Классы и объекты

Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса.

Пример 1. Класс Person

Как правило, классы определяются в разных файлах. В данном случае для простоты мы определяем два класса в одном файле. Стоит отметить, что в этом случае только один класс может иметь модификатор public.

```
public class Main {

        public static void main(String... args) {

// вызов первого конструктора без параметров
            Person bob = new Person();
            bob.displayInfo();

// вызов второго конструктора с одним параметром
            Person tom = new Person("Tom");
            tom.displayInfo();

// вызов третьего конструктора с двумя параметрами
            Person sam = new Person("Sam", 25);
            sam.displayInfo();
        }
    }

class Person{

    String name;    // имя
    int age;        // возраст

    Person()
    {
        name = "Undefined";
        age = 18;
    }
    Person(String n)
    {
        name = n;
        age = 18;
    }
    Person(String n, int a)
    {
        name = n;
        age = a;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Пример 2. Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы

```
class Person{

    String name;    // имя
    int age;        // возраст
    Person()
    {
        this("Undefined", 18);
    }
    Person(String name)
    {
        this(name, 18);
    }
    Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Пример 3. Инициализаторы

Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта. Инициализатор выполняется до любого конструктора. То есть в инициализатор мы можем поместить код, общий для всех конструкторов.

```
class Person{

    String name;    // имя
    int age;        // возраст

    /*начало блока инициализатора*/
    {
        name = "Undefined";
        age = 18;
    }
    /*конец блока инициализатора*/
    Person(){

    }
    Person(String name){

        this.name = name;
    }
    Person(String name, int age){

        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }}
```

Пакеты

Как правило, в Java классы объединяются в **пакеты**. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. **Принадлежность к пакету позволяет гарантировать однозначность имен**.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву **package**, после которой указывается имя пакета:

```
package название_пакета;
```

Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом. А путь к файлам внутри проекта соответствует названию пакета этих файлов. Например, если классы принадлежат пакету **mypack**, то эти классы помещаются в проекте в папку **mypack**.

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится **в пакете по умолчанию, который не имеет имени**.

Подключение встроенных классов

Директива **import** указывается в самом начале кода, после чего идет имя подключаемого класса


```
import java.util.*; // импорт всех классов из пакета java.util
```

Возможна ситуация, когда мы используем два класса с одним и тем же названием из двух разных пакетов, например, класс **Date** имеется и в пакете java.util, и в пакете java.sql. И если нам надо одновременно использовать два этих класса, то необходимо указывать полный путь к этим классам в пакете:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date();
```

Статический импорт

В java есть также особая форма импорта - статический импорт. Для этого вместе с директивой import используется модификатор static

```
import static java.lang.System.*;
import static java.lang.Math.*;

public class Main {

        public static void main(String... args) {
            double result = sqrt(20);
            out.println(result);
}}
```

Здесь происходит статический импорт классов System и Math. Эти классы имеют статические методы. Благодаря операции статического импорта мы можем использовать эти методы без названия класса. Например, писать не Math.sqrt(20), а sqrt(20), так как функция sqrt(), которая возвращает квадратный корень числа, является статической.

То же самое в отношении класса System: в нем определен статический объект out, поэтому мы можем его использовать без указания класса.

Модификаторы доступа

- **public**: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.
- **private**: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.
- **protected**: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах.
- **Модификатор по умолчанию**. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.

```
public class Main {

        public static void main(String... args) {
            Person kate = new Person("Kate", 32, "Baker Street", "+12334567");
            kate.displayName();     // метод public
            kate.displayAge();      // метод имеет модификатор по умолчанию
            kate.displayPhone();    // метод protected
            //kate.displayAddress();  // ! Ошибка, метод private

            System.out.println(kate.name);      // модификатор по умолчанию
            System.out.println(kate.address);   // модификатор public
            System.out.println(kate.age);       // модификатор protected
        //    System.out.println(kate.phone);   // !Ошибка, модификатор private
        }
    }

class Person{

    String name;
    protected int age;
    public String address;
    private String phone;

    public Person(String name, int age, String address, String phone){
        this.name = name;
        this.age = age;
        this.address = address;
        this.phone = phone;
    }
    public void displayName(){
        System.out.printf("Name: %s \n", name);
    }
    void displayAge(){
```

```
        System.out.printf("Age: %d \n", age);
    }
    private void displayAddress(){
        System.out.printf("Address: %s \n", address);
    }
    protected void displayPhone(){
        System.out.printf("Phone: %s \n", phone);
    }}
```

В данном случае оба класса расположены в одном пакете - пакете по умолчанию, поэтому в классе **Main** мы можем использовать все методы и переменные класса Person, которые имеют модификатор по умолчанию, **public** и **protected**. А поля и методы с модификатором **private** в классе **Main** не будут доступны.

Если бы класс **Main** располагался бы в другом пакете, то ему были бы доступны только поля и методы с модификатором **public**.

Модификатор доступа должен предшествовать остальной части определения переменной или метода.

Модификатор final в Java

Суть **модификатора final** - сделать дальнейшее изменение объекта невозможным. С английского "final" можно перевести как **"последний, окончательный"**:

Можно применять этот модификатор **тремя способами**: для класса, для поля (переменной) и для метода.

Final для полей

Если вы хотите, чтобы **после инициализации** никто **не мог бы изменить** вашу переменную, напишите слово "final":

```
final int I = 1;
System.out.println(I);
```

или так

```
final int I;
I = 10;

System.out.println(I);
```

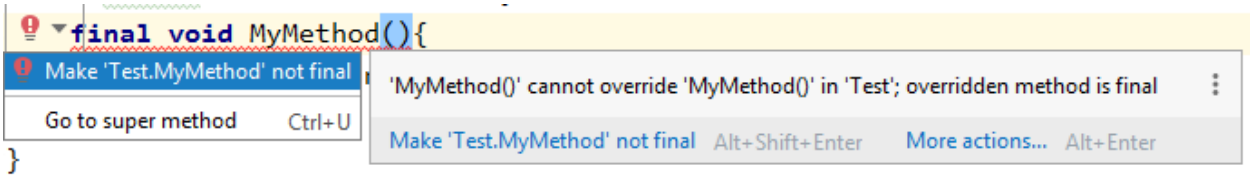
Final для методов

```
public class Main {

     final static void MyMethod(){
        System.out.println("Hello!");
     }
    public static void main(String args[]) {
```

```
        MyMethod();
}}
```

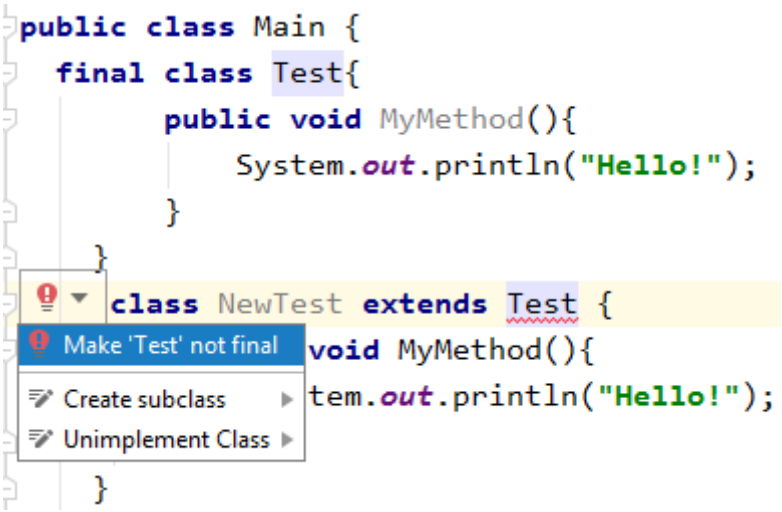
Это будет значить, что при наследовании данный метод **нельзя переопределить**:




```
class Test{
    final void MyMethod(){
        System.out.println("Hello!");
    }
}
class ChildTest extends Test {
    final void MyMethod(){
        System.out.println("Hello!");
    }
}
```

Final для классов

Модификатор **final** может применяться к классам тоже. Это будет означать, что нельзя создать наследников этого класса:



Инкапсуляция

Казалось бы, почему бы не объявить все переменные и методы с модификатором **public**, чтобы они были доступны в любой точке программы вне зависимости от пакета или класса? Возьмем, например, поле age, которое представляет возраст. Если другой класс имеет прямой доступ к этому полю, то есть вероятность, что в процессе работы программы ему будет передано некорректное значение, например, отрицательное число. Подобное изменение данных не является желательным. Либо же мы хотим, чтобы некоторые данные были доступны напрямую, чтобы их можно было вывести на консоль или просто узнать их значение. В этой связи

рекомендуется **как можно больше ограничивать доступ к данным**, чтобы защитить их от нежелательного доступа извне (как для получения значения, так и для его изменения). Использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом. Подобное **сокрытие данных внутри некоторой области видимости называется инкапсуляцией**.

Так, как правило, вместо непосредственного использования полей, как правило, **используют методы доступа**.

```
public class Main {

    public static void main(String[] args) {

        Person kate = new Person("Kate", 30);
        System.out.println(kate.getAge());      // 30
        kate.setAge(33);
        System.out.println(kate.getAge());      // 33
        kate.setAge(123450);
        System.out.println(kate.getAge());      // 33
    }
}
class Person{

    private String name;
    private int age;

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = name;
    }
    public int getAge(){
        return this.age;
    }
    public void setAge(int age){
        if(age > 0 && age < 110)
            this.age = age;
    }
}
```

Вместо непосредственной работы с полями **name** и **age** в классе **Person** мы будем работать с методами, которые устанавливают и возвращают значения этих полей.

Объекты как параметры методов

Объекты классов, как и данные примитивных типов, могут передаваться в методы. Однако в данном случае есть одна особенность - при передаче объектов в качестве значения передается копия ссылки на область в памяти, где расположен этот объект.

Пример. Класс Box

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

    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);

    Box myclone = new Box(mybox1);

    double vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);

}}


class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
    }
}
```

Пример. Класс Person

Main.java

```
public class Main {
    static void changeName(Person p, String str){
        p.setName(str);
    }
```

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

        Person kate = new Person("Kate");
        System.out.println(kate.getName());     // Kate

        changeName(kate,"Alice");
        System.out.println(kate.getName());     // Alice
    }

}
```

Person.java

```
class Person{

    private String name;

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

        return this.name;
    }
}
```


Здесь в метод **changeName** передается объект **Person**, у которого изменяется имя. Так как в метод будет передаваться копия ссылки на область памяти, в которой находится объект **Person**, то переменная **kate** и параметр p метода **changeName** будут указывать на один и тот же объект в памяти. Поэтому после выполнения метода у объекта **kate**, который передается в метод, будет изменено имя с "**Kate**" на "**Alice**".

Внутренние и вложенные классы

Классы могут быть вложенными (nested), то есть могут быть определены внури других классов. Частным случаем вложенных классов являются внутренние классы (inner class).

Например, имеется класс Person, внутри которого определен класс Account:

Person.java

```
class Person{

    private String name;
    Account account;

    Person(String name, String password){
        this.name = name;
        account = new Account(password);
    }
    public void displayPerson(){
```

```
        System.out.printf("Person \t Name: %s \t Password: %s \n", name,
account.password);
    }

    public class Account{
        private String password;

        Account(String pass){
            this.password = pass;
        }
        void displayAccount(){
            System.out.printf("Account Login: %s \t Password: %s \n",
Person.this.name, password);
        }
    }
}
```

Main.java

```
public class Main {

    public static void main(String[] args) {
        Person tom = new Person("Tom", "qwerty");
        tom.displayPerson();
        tom.account.displayAccount();
    }
}
```

Внутренний класс ведет себя как обычный класс за тем исключением, что его объекты могут быть созданы только внутри внешнего класса.

Внутренний класс имеет доступ ко всем полям внешнего класса, в том числе закрытым с помощью модификатора private. Аналогично внешний класс имеет доступ ко всем членам внутреннего класса, в том числе к полям и методам с модификатором private.

Ссылку на объект внешнего класса из внутреннего класса можно получить с помощью выражения Внешний_класс.this, например, Person.this.

Объекты внутренних классов могут быть созданы только в том классе, в котором внутренние классы определены. В других внешних классах объекты внутреннего класса создать нельзя.

Еще одной особенностей внутренних классов является то, что их можно объявить внутри любого контекста, в том числе внутри метода и даже в цикле:

Person.java

```
class Person{

    private String name;

    Person(String name){
        this.name = name;
    }

    public void setAccount (String password){

        class Account{

            void display(){
                System.out.printf("Account Login: %s \t Password: %s \n", name,
password);
            }
```

```
        }
        Account account = new Account();
        account.display();
    }
}
```

Main.java

```
public class Main {

    public static void main(String[] args) {
        Person tom = new Person("Tom");
        tom.setAccount("qwerty");
    }
}
```

Задания

1. Создать класс прямоугольного треугольника. Поля – основание и высота. Методы – площадь, гипотенуза, периметр.
2. Создать класс окружность. Поля – координаты центра и радиус. Предусмотреть конструктор для вырожденной окружности, когда радиус равен нулю. Методы – площадь, длина окружности.
3. Создать класс студент. Поля – фамилия, имя, курс, группа, средний балл. Методы – перевести на следующий курс. Изменить средний балл. Стипендия в зависимости от среднего балла.
4. Создать класс книга. Поля – название, автор, год, цена, количество. Методы – изменить количество. Изменить цену в зависимости от количества. Стоимость всего количества книг.