Урок 1

Объектно-ориентированное программирование Java

Углубленное изучение вопросов ООП в Java: принципы ООП, классы, объекты, интерфейсы, перечисления, внутренние/вложенные/анонимные/локальные классы.

Оглавление

Что такое класс

Класс определяет форму и сущность объекта, и является логической конструкцией на основе которой построен весь язык Java. Наиболее важная особенность класса состоит в том, что он определяет новый тип данных, которым можно воспользоваться для создания объектов этого типа, т.е. класс - это шаблон(чертеж) по которому создаются объекты, а объекты - экземпляры класса. Для определения формы и сущности класса указываются данные, которые он должен содержать, а также код, воздействующий на эти данные. Ниже представлена упрощенная форма объявления класса:

```
модификатор_доступа class имя_класса {
    тип_переменной поле1;
    тип_переменной поле2;
    ...
    тип_переменной полеN;

    тип_метода имя_метода1(список_параметров) {
        // тело метода
    }

    ...

    тип_метода имя_методаN(список_параметров) {
        // тело метода
    }
}
```

Переменные, определенные в классе, называются полями экземпляра, поскольку каждый объект класса содержит собственные копии этих переменных. Таким образом, данные одного объекта отделены и отличаются от данных другого объекта. Код содержится в теле методов. Поля экземпляра и методы, определенные в классе, называются членами класса. В большинстве классов действия над полями осуществляются через методы, определенные в этом классе.

Первый класс

Ниже приведен код класса Cat, который определяет три поля: name(кличка), color(цвет) и age(возраст), и не содержит пока никаких методов. Следует заметить, что имя класса должно совпадать с именем файла в котором он объявлен, т.е. класс Cat должен находится в файле Cat.java

```
public class Cat {
    String name;
    String color;
    int age;
}
```

Как пояснялось выше, класс определяет новый тип данных, в данном случае это Cat. Поскольку класс является лишь чертежом, приведенный выше код не приводит к появлению каких-либо объектов. Чтобы создать объект класса Cat, нужно воспользоваться оператором наподобие следующего:

```
Cat cat1 = new Cat(); // создать объект класса Cat
```

После выполнения этого оператора объект cat1 станет экземпляром класса Cat. Всякий раз, когда получается экземпляр класса, создается объект, который содержит собственную копию каждой переменной экземпляра, определенной в данном классе. Таким образом, каждый объект класса Cat будет содержать собственные копии полей name, color и age. Для доступа к этим полям служит

операция-точка, которая связывает имя объекта с именем поля. Например, чтобы присвоить полю color объекта cat1 значение *White*, нужно выполнить следующий оператор:

```
cat1.color = "White";
```

В общем, операция-точка служит для доступа к полям и методам объекта. Ниже приведен полноценный пример программы, в которой используется класс Cat.

```
public class CatDemo {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
        Cat cat2 = new Cat();
        cat1.name = "Barsik";
        cat1.color = "White";
        cat1.age = 4;
        cat2.name = "Murzik";
        cat2.color = "Black";
        cat2.age = 6;
        System.out.println("Cat1 name: " + cat1.name + " color: " + cat1.color +
" age: " + cat1.age);
        System.out.println("Cat2 name: " + cat2.name + " color: " + cat2.color +
" age: " + cat2.age);
    }
}
```

Результат работы программы:

```
Cat1 name: Barsik color: White age: 4
Cat2 name: Murzik color: Black age: 6
```

При наличии двух объектов класса Cat, каждый из них будет содержать собственные копии полей name, color и age, т.е. изменение полей одного объекта не повлияет на поля другого. Данные из объекта cat1 изолированы от данных, содержащихся в объекте cat2.

Создание объектов

При создании класса появляется новый тип данных, который можно использовать для создания объектов данного типа. Но создание объектов класса представляет собой двухэтапный процесс. Сначала следует объявить переменную типа класса. Эта переменная не определяет объект. Она является лишь переменной, которая может ссылаться на объект. Затем нужно получить конкретную, физическую копию объекта и присвоить ее этой переменной. Это можно сделать с помощью оператора new, который динамически резервирует память для объекта и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором new. Затем эта ссылка сохраняется в переменной.

```
public static void main(String[] args) {
    Cat cat1;
    cat1 = new Cat();
}
```

В первой строке кода переменная cat1 объявляется как ссылка на объект типа Cat и пока еще не ссылается на конкретный объект. В следующей строке выделяется память для объекта, а переменной cat1 присваивается ссылка на него. После выполнения второй строки кода переменную cat1 можно использовать так, как если бы она была объектом типа Cat.

Подробное рассмотрение оператора new

Оператор new динамически выделяет оперативную память для объекта. Общая форма этого оператора имеет следующий вид:

```
Переменная_класса = new Имя_класса();
```

где переменная класса обозначает переменную создаваемого класса, а имя_класса - конкретное имя класса, экземпляр которого получается. Имя класса, за которым следуют круглые скобки, обозначает конструктор данного класса. Конструктор определяет действия, выполняемые при создании объекта класса. В большинстве классов, явно объявляются свои конструкторы, если ни один из явных конструкторов не указан, то Java автоматически сформирует конструктор по умолчанию.

При присваивании переменные ссылок на объекты действуют иначе, чем можно было бы предположить.

```
public static void main(String[] args) {
    Cat cat1 = new Cat();
    Cat cat2 = cat1;
}
```

На первый взгляд, может показаться что переменной cat2 присваивается ссылка на копию объекта cat1, т.е. переменные cat1 и cat2 будут ссылаться на разные объекты в памяти, но это не так. На самом деле cat1 и cat2, будут ссылаться на один и тот же объект. Присваивание переменной cat1 значения переменной cat2 не привело к выделению области памяти или копированию объекта, лишь к тому, что переменная cat2 ссылается на тот же объект, что и переменная cat1. Таким образом, любые изменения, внесенные в объекте по ссылке cat2, окажут влияние на объект, на который ссылается переменная cat1, поскольку это один и тот же объект в памяти.

Конструкторы

Конструктор позволяет инициализировать объект непосредственно во время его создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только конструктор определен, он автоматически вызывается при создании объекта перед окончанием выполнения оператора new.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat() {
        System.out.println("Это конструктор класса Cat") ;
        name = "Barsik";
        color = "White";
        age = 2;
    }
}

public class MainClass {
    public static void main(String[] args) {
        Cat cat1 = new Cat();
    }
}
```

Теперь при создании объектов класса Cat, все коты будут иметь одинаковые имя, цвет и возраст. Глядя на строку с созданием объекта cat1 должно быть ясно, почему после имени класса требуется указывать круглые скобки. В действительности оператор new вызывает конструктор класса. Если конструктор класса не определен явно, то создается конструктор по умолчанию.

Параметризированные конструкторы

В предыдущем примере конструктор позволяет создавать только одинаковых котов, что не имеет особого смысла. Для того, чтобы можно было с помощью конструктора создавать отличающиеся объекты, достаточно добавить в конструктор набор параметров.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String _name, String _color, int _age) {
        name = _name;
        color = _color;
        age = _age;
    }
}
```

Как правило, удобно чтобы имя параметра совпадало с именем поля, которое он заполняет. В данном случае так и есть, только перед именами параметров стоит нижнее подчеркивание (это не является каким-то правилом), это сделано для того, чтобы внутри конструктора можно было легко отличить имя поля от имени передаваемого параметра. Ниже приведен пример создания двух объектов класса Cat:

```
public static void main(String[] args) {
    Cat cat1 = new Cat("Barsik", "Brown", 4);
    Cat cat2 = new Cat("Murzik", "White", 5);
}
```

Перегрузка конструкторов

Наряду с перегрузкой обычных методов возможна перегрузка и конструкторов.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String _name, String _color, int _age) {
        name = _name;
        color = _color;
        age = _age;
    }

    public Cat(String _name) {
        name = _name;
        color = "Unknown";
        age = 1;
    }

    public Cat() {
        name = "Unknown";
        color = "Unknown";
```

```
        age = 1;
    }
}
```

В этом случае допустимы будут следующие варианты создания объектов:

```
public static void main(String[] args) {
    Cat cat1 = new Cat();
    Cat cat2 = new Cat("Barsik");
    Cat cat3 = new Cat("Murzik", "White", 5);
}
```

Соответствующий перегружаемый конструктор вызывается в зависимости от аргументов, указываемых при выполнении оператора new.

Ключевое слово this

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Для этой цели в Java определено ключевое слово this. Им можно пользоваться в теле любого метода для ссылки на текущий объект, т.е. объект у которого был вызван этот метод.

```
public class Cat {
    private String name;
    private String color;
    private int age;

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }
}
```

Эта версия конструктора действует точно так же, как и предыдущая. Ключевое слово this применяется в данном случае для того чтобы отличить имя параметра, от имени поля объекта.

Инкапсуляция

Инкапсуляция связывает данные с манипулирующим ими кодом и позволяет управлять доступом к членам класса из отдельных частей программы, предоставляя доступ к данным только с помощью определенного ряда методов, что позволяет предотвратить злоупотребление этими данными. Так, если класс реализован правильно, он создает своего рода "черный ящик", которым можно пользоваться, но его внутренний механизм защищен от повреждений.
Способ доступа к члену класса определяется модификатором доступа, присутствующим в его объявлении. Некоторые аспекты управления доступом связаны главным образом с наследованием и пакетами, и будут рассмотрены позднее. В Java определяются следующие модификаторы доступа: public, private и protected, а также уровень доступа, предоставляемый по умолчанию. Любой public член класса доступен из любой части программы. А любой компонент, объявленный как private, недоступен для компонентов, находящихся за пределами его класса. Если в объявлении члена класса отсутствует явно указанный модификатор доступа, он доступен для подклассов и других классов из данного пакета. Этот уровень доступа используется по умолчанию. Если же требуется, чтобы элемент был доступен за пределами его текущего пакета, но только классам, непосредственно производным от данного класса, такой элемент должен быть объявлен как protected.

Модификатор доступа предшествует остальной спецификации типа члена.

```
public int a;
protected char b;
private void cMethod(float x1, float x2) {
    // ...
}
```

Как правило, доступ к данным объекта должен осуществляться только через методы, определенные в классе этого объекта. Поле экземпляра вполне может быть открытым, но на то должны иметься веские основания. Для доступа к данным с модификатором private обычно используются геттеры и сеттеры. Геттер позволяет узнать содержимое поля, как правило, его имя такое же как у поля для которого он создан с добавлением слова get в начале, тип геттера также должен совпадать с типом поля. Сеттер используется для изменения значения поля, объявляется как void, и именуется по аналогии с геттером (только вместо get в начале имени метода стоит слово set). Кроме того, сеттер позволяет добавлять ограничения на изменение полей, в примере ниже, с помощью сеттера не получится указать коту отрицательный возраст. Если для поля сделать только геттер, то вне класса это поле будет доступно только для чтения.

```
public class Cat {
    ...
    private String name;
    private int age;

    public void setAge(int age) {
        if (age > 0)
            this.age = age;
        else
            System.out.println("Введен некорректный возраст");
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

В Java представлено три модификатора доступа: private, public, protected:

	private	Модификатор отсутствует	protected	public
Один и тот же класс	+	+	+	+
Подкласс, производный от класса из того же самого пакета	-	+	+	+
Класс из того же самого пакета, не являющийся подклассом	-	+	+	+
Подкласс, производный от класса другого пакета	-	-	+	+
Класс из другого пакета, не являющийся подклассом, производный от класса данного пакета	-	-	-	+

Дополнительные вопросы

Сборка «мусора». Поскольку выделение оперативной памяти для объектов осуществляется динамически с помощью оператора new, в процессе выполнения программы необходимо периодически и удалять объекты из памяти. В Java освобождение оперативной памяти осуществляется автоматически и называется сборкой «мусора». В отсутствие любых ссылок на объект считается, что этот объект больше не нужен и занимаемую им память можно освободить. Во время выполнения программы сборка «мусора» выполняется только изредка и не будет выполняться лишь потому, что один или несколько объектов больше не используются.
Ключевое слово static. Иногда возникает необходимость создать поле класса, общее для всех объектов этого класса, или метод, который можно было бы использования без создания объектов класса, в котором прописан этот метод. Обращение к такому полю или методу должно осуществляться через имя класса. Для этого, в начале объявления поля или метода ставится ключевое слово static. Когда член класса объявлен как static(статический), он доступен до создания любых объектов его класса и без ссылки на конкретный объект. Наиболее распространенным примером статического члена служит метод main(). При создании объектов класса, копии статических полей не создаются, и все объекты этого класса используют одно и то же статическое поле.
На методы, объявленные как static, накладывается следующие ограничения:

- Они могут непосредственно вызывать только другие статические методы;
- Им непосредственно доступны только статические переменные;
- Они никоим образом не могут делать ссылки типа this или super.

Основы наследования и полиморфизм

Одним из основополагающих принципов объектно-ориентированного программирования является наследование, используя которое, можно создать класс(суперкласс), который определяет какие-то общие характеристики. Затем этот общий класс может наследоваться другими, более специализированными классами(подклассами), каждый из которых будет добавлять свои особые характеристики. Подкласс наследует все члены, определенные в суперклассе, добавляя к ним собственные. Для реализации наследования используется ключевое слово extends в следующей форме:

```
class имя_подкласса extends имя_суперкласса
```

Пример создания класса Cat, который является подклассом класса Animal:

```
public class Animal {
    protected String name;

    public Animal() {
    }

    public Animal(String name) {
        this.name = name;
    }

    public void animalInfo() {
        System.out.println("Animal: " + name);
    }
}
```

```
public class Cat extends Animal {
    protected String color;

    public Cat(String name, String color) {
        this.name = name;
        this.color = color;
    }

    public void catInfo() {
        System.out.println("Cat: " + name + " " + color);
    }
}

public class MainClass {
    public static void main(String[] args) {
        Animal animal = new Animal("Animal");
        Cat cat = new Cat("Barsik", "White");
        animal.animalInfo();
        cat.animalInfo();
        cat.catInfo();
    }
}
```

Результат:

```
Animal: animal
Animal: Barsik
Cat: Barsik White
```

Подкласс Cat включает в себя все члены своего суперкласса Animal. Именно поэтому объект cat имеет доступ к методу animalInfo(), и в методе catInfo() возможна непосредственная ссылка на переменную color, как если бы они была частью класса Cat.
Несмотря на то что класс Animal является суперклассом для класса Cat, он в то же время остается полностью независимым и самостоятельным классом. То, что один класс является суперклассом для другого класса, совсем не исключает возможность его самостоятельного использования. Более того, один подкласс может быть суперклассом другого подкласса. Для каждого создаваемого подкласса можно указать только один суперкласс, в Java не поддерживается множественное наследование.

Ключевое слово super

У ключевого слова super имеются две общие формы. Первая форма служит для вызова конструктора суперкласса, вторая - для обращения к члену суперкласса, скрываемому членом подкласса. Из подкласса можно вызывать конструктор, определенный в его суперклассе, используя следующую форму ключевого слова super:

```
super(список_аргументов)
```

где список_аргументов определяет аргументы, требующиеся конструктору суперкласса. Вызов метода super() **всегда должен быть** первым оператором, выполняемым в конструкторе подкласса. Если в конструкторе подкласса явно не использовать super(), то автоматически первой строкой будет вызываться конструктор по умолчанию из суперкласса. Такая конструкция позволяет заполнять даже поля суперкласса с модификатором доступа private. Например:

```
public class Animal {
    private int a;
    protected int z;
    public Animal(int a) {
        this.a = a;
    }
}

public class Cat extends Animal {
    private int b;
    protected int z;
    public Cat(int a, int b) {
        super(a); // первым делом вызываем конструктор Animal
        this.b = b;
    }
    public void test() {
        z = 10; // Обращение к полю z класса Cat
        super.z = 20; // Обращение к полю z класса Animal
    }
}

public class SuperCat extends Cat {
    private int c;
    public SuperCat(int a,int b, int c) {
        super(a, b); // первым делом вызываем конструктор Cat
        this.c = c;
    }
}
```

Вторая форма применения ключевого слова super действует подобно ключевому слову this, за исключением того, что ссылка всегда делается на суперкласс. Вторая форма наиболее пригодна в тех случаях, когда имена членов подкласса скрывают члены суперкласса с такими же именами, в примере выше поле z класса Cat, скрывает поле z суперкласса, поэтому для доступа к полю суперкласса используется запись super.z, то же справедливо и для методов.

Порядок вызова конструкторов. При вызове конструктора SuperCat будут по цепочке вызваны конструкторы родительских классов, начиная с самого первого класса.

```
SuperCat sc = new SuperCat();
// Animal() => Cat() => SuperCat()
```

Конструкторы вызываются в порядке наследования, поскольку суперклассу ничего неизвестно о своих подклассах, и поэтому любая инициализация должна быть выполнена в нем совершенно независимо от любой инициализации, выполняемой подклассом. Следовательно, она должна выполняться в первую очередь.

Переопределение методов

Если у супер- и подкласса совпадают имена и сигнатуры типов методов, то говорят, что метод из подкласса переопределяет метод из суперкласса. Когда переопределенный метод вызывается из своего подкласса, он всегда ссылается на свой вариант, определенный в подклассе. А вариант метода, определенный в суперклассе, будет скрыт. Рассмотрим следующий пример:

```
public class Animal {
    void voice() {
        System.out.println("Животное издало звук");
    }
}
```

```
public class Cat extends Animal {
    @Override
    void voice() {
        System.out.println("Кот мяукнул");
    }
}

public class MainClass {
    public static void main(String[] args) {
        Animal a = new Animal();
        Cat c = new Cat();
        a.voice();
        c.voice();
    }
}
```

Результат:

```
Животное издало звук
Кот мяукнул
```

Когда метод voice() вызывает объект типа Cat, выбирается вариант этого метода, определенный в классе Cat. Если при переопределении метода необходим функционал из этого метода суперкласса, можно использовать конструкцию *super.method()*. Например:

```
public class Cat extends Animal {
    @Override
    public void voice() {
        super.voice(); // вызываем метод voice() суперкласса
        System.out.println("Кот мяукнул");
    }
}
```

Переопределение методов выполняется только в том случае, если имена и сигнатуры типов обоих методов одинаковы. В противном случае оба метода считаются перегружаемыми. Переопределенные методы позволяют поддерживать в Java полиморфизм во время выполнения, он позволяет определить в общем классе методы, которые станут общими для всех производных от него классов, а в подклассах - конкретные реализации некоторых или всех этих методов.

Абстрактные классы и методы

Иногда суперкласс требуется определить таким образом, чтобы объявить в нем структуру заданной абстракции, не предоставляя полную реализацию каждого метода. Например, определение метода voice() в классе Animal служит лишь в качестве шаблона, поскольку все животные издают разные звуки, а значит нет возможности прописать хоть какую-то реализацию этого метода в классе Animal. Для этой цели служит абстрактный метод (с модификатором типа abstract). Иногда они называются методами под ответственностью подкласса, поскольку в суперклассе для них никакой реализации не предусмотрено, и они обязательно должны быть переопределены в подклассе.

```
abstract void voice();
```

Как видите, в этой форме тело метода отсутствует. Класс, содержащий хоть один абстрактный метод, должен быть объявлен как абстрактный. Нельзя создавать объекты абстрактного класса, поскольку он определен не полностью. Кроме того, нельзя объявлять абстрактные конструкторы или

абстрактные статические методы. Любой подкласс, производный от абстрактного класса, обязан реализовать все абстрактные методы из своего суперкласса. При этом абстрактный класс вполне может содержать конкретные реализации методов. Пример:

```
public abstract class Animal {
    public abstract void voice();
    public void jump() {
        System.out.println("Животное подпрыгнуло");
    }
}

public class Cat extends Animal {
    @Override
    public void voice() {
        System.out.println("Кот мяукнул");
    }
}
```

Несмотря на то что абстрактные классы не позволяют получать экземпляры объектов, их все же можно применять для создания ссылок на объекты подклассов.

```
Animal a = new Cat();
```

Ключевое слово final в сочетании с наследованием

Существует несколько способов использования ключевого слова final:
Первый способ: создание именованной константы.

```
final int MONTHS_COUNT = 12; // final в объявлении поля или переменной
```

Второй способ: предотвращение переопределения методов.

```
public final void run() { // final в объявлении метода
}
```

Третий способ: запрет наследования от текущего класса.

```
public final class A { // final в объявлении класса
}
public class B extends A { // Ошибка, класс A не может иметь подклассы
}
```

Интерфейсы

С помощью ключевого слова interface можно полностью абстрагировать интерфейс класса от его реализации, то есть указать что именно должен выполнять класс, но не как это делать. Синтаксически интерфейсы аналогичны классам, но не содержат переменные экземпляра, а объявления их методов, как правило, не содержат тело метода. Как только интерфейс определен, его может реализовать любое количество классов. Кроме того, один класс может реализовать любое количество интерфейсов. Чтобы реализовать интерфейс, в классе должен быть переопределен весь набор методов интерфейса. Ключевое слово interface позволяет в полной мере использовать принцип полиморфизма "один интерфейс, несколько методов".

Объявление интерфейса

Определение интерфейса подобно определению класса. Упрощенная форма объявления:

```
Модификатор_доступа interface имя_интерфейса {
   возвращаемый_тип имя_метода1(список_параметров);
   возвращаемый_тип имя_метода2(список_параметров);
   тип имя_переменной1 = значение;
   тип имя_переменной2 = значение;
}
```

Методы интерфейса являются открытыми (public) и абстрактными (при этом не обязательно использовать ключевое слово abstract). Каждый класс, который включает в себя интерфейс, должен реализовать все его методы. В интерфейсах могут быть объявлены переменные, но они неявно объявляются как public static final, т.е. их нельзя изменить в классе, реализующем интерфейс. Кроме того, они должны быть инициализированы. Ниже приведен пример объявления интерфейса.

```
public interface Callback {
    void callback(int param);
}
```

Реализация интерфейсов

Как только интерфейс определен, он может быть реализован в одном или нескольких классах, для этого в объявлении класса необходимо добавить ключевое слово implements (как показано ниже), а затем переопределить методы интерфейса.

```
Модификатор_доступа class имя_класса [extend суперкласс] [implements
имя_интерфейса, ...] {
   // ...
}
```

Если в классе реализуется больше одного интерфейса, имена интерфейсов разделяются запятыми. Так, если в классе реализуются два интерфейса, в которых объявляется один и тот же метод, то этот же метод будет использоваться клиентами любого из двух интерфейсов. Рассмотрим небольшой пример класса, где реализуется приведенный ранее интерфейс Callback.

```
public class Client implements Callback {
    public void callback(int param) { // метод интерфейса
        System.out.println("param: " + param);
    }
    public void info() { // метод самого класса
        System.out.println("Client Info");
    }
}
```

Доступ к реализациям через ссылки на интерфейсы

По аналогии с тем, что ссылку на объект подкласса можно записать в ссылку на суперкласс (`Animal a = new Cat(...)`), можно сделать и ссылку на объект любого класса, который реализует указанный интерфейс (`Flyable f = new Bird(...);` где `class Bird implements Flyable`). При вызове метода по одной из таких ссылок нужный вариант будет выбираться в зависимости от конкретного экземпляра интерфейса, на который делается ссылка.

```
public interface Callback {
    void callback(int param);
}

public class ClientOne implements Callback {
    public void callback(int param) {
        System.out.println("ClientOne param: " + param);
    }
}

public class ClientTwo implements Callback {
    public void callback(int param) {
        System.out.println("ClientTwo param: " + param);
    }
}

public class TestClass {
    public static void main(String[] args) {
        Callback c1 = new ClientOne();
        Callback c2 = new ClientTwo();
        c1.callback(1);
        c2.callback(2);
    }
}
```

Результат:

```
ClientOne param: 1
ClientTwo param: 2
```

Вызываемый вариант метода callback() выбирается в зависимости от класса объекта, на который переменные c1, c2 ссылаются во время выполнения.

Перечисления

В простейшей форме *перечисление* - это список именованных констант, определяющих новый тип данных. В объектах перечислимого типа могу храниться лишь значения, содержащиеся в этом списке. Таким образом, перечисления позволяют определять новый тип данных, характеризующийся строго определенным рядом допустимых значений. В качестве примера можно привести названия дней недели или месяцев в году - все они являются перечислениями.
Перечисления создаются с использованием ключевого слова enum. Вот так, например, может выглядеть простое перечисление, представляющее различные виды фруктов.

```
public enum Fruit {
    ORANGE, APPLE, BANANA, CHERRY
}
```

Идентификаторы ORANGE, APPLE и т.д. - *константы перечисления.* Каждый из них неявно объявлен как открытый (public), статический (static) член перечисления Fruit. Типом этих констант является тип перечисления (в данном случае Fruit). В Java подобные константы называют *самотипизированными*.
Определив перечисление, можно создавать переменные этого типа. Однако, несмотря на то что перечисление - это тип класса, объекты этого класса создаются без привлечения оператора new. Переменные перечислимого типа создаются подобно переменным элементарных типов. Поскольку переменная f относится к типу Fruit, ей можно присваивать только те значения, которые определены для данного типа.

```
public static void main(String[] args) {
    Fruit f = Fruit.APPLE;
    System.out.println(f);
    if (f == Fruit.APPLE) {
        System.out.println("f действительно является яблоком");
    }
    switch (f) {
        case APPLE:
            System.out.println("f - яблоко");
            break;
        case ORANGE:
            System.out.println("f - апельсин");
            break;
        case CHERRY:
            System.out.println("f - вишня");
            break;
    }
}
```

Результат:

```
APPLE
f действительно является яблоком
f - яблоко
```

Для проверки равенства констант перечислимого типа используется операция сравнения ==. Перечисления можно использовать в качестве селектора в блоке switch. Заметьте, что в ветвях case оператора switch используются простые имена констант, а не уточненные. Так, в приведенном выше коде вместо Fruit.APPLE используется APPLE.

При отображении константы перечислимого типа, например, с помощью метода System.out.println(), выводится ее имя. Имена констант в перечислении Fruit указываются прописными буквами. Однако это требование не является обязательным. Но поскольку константы перечислимого типа обычно играют ту же роль, что и final переменные, которые традиционно обозначаются прописными буквами, для записи имен констант принято использовать тот же способ.

В Java перечисления реализованы как типы классов. И хотя для создания экземпляров класса enum не требуется использовать оператор new, во всех остальных отношениях они ничем не отличаются от классов. В частности, допускается определение конструкторов перечислений, добавление в них объектных переменных и методов и даже реализацию интерфейсов. При этом они не могут быть подклассом другого класса, или выступать в роли суперкласса.

Все перечисления автоматически включают два метода: values() – возвращает массив, содержащий список констант перечисления, и valueOf() – константу перечисления, значение которой соответствует строке str, переданной методу в качестве аргумента. Ниже приведен пример использования этих методов.

```
public static void main(String[] args) {
    System.out.println("Все элементы перечисления:");
    for(Fruit o : Fruit.values()) {
        System.out.println(o);
    }
    System.out.println("Поиск элемента по названию: " +
Fruit.valueOf("BANANA"));
}
```

Результат:

```
Все элементы перечисления:
ORANGE
APPLE
BANANA
CHERRY
Поиск элемента по названию: BANANA
```

Конструкторы, методы, переменные экземпляра и перечисления. В перечислении каждая константа является объектом класса данного перечисления. Таким образом, перечисление может иметь конструкторы, методы и переменные экземпляра. Если определить для объекта перечислимого типа конструктор, он будет вызываться всякий раз при создании константы перечисления. Для каждой константы перечислимого типа можно вызвать любой метод, определенный в перечислении. Кроме того, у каждой константы перечислимого типа имеется собственная копия любой переменной экземпляра, определенной в перечислении. Ниже приведен пример перечисления Fruit, к которому было добавлено название фрукта на русском языке и вес в условных единицах.

```
public enum Fruit {
    ORANGE("Апельсин", 3), APPLE("Яблоко", 3), BANANA("Банан", 2),
CHERRY("Вишня", 1);

    private String rus;
    private int weight;

    public String getRus() {
        return rus;
    }

    public int getWeight() {
        return weight;
    }

    Fruit(String rus, int weight) {
        this.rus = rus;
        this.weight = weight;
    }
}

public class Main {
    public static void main(String[] args) {
        for(Fruit o : Fruit.values()) {
            System.out.printf("Средний вес фрукта %s составляет: %d ед.\n",
o.getRus(), o.getWeight());
        }
    }
}
```

Результат:

```
Средний вес фрукта Апельсин составляет: 3 ед.
Средний вес фрукта Яблоко составляет: 3 ед.
Средний вес фрукта Банан составляет: 2 ед.
Средний вес фрукта Вишня составляет: 1 ед.
```

Итак, перечисление Fruit претерпело ряд изменений. Во-первых, появились две переменные экземпляра rus – название фрукта на русском и weight – средний вес фрукта в условных единицах. Во-вторых, добавлен конструктор, заполняющий поля. В-третьих, добавлены геттеры. И в-четвертых, список констант перечислимого типа стал завершаться точкой с запятой, которая требуется в том случае, если класс перечисления содержит наряду с константами и другие члены.

Внутренние и вложенные классы

Начиная с версии Java 1.1 допускается определять один класс в другом классе. Такие классы называются вложенными. Область действия вложенного класса ограничена областью действия внешнего касса. Так, если класс B определен в кассе A, то класс B не может существовать независимо от класса A. Вложенный класс имеет доступ к членам (в том числе закрытым) того класса,

в который он вложен. Но внешний класс не имеет доступа к членам вложенного класса. Вложенный класс, объявленный непосредственно в области действия своего внешнего класса, считается его членом. Классы, объявленные внутри кодовых блоков, называются локальными.
Существуют два типа вложенных классов: статический и не статический.
Статическим называется такой вложенный класс, объявленный с модификатором static, поэтому он должен обращаться к нестатическим членам своего внешнего класса посредством объекта. Это означает, что вложенный статический класс не может непосредственно ссылаться на нестатические члены своего внешнего класса.
Внутренний класс - это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них таким же образом, как это делают остальные нестатические члены внешнего класса. Ниже приведен пример работы с внутренним классом.

```
public class Outer {
    class Inner {
        private int innerVar;

        public Inner(int innerVar) {
            this.innerVar = innerVar;
        }

        void innerTest() {
            System.out.println("innerVar: " + innerVar);
            System.out.println("outerVar: " + outerVar);
        }
    }

    private int outerVar;

    public Outer(int outerVar) {
        this.outerVar = outerVar;
    }

    public void outerTest() {
        System.out.println("outerVar: " + outerVar);
        // System.out.println("innerVar: " + innerVar); тут ошибка
        Inner io = new Inner(20);
        System.out.println("io.innerVar = " + io.innerVar);
    }
}
```

Домашнее задание

- Разобраться с имеющимся кодом;
- Добавить класс Team, который будет содержать: название команды, массив из 4х участников (т.е. в конструкторе можно сразу всех участников указывать), метод для вывода информации о членах команды прошедших дистанцию, метод вывода информации обо всех членах команды.
- Добавить класс Course (полоса препятствий), в котором будут находиться: массив препятствий, метод который будет просить команду пройти всю полосу;
  То есть в итоге должно быть что-то вроде:

```
public static void main(String[] args) {
      Course c = new Course(...); // Создаем полосу препятствий
      Team team = new Team(...); // Создаем первую команду
      c.doIt(team); // Просим первую команду пройти полосу
      team.showResults(); // Показываем результаты первой команды
}
```

Дополнительные материалы

1 Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. - М.: Вильямс, 2014. - 864 с.

2 Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1168 с.

3 Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. - М.: Вильямс, 2015. - 1376 с.

4 Г. Шилдт Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. - М.: Вильямс, 2015. - 720 с.