Объектно-ориентированное программирование в JAVA

Темы рассматриваемые в уроке:

- Наследование
- Полиморфизм
- Абстрактные классы
- Интерфейсы
- Утилиты
- Настраиваемые типы

Объектно-ориентированное программи́рование (ООП) — парадигма программирования, основной особеностью которой, является то что предметная область в разрабатываемом приложении описывается объектами, которые взаимодействуют между собой. Из предыдущего урока Вы знаете что объектом называется экземпляр класса. Так же Вам известно что в основе ООП, лежат три понятия:

- Наследование (Inheritance)
- Инкапсуляция (Encapsulation)
- Полиморфизм (Polymorphism)

Наследование

Наследованием называется возможность создавать класс путем расширения функциональности другого класса, с сохранением свойств и методов класса-предка (прародителя, иногда его называют суперклассом) . Набор классов, связанных отношением наследования, называют иерархией.

Для того что бы расширить функциональность класса используется ключевое слово extends.

```
class A extends B
{
//определение класса a
}
```

Из курса по C++ Вы знаете что доступ к полям и методам базового класса зависит от

спецификаторов доступа. Такая же ситуация и в Java:

Спецификатор доступа в суперклассе	Доступ
private	Доступ к полям и методам со спецификатором private, класс наследник не имеет
protected	Поля и методы со спецификатором protected доступны только в методах класса наследника
public	Доступ к полям и методом со спецификатором public можно получить через объект производного класса

Со ссылкой this мы уже знакомились, а вот что бы явно указать обращение к методу или полю суперкласса необходимо использовать ключевое слово super.

Рассмотри пример наследования, реализуем класс студент и класс аспирант, аспирант отличается от студента наличием некой научной работы.

```
/*
* Класс описывает студента
*/
public class Student {
    //ФИО студента
    private String fullName;
    //Возраст студента
    private int age;

    public String getFullName() {
        return fullName;
    }

    public void setFullName(String fullName) {
        this.fullName = fullName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

```
    public Student(String fullName, int age) {
        this.fullName = fullName;
        this.age = age;
    }

    /*
    * Возвращает строковое представление
    * класса Student
    */
    public String toString(){
        return fullName+" "+age;
    }
}
```

```
/*
* Класс описывает аспиранта,
* Аспиарнтом - называеться студент
* выполняющий некоторую научную работу
*/
public class Aspirant extends Student {

    //Название научной работы
    private String workName;

    public Aspirant(String fullName, int age, String workName) {
        //Вызов конструктора супер класса
        super(fullName, age);
        this.workName = workName;
    }

    public String getWorkName() {
        return workName;
}

    public void setWorkName(String workName) {
        this.workName = workName;
    }

    public String toString(){
        return super.toString() //Вызов метода суперкласса
            +" "+workName;
    }
}
```

```
Student student = new Student ("Иванов Константин Евгеньевич",22);
System.out.println(student.toString());

Aspirant apsirant = new Aspirant("Сидоров Константин Сергеевич
",28,"Исследование объектно-реляционных баз данных");
System.out.println(apsirant.toString());

//Пример использования позднего связывания в JAVA
Student student1 = new Aspirant ("Петров Сергей
Сергеевич",22,"Исследование объектно-реляционных баз данных");

System.out.println(student1.toString());

// Происходит преобразование переменной типа
// Student к строке. Неявным образом вызывается
// метод toString
System.out.println(student);
```

Вызов конструкторов в иерархии происходит начиная с базового класса и заканчивая последним классом наследником.

Немного больше о наследовании. Дело в том, что все классы неявным образом наследуются от класса Object по мере изучения нашего курса, мы будем более детально рассматривать его методы.

Object clone()	Создает копию (клон) объекта
boolean equals(Object obj)	Сравнивает два объекта, и возвращает истину, если они равны
void finalize()	Вызывается средой выполнения при уничтожении объекта
int hashCode()	Представляет объект целым числом. Может использовать для идентификации объектов в приложении.
String toString()	Возвращает строковое представление объекта. Так же метод вызывается при преобразовании переменной объектного типа данных к строке

В отличии от языка C++ в Java нет возможности реализовывать множественое наследование. Но класс, помимо супер класса, может реализовывать неограниченный набор интерфейсов (интерфейсы рассматриваются позже)

Полиморфизм

Полиморфизмом называется взаимозаменяемость объектов имеющих одинаковый набор методов. Достигается полиморфизм созданием метода в классе наследнике,

сигнатура которого совпадает с методом в супер классе. Замете что в строчке System.out.println(student1.toString()); вызвался метод toString класса Aspirant, а не Student. Произошло это из-за переопределения метода toString(). Другими словами, вызывается метод объекта который находится в памяти, а не объекта ссылки.

Финализация

Финализация – термин присущий только Java программам. Финализацией называют запрет на дальнейшие использования элемента языка. Ранее мы сталкивались с финализацией в теме «константы», добавляя ключевое слово final к переменной (полю классы), мы запрещали изменять её значение. Другими словами мы её финализировали. Так же ключевое слово final можно применять к методам суперкласса, что приводит к запрету на переопределение это метода в классе наследнике, а использование ключевое слово final совместно с объявлением класса, запрещает наследовать класс.

Пример. Создадим иерархию наследования из двух классов, и финализируем некоторые методы.

```
public class First {
    void method1(){
        System.out.println("class First method1");
    }

    // Указывая ключевое слово final,
    // мы запрещаем переопределение этого
    // метода в классе наследнике
    final void method2(){
        System.out.println("class First method2");
    }
}
// Обозначая класс финализацией мы
// запрещаем его использование в качестве
// суперкласса
final public class Second extends First {
    // Мы можем переопределить только method1
    // т.к. он не объявлен в суперклассе как финальный
    void method1() {
        System.out.println("class Second method1");
    }
```

Статические поля и методы

Объявляя поле класса, мы декларируем то, что каждый экземпляр класса будет

иметь такое поле. Но, у нас есть возможность создавать поля и методы, которые будут являться общими для всех экземпляров класса. Такие поля называются статическими. Создается такое поле с помощью спецификатора **static** .

К примеру, создадим класс, который будет считать количество своих экземпляров. В классе сделаем статическую переменную (счетчик) count и будем инкрементировать её в конструкторе класса и декрементировать в деструкторе. Так же создадим статический метод, который будет возвращать значение переменной счетчика.

```
public class Ball {

    /*
    * Сатическое поле для хранения кол-ва
    * существующих экземпляров класса
    */
     private static int count =0;

    /*
    * Метод возвращает кол-ва
    * существующих экземпляров класса
    */
    static int getCountInstance(){
         return count;
    }

    private String color = "none";

    public Ball(String color){
        this.color = color;
        //Увеличиваем значение счетчика
        count++;
    }

    protected void finalize() {
        System.out.println("ASD");
        //Уменьшаем значение счетчика
        count--;
    }
}
```

Абстрактны классы

Иногда, при проектировании объектной модели, создается ситуация при которой мы четко можем выделить некий суперкласс, но нет возможности реализовать некоторые методы в нём, т.к. реализация этих методов в классах наследниках

существенно различается. Ярким примером является иерархия транспортных средств. Мы можем выделить некий суперкласс «Транспортное средство» и два класса наследника «Самолет» и «Пароход». Каждому транспортному средству присущий метод «Движение», но движение самолета и парохода различно по своей природе (а следовательно и в нашей реализации объектной модели так же должно быть различно). Следовательно, рождается два вопроса:

- Что должен содержать в своем теле метод «Движение» класса «Транспортное средство» ?
- Нужна ли возможность создавать класс «Транспортное средство» в нашем приложении имеющий неполную функциональность (частично описывающий объект предметной области) ?

На первый вопрос можно смело ответить «незнаем». По поводу второго, ответ однозначно «нет». Ситуация сложилась патовая, с одной стороны нам бы не помешал суперкласс, общий для нашей объектной модели, а с другой стороны класс который частично реализовывает функциональность нам однозначно не нужен. Решить нашу проблему помогут абстрактные классы.

Абстрактным называется класс, содержащий хотя бы один абстрактный метод.

Абстрактным называется метод, не содержащий реализации, другими словами метод в классе существует (тип возвращаемого значение, имя, параметры) а его тело отсутствует.

Следовательно, если мы имеем метод без реализации то создать экземпляр такого класса не можем, т.к. вызвать такой метод невозможно.

Рассмотрим пример. Иерархия классов описывающих фигуры. Базовым классом для всех фигур является класс Shape, который содержит поле color, хранящее цвет фигуры. Так же в классе определен метод draw(), который является абстрактным, т. к. когда говорим о фигуре мы понятия не имеем как она выглядит. Классы наслденики Rectangle(Прямоугольник) и Circle(Круг) переопределяют абстрактный метод draw, в котором собственно и реализовывают отрисовку круга и прямоугольника.

```
public abstract class Shape{
        protected String color = "red";
        public abstract void draw();
        public Shape(String color){
        this.color = color;
    }
}
```

```
public class Circle extends Shape{
    protected int x,y,r;

    public Circle(int x, int y, int r,String color) {
        super(color);
        this.x = x;
        this.y = y;
        this.r = r;
    }
    public void draw() {
        System.out.println("Drawing circle(x="+x+", y="+y+", radius="+r
+", color="+super.color+")");
    }
}
```

```
public class Rectangle extends Shape{
    protected int x,y,x1,y1;

    public Rectangle(int x, int y, int x1,int y1,String color) {
        super(color);
        this.x = x;
        this.y = y;
        this.x1 = x1;
        this.y1 = y1;
    }

    public void draw() {
        System.out.println("Drawing rectangle(x="+x+", y="+y+", x1="+x1
+", y1="+
        y1+" color="+super.color+")");
    }
}
```

Пример использования иерархии

```
Shape s[] = new Shape[3];
s[0] = new Circle(10,10,5,"black");
s[1] = new Rectangle(10,10,50,50,"yellow");
s[2] = new Circle(10,10,5,"green");

for(int i=0;i<s.length;i++)
    s[i].draw();
```

Интерфейсы

Интерфейс — это элемент языка программирования который, специфицирует набор услуг, предоставляемых классом. Другими словами интерфейс - это элемент языка программирования который содержит описание методов, но не содержит их реализацию. Объявление интерфейсов очень похоже на упрощенное объявление классов. Оно начинается с заголовка. Сначала указываются модификаторы. Интерфейс может быть объявлен как public и тогда он будет доступен для общего использования, либо модификатор доступа может не указываться, в этом случае интерфейс доступен только для типов своего пакета. Модификатор abstract для интерфейса не требуется, поскольку все интерфейсы являются абстрактными. Его можно указать, но делать этого не рекомендуется, чтобы не загромождать код. Далее записывается ключевое слово interface и имя интерфейса. После этого может следовать ключевое слово extends и список интерфейсов, от которых будет наследоваться объявляемый интерфейс. Родительских типов может быть много, главное, чтобы не было повторений и чтобы отношение наследования не образовывало циклической зависимости.

```
public interface Drawable extends Colorable, Resizable {
}
```

Тело интерфейса состоит из объявления элементов, то есть полей-констант и абстрактных методов. Все поля интерфейса должны быть public final static , так что эти модификаторы указывать необязательно и даже нежелательно, чтобы не загромождать код. Поскольку поля объявляются финальными, необходимо их сразу инициализировать.

```
public interface Directions {
    int RIGHT=1;
    int LEFT=2;
    int UP=3;
    int DOWN=4;
}
```

Все методы интерфейса являются public abstract и эти модификаторы также необязательны.

```
public interface Moveable {
    void moveRight();
    void moveLeft();
    void moveUp();
    void moveDown();
```

```
}
```

Каждый класс может реализовывать любые доступные интерфейсы. При этом в классе должны быть реализованы все абстрактные методы, появившиеся при наследовании от интерфейсов или родительского класса, чтобы новый класс мог быть объявлен неабстрактным.

Если из разных интерфейсов наследуются методы с одинаковой сигнатурой, то достаточно один раз описать реализацию, и она будет применяться для всех методов. Однако если у них различное возвращаемое значение, то возникает конфликт:

```
interface A {
    int getValue();
}

interface B {
    double getValue();
}
```

Пример

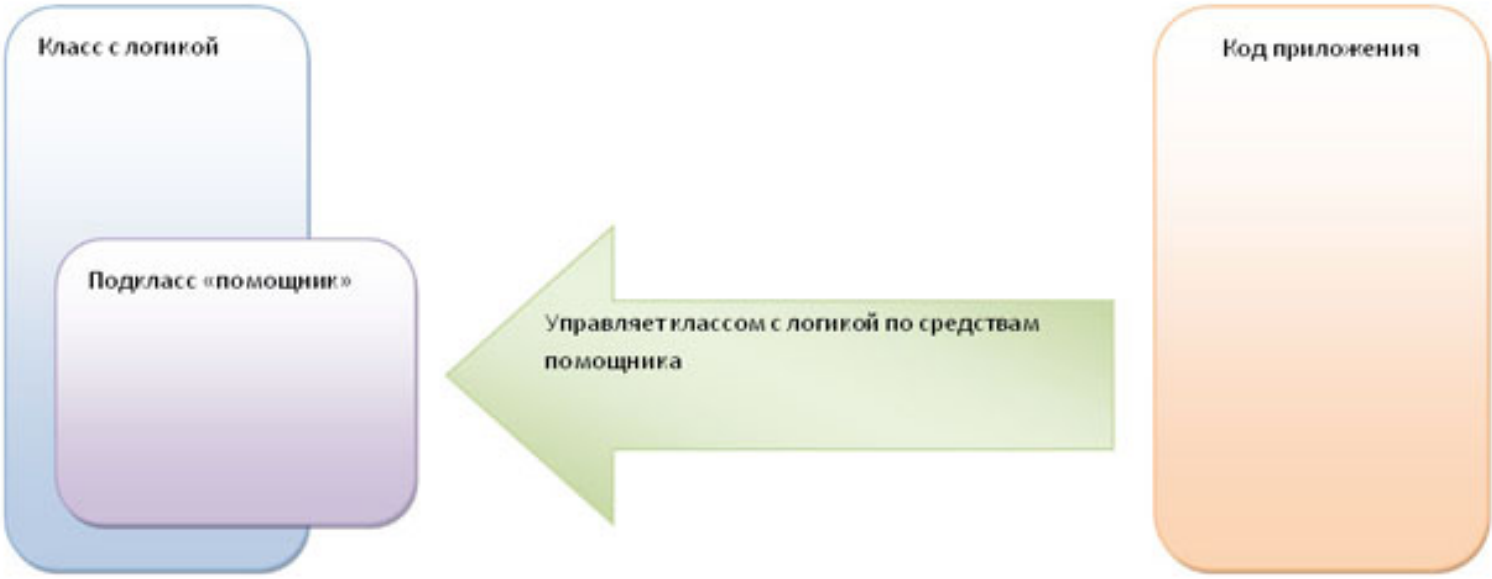
```
public interface Moveable {

    int defaultX = 0, defaultY=0;

    void moveRight();
    void moveLeft();
    void moveUp();
    void moveDown();
    void moveToDefaultPosition();
}

public class Circle extends Shape implements Moveable {

    protected int x,y,r;

    public Circle(int x, int y, int r,String color) {
        super(color);
        this.x = x;
        this.y = y;
        this.r = r;
    }
```

```
    public void draw() {
        System.out.println("Drawing circle(x="+x+", y="+y+", radius="+r
+", color="+super.color+")");
    }

    public void moveDown() {
        y=-1;
    }

    public void moveLeft() {
        x=-1;
}

    public void moveRight() {
        x+=1;
}

    public void moveUp() {
        y+=1;
    }

    public void moveToDefaultPosition() {
        x = defaultX;
        y = defaultY;
    }
}
```

Вложенные классы (Subclassing)

Подклассом (subclass) – называется класс, объявленный внутри другого класса. Подклассы используются:

- При реализации класса со сложной логикой работы. Возможна ситуация в которой Вам понадобятся некоторые вспомогательные классы (для расчетов, специфического вывода информации и т.д.). Что бы «не засорять» проект служебными классами, как правила их реализуют в виде подклассов.
- Для реализации класса помощника, который позволяет управлять экземпляром другого класса.



Подкласс имеет доступ ко всем полям внешнего класса, а так же спецификатор доступа который определяет, может ли быть доступен нам экземпляр подкласса за пределами внешнего класса.

Объявление подкласса:

```
СПЕЦИФИКАТОР_ДОСТУПА class ИМЯ_ВНЕШНЕГО_КЛАССА {
    СПЕЦИФИКАТОР_ДОСТУПА class ИМЯ_ПОДКЛАССА_КЛАССА
    {

    }
}
```

Спецификатор доступа подкласса определяет доступность класса за пределами внешнего. Спецификатор доступа private указывает на то, что доступ к полям и методам внутреннего класса имеет только внешний класс, а спецификаторы public и protected указывают что доступ к полям и методам может получить любой класс программы. Так же следует отметить, что наличие спецификатора доступа private запрещает использовать подкласс в классах наследниках внешнего.

Создание экземпляра подкласса

Создание подкласса во внешнем классе ничем не отличается от создания обычного экземпляра класса, а вот создание подкласса «из вне» происходит несколько иначе. Т.к. подкласс объявлен внутри другого то, для того что бы создать экземпляр подкласса вначале надо создать объект внешнего класса, а затем через него создать объект подкласса. При этом между классами устанавливается связь «Внешний класс» -> «Подкласс», другими словами созданный подкласс будет иметь доступ к полям и

методам того внешнего класса от которого был порожден. Создание объекта подкласса «из вне», происходит следующим образом:

```
ИМЯ_ВНЕШНЕГО_КЛАССА.ИМЯ_ВНУТРЕНЕГО_КЛАССА ИМЯ_ОБЪЕКТА =
ИМЯ_ОБЪЕКТА_ВНЕШНЕГО_КЛАССА.new КОНСТРУКТОР_ВНУТРЕНЕГО_КЛАССА;
```

Пример

```
public class Outer {
    private int field = 0;

    /*
    * Класс для управления полем внешнего класса
    */
    class Inner{
        /*
        * Метод устанавливает значение внешнего класса
        */
        public void setField(int f){
            field = f;
        }
    }

    public void showField(){
        System.out.println(field);
    }
}
```

```
class Start {
    public static void main(String args[]) {

        Outer out = new Outer();
        out.showField();

        Outer.Inner in = out.new Inner();
        in.setField(1);
        out.showField();

        Outer.Inner in2= out.new Inner();
        in2.setField(2);
        out.showField();

        Outer out2 = new Outer();
```

```
        out2.showField();
        Outer.Inner in3 = out2.new Inner();
        in3.setField(1);
        out2.showField();
    }
}
```

Объекты in и in2 созданы через объект out, следовательно, они могут получать доступ к полям и методам класса out. Объект in3 создан через экземпляр класса out2, т.е. in3 может получить доступ к полям и методам объекта out2.

Утилиты

Библиотека языка JAVA включает в себя набор вспомогательных классов, широко используемых в других встроенных пакетах Java. Эти классы расположены в пакетах java.lang и java.util. Они используются для работы с наборов объектов, взаимодействия с системными функциями низкого уровня, для работы с математическими функциями, генерации случайных чисел и манипуляций с датами и временем.

Простые оболочки для типов

Как вы уже знаете, Java использует встроенные примитивные типы данных, например, int и char. Эти типы данных не принадлежат к классовой иерархии Java. Они передаются методам по значению, передать их по ссылке невозможно. По этой причине для каждого примитивного типа в Java реализован специальный класс.

Number

Абстрактный класс Number представляет собой интерфейс для работы со всеми стандартными скалярными типами: — long, int, float и double. Класс имеет набор методов для доступа к содержимому объекта, которые возвращают (возможно округленное) значение объекта в виде значения каждого из примитивных типов:

- doubleValue() - возвращает содержимое объекта в виде значения типа double.
- floatValue() - возвращает значение типа float.
- intValue() - возвращает значение типа int
- longValue() - возвращает значение типа long

Double и Float

Double и Float — наследники класса Number. В дополнение к четырем методам, объявленным в суперклассе, эти классы содержат несколько методов, которые предназначены для работы со значениями типа double и float. У каждого из классов

есть конструкторы, позволяющие инициализировать объекты значениями типов double и float, кроме того, для удобства пользователя, эти объекты можно инициализировать и объектом типа String, содержащим текстовое представление вещественного числа. Приведенный ниже пример иллюстрирует создание представителей класса Double с помощью обоих конструкторов.

```
class DoubleDemo {
    public static void main(String args[]) {
        Double d1 = new Double(3.14159);
        Double d2 = new Double("314159E-5");
        System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
    }
}
```

Как Вы можете видеть из результата работы этой программы, метод equals возвращает значение true, а это означает, что оба использованных в примере конструктора создают идентичные объекты класса Double.

Бесконечность и NaN

В спецификации IEEE для чисел с вещественной точкой есть два значения типа double, которые трактуются специальным образом: бесконечность и NaN (Not a Number — неопределенность). Класс Double содержит методы для проверки этих условий, причем методы сущетсвуют в двух видах, в виде статических методов и в веде обычных методов. Статическим методам, значение передается в качестве аргументов, ну а обыный метод вызывается у объекта типа Double.

- (статический)isInfinite(d) возвращает true, если абсолютное значение указанного числа типа double бесконечно велико
- isInfinite() возвращает true, если абсолютное значение числа, хранящегося в данном объекте Double, бесконечно велико
- (статический) isNaN(d) возвращает true, если значение указанного числа типа double неопределено
- isNaN() возвращает true, если значение числа, хранящегося в данном объекте Double, неопределено

В прмере создаются два объекта типа Double, один с бесконечным, другой с неопределенным значением.

```
class InfNaN {
    public static void main(String args[]) {
        Double d1 = new Double(1/0.);
        Double d2 = new Double(0/0.);
        System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN
());
```

```
        System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN
());
    }
}
```

Результат работы программы:

```
Infinity: true, false
NaN: false, true
```

Integer и Long

Класс Integer — класс наследник Number, для чисел типов int, short и byte, а класс Long — соответственно для типа long. Помимо наследуемых методов своего суперкласса Number, классы Integer и Long содержат методы для разбора текстового представления чисел, и наоборот, для представления чисел в виде текстовых строк. Различные варианты этих методов позволяют указывать основание (систему счисления), используемую при преобразовании. Обычно используются двоичная, восьмеричная, десятичная и шестнадцатиричная системы счисления.

- parseInt(String) преобразует текстовое представление целого числа, содержащееся в переменной String, в значение типа int. Если строка не содержит представления целого числа, записанного в допустимом формате, Вы получите исключение NumberFormatException.
- parseInt(String, radix) аналог предыдущего метода за исключением того что Вы можете указывать основание, отличное от 10.
- toString(int) преобразует переданное в качестве параметра целое число в текстовое представление в десятичной системе.
- toString(int, radix) преобразует переданное в качестве первого параметра целое число в текстовое представление в задаваемой вторым параметром системе счисления.

Character

Character — простой класс-оболочка типа char. У него есть несколько полезных статических методов, с помощью которых можно выполнять над символом различные проверки и преобразования.

- isLowerCase(char ch) возвращает true, если символ-параметр принадлежит нижнему регистру (имеется в виду не просто диапазон a-z, но и символы нижнего регистра в кодировках, отличных от ISO-Latin-1).
- isUpperCase(char ch) делает то же самое в случае символов верхнего регистра.
- isDigit(char ch) и isSpace(char ch) возвращают true для цифр и пробелов,

соответственно.
- toLowerCase(char ch) и toupperCase(char ch) выполняют преобразования символов из верхнего в нижний регистр и обратно.

Boolean

Класс Boolean — это очень тонкая оболочка вокруг логических значений, она бывает полезна лишь в тех случаях, когда тип boolean требуется передавать по ссылке, а не по значению.

Перечисления

В Java для хранения групп однородных данных имеются массивы. Они очень полезны при использовании простых моделей доступа к данным. Перечисления же предлагают более совершенный объектно-ориентированный путь для хранения наборов данных сходных типов. Перечисления используют свой собственный механизм резервирования памяти, и их размер может увеличиваться динамически. У них есть интерфейсные методы для выполнения итераций и для просмотра. Их можно индексировать чем-нибудь более полезным, нежели простыми целыми значениями.

Интерфейс Enumeration

Enumeration — простой интерфейс, позволяющий вам обрабатывать элементы любой коллекции объектов. В нем задается два метода. Первый из них — метод hasMoreElements, возвращающий значение типа boolean. Он возвращает значение true, если в перечислении еще остались элементы, и false, если у данного элемента нет следующего. Второй метод — nextElement — возвращает обобщенную ссылку на объект класса Object, которую, прежде чем использовать, нужно преобразовать к реальному типу содержащихся в коллекции объектов.

Ниже приведен пример, в котором используется класс Enum, реализующий перечисление объектов класса Integer, и класс EnumerateDemo, создающий объект типа Enum, выводящий все значения перечисления. Обратите внимание на то, что в объекте Enum не содержится реальных данных, он просто возвращает последовательность создаваемых им объектов Integer.

```
import java.util.Enumeration;
class Enum implements Enumeration {
    private int count = 0;
    private boolean more = true;

    public boolean hasMoreElements() {
        return more;
    }
```

```
public Object nextElement() {
    count++;
    if (count > 4) more = false;
        return new Integer(count);
    }
}
class EnumerateDemo {
    public static void main(String args[]) {
        Enumeration enum = new Enum();
        while (enum.hasMoreElements()) {
            System.out.println(enum.nextElement());
        }
    }
}
```

Vector

Vector — это способный увеличивать число своих элементов массив ссылок на объекты. Внутри себя Vector реализует стратегию динамического расширения, позволяющую минимизировать неиспользуемую память и количество операций по выделению памяти. Объекты можно либо записывать в конец объекта Vector с помощью метода addElement, либо вставлять в указанную индексом позицию методом insertElementAt. Вы можете также записать в Vector массив объектов, для этого нужно воспользоваться методом copyInto. После того, как в Vector записана коллекция объектов, можно найти в ней индивидуальные элементы с помощью методов Contains, indexOf и lastIndexOf. Кроме того методы elementAt, firstElement и lastElement позволяют извлекать объекты из нужного положения в объекте Vector.

Stack

Stack — подкласс класса Vector, который реализует простой механизм типа "первым вошел — первым вышел" (FIFO). В дополнение к стандартным методам своего родительского класса, Stack предлагает метод push для помещения элемента в вершину стека и pop для извлечения из него верхнего элемента. С помощью метода peek вы можете получить верхний элемент, не удаляя его из стека. Метод empty служит для проверки стека на наличие элементов — он возвращает true, если стек пуст. Метод search ищет заданный элемент в стеке, возвращая количество операция pop, которые требуются для того чтобы перевести искомый элемент в вершину стека. Если заданный элемент в стеке отсутствует, этот метод возвращает -1. Ниже приведен пример программы, которая создает стек, заносит в него несколько объектов типа Integer, а затем извлекает их.

```
import java.util.Stack;
```

```
import java.util.EmptyStackException;

class StackDemo {
    static void showpush(Stack st, int a) {
        st.push(new Integer(a));
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }

    static void showpop(Stack st) {
        System.out.print("pop -> ");
        Integer a = (Integer) st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }

    public static void main(String args[]) {
        Stack st = new Stack();
        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);
        try {
            showpop(st);
        }
        catch (EmptyStackException e) {
            System.out.println("empty stack");
        }
    }
}
```

Ниже приведен результат, полученный при запуске этой программы. Обратите внимание на то, что обработчик исключений реагирует на попытку извлечь данные из пустого стека. Благодаря этому мы можем аккуратно обрабатывать ошибки такого рода.

```
stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
```

```
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: []
pop -> empty stack
```

Dictionary

Dictionary (словарь) — абстрактный класс, представляющий собой хранилище информации типа "ключ-значение". Ключ — это имя, по которому осуществляется доступ к значению. Имея ключ и значение, вы можете записать их в словарь методом put(key, value). Для получения значения по заданному ключу служит метод get(key). И ключи, и значения можно получить в форме перечисления (объект Enumeration) методами keys и elements. Метод size возвращает количество пар "ключ-значение", записанных в словаре, метод isEmpty возвращает true, если словарь пуст. Для удаления ключа и связанного с ним значения предусмотрен метод remove(key).

HashTable

HashTable — это подкласс Dictionary, являющийся конкретной реализацией словаря. Представителя класса HashTable можно использовать для хранения произвольных объектов, причем для индексации в этой коллекции также годятся любые объекты. Наиболее часто HashTable используется для хранения значений объектов, ключами которых служат строки (то есть объекты типа String). В очередном нашем примере в HashTable хранится информация об автомобиле.

```
import java.util.Dictionary;
import java.util.Hashtable;

class HTDemo {
    public static void main(String args[]) {
        Hashtable ht = new Hashtable();
        ht.put("brand", "Opel");
        ht.put("mark", "Corsa");
        ht.put("engine", "1.2 ecotec");
        ht.put("price", new Double(95000));
        show(ht);
    }
    static void show(Dictionary d) {
        System.out.println("Brand: " + d.get("brand"));
        System.out.println("Mark: " + d.get("mark"));
        System.out.println("Engine: " + d.get("engine"));
```

```
        System.out.println("Price: " + d.get("price"));
    }
}
```

Properties

Properties — подкласс HashTable, в который для удобства использования добавлено несколько методов, позволяющих получать значения, которые, возможно, не определены в таблице. В методе getProperty вместе с именем можно указывать значение по умолчанию:

```
getProperty("имя","значение_по_умолчанию");
```

При этом, если в таблице свойство "имя" отсутствует, метод вернет "значение_по_умолчанию". Кроме того, при создании нового объекта этого класса конструктору в качестве параметра можно передать другой объект Properties, при этом его содержимое будет использоваться в качестве значений по умолчанию для свойств нового объекта. Объект Properties в любой момент можно записать либо считать из потока — объекта Stream (потоки будут обсуждаться в следующих уроках). Ниже приведен пример, в котором создаются и впоследствии считываются некоторые свойства:

```
import java.util.Properties;

class PropDemo {
    static Properties prop = new Properties();

    public static void main(String args[]) {
        prop.put("Brand", "put brand here");
        prop.put("Mark", "put mark here");
        prop.put("Engine", "engine not set");
        Properties book = new Properties(prop);
        book.put("Brand", "Opel");
        book.put("Mark", "Corsa");
        System.out.println("Brand: " +book.getProperty("Brand"));
        System.out.println("Mark: " +book.getProperty("Mark"));
        System.out.println("Engine: " +book.getProperty("Engine"));
        System.out.println("other: " +book.getProperty("other", "???"));
    }
}
```

Здесь мы создали объект prop класса Properties, содержащий три значения по умолчанию для полей Title, Author и isbn. После этого мы создали еще один объект

Properties с именем book, в который мы поместили реальные значения для полей Title и Author. В следующих трех строках примера мы вывели результат, возвращенный методом getProperty для всех трех имеющихся ключей. В четвертом вызове getProperty стоял несуществующий ключ “ean”. Поскольку этот ключ отсутствовал в объекте book и в объекте по умолчанию prop, метод getProperty выдал нам указанное в его вызове значение по умолчанию, то есть “???”:

```
Brand: Opel
Mark: Corsa
Engine: engine not set
other: ???
```

StrinsTokenizer

Обработка текста часто подразумевает разбиение текста на последовательность лексем - слов (tokens). Класс StringTokenizer предназначен для такого разбиения, часто называемого лексическим анализом или сканированием. Для работы StringTokenizer требует входную строку и строку символов-разделителей. По умолчанию в качестве набора разделителей используются обычные символы-разделители: пробел, табуляция, перевод строки и возврат каретки. После того, как объект StringTokenizer создан, для последовательного извлечения лексем из входной строки используется его метод nextToken. Другой метод — hasMoreTokens — возвращает true в том случае, если в строке еще остались неизвлеченные лексемы. StringTokenizer также реализует интерфейс Enumeration, а это значит, что вместо методов hasMoreTokens и nextToken вы можете использовать методы hasMoreElements и nextElement, соответственно.

Ниже приведен пример, в котором для разбора строки вида "ключ=значение" создается и используется объект StringTokenizer. Пары "ключ=значение" разделяются во входной строке двоеточиями.

```
import java.util.StringTokenizer;
class STDemo {
    static String in = "Brand=Opel:" + "Mark=Corsa:" + "Engine=1.2
ecotec";
    public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer(in, "=:");
        while (st.hasMoreTokens()) {
        String key = st.nextToken();
        String val = st.nextToken();
        System.out.println(key + "\t" + val);
    }
}
}
```

Runtime

Класс Runtime инкапсулирует интерпретатор Java. Вы не можете создать нового представителя этого класса, но можете, вызвав его статический метод, получить ссылку на работающий в данный момент объект Runtime. Обычно апплеты (JAVA приложения загруженые через WEB и выполняемые в окне браузера) и другие непривелигированные программы не могут вызвать ни один из методов этого класса, не возбудив при этом исключения SecurityException. Одна из простых вещей, которую вы можете проделать с объектом Runtime — его останов, для этого достаточно вызвать метод exit(int code).

Управление памятью

Хотя Java и представляет собой систему с автоматической сборкой мусора, вы для проверки эффективности своего кода можете захотеть узнать, каков размер “кучи” и как много в ней осталось свободной памяти. Для получения этой информации нужно воспользоваться методами totalMemory и freeMemory

.

ВНИМАНИЕ!
При необходимости Вы можете "вручную" запустить сборщик мусора, вызвав метод gc. Если вы хотите оценить, сколько памяти требуется для работы вашему коду, лучше всего сначала вызвать gc, затем freeMemory, получив тем самым оценку свободной памяти, доступной в системе. Запустив после этого свою программу и вызвав freeMemory внутри нее, вы увидите, сколько памяти использует ваша программа.

Выполнение других программ

В безопасных средах вы можете использовать Java для выполнения других полновесных процессов в своей многозадачной операционной системе. Несколько форм метода exec позволяют задавать имя программы и ее параметры.

В очередном примере используется специфичный для Windows вызов exec, запускающий процесс notepad — простой текстовый редактор. В качестве параметра редактору передается имя одного из исходных файлов Java. Обратите внимание — exec автоматически преобразует в строке-пути символы "/" в разделители пути в Windows — "\".

```
class ExecDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
```

```
        Process p = null;
        String cmd[] = { "notepad", "/java/src/java/lang/Runtime.java" };
        try {
            p = r.exec(cmd);
        } catch (Exception e) {
            System.out.println("error executing " + cmd[0]);
        }
    }
}
```

System

Класс System содержит любопытную коллекцию глобальных функций и переменных. В большинстве примеров этой книге для операций вывода мы использовали метод System.out.println(). Метод currentTimeMillis возвращает текущее системное время в виде миллисекунд, прошедших с 1 января 1970 года. Метод arraycopy можно использовать для быстрого копирования массива любого типа из одного места в памяти в другое. Ниже приведен пример копирования двух массивов с помощью этого метода.

```
class ACDemo {
    static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
    static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };

    public static void main(String args[]) {
        System.out.println("a = " + new String(a, 0));
        System.out.println("b = " + new String(b, 0));
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println("a = " + new String(a, 0));
        System.out.println("b = " + new String(b, 0));
        System.arraycopy(a, 0, a, 1, a.length - 1);
        System.arraycopy(b, 1, b, 0, b.length - 1);
        System.out.println("a = " + new String(a, 0));
        System.out.println("b = " + new String(b, 0));
    }
}
```

Как вы можете заключить из результата работы этой программы, копирование можно выполнять в любом направлении, используя в качестве источника и приемника один и тот же объект.

```
a = ABCDEFGHIJ
b = MMMMMMMMMM
```

```
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Свойства окружения

Исполняющая среда Java предоставляет доступ к переменным окружения через представителя класса Properties (описанного ранее в этой главе), с которым можно работать с помощью метода System.getProperty. Для получения полного списка свойств можно вызвать метод System.getProperties()

Имя	Значение
java.version	Версия интерпретатора Java
java.vendor	Строка идентификатора, заданная разработчиком
java.vendor.url	URL разработчика
java.class.version	Версия Java API
java.class.path	Значение переменной CLASSPATH
java.home	Каталог, в котором инсталлирована среда Java
java.compiler	Компилятор JIT
os.name	Название операционной системы
os.arch	Архитектура компьютера, на котором выполняется программа
os.version	Версия операционной системы Web-узла
file.separator	Зависящие от платформы разделители файлов (/ или \)
path.separator	Зависящие от платформы разделители пути (: или ;)
line.separator	Зависящие от платформы разделители строк (\n или \r\n)
user.name	Имя текущего пользователя
user.home	Домашний каталог пользователя
user.dir	Текущий рабочий каталог
user.language	2-символьный код языка для местности по умолчанию
user.region	2-символьный код страны для местности по умолчанию
user.timezone	Временной пояс по умолчанию
user.encoding	Кодировка сиволов для местности по умолчанию
user.encoding.pkg	Пакет, содержащий конверторы для преобразования символов из местной кодировки в Unicode

Настраиваемые типы(Generic types)

Настраиваемым типом называется класс который описывает логику работу на не описывает типы данных, с которыми происходят действия. Настраиваемые типы в Java, являются аналогами шаблонных классов в C++.

Создание настраиваемого типа данных

```
class ИМЯ_КЛАССА <псевдотип 1, псевдотип 2,…, псевдотип N>
{
    ТЕЛО_КЛАССА
}
```

Пример. Настраиваемый тип данных хранящий пару значений

```
public class Pair <T,T2>{
    T v1;
    T2 v2;

    Pair(T v1,T2 v2)
    {
        this.v1 = v1;
        this.v2 = v2;
    }

    public T getV1() {
        return v1;
    }

    public void setV1(T v1) {
        this.v1 = v1;
    }

    public T2 getV2() {
        return v2;
    }

    public void setV2(T2 v2) {
        this.v2 = v2;
    }
}
```

Ограничения в настраиваемых типах

Создавая настраиваемый тип, мы можем создавать ограничения на псевдотипы, к

примеру, может определить псевдотип который должен реализовывать определенный интерфейс или быть наследником какого-то класса. К примеру, перепишем наш класс Pair таким образом, что бы объект псевдотипа T обязательно реализовывал интерфейс Moveable.

```
public class Pair < T extends Moveable,T2 > {
    T v1;
    T2 v2;

    Pair(T v1,T2 v2)
    {
        this.v1 = v1;
        this.v2 = v2;
    }

    public T getV1() {
        return v1;
    }

    public void setV1(T v1) {
        this.v1 = v1;
    }

    public T2 getV2() {
        return v2;
    }

    public void setV2(T2 v2) {
        this.v2 = v2;
    }
}
```

Создание объектов настраиваемого типа

```
ИМЯ_КЛАССА<ОПРЕДЕЛЕНИЕ_ТИПОЗАМЕНИТЕЛЕЙ> ИМЯ_ОБЪЕКТА = new
ИМЯ_КЛАССА<ОПРЕДЕЛЕНИЕ_ТИПОЗАМЕНИТЕЛЕЙ>(ПАРАМЕТРЫ_КОНСТРУКТОРА);
```

```
Pair<Circle,Integer> pair = new Pair<Circle,Integer>(new Circle
(10,100,100,"blue"),4);
```

Настраиваемые контейнеры

Рассмотри пример

```
Vector v = new Vector();
v.add(10);
v.add("Hello");

Integer a = (Integer)v.get(1);
```

В последней строчке кода мы получаем исключительную ситуацию java.lang. ClassCastException, из-за того что неможем преобразовать значение типа String в значение типа Integer. Для того что бы избегать подобных ситуаций в Java используют настраиваемые контейнеры (так же они известны как типизируемые контейнры), при создании которых явным образом указывается тип данных который будет помещатся в контейнер.

```
Vector<Integer> v = new Vector<Integer>();
v.add(10);
/* Ошибка на этапе компиляции, т.к. контейнер
может хранить только целочисленные значения */
v.add("Hello"); Integer a = (Integer)v.get(1);
```

Набор настраиваемых контейнеров достаточно велик. Все контейнеры которые рассматривалисб выше, существуют в виде настраиваемых контейнеров.

Практический пример

Разработаем объектную модель описывающею работу банка (Клиенты, Счета, Операции над счетами).
Классы приложения:

Название	Описание
Bank	Класс описывающий банк. Содержит коллекции для хранения клиентов, счетов, операций над счетами.
Account	Класс описывающий счет в банке. Конструктор этого класса имеет спецификатор доступа private, для того чтобы запретить создание экземпляра класса (нельзя создать счет вне банка, другими словами созданный счет в обязательном порядке должен принадлежать банку). Создать счет можно с помощью метода createAccount класса Bank, метод создает счет и размещает его в коллекции счетов. Если счет будет создаваться для существующего клиента, то повторно объект клиента (см. ниже) создаваться небудет

Client	Класс описывающий клиента банка
Operation	Абстрактный класс, который является базовым классом над всеми операциями банка, имеет поле amount которое хранит количество денежных средств учавствующее в операции и абстрактный метод doWork, который выполняет оперцию
AccountOperation	Класс который реализуют операцию перемещения денежных средств по счету, наследник класса Operation
CrossAccountOperation	Класс который реализуют операцию перемещения денежных средств между счетами, наследник класса Operation
OperationException	Класс который описывает исключительную ситуацию, которая может возникнуть при выполнении операции

Листинг. Класс Bank

```
package BankSystem;

import java.util.*;

public class Bank{
    private static int accountNum = 1;

    public class Account{
        private double balance;
        private final String number;
        private Client client;

        public Account(Client client,double startBalance){
            this(client);
            balance = startBalance;
        }

        public Account(Client client){
            this.client = client;
            balance = 0;
            number = "Acc "+accountNum++;
        }

        public double getBalance() {
            return balance;
        }

        public void setBalance(double balance) {
            this.balance = balance;
        }

        public String toString(){
```

```
            return "Счет № "+accountNum+" Владелец:"+client.toString()+"
Баланс:"+balance;
        }
    }

    private Vector<Client> clients = new Vector<Client>();
    private Vector<Account> accounts = new Vector<Account>();
    private Vector<Operation> operations = new Vector<Operation>();

    public void addOperation(Operation operation){
        operations.add(operation);
    }

    public void runOperations( ){
        for (int i=0; i<operations.size(); i++){
            try {
                operations.get(i).doWork();
            }catch(OperationException ex){
                System.out.println(ex.getMessage());
            }
        }
        operations.clear();
    }

    public Bank.Account createAccount(String clientName,String passport,
double initialBalance){
        Client cl = null;

        for (int i=0;i<clients.size();i++)
            if (clients.get(i).getPassport() == passport){
                cl = clients.get(i);
                break;
        }
        if (cl == null) {
            cl = new Client(clientName, passport);
            clients.add(cl);
        }
        Account acc= new Account(cl,initialBalance);
        accounts.add(acc);

        return acc;
    }

    public Bank.Account createAccount(String clientName,String passport){
        return createAccount(clientName,passport,0);
    }

    public String toString(){
```

```
        StringBuilder builder = new StringBuilder("Состояния счетов
\n-------------------------\n");
        for (int i=0; i < accounts.size(); i++){
            builder.append(accounts.get(i));
            builder.append("\n");
        }

        return builder.toString();
    }
}
```

Листинг. Класс Client

```
package BankSystem;

public class Client{
    private String passport;
    private String name;
    private Bank.Account account;
    public Client(String name, String passport){
        this.passport = passport;
        this.name = name;
    }

    public String getPassport() {
        return passport;
    }
    public String getName() {
        return name;
    }
    public String toString(){
        return this.name+" "+this.passport;
    }
}
```

Листинг. Класс Operation

```
package BankSystem;

/**
* Абстрактный класс, который описывает
* банковскую операцию
*/
public abstract class Operation{
    //Денежные средства в банковской операции
    protected double amount;
    //Выполняет банковскую операцию
```

```
    public abstract void doWork( ) throws OperationException;
}
```

Листинг. Класс AccountOperation

```
package BankSystem;
/**
* Операция перемещения денежных
* средств по счету
*/
public class AccountOperation extends Operation{
    //Счет учавствующий в операции
    private Bank.Account account;

    public AccountOperation(Bank.Account account, double amount) {
        this.account = account;
        this.amount = amount;
    }
    public void doWork() throws OperationException {
        double balance = account.getBalance();

        double result = balance + amount ;
        if (result < 0)
            throw new OperationException("Недостаточно денежных средств
на счете "+account);

        account.setBalance(result);
    }
}
```

Листинг. Класс CrossAccountOperation

```
package BankSystem;

/**
* Операция перемещения денежных
* средств между счетами
*/
public class CrossAccountOperation extends Operation{
    //Счет с которого уходят денежные средства
    private Bank.Account outAccount;

    //Счет на который поступают денежные средства
    private Bank.Account inAccount;

    public CrossAccountOperation(Bank.Account inAccount,Bank.Account
outAccount,double amount){
```

```
        this.inAccount = inAccount;
        this.outAccount = outAccount;
        super.amount = amount;
    }

    public void doWork() throws OperationException{
        double balance = outAccount.getBalance();
        if (balance < amount)
            throw new OperationException("Недостаточно денежных средств
на счете "+outAccount);
        inAccount.setBalance(balance+amount);
        outAccount.setBalance(balance-amount);
    }
}
```

```
Листинг. Пример работы с объектной моделью
package BankSystem; public class Banking {

    public static void main(String[] args) {
        Bank bank = new Bank();
        Bank.Account acc = bank.createAccount("Иванов", "АК123456");
        Bank.Account acc1 = bank.createAccount("Петров", "АЕ123456",50);
        AccountOperation oper = new AccountOperation(acc,100);
        CrossAccountOperation oper2 = new CrossAccountOperation(acc,
acc1,10);
        bank.addOperation(oper);
        bank.addOperation(oper2);
        bank.runOperations();

        System.out.print(bank);
    }
}
```

Домашнее задание

Домашнее задание этого урока строится на практическом примере описаном выше.

1. Создайте класс Tree<T> (настраиваемое бинарное дерево) и используйте его для хранения клиентов банка (вместо Vector).Обратите внимание что код приведеный ниже работать небудет, т.к. Java-компилятор незнает как сравнить две переменные неизвестного типа. Для того что реализовать сравнение в классе дерева воспользуйтесь стандартным интерфейсом Comparator (более детальную информацию смотрите в Java документации).

```
class GenericTest<T>{
```

```
    T a,b;
    GenericTest(T a,T b){
        this.a = a;
        this.b = b;
    }

    public boolean aIsGrant(){
        return a>b;
    }
}
```

2. Необходимо логировать ошибки происходящие во время банковских операций. Ошибка должна хранится в виде (Дата, Сообщение). Разработайте объектную модель которая позволит хранить ошибки. При создании модели, обратите внимание на то что неплохобы было добавлять информацию об ошибках с любого места программы. Таким образом унифицировать процесс обработки ошибок.