

# Объектная модель в Java

В этой лекции рассматриваются особенности объектной модели Java:

- использование ключевых слов `this` и `super`
- понятие абстрактного метода и класса
- особый тип в Java, *интерфейс*
- понятие *полиморфизма*

# Темы лекции 13. Дополнительные сведения о методах и классах

- Ключевые слова `this` и `super`
- Ключевое слово `abstract`
- Интерфейсы
- Объявление интерфейсов
- Реализация интерфейса
- Применение интерфейсов
- Полиморфизм
- Поля
- Методы
- Полиморфизм и объекты

# Ключевые слова `this` и `super`

```
class Test {  
    public Object getThis()  
    {  
        return this;  
        // Проверим, куда указывает эта ссылка  
    }  
    public static void main(String s[])  
    {  
        Test t = new Test();  
        System.out.println(t.getThis()==t);  
        // Сравнение  
    }  
}
```

Результатом работы программы будет:

**true**

# Ключевые слова `this` и `super`

```
class Human {  
    public static void register(Human h) {  
        System.out.println(h.name+  
            " is registered.");    }  
  
    private String name;  
    public Human (String s) {  
        name = s;  
        register(this); // саморегистрация  
    }  
    public static void main(String s[]) {  
        new Human("John");  
    }  
}
```

# Ключевые слова `this` и `super`

Другое применение `this` рассматривалось в случае "затемняющих" объявлений:

```
class Human {  
    private String name;  
  
    public void setName(String name) {  
        this.name=name;  
    }  
}
```

# Ключевые слова `this` и `super`

Переопределением называют объявление метода, сигнатура которого совпадает с одним из методов родительского класса.

```
class Parent {  
    public int getValue() {  
        return 5;  
    }  
}  
  
class Child extends Parent {  
    public int getValue() { // Переопределение метода  
        return 3;  
    }  
  
    public static void main(String s[]) {  
        Child c = new Child();  
        // пример вызова переопределенного метода  
        System.out.println(c.getValue());    }  
}
```

# Ключевые слова `this` и `super`

```
class Parent {
    public int getValue() {
        return 5; }}
class Child extends Parent {
    // переопределение метода
    public int getValue() {
        // обращение к методу родителя
        return super.getValue()+1;
    }

    public static void main(String s[]) {
        Child c = new Child();
        System.out.println(c.getValue());
    }
}
```

# Ключевые слова `this` и `super`

Важно:

- Если родительский класс переопределил функциональность своего родителя, значит, она не будет доступна его наследникам.
- Поскольку ключевые слова `this` и `super` требуют наличия ассоциированного объекта, т.е. динамического контекста, использование их в статическом контексте запрещено.



# Ключевое слово `abstract`

Иногда имеет смысл описать только заголовок метода, без его тела, и таким образом объявить, что данный метод будет существовать в этом классе.

***Реализацию*** этого метода, то есть его тело, можно описать позже.

# Ключевое слово `abstract`

```
// Базовая арифметическая операция
abstract class Operation
{public abstract int calculate(int a, int b);}
class Addition extends Operation // Сложение
{public int calculate(int a, int b)
{return a+b;    }}
class Subtraction extends Operation // Вычитание
{public int calculate(int a, int b)
{ return a-b;    }}
class Test
{   public static void main(String s[])
{       Operation o1 = new Addition();
        Operation o2 = new Subtraction();
        o1.calculate(2, 3);
        o2.calculate(3, 5);    }}
```

# Ключевое слово `abstract`

- Поскольку абстрактный метод не имеет тела, после описания его заголовка ставится точка с запятой.
- А раз у него нет тела, то к нему нельзя обращаться, пока его наследники не опишут *реализацию*.
- Это означает, что нельзя создавать экземпляры класса, у которого есть абстрактные методы. Такой класс сам объявляется абстрактным.
- Разработчик может указать ключевое слово `abstract` в списке модификаторов класса, если хочет запретить создание экземпляров этого класса.
- Класс не может быть одновременно `abstract` и `final`
- Сам класс может без ограничений пользоваться своими абстрактными методами

# Ключевое слово abstract

```
abstract class Test
{
    public abstract int getX();
    public abstract int getY();
    public double getLength()
    {
        return Math.sqrt(getX()*getX()+getY()*getY());
    }
}
```

Это корректно, поскольку метод `getLength()` может быть вызван только у объекта. Объект может быть порожден только от неабстрактного класса, который является наследником от `Test`, и должен был реализовать все абстрактные методы.

# Интерфейсы

- В Java класс может иметь только одного родителя, поскольку при множественном наследовании могут возникать конфликты, которые запутывают объектную модель.
- Даже если унаследовано несколько одинаковых абстрактных методов, все равно у них нет *реализации* и можно один раз описать тело метода, которое будет использоваться при вызове любого из этих методов.
- Именно так устроены *интерфейсы* в Java. От них нельзя порождать объекты, но другие классы могут реализовывать их.

# Объявление интерфейсов

- *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.
- Родительских типов может быть много, главное, чтобы не было повторений и чтобы отношение наследования не образовывало циклической зависимости.
- Если есть два *интерфейса*, А и В, причем В наследуется от А, то новый *интерфейс* С может наследоваться от них обоих.

# Объявление интерфейсов

```
public interface Drawble extends Colorable, Resizable  
{ //тело интерфейса}
```

Тело *интерфейса* состоит из объявления элементов, то есть полей-констант и абстрактных методов. Все поля *интерфейса* должны быть `public final static` и их 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();}
```

# Реализация интерфейса

- Каждый класс может реализовывать любые доступные *интерфейсы*
- 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 источников наследуются методы с одинаковой сигнатурой, то достаточно один раз описать *реализацию* и она будет применяться для всех этих методов



# Реализация интерфейса

```
interface A
{   int getValue(); }
interface B
{   double getValue(); }
class C implements A, B {...} // ошибка компиляции
```

В классе оказывается два разных метода с одинаковой сигнатурой, что является неразрешимым конфликтом. Это единственное ограничение на набор *интерфейсов*, которые может реализовывать класс.

# Реализация интерфейса

Конфликт с полями-константами не столь критичен

```
interface A {  
    int value=3;  
}  
  
interface B {  
    double value=5.4;  
}  
  
class C implements A, B {  
    public static void main(String s[]) {  
        C c = new C();  
        // System.out.println(c.value); - ошибка!  
        System.out.println(((A)c).value);  
        System.out.println(((B)c).value);  
    }  
}
```

# Применение интерфейсов

```
public interface InsectConsumer
{   void consumeInsect(Insect i);}
// росянка расширяет класс растение
public class Sundew extends
    Plant implements InsectConsumer {
    public void consumeInsect(Insect i) {...}}
// ласточка расширяет класс птица
public class Swallow extends
    Bird implements InsectConsumer {
    public void consumeInsect(Insect i) {...}}
// муравьед расширяет класс млекопитающее
public class AntEater extends
    Mammal implements InsectConsumer {
    public void consumeInsect(Insect i) {...}}
```

# Применение интерфейсов

// служащий, отвечающий за кормление, расширяет класс  
служащий

```
class FeedWorker extends Worker {  
    // с помощью этого метода можно накормить  
    // и росянку, и ласточку, и муравьеда  
    public void feedOnInsects(InsectConsumer consumer)  
    {  
        ...  
        consumer.consumeInsect(insect);  
        ...  
    }  
}
```

В результате удалось свести работу с одним свойством трех разнородных классов в одно место, сделать код более универсальным.

# Полиморфизм

- При объявлении одноименных полей или методов с совпадающими сигнатурами происходит перекрытие элементов из родительского и наследующего класса.
- Наследники могут объявлять поля с именами, совпадающими с родительскими полями. Такие *объявления* называют *скрывающими*.
- При этом объекты будут содержать оба значения, а компилятор будет каждый раз определять, с каким из них надо работать.
- Рассмотрим, как функционируют классы и объекты в таких ситуациях.

# Поля

Поля могут быть статическими или динамическими.  
Рассмотрим пример:

```
class Parent
{ int a=2; }
class Child extends Parent
{ int a=3;}
Child c = new Child();
System.out.println(c.a);
Parent p = c;
System.out.println(p.a);
```

3

2

Объявление поля в классе-наследнике "скрыло" родительское поле.

# Поля

Явное обращение к родительскому полю:

```
class Parent
{   int a=2; }
class Child extends Parent
{
    int a=3;                // скрывающее объявление
    int b=((Parent)this).a; // более громоздкое объявление
    int c=super.a;          // более простое
}
```

- Переменные b и c получают значение, хранящееся в родительском поле a
- Выражение с **super** более простое, оно не позволит обратиться на два уровня вверх по дереву наследования
- Можно обратиться явным приведением, как это делается для b

# Поля

```
class Parent {  
    int x=0;  
    public void printX() {  
        System.out.println(x);  
    }  
}  
class Child extends Parent {  
    int x=-1;  
}  
new Child().printX();
```

Вызывается метод, определённый в классе Parent и компилятор расценивает обращение к полю **x** в этом методе именно как к полю класса Parent. Поэтому результатом будет **0**.



# Поля

```
class Parent {  
    static int a=2;  
}  
class Child extends Parent {  
    static int a=3;  
}  
Child c = new Child();  
System.out.println(c.a);  
Parent p = c;  
System.out.println(p.a);
```

**Рассматриваемый пример эквивалентен:**

```
System.out.println(Child.a);    // 3  
System.out.println(Parent.a);  // 2
```

**Компилятор всегда может определить, через ссылку какого типа происходит обращение к нему.**

# Поля

```
class Parent {  
    static int a;  
}
```

```
class Child extends Parent {  
}
```

**Каков будет результат следующих строк?**

```
Child.a=10;
```

```
Parent.a=5;
```

```
System.out.println(Child.a);
```

**Несмотря на то, что к полю класса идут обращения через разные классы, переменная всего одна.**

# Методы

```
class Parent {  
    public int getValue() {  
        return 0;  
    }  
}  
class Child extends Parent {  
    public int getValue() {  
        return 1;  
    }  
}  
Child c = new Child();  
System.out.println(c.getValue());  
Parent p = c;  
System.out.println(p.getValue());
```

1

1

# Методы

- Ключевая особенность полиморфизма – наследники могут изменять родительское поведение, даже если обращение к ним производится по ссылке родительского типа.
- Хотя старый метод снаружи уже недоступен, внутри класса-наследника к нему все же можно обратиться с помощью **super**.

# Методы

```
class Parent {  
    public int getValue() {  
        return 0;  
    }  
    public void print() {  
        System.out.println(getValue());  
    }  
}  
  
class Child extends Parent {  
    public int getValue() {  
        return 1;  
    }  
}  
  
Parent p = new Child();  
p.print();
```

# Методы

- С помощью ссылки типа Parent вызывается метод `print()`, объявленный в классе Parent.
- Из этого метода делается обращение к `getValue()`, которое в классе Parent возвращает 0.
- Но компилятор уже не может предсказать, к динамическому методу какого класса произойдет обращение во время работы программы.
- Это определяет виртуальная машина на основе объекта, на который указывает ссылка.
- И раз этот объект порожден от Child, то существует лишь один метод `getValue()`.

# Методы

```
class Parent {  
    protected int getValue() {  
        return 0;  
    }  
}
```

```
class Child extends Parent {  
    /* ??? */ protected int getValue() {  
        return 1;  
    }  
}
```

```
Parent p = new Child();  
p.getValue();
```

# Методы

- Доступ к переопределенному методу не может быть более ограниченным, чем к исходному
- Методы с доступом по умолчанию можно переопределять с таким же доступом, либо `protected` или `public`.
- `Protected`-методы переопределяются такими же, или `public`, а для `public` менять модификатор доступа и вовсе нельзя.
- `Private`-методы определены только внутри класса, снаружи не видны, а потому наследники могут без ограничений объявлять методы с такими же сигнатурами и произвольными возвращаемыми значениями, модификаторами доступа и т.д.



# Методы

```
class Parent {  
    static public int getValue() {  
        return 0;}}  
class Child extends Parent {  
    static public int getValue() {  
        return 1;}}  
Child c = new Child();  
System.out.println(c.getValue());  
Parent p = c;  
System.out.println(p.getValue());  
// аналогично  
System.out.println(Child.getValue()); // 1  
System.out.print(Parent.getValue()); // 0
```

**Статические методы, как и поля, принадлежат классу и появление наследников на них не сказывается!**

# Полиморфизм и объекты

Ссылочный тип обладает следующими элементами:

- непосредственно объявленными в его теле;
- объявленными в его родительском классе и реализуемых интерфейсах, кроме:
  - `private`-элементов;
  - "скрытых" элементов (полей и статических методов, скрытых одноименными элементами);
  - переопределённых (динамических) методов.

Переменная типа абстрактный класс может ссылаться на объекты, порождённые неабстрактным наследником этого класса.

Переменная типа *интерфейс* может ссылаться на объекты, порождённые от класса, реализующего данный *интерфейс*.

# Полиморфизм и объекты

Тип переменной	Допустимые типы ее значения
Абстрактный класс	<ul style="list-style-type: none"><li>• null</li><li>• неабстрактный наследник</li></ul>
Интерфейс	<ul style="list-style-type: none"><li>• null</li><li>• классы, реализующие интерфейс, а именно:</li><li>• реализующие напрямую (заголовок содержит implements);</li><li>• наследуемые от реализующих классов;</li><li>• реализующие наследников этого интерфейса;</li><li>• смешанный случай - наследование от класса, реализующего наследника интерфейса</li></ul>