможность коллекций использовать лямбда-выражения.

·Если какой-либо класс в иерархии имеет метод с той же сигнатурой, то методы по умолчанию становятся неактуальными. Метод по умолчанию не может переопределить метод класса java.lang.Object. Аргументация очень проста: это потому, что объект является базовым классом для всех Java-классов. Таким образом, даже если у нас есть методы класса Object, определенные в качестве методов по умолчанию в интерфейсах, это будет бесполезно, потому что всегда будет использоваться метод класса объекта. Вот почему, чтобы избежать путаницы, мы не можем писать стандартные методы, которые переопределяли бы методы класса Object.

5)Статические методы в интерфейсах:

Статические методы похожи на методы по умолчанию, за исключением того, что мы не можем переопределить их в классах, реализующих интерфейс. Этот функционал помогает нам избежать нежелательных результатов, которые могут появиться в дочерних классах. Статические методы видны только для методов интерфейса.

6) Коротко о главном. Статические методы в интерфейсах:

·Статические методы в интерфейсе являются частью интерфейса, мы не можем использовать его для объектов класса реализации.

·Статические методы в интерфейсе хороши для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.

·Статические методы в интерфейсе помогают обеспечивать безопасность, не позволяя классам, которые реализуют интерфейс, переопределить их.

·Мы не можем определить статические методы для методов класса Object, потому что получим ошибку компиляции: «*This static method cannot hide the instance method from Object*«. Это потому, что в Java так делать нельзя . То есть Object является базовым классом для всех классов и мы не можем использовать статический метод и еще такой метод с одинаковой сигнатурой.

·Мы можем использовать статические методы интерфейса, чтобы не создавать вспомогательные классы, то есть переместить все статические методы в соответствующий интерфейс. Такой метод легко использовать и быстро находить.

7)Разрешение конфликтов с методом по умолчанию:

7.1) А в Java подобные правила оказываются намного более простыми и заключаются в следующем:

1. В конфликте верх одерживает суперкласс. Если суперкласс предоставляет конкретный метод, то методы по умолчанию с одинаковыми именами и типами параметров просто игнорируются.

2. Интерфейсы вступают в конфликт. Если суперинтерфейс предоставляет метод, а другой интерфейс — метод (по умолчанию или иначе) с таким же самым именем и типами параметров, то для разрешения конфликта необходимо переопределить этот метод.

7.2) Этот класс наследует оба конфликтующих метода getName (), предоставляемых интерфейсами Person и Named. Вместо того чтобы выбрать один из этих методов, компилятор Java выдаст сообщение об ошибке, предоставив программисту самому разрешать возникшую неоднозначность. Для этого достаточно предоставить метод getName () в классе Student и выбрать в нем один из конфликтующих методов следующим образом:

class Student implements Person, Named {

public String getName () { return Person.super.getName(); }

}

8) Comparator интерфейс :

**Интерфейс Java Comparator**используется для упорядочивания объектов определенного пользователем класса. Этот интерфейс находится в пакете java.util и содержит 2 метода сравнения (Object obj1, Object obj2) и equals (Элемент Object). Он обеспечивает несколько последовательностей сортировки, т. Е. Вы можете сортировать элементы на основе любого элемента данных, например, rollno, name, age или чего-либо еще.

9) Клонирование объектов:

9.1)

Этом разделе рассматривается интерфейс Cloneable, который обозначает, что в классе предоставляется надежный метод clone (). Клонирование объектов производится нечасто, а подробности данного процесса носят совершенно технический характер, поэтому вы можете не обращаться к материалу этого раздела до тех пор, пока он вам не понадобится. Чтобы понять назначение клонирования объектов, напомним, что же происходит, когда создается копия переменной, содержащей ссылку на объект. В этом случае оригинал и копия переменной содержат ссылки на один и тот же объект. Это означает, что изменение одной переменной повлечет за собой изменение другой:

Employee original = new Employee("John Public", 50000);

Employee copy = original;

copy.raiseSalary(10); // Оригинал тоже изменился!

Если же требуется, чтобы переменная сору представляла новый объект, который в первый момент своего существования идентичен объекту original, но совершенно независим от него, в таком случае нужно воспользоваться методом clone () следующим образом:

Employee сору = original.clone();

copy.raiseSalary(10); // Теперь оригинал не изменился!

9.2)

Но не все так просто. Метод clone () является защищенным (protected) в классе Object, т.е. его нельзя вызвать непосредственно. И только класс Employee может клонировать объекты своего класса. Для этого ограничения имеется своя веская причина. Проанализируем, каким образом класс Object может реализовать метод clone (). Ему вообще ничего не известно об объекте, поэтому он может копировать лишь поля. Если все поля класса являются числовыми или имеют другой основной тип, их копирование выполняется нормально. Но если объект содержит ссылку на подобъект, то оригинал и клонированные объекты будут совместно использовать одни и те же данные.

9.3)

Для каждого класса нужно принять следующие решения.

1. Достаточно ли метода clone (), предоставляемого по умолчанию?

2.Можно ли доработать предоставляемый по умолчанию метод clone () таким образом, чтобы вызывать его для изменяемых объектов?

3. Следует ли вообще отказаться от применения метода clone () ?

По умолчанию принимается последнее решение. А для принятия первого и второго решения класс должен удовлетворять следующим требованиям.

1. Реализация интерфейса Cloneable.

2. Переопределение метода clone () с модификатором доступа public.

Метод clone () из класса Object может генерировать исключение типа CloneNotSupportedException. Это происходит в том случае, если метод clone () вызывается для объекта, не реализующего интерфейс Cloneable.

10)Лямбда- выражения:

Лямбда-выражение — это блок кода, который передается для последующего выполнения один или несколько раз.

Обратимся снова к примеру сортировки символьных строк из раздела 6.2.2. В этом примере определяется, является ли одна символьная строка короче другой. С этой целью вычисляется следующее выражение:

first.length() - second.length()

что обозначают ссылки first и second? Они обозначают символьные строки. Язык Java является строго типизированным, и поэтому приведенное выше выражение можно написать следующим образом:

(String first, String second) -> first.length() - second.length()

Собственно говоря, это и есть лямбда-выражение. Такое выражение представляет собой блок кода вместе с указанием любых переменных, которые должны быть переданы коду.

Давайте изучим структуру lambda-выражений:

* Lambda-выражения могут иметь от 0 и более входных параметров.
* Тип параметров можно указывать явно либо может быть получен из контекста. Например (int a) можно записать и так (a)
* Параметры заключаются в круглые скобки и разделяются запятыми.
* Например (a, b) или (int a, int b) или (String a, int b, float c)
* Если параметров нет, то нужно использовать пустые круглые скобки.
* Например () -> 42
* Когда параметр один, если тип не указывается явно, скобки можно опустить. Пример: a -> return a\*a
* Тело Lambda-выражения может содержать от 0 и более выражений.
* Если тело состоит из одного оператора, его можно не заключать в фигурные скобки, а возвращаемое значение можно указывать без ключевого слова return.
* В противном случае фигурные скобки обязательны (блок кода), а в конце надо указывать возвращаемое значение с использованием ключевого слова return (в противном случае типом возвращаемого значения будет void).

11)Функциональные интерфейсы:

Функциональный интерфейс в Java – это интерфейс, который содержит только 1 абстрактный метод. Основное назначение – использование в лямбда выражениях и method reference.

Наличие 1 абстрактного метода - это единственное условие, таким образом функциональный интерфейс может содержать так же default и static методы. К функциональному интерфейсу можно добавить аннотацию @FunctionalInterface. Это не обязательно, но при наличии данной аннотации код не скомпилируется, если будет больше или меньше, чем 1 абстрактный метод.

12)Внутренние классы:

Внутренним называется один класс, определенный в другом классе. А зачем он вообще нужен? На то имеются следующие причины:

• Объект внутреннего класса имеет доступ к данным объекта, в котором он определен, включая закрытые данные.

• Внутренний класс можно скрыть от других классов того же пакета.

• Анонимный внутренний класс оказывается удобным в тех случаях, когда требуется определить обратный вызов в процессе выполнения программы, не прибегая к необходимости писать много кода.

Довольно сложная тема внутренних классов в этом разделе будет обсуждаться в следующем порядке:

1. Сначала будет рассмотрен простой пример внутреннего класса, способного обращаться к полям экземпляра внешнего класса.

2. Затем будут обсуждаться специальные синтаксические правила, применяемые при объявлении внутренних классов.

3. Далее речь пойдет о преобразовании внутренних классов в обычные. (Слабонервные читатели могут пропустить этот материал.)

4. После этого рассматриваются локальные внутренние классы, способные обращаться к локальным переменным в области действия объемлющего класса.

5. Затем будет введено понятие анонимного внутреннего класса и показано, как пользоваться такими классами при организации обратных вызовов.

6. И наконец, будет показано, как пользоваться статическими внутренними классами для формирования вложенных вспомогательных классов.

Если один класс объявить внутри другого, то такой класс называется внутренним. Объекты внутренних классов при этом вложены в объекты внешних классов и могут обращаться к переменным внешних классов. Объект типа Door не может существовать отдельно от объекта типа Car – ведь он использует его переменные. Компилятор незаметно добавляет в конструктор и в класс Door ссылку на объект внешнего класса Car, чтобы методы внутреннего класса Door могли обращаться к переменным и вызвать методы внешнего класса Car. Во внутреннем классе Door имеется ссылка на объект класса Car, поэтому:

1) Нельзя создать объект Door внутри статического метода в классе Car: негде взять ссылку на объект типа Car, который неявно передается в конструктор типа Door.

**2) Класс Door не может содержать статические переменные и методы.**

3) Обрати внимание, что **если внутренний класс объявлен как public, то его объекты можно создавать вне внешнего класса, но объект внешнего класса при этом обязан присутствовать**

**4)**Т.к. у нас два вложенных объекта, то в методах внутреннего объекта доступно две ссылки this. Чтобы обратиться к переменной внешнего класса, когда она скрыта, или к this внешнего класса, достаточно написать «имя класса» точка this (Car.Door.this , Car.this , Car.Door.InnerClass2.InnerClass3.this)