Программирование на Java

Лекция 8. Объектная модель в Java

20 января 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

Аннотация

Эта лекция является некоторым отступлением от рассмотрения технических особенностей Java и посвящена в основном изучению ключевых свойств объектной модели Java, таких как статические элементы, абстрактные методы и классы, интерфейсы, заменяющие множественное наследование. Без этих мощных конструкций, язык Java был бы не способен решать серьезные задачи.

В заключение рассматриваются принципы работы полиморфизма для полей и методов, статических и динамических. Уточняется классификация типов переменных и типов значений, которые они могут хранить.

Оглавление

Лекция 8. Объектная модель в Java

Содержание лекции.

1. Введение

В предыдущих главах были рассмотрены основы объявления классов, а затем взаимоотношения классов, связанных механизмом наследования. В этой главе будет продолжено изложение особенностей объектной модели Java, в том числе и альтернативный множественному наследованию подход - интерфейсы.

2. Статические элементы

До этого момента под полями объекта 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:

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

Для удобства позволяется 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 убедится, изучив следующий пример:

```
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 этого класса, и был 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 не требуется ссылки на объект.

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

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

```
Human h=null;
h.getTotalCount(); // Два эквивалентных обращения к
Human.getTotalCount(); // 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 запутывает 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, рассматриваемых 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 аналогичные 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, что 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, что 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 ни одного экземпляра MyClass может не быть. 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
   }
}
```

3. Ключевые слова this и super

Эти ключевые слова уже упоминались, были рассмотрены некоторые ситуации их применения. Здесь они будут описаны более полно.

Если выполнение кода происходит в динамическом контексте, то должен быть объект, ассоциированный с ним. В этом случае ключевое слово this возвращает ссылку на этот 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 этот 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.our.println(c.getValue());
   }
}
```

Вызов переопределенного метода использует механизм полиморфизма, который подробно рассматривается в конце этой главы. Однако ясно, что результатом выполнения примера будет значение 3. Невозможно, используя ссылку типа Child, получить из метода getVaue() значение 5, родительский метод перекрыт и больше не доступен.

Часто бывают случаи, когда при переопределении было бы полезно воспользоваться результатом работы родительского метода. Предположим, он делал сложные вычисления, а переопределенный метод должен вернуть округленный результат этих вычислений. Понятно, что гораздо удобнее как-то 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.our.println(c.getValue());
   }
}
```

Результатом работы программы будет значение 6.

Обращаться с помощью ключевого слова super к переопределенному методу родителя 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.

4. Ключевое слово abstract

Следующее важное понятие, которое необходимо рассмотреть - ключевое слово abstract.

Иногда бывает удобным описать только заголовок метода, без его тела, и таким образом объявить, что такой 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 этого, есть специальный контейнер, который занимается их отрисовкой. Понятно, что внешний вид каждой компоненты уникален, не похож на других, а значит, соответствующий метод (назовем его paint()) будет реализован в разных элементах совсем 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 размер контейнера и т.д. Каждая может иметь цвет, которым ее надо рисовать, может быть видимой или не видимой, 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 для анализа того, какая именно компонента сейчас обрабатывается, делать 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 {
   public int calculate(int a, int b) {
      return a+b;
   }
}

// Вычитание
class Subtraction {
   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 совершенно одинаковым образом.

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 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 не абстрактными 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();
   pulic 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.

5. 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 серьезно 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 как будет работать функциональность родительского класса, который лишился своего метода?

Все эти проблемы не возникают в случае, если наследуются только абстрактные методы от нескольких родителей. Даже если будет унаследовано несколько одинаковых методов, все равно у них нет реализации, и можно один раз описать тело метода, которое будет использовано при вызове любого из этих методов.

Именно так устроены интерфейсы в Java. От них нельзя порождать объекты, но другие классы могут реализовывать их.

5.1. Объявление интерфейсов

Объявление интерфейсов очень похоже на упрощенное объявление классов.

Объявление начинается с заголовка. Сначала указываются модификаторы. Интерфейс может быть объявлен как public, и тогда он будет доступен для всеобщего 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 рекомендуется этого не делать, 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 Drawble extends Colorable, Resizable {
}
```

Тело интерфейса состоит из объявления элементов, то есть полей-констант и абстрактных методов.

Все поля интерфейса должны быть public final static, поэтому эти модификаторы указывать необязательно и даже не желательно, чтобы не загромождать код. Поскольку поля объявляются финальными, необходимо их сразу инициализировать.

```
public interface Directions {
   int RIGTH=1;
   int LETFT=2;
   int UP=3;
   int DOWN=4;
}
```

Все методы интерфейса являются public abstract, и эти модификаторы также являются необязательными и нежелательными.

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

Как видно, описание интерфейса гораздо проще, чем объявление класса.

5.2. Реализация интерфейса

Любой 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 них. Однако, если у этих методов 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 неверно, 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 этого 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, а затем и классу.

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

Вышесказанное 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, то это означает, что значения типа интерфейс обладают всеми элементами класса Object.

5.3. 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 выражение, 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.

Рассмотрим пример. 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. Среди животных есть две больших 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 возможное 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 такой группы, 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 независимо, 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 весьма 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.

6. Полиморфизм

Ранее были рассмотрены правила объявления классов с учетом их наследования. В этой главе 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 наследующегося класса. Рассмотрим, как именно функционируют классы и объекты в таких ситуациях.

6.1. Поля

Во-первых, 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 делается 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. Это объявление так и называется - "скрывающим" (hiding). Это особый случай перекрытия областей видимости, отличающийся от "затеняющего" (shadowing) и "заслоняющего" (obscuring) объявлений. Тем не менее, как хорошо видно, 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 получат 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, вообще 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. Не имеет никакого значения, 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 появление классов-наследников с 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:

```
5
```

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 определять в каждом случае, с которым из них надо работать.

6.2. 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 уже не доступен снаружи, 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 будет произведено 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 станет:

```
1
```

Данный пример демонстрирует, что переопределение методов должно производится 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), то 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. Понято, 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 идет с помощью ссылки типа Parent. Именно компилятор делает 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 последующих главах.

Если абстрактный метод переопределяется неабстрактным, то говорят, что он его реализовал (implements). Как ни странно, но 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 очевидно 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.

6.3. 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 интерфейсов, кроме:
  - private-элементов;
  - "скрытых" элементов (полей и статических методов, скрытых одноименными элементами);
  - переопределенных методов (динамических методов).

А во-вторых, продолжим рассмотрение взаимосвязи типа переменной и типов ее возможных значений. К случаям, рассмотренным в прошлой главе, добавляются два особых случая. Переменная типа абстрактный класс может ссылаться на объекты, порожденные неабстрактным наследником этого класса. Переменная типа интерфейс может ссылаться на объекты, порожденные от класса, реализующего этот интерфейс.

Сведем эти данные в таблицу:

Тип переменной	Допустимые типы ее значения
Абстрактный класс	• null • неабстрактный наследник
Интерфейс	• null • классы, реализующие интерфейс, а именно: - напрямую реализующие (заголовок содержит implements); - наследующиеся от реализующих классов; - реализующие наследников этого интерфейса; - смешанный случай - наследование от класса, реализующего наследника интерфейса

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.

7. Заключение

В этой главе 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 использовать 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. Оно помогает избегать конфликтов имен, а также применяется в конструкторах.

Слово super предоставляет возможность использовать 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, пускай 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. Были подробно изучены детали поведения полей и методов, как статических, так и динамических, при переопределении. После этого рассмотрения становится возможным развить вопрос соответствия типов переменной и ее значения.

8. Контрольные вопросы

8-1. Предположим, вы моделируете автомобиль, описывая его свойства в формате Java-класса. Какие из следующих полей нужно объявить динамическими, а какие – статическими?

- количество колес автомобиля;
- необходимое количество колес, полагающееся по проектной документации;
- максимально допустимая масса для этого класса автомобилей;
- максимально большое количество пассажиров, когда-либо одновременно перевозимых автомобилем;
- дата начала выпуска автомобилей;
- дата выпуска автомобиля.

a.) 1, 4, 6 – динамические, поскольку описывают свойства конкретного автомобиля

2, 3, 5 – статические, поскольку описывают свойства, присущие всем автомобилям этого класса.

8-2. Корректно ли следующее обращение к переменной x?

```
public class Test {
   static void perform() {
      ...
   }
   private Test x;

   public static void main(String s[]) {
      x.perform(); // корректно ли это выражение?
   }
}
```

a.) Нет, не корректно. Хотя при обращении к статическим элементам через имя переменной, используется лишь ее тип, а не значение, в данном примере производится попытка обратиться к динамической переменной из статического метода, чего делать нельзя, несмотря на то, что для вычисления выражения требуется лишь тип переменной.

8-3. Что окажется на консоли после выполнения следующей программы?

```
public class Parent {
   int x=2;

}

public class Child extends Parent {
   int x=3;

   void print(int x) {
      System.out.println(x);
      System.out.println(this.x);
      System.out.println(super.x);
   }

   public static void main(String s[]) {
      new Child().print(0);
   }
}
```

a.) Результатом будет:

```
0
3
2
```

В первом случае распечатывается значение аргумента метода, который передается из метода main, то есть 0. Во втором случае распечатывается значение переменной, объявленной в классе Child, то есть 3. Наконец, в

последнем случае распечатывается значение переменной, унаследованной от родительского класса Parent, то есть 2.

8-4. Для каких целей может быть использовано ключевое слово this?

a.) Для следующих целей:

- обращение из первой строки конструктора к другому конструктору этого же класса.
- получение ссылки на текущий объект, что может потребоваться в следующих случаях:
  - передача ссылки на сам объект в качестве аргумента вызываемого метода
  - разрешения конфликта имен в случае «затеняющих» объявлений
  - использование для инициализации одного поля другого поля, объявленного ниже
- также это слово применяется при работе с внутренними типами, что выходит за рамки этого курса

8-5. Можно ли при переопределении некоторого абстрактного метода perform() использовать выражение super.perform()?

a.) Нет, выражение super.perform() означает полноценный вызов родительского метода, что невозможно, если у него отсутствует тело, что верно для абстрактных методов.

8-6. Можно ли при наследовании не реализовывать абстрактный метод родительского класса?

a.) Можно, но тогда наследник должен оставаться абстрактным.

8-7. Если есть переменная типа абстрактный класс, можно ли с ее помощью обращаться к абстрактным методам этого класса?

a.) Да, поскольку ее значение, не равное null, будет ссылаться на объект, порожденный от неабстрактного класса-наследника. Следовательно, в нем реализованы все абстрактные методы.

8-8. Какие модификаторы элементов интерфейса подставляются по умолчанию, а потому не рекомендованы для явного указания?

a.) Для полей – public final static.

Для методов – public abstract.

8-9. Возможно ли не реализовывать все методы из интерфейса, указанного в выражении implements?

a.) Да, но такой класс должен быть объявлен абстрактным.

Для методов – public abstract.

8-10. Есть ли какие-либо ограничения на набор интерфейсов, которые может реализовывать класс?

a.) Да, они не могут иметь различных методов с одинаковыми сигнатурами, то есть различающихся типом возвращаемого значения.

8-11. Для каких элементов класса работает полиморфизм?

a.) Только для динамических методов.

8-12. Какое значение появится на консоли после выполнения следующей программы?

```
public class Parent {
   int x = 2;
   public void print() {
      System.out.println(x);
   }
}

public class Child extends Parent {
   int x = 3;

   public static void main(String s[]) {
      new Child().print();
   }
}
```

a.) Появится число 2, так как выводом занимается метод класса Parent, то и переменная будет использована та, что объявлена в этом классе.

8-13. Изменится ли результат программы из предыдущего вопроса, если добавить в объявление класса Child следующие строки?

```
public void print() {
   System.out.println(x);
}
```

a.) Хотя переопределенный метод выглядит точно так же, как и родительский, однако теперь он использует переменную класса Child, поэтому результатом будет 3.

8-14. Корректен ли следующий пример, и если да, то что появится после его выполнения?

```
public class Test {
   public static void test(Test t) {
      System.out.println("test "+t);
   }

   public static void main(String s[]) {
      Test t = null;
      t.test(t);
```

```
        }
    }
```

а.) Пример корректен, значение типа null подходит как для обращения к статическому элементу, так и для передачи в качестве аргумента. Результатом работы метода test станет текст test null.

8-15. Может ли переменная иметь тип абстрактный класс? Интерфейс? Если да, то какие значения она может хранить?

а.) Оба ответа – да, может. В обоих случаях переменные могут принимать значение null. Также переменная типа абстрактный класс может ссылаться на объекты, порожденные от неабстрактных классов-наследников. В случае переменных типа интерфейс, они могут ссылаться на объекты неабстрактных классов, реализующих этот интерфейс.