

## Лекция 7: Объектная модель в Java

### Статические элементы

До этого момента под полями объекта мы всегда понимали значения, которые имеют смысл только в контексте некоторого экземпляра класса. Например:

```
class Human {  
    private String name;  
}
```

Прежде, чем обратиться к полю `name`, необходимо получить ссылку на экземпляр класса `Human`, невозможно узнать имя вообще, оно всегда принадлежит какому-то конкретному человеку.

Но бывают данные и иного характера. Предположим, необходимо хранить количество всех людей (экземпляров класса `Human`, существующих в системе). Понятно, что общее число людей не является характеристикой какого-то одного человека, оно относится ко всему типу в целом. Отсюда появляется название "поле класса", в отличие от "поля объекта". Объявляются такие поля с помощью модификатора `static`:

```
class Human {  
    public static int totalCount;  
}
```

Чтобы обратиться к такому полю, ссылка на объект не требуется, вполне достаточно имени класса:

```
Human.totalCount++;  
    // рождение еще одного человека
```

Для удобства разрешено обращаться к статическим полям и через ссылки:

```
Human h = new Human();  
h.totalCount=100;
```

Однако такое обращение конвертируется компилятором. Он использует тип ссылки, в данном случае переменная `h` объявлена как `Human`, поэтому последняя строка будет неявно преобразована в:

```
Human.totalCount=100;
```

В этом можно убедиться на следующем примере:

```
Human h = null;  
h.totalCount+=10;
```

Значение ссылки равно `null`, но это не имеет значения в силу описанной конвертации. Данный код успешно скомпилируется и корректно исполнится. Таким образом, в следующем примере

```
Human h1 = new Human(), h2 = new Human();  
Human.totalCount=5;
```

```
h1.totalCount++;  
System.out.println(h2.totalCount);
```

все обращения к переменной `totalCount` приводят к одному единственному полю, и результатом работы такой программы будет 6. Это поле будет существовать в единственном экземпляре независимо от того, сколько объектов было порождено от данного класса, и были ли вообще создан хоть один объект.

Аналогично объявляются статические методы.

```
class Human {  
    private static int totalCount;  
  
    public static int getTotalCount() {  
        return totalCount;  
    }  
}
```

Для вызова статического метода ссылки на объект не требуется.

```
Human.getTotalCount();
```

Хотя для удобства обращения через ссылку разрешены, но принимается во внимание только тип ссылки:

```
Human h=null;  
h.getTotalCount();           // два эквивалентных  
Human.getTotalCount();       // обращения к одному  
                               // и тому же методу
```

Хотя приведенный пример технически корректен, все же использование ссылки на объект для обращения к статическим полям и методам не рекомендуется, поскольку это усложняет код.

Обращение к статическому полю является корректным независимо от того, были ли порождены объекты от этого класса и в каком количестве. Например, стартовый метод `main()` запускается до того, как программа создаст хотя бы один объект.

Кроме полей и методов, статическими могут быть инициализаторы. Они также называются инициализаторами класса, в отличие от инициализаторов объекта, рассматривавшихся ранее. Их код выполняется один раз во время загрузки класса в память виртуальной машины. Их запись начинается с модификатора `static`:

```
class Human {  
    static {  
        System.out.println("Class loaded");  
    }  
}
```

Если объявление статического поля совмещается с его инициализацией, то поле инициализируется также однократно при загрузке класса. На объявление и применение статических полей накладываются те же ограничения, что и для динамических,— нельзя

использовать поле в инициализаторах других полей или в инициализаторах класса до того, как это поле объявлено:

```
class Test {
    static int a;
    static {
        a=5;
        // b=7;    // Нельзя использовать до
                  // объявления!
    }
    static int b=a;
}
```

Это правило распространяется только на обращения к полям по простому имени. Если использовать составное имя, то обращаться к полю можно будет раньше (выше в тексте программы), чем оно будет объявлено:

```
class Test {
    static int b=Test.a;
    static int a=3;
    static {
        System.out.println("a="+a+", b="+b);
    }
}
```

Если класс будет загружен в систему, на консоли появится текст:

```
a=3, b=0
```

Видно, что поле `b` при инициализации получило значение по умолчанию поля `a`, т.е. 0. Затем полю `a` было присвоено значение 3.

Статические поля также могут быть объявлены как `final`, это означает, что они должны быть проинициализированы строго один раз и затем уже больше не менять своего значения. Аналогично, статические методы могут быть объявлены как `final`, а это означает, что их нельзя перекрывать в классах-наследниках.

Для инициализации статических полей можно пользоваться статическими методами и нельзя обращаться к динамическим. Вводят специальные понятия – статический и динамический контексты. К статическому контексту относят статические методы, статические инициализаторы, инициализаторы статических полей. Все остальные части кода имеют динамический контекст.

При выполнении кода в динамическом контексте всегда есть объект, с которым идет работа в данный момент. Например, для динамического метода это объект, у которого он был вызван, и так далее.

Напротив, со статическим контекстом ассоциированных объектов нет. Например, как уже указывалось, стартовый метод `main()` вызывается в тот момент, когда ни один объект еще не создан. При обращении к статическому методу, например, `MyClass.staticMethod()`, также может не быть ни одного экземпляра `MyClass`. Обращаться к статическим методам класса `Math` можно, а создавать его экземпляры нельзя.

А раз нет ассоциированных объектов, то и пользоваться динамическими конструкциями нельзя. Можно только ссылаться на статические поля и вызывать статические методы. Либо обращаться к объектам через ссылки на них, полученные в результате вызова конструктора или в качестве аргумента метода и т.п.

```
class Test {
    public void process() {
    }
    public static void main(String s[]) {
        // process(); - ошибка!
        // у какого объекта его вызывать?

        Test test = new Test();
        test.process(); // так правильно
    }
}
```

### Ключевые слова this и super

Эти ключевые слова уже упоминались, рассматривались и некоторые случаи их применения. Здесь они будут описаны более подробно.

Если выполнение кода происходит в динамическом контексте, то должен быть объект, ассоциированный с ним. В этом случае ключевое слово this возвращает ссылку на данный объект:

```
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
```

То есть внутри методов слово this возвращает ссылку на объект, у которого этот метод вызван. Оно необходимо, если нужно передать аргумент, равный ссылке на данный объект, в какой-нибудь метод.

```
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");
}
}

```

Результатом будет:

John is registered.

Другое применение this рассматривалось в случае "затемняющих" объявлений:

```

class Human {
    private String name;

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

```

Слово this можно использовать для обращения к полям, которые объявляются ниже:

```

class Test {
    // int b=a; нельзя обращаться к
    // необъявленному полю!
    int b=this.a;
    int a=5;
    {
        System.out.println("a="+a+", b="+b);
    }
    public static void main(String s[]) {
        new Test();
    }
}

```

Результатом работы программы будет:

a=5, b=0

Все происходит так же, как и для статических полей – b получает значение по умолчанию для a, т.е. ноль, а затем a инициализируется значением 5.

Наконец, слово this применяется в конструкторах для явного вызова в первой строке другого конструктора этого же класса. Там же может применяться и слово super, только уже для обращения к конструктору родительского класса.

Другие применения слова `super` также связаны с обращением к родительскому классу объекта. Например, оно может потребоваться в случае *переопределения* (overriding) родительского метода.

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

```
class Parent {
    public int getValue() {
        return 5;
    }
}

class Child extends Parent {
    // Переопределение метода
    public int getValue() {
        return 3;
    }

    public static void main(String s[]) {
        Child c = new Child();

        // пример вызова переопределенного метода
        System.out.println(c.getValue());
    }
}
```

Вызов переопределенного метода использует механизм полиморфизма, который подробно рассматривается в конце этой лекции. Однако ясно, что результатом выполнения примера будет значение 3. Невозможно, используя ссылку типа `Child`, получить из метода `getValue()` значение 5, родительский метод перекрыт и уже недоступен.

Иногда при *переопределении* бывает полезно воспользоваться результатом работы родительского метода. Предположим, он делал сложные вычисления, а переопределенный метод должен вернуть округленный результат этих вычислений. Понятно, что гораздо удобнее обратиться к родительскому методу, чем заново описывать весь алгоритм. Здесь применяется слово `super`. Из класса наследника с его помощью можно обращаться к переопределенным методам родителя:

```
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());
    }
}

```

Результатом работы программы будет значение 6.

Обращаться с помощью ключевого слова `super` к переопределенному методу родителя, т.е. на два уровня наследования вверх, невозможно. 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`

Следующее важное понятие, которое необходимо рассмотреть, – 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.

Рассмотрим пример. Предположим, необходимо создать набор графических элементов, неважно, каких именно. Например, они могут представлять собой геометрические фигуры – круг, квадрат, звезда и т.д.; или элементы пользовательского *интерфейса* – кнопки, поля ввода и т.д. Сейчас это не имеет решающего значения. Кроме того, существует специальный контейнер, который занимается их отрисовкой. Понятно, что внешний вид каждой компоненты уникален, а значит, соответствующий метод (назовем его `paint()`) будет реализован в разных элементах по-разному.

Но в то же время у компонент может быть много общего. Например, любая из них занимает некоторую прямоугольную область контейнера. Сложные контуры фигуры необходимо вписать в прямоугольник, чтобы можно было анализировать перекрытия, проверять, не вылезает ли компонент за границы контейнера, и т.д. Каждая фигура может иметь цвет, которым ее надо рисовать, может быть видимой, или невидимой и т.д. Очевидно, что полезно создать родительский класс для всех компонент и один раз объявить в нем все общие свойства, чтобы каждая компонента лишь наследовала их.

Но как поступить с методом отрисовки? Ведь родительский класс не представляет собой какую-либо фигуру, у него нет визуального представления. Можно объявить метод `paint()` в каждой компоненте независимо. Но тогда контейнер должен будет обладать сложной функциональностью, чтобы анализировать, какая именно компонента сейчас обрабатывается, выполнять приведение типа и только после этого вызывать нужный метод.

Именно здесь удобно объявить абстрактный метод в родительском классе. У него нет внешнего вида, но известно, что он есть у каждого наследника. Поэтому заголовок метода описывается в родительском классе, тело метода у каждого наследника свое, а контейнер может спокойно пользоваться только базовым типом, не делая никаких приведений.

Приведем упрощенный пример:

```
// Базовая арифметическая операция
abstract class Operation {
    public abstract int calculate(int a, int b);
}

// Сложение
class Addition extends Operation {
    public int calculate(int a, int b) {
        return a+b;
    }
}

// Вычитание
class Subtraction extends Operation {
    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);
    }
}
```

Видно, что выполнения операций сложения и вычитания в методе `main()` записываются одинаково.

Обратите внимание – 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.

Класс может быть абстрактным и в том случае, если у него нет абстрактных методов, но должен быть абстрактным, если такие методы есть. Разработчик может указать ключевое слово `abstract` в списке модификаторов класса, если хочет запретить создание экземпляров этого класса. Классы-наследники должны реализовать (*implements*) все абстрактные методы (если они есть) своего абстрактного родителя, чтобы их можно было объявлять неабстрактными и порождать от них экземпляры.

Конечно, класс не может быть одновременно `abstract` и `final`. Это же верно и для методов. Кроме того, абстрактный метод не может быть `private`, `native`, `static`.

Сам класс может без ограничений пользоваться своими абстрактными методами.

```
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.

По этой же причине можно объявлять переменные типа абстрактный класс. Они могут иметь значение `null` или ссылаться на объект, порожденный от неабстрактного наследника этого класса.

## Интерфейсы

Концепция абстрактных методов позволяет предложить альтернативу множественному наследованию. В Java класс может иметь только одного родителя, поскольку при множественном наследовании могут возникать конфликты, которые запутывают объектную модель. Например, если у класса есть два родителя, которые имеют одинаковый метод с различной *реализацией*, то какой из них унаследует новый класс? И какая будет функциональность родительского класса, который лишился своего метода?

Все эти проблемы не возникают в том случае, если наследуются только абстрактные методы от нескольких родителей. Даже если унаследовано несколько одинаковых 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

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.

Наследование *интерфейсов* действительно очень гибкое. Так, если есть два *интерфейса*, `A` и `B`, причем `B` наследуется от `A`, то новый *интерфейс* `C` может наследоваться от них обоих. Впрочем, понятно, что указание наследования от `A` является избыточным, все элементы этого *интерфейса* и так будут получены по наследству через *интерфейс* `B`.

Затем в фигурных скобках записывается тело *интерфейса*.

```
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();  
}
```

Как мы видим, описание *интерфейса* гораздо проще, чем объявление класса.

## Реализация интерфейса

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

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

Если попытаться объявить класс, реализующий оба эти *интерфейса*, то возникнет 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.

Подобный 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);
    }
}
```

Как видно из примера, обращаться к такому полю через сам класс нельзя, компилятор не сможет понять, какое из двух полей нужно использовать. Но можно с помощью явного приведения сослаться на одно из них.

Итак, если имя *интерфейса* указано после `implements` в объявлении класса, то класс реализует этот *интерфейс*. Наследники данного класса также реализуют *интерфейс*, поскольку им достаются по наследству его элементы.

Если *интерфейс* A наследуется от *интерфейса* B, а класс реализует A, то считается, что *интерфейс* B также реализуется этим классом по той же причине – все элементы передаются по наследству в два этапа – сначала *интерфейсу* A, затем классу.

Наконец, если класс C1 наследуется от класса C2, класс C2 реализует *интерфейс* A1, а *интерфейс* A1 наследуется от *интерфейса* A2, то класс C1 также реализует *интерфейс* A2.

Все это позволяет утверждать, что переменные типа интерфейс также допустимы. Они могут иметь значение `null`, или ссылаться на объекты, порожденные от классов, реализующих этот *интерфейс*. Поскольку объекты порождаются только от классов, а все они наследуются от `Object`, это означает, что значения типа интерфейс обладают всеми элементами класса `Object`.

## Применение интерфейсов

До сих пор *интерфейсы* рассматривались с технической точки зрения – как их объявлять, какие конфликты могут возникать, как их разрешать. Однако важно понимать, как применяются *интерфейсы* с концептуальной точки зрения.

Распространенное мнение, что *интерфейс* – это полностью абстрактный класс, в целом верно, но оно не отражает всех преимуществ, которые дают *интерфейсы* объектной модели. Как уже отмечалось, множественное наследование порождает ряд конфликтов, но отказ от него, хоть и делает язык проще, но не устраняет ситуации, в которых требуются подобные подходы.

Возьмем в качестве примера дерева наследования классификацию живых организмов. Известно, что растения и животные принадлежат к разным царствам. Основным различием между ними является то, что растения поглощают неорганические элементы, а животные питаются органическими веществами. Животные делятся на две большие группы – птицы и

млекопитающие. Предположим, что на основе этой классификации построено дерево наследования, в каждом классе определены элементы с учетом наследования от родительских классов.

Рассмотрим такое свойство живого организма, как способность питаться насекомыми. Очевидно, что это свойство нельзя приписать всей группе птиц, или млекопитающих, а тем более растений. Но существуют представители каждой из названных групп, которые этим свойством обладают, – для растений это росянка, для птиц, например, ласточки, а для млекопитающих – муравьеды. Причем, очевидно, "реализовано" это свойство у каждого вида совсем по-разному.

Можно было бы объявить соответствующий метод (скажем, `consumeInsect(Insect)`) у каждого представителя независимо. Но если задача состоит в моделировании, например, зоопарка, то однотипную процедуру – кормление насекомыми – пришлось бы описывать для каждого вида отдельно, что существенно осложнило бы код, причем без какой-либо пользы.

Java предлагает другое решение. Объявляется *интерфейс* `InsectConsumer`:

```
public interface InsectConsumer {
    void consumeInsect(Insect i);
}
```

Его реализуют все подходящие животные и растения:

```
// 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) {
        ...
    }
}
```

В результате в классе, моделирующем служащего зоопарка, можно объявить соответствующий метод:

```
// 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. Обратите внимание, что при добавлении еще одного насекомоядного такая модель зоопарка не потребует никаких изменений, чтобы обслуживать новый вид, в отличие от первоначального громоздкого решения. Благодаря введению *интерфейса* удалось отделить классы, реализующие его (живые организмы) и использующие его (служащий зоопарка). После любых изменений этих классов при условии сохранения *интерфейса* их взаимодействие не нарушится.

Данный пример иллюстрирует, как *интерфейсы* предоставляют альтернативный, более строгий и гибкий подход вместо множественного наследования.

## Полиморфизм

Ранее были рассмотрены правила объявления классов с учетом их наследования. В этой лекции было введено понятие переопределенного метода. Однако полиморфизм требует более глубокого изучения. При объявлении одноименных полей или методов с совпадающими сигнатурами происходит перекрытие элементов из родительского и наследующего класса. Рассмотрим, как функционируют классы и объекты в таких ситуациях.

### Поля

Начнем с полей, которые могут быть статическими или динамическими. Рассмотрим пример:

```

class Parent {
    int a=2;
}
class Child extends Parent {
    int a=3;
}

```

Прежде всего, нужно сказать, что такое объявление корректно. Наследники могут объявлять поля с любыми именами, даже совпадающими с родительскими. Затем, необходимо понять, как два одноименных поля будут сосуществовать. Действительно, объекты класса `Child` будут содержать сразу две переменных, а поскольку они могут отличаться не только значением, но и типом (ведь это два независимых поля), именно компилятор будет определять, какое из значений использовать. Компилятор может опираться только на тип ссылки, с помощью которой происходит обращение к полю:

```

Child c = new Child();
System.out.println(c.a);
Parent p = c;

```

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

Обе ссылки указывают на один и тот же объект, порожденный от класса Child, но одна из них имеет такой же тип, а другая – Parent. Отсюда следуют и результаты:

```
3
2
```

Объявление поля в классе-наследнике "скрыло" родительское поле. Данное *объявление* так и называется – "*скрывающим*" (hiding). Это особый случай перекрытия областей видимости, отличный от "затеняющего" (shadowing) и "заслоняющего" (obscuring) объявлений. Тем не менее, родительское поле продолжает существовать. К нему можно обратиться и явно:

```
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. Хотя выражение с super более простое, оно не позволит обратиться на два уровня вверх по дереву наследования. А ведь вполне возможно, что в родительском классе это поле также было скрывающим и в родителе родителя хранится еще одно значение. К нему можно обратиться явным приведением, как это делается для b.

Рассмотрим следующий пример:

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

Каков будет результат следующих строк?

```
new Child().printX();
```

Значение какого поля будет распечатано? Метод вызывается с помощью ссылки типа Child, но это не сыграет никакой роли. Вызывается метод, определенный в классе Parent, и компилятор, конечно, расценил обращение к полю x в этом методе именно как к полю класса Parent. Поэтому результатом будет 0.

Перейдем к статическим полям. На самом деле, для них проблем и конфликтов, связанных с полиморфизмом, не существует.

Рассмотрим пример:

```
class Parent {
    static int a=2;
}
class Child extends Parent {
    static int a=3;
}
```

Каков будет результат следующих строк?

```
Child c = new Child();
System.out.println(c.a);
Parent p = c;
System.out.println(p.a);
```

Нужно вспомнить, как компилятор обрабатывает обращения к статическим полям через ссылочные значения. Неважно, на какой объект указывает ссылка. Более того, она может быть даже равна null. Все определяется типом ссылки.

Поэтому рассматриваемый пример эквивалентен:

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

А его результат сомнений уже не вызывает:

```
3
2
```

Можно привести следующее пояснение. Статическое поле принадлежит классу, а не объекту. В результате появления классов-наследников со *скрывающими* (hiding) *объявлениями* никак не сказывается на работе с исходным полем. Компилятор всегда может определить, через ссылку какого типа происходит обращение к нему.

Обратите внимание на следующий пример:

```
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);
```

В этом примере поле a не было скрыто и передалось по наследству классу Child. Однако результат показывает, что это все же одно поле:

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

Итак, 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.

## Методы

Рассмотрим случай *переопределения* (overriding) методов:

```
class Parent {
    public int getValue() {
        return 0;
    }
}
class Child extends Parent {
    public int getValue() {
        return 1;
    }
}
```

И строки, демонстрирующие работу с этими методами:

```
Child c = new Child();
System.out.println(c.getValue());
Parent p = c;
System.out.println(p.getValue());
```

Результатом будет:

```
1
1
```

Можно видеть, что родительский метод полностью перекрыт, значение 0 никак нельзя получить через ссылку, указывающую на объект класса Child. В этом ключевая особенность полиморфизма – наследники могут изменять родительское поведение, даже если обращение к ним производится по ссылке родительского типа. Напомним, что, хотя старый метод снаружи уже недоступен, внутри класса-наследника к нему все же можно обратиться с помощью `super`.

Рассмотрим более сложный пример:

```
class Parent {
    public int getValue() {
        return 0;
    }
    public void print() {
        System.out.println(getValue());
    }
}
```



```
class Child extends Parent {
    public int getValue() {
        return 1;
    }
}
```

Что появится на консоли после выполнения следующих строк?

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

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().

Результатом работы примера будет:

1

Данный пример демонстрирует, что *переопределение* методов должно производиться с осторожностью. Если слишком сильно изменить логику их работы, нарушить принятые соглашения (например, начать возвращать null в качестве значения ссылочного типа, если родительский метод такого не допускал), это может привести к сбоям в работе родительского класса, а значит, объекта наследника. Более того, существуют и некоторые обязательные ограничения.

Вспомним, что заголовок метода состоит из модификаторов, возвращаемого значения, сигнатуры и throws-выражения. Сигнатура (имя и набор аргументов) остается неизменной, если говорить о *переопределении*. Возвращаемое значение также не может меняться, иначе это приведет к появлению двух разных методов с одинаковыми сигнатурами.

Рассмотрим модификаторы доступа.

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

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

Пусть родительский метод был объявлен как protected. Понятно, что метод наследника можно оставить с таким же уровнем доступа, но можно ли его расширить (public), или сузить (доступ по умолчанию)? Несколько строк для проверки:

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

Обращение к методу осуществляется с помощью ссылки типа Parent. Именно компилятор выполняет проверку уровня доступа, и он будет ориентироваться на родительский класс. Но ссылка-то указывает на объект, порожденный от Child, и по правилам полиморфизма исполняться будет метод именно этого класса. А значит, доступ к переопределенному методу не может быть более ограниченным, чем к исходному. Итак, 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.

Что касается private-методов, то они 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.

Аналогичные ограничения накладываются и на throws-выражение, которое будет рассмотрено в следующих лекциях.

Если абстрактный метод переопределяется неабстрактным, то говорят, что он его реализовал (implements). Как ни странно, абстрактный метод может переопределить другой абстрактный, или даже неабстрактный, метод. В первом случае такое действие может иметь смысл только при изменении модификатора доступа (расширении), либо throws-выражения. Во втором случае полностью утрачивается старая *реализация* метода, что может потребоваться в особенных случаях.

Перейдем к статическим методам. Рассмотрим пример:

```
class Parent {  
    static public int getValue() {  
        return 0;  
    }  
}  
  
class Child extends Parent {  
    static public int getValue() {  
        return 1;  
    }  
}
```

И строки, демонстрирующие работу с этими методами:

```
Child c = new Child();  
System.out.println(c.getValue());  
Parent p = c;  
System.out.println(p.getValue());
```

Аналогично случаю со статическими переменными, вспоминаем алгоритм обработки компилятором таких обращений к статическим элементам и получаем, что код эквивалентен следующим строкам:

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

```
System.out.println(Parent.getValue());
```

Результатом будет:

```
1
0
```

То есть статические методы, подобно статическим полям, принадлежат классу и появление наследников на них не сказывается.

Статические методы не могут перекрывать обычные, и наоборот.

## Полиморфизм и объекты

В заключение рассмотрим несколько особенностей, вытекающих из свойств полиморфизма.

Во-первых, теперь можно точно сформулировать, что является элементами ссылочного типа. 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);
  - переопределенных (динамических) методов.

Во-вторых, продолжим рассматривать взаимосвязь типа переменной и типов ее возможных значений. К случаям, описанным в предыдущей лекции, добавляются еще два. Переменная типа абстрактный класс может ссылаться на объекты, порожденные неабстрактным наследником этого класса. Переменная типа *интерфейс* может ссылаться на объекты, порожденные от класса, реализующего данный *интерфейс*.

Сведем эти данные в таблицу.

**Таблица 7.1. Взаимосвязь типа переменной и типов ее возможных значений.**

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

Таким образом, Java предоставляет гибкую и мощную модель объектов, позволяющую проектировать самые сложные системы. Необходимо хорошо разбираться в ее основных

свойствах и механизмах – наследование, статические элементы, абстрактные элементы, *интерфейсы*, полиморфизм, разграничения доступа и другие. Все они позволяют избегать дублирующего кода, облегчают развитие системы, добавление новых возможностей и изменение старых, помогают обеспечивать минимальную связность между частями системы, то есть повышают модульность. Также удачные технические решения можно многократно использовать в различных системах, сокращая и упрощая процесс их создания.

Для достижения таких важных целей требуется не только знание Java, но и владение объектно-ориентированным подходом, основными способами проектирования систем и проверки качества архитектурных решений. Платформа Java является основой и весьма удобным инструментом для применения всех этих технологий.

## Заключение

В этой лекции были рассмотрены особенности объектной модели Java. Это, во-первых, статические элементы, позволяющие использовать *интерфейс* класса без создания объектов. Нужно помнить, что, хотя для обращения к статическим элементам можно задействовать ссылочную переменную, на самом деле ее значение не используется, компилятор основывается только на ее типе.

Для правильной работы со статическими элементами вводятся понятия статического и динамического контекста.

Далее рассматривалось использование ключевых слов `this` и `super`. Выражение `this` предоставляет ссылку, указывающую на объект, в контексте которого оно встречается. Эта конструкция помогает избегать конфликтов имен, а также применяется в конструкторах.

Слово `super` позволяет задействовать свойства родительского класса, что необходимо для *реализации* переопределенных методов, а также в конструкторах.

Затем было введено понятие абстрактного метода и класса. Абстрактный метод не имеет тела, он лишь указывает, что метод с такой сигнатурой должен быть реализован в классе-наследнике. Поскольку он не имеет собственной *реализации*, классы с абстрактными методами также должны быть объявлены с модификатором `abstract`, который указывает, что от них нельзя порождать объекты. Основная цель абстрактных методов – описать в родительском классе как можно больше общих свойств наследников, пусть даже и в виде заголовков методов без *реализации*.

Следующее важное понятие – особый тип в Java, *интерфейс*. Его еще называют полностью абстрактным классом, так как все его методы обязательно абстрактные, а поля `final static`. Соответственно, на основе *интерфейсов* невозможно создавать объекты.

*Интерфейсы* являются альтернативой множественному наследованию. Классы не могут иметь более одного родителя, но они могут реализовывать сколько угодно *интерфейсов*. Таким образом, *интерфейсы* описывают общие свойства классов, не находящихся на одной ветви дерева наследования.

Наконец, важным свойством объектной модели является полиморфизм. Было подробно изучено поведение полей и методов, как статических, так и динамических, при *переопределении*. Что позволило перейти к вопросу соответствия типов переменной и ее значения.