## Модификаторы доступа

Во многих языках существуют права доступа, которые ограничивают возможность использования, например, переменной в классе. Например, легко представить два крайних вида прав доступа: это public, когда поле доступно из любой точки программы, и private, когда поле может использоваться только внутри того класса, в котором оно объявлено.

Однако прежде, чем переходить к подробному рассмотрению этих и других модификаторов доступа, необходимо внимательно разобраться, зачем они вообще нужны.

### Предназначение модификаторов доступа

Очень часто права доступа расцениваются как некий элемент безопасности кода: мол, необходимо защищать классы от "неправильного" использования. Например, если в классе Human (человек) есть поле age (возраст человека), то какой-нибудь программист намеренно или по незнанию может присвоить этому полю отрицательное значение, после чего объект станет работать неправильно, могут появиться ошибки. Для защиты такого поля age необходимо объявить его private.

Это довольно распространенная точка зрения, однако нужно признать, что она далека от истины. Основным смыслом разграничения прав доступа является обеспечение неотъемлемого свойства объектной модели – инкапсуляции, то есть сокрытия реализации. Исправим пример таким образом, чтобы он корректно отражал предназначение модификаторов доступа. Итак, пусть в классе Human есть поле age целочисленного типа, и чтобы все желающие могли пользоваться этим полем, оно объявляется public.

```
public class Human {
   public int age;
}
```

Проходит время, и если в группу программистов, работающих над системой, входят десятки разработчиков, логично предположить, что все или многие из них начнут использовать это поле.

Может получиться так, что целочисленного типа данных будет уже недостаточно и захочется сменить тип поля на дробный. Однако если просто изменить int на double, вскоре все разработчики, которые пользовались классом Human и его полем age, обнаружат, что в их коде появились ошибки, потому что поле вдруг стало дробным, и в строках, подобных этим:

```
Human h = getHuman();    // получаем ссылку
int i=h.age;    // ошибка!!
```

будет возникать ошибка из-за попытки провести неявным образом сужение примитивного типа.

Получается, что подобное изменение (в общем, небольшое и локальное) потребует модификации многих и многих классов. Поэтому внесение его окажется недопустимым, неоправданным с точки зрения количества усилий, которые необходимо затратить. То есть, объявив один раз поле или метод как public, можно оказаться в ситуации, когда малейшие изменения (имени, типа, характеристик, правил использования) в дальнейшем станут невозможны.

Напротив, если бы поле было объявлено как private, а для чтения и изменения его значения были бы введены дополнительные методы, ситуация поменялась бы в корне:

```
public class Human {
   private int age;
   // метод, возвращающий значение age
   public int getAge() {
      return age;
   }
   // метод, устанавливающий значение age
   public void setAge(int a) {
      age=a;
   }
}
```

В этом случае с данным классом могло бы работать множество программистов и могло быть создано большое количество классов, использующих тип Human, но модификатор private дает гарантию, что никто напрямую этим полем не пользуется и изменение его типа было бы совсем несложной операцией, связанной с изменением только в одном классе.

Получение величины возраста выглядело бы следующим образом:

```
Human h = getHuman();
int i=h.getAge();   // обращение через метод
```

Рассмотрим, как выглядит процесс смены типа поля age:

```
public class Human {

   // поле получает новый тип double
   private /*int*/ double age;

   // старые методы работают с округлением
   // значения
```

```
   public int getAge() {
      return (int)Math.round(age);
   }
   public void setAge(int a) {
      age=a;
   }
   // добавляются новые методы для работы
   // с типом double

   public double getExactAge() {
      return age;
   }
   public void setExactAge(double a) {
   age=a;
   }
}
```

Видно, что старые методы, которые, возможно, уже применяются во многих местах, остались без изменения. Точнее, остался без изменений их внешний формат, а внутренняя реализация усложнилась. Но такая перемена не потребует никаких модификаций остальных классов системы. Пример использования

```
Human h = getHuman();
int i=h.getAge();   // корректно
```

остается верным, переменная i получает корректное целое значение. Однако изменения вводились для того, чтобы можно было работать с дробными величинами. Для этого были добавлены новые методы и во всех местах, где требуется точное значение возраста, необходимо обращаться к ним:

```
Human h = getHuman();
double d=h.getExactAge();
   // точное значение возраста
```

Итак, в класс была добавлена новая возможность, не потребовавшая никаких изменений кода.

За счет чего была достигнута такая гибкость? Необходимо выделить свойства объекта, которые потребуются будущим пользователям этого класса, и сделать их доступными (в данном случае, public ). Те же элементы класса, что содержат детали внутренней реализации логики класса, желательно скрывать, чтобы не образовались нежелательные зависимости, которые могут сдерживать развитие системы.

Этот пример одновременно иллюстрирует и другое теоретическое правило написания объектов, а именно: в большинстве случаев доступ к полям лучше

реализовывать через специальные методы (accessors) для чтения (getters) и записи (setters). То есть само поле рассматривается как деталь внутренней реализации. Действительно, если рассматривать внешний интерфейс объекта как целиком состоящий из допустимых действий, то доступными элементами должны быть только методы, реализующие эти действия. Один из случаев, в котором такой подход приносит необходимую гибкость, уже рассмотрен.

Есть и другие соображения. Например, вернемся к вопросу о корректном использовании объекта и установке верных значений полей. Как следствие, правильное разграничение доступа позволяет ввести механизмы проверки входных значений:

```
public void setAge(int a) {
   if (a>=0) {
      age=a;
   }
}
```

В этом примере поле age никогда не примет некорректное отрицательное значение. (Недостатком приведенного примера является то, что в случае неправильных входных данных они просто игнорируются, нет никаких сообщений, позволяющих узнать, что изменения поля возраста на самом деле не произошло; для полноценной реализации метода необходимо освоить работу с ошибками в Java.)

Бывают и более существенные изменения логики класса. Например, данные можно начать хранить не в полях класса, а в более надежном хранилище, например, файловой системе или базе данных. В этом случае методы - аксессоры опять изменят свою реализацию и начнут обращаться к persistent storage (постоянное хранилище, например, БД) для чтения/записи значений. Если доступа к полям класса не было, а открытыми были только методы для работы с их значениями, то можно изменить код этих методов, а наружные типы, которые использовали данный класс, совершенно не изменятся, логика их работы останется той же.

Подведем итоги. Функциональность класса необходимо разделять на открытый интерфейс, описывающий действия, которые будут использовать внешние типы, и на внутреннюю реализацию, которая применяется только внутри самого класса. Внешний интерфейс в дальнейшем модифицировать невозможно, или очень сложно, для больших систем, поэтому его требуется продумывать особенно тщательно. Детали внутренней реализации могут быть изменены на любом этапе, если они не меняют логику работы всего класса. Благодаря такому подходу реализуется одна из базовых характеристик объектной модели — инкапсуляция, и обеспечивается важное преимущество технологии ООП — модульность.

Таким образом, модификаторы доступа вводятся не для защиты типа от внешнего пользователя, а, напротив, для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации. Что же касается неправильного применения класса, то его создателям нужно стремиться к тому, чтобы класс был прост в применении, тогда таких проблем не возникнет, ведь программист не станет намеренно писать код, который порождает ошибки в его программе.

Конечно, такое разбиение на внешний интерфейс и внутреннюю реализацию не всегда очевидно, часто условно. Для облегчения задачи технических дизайнеров классов в Java введено не два ( public и private ), а четыре уровня доступа. Рассмотрим их и весь механизм разграничения доступа в Java более подробно.

**Разграничение доступа в Java**

Уровень доступа элемента языка является статическим свойством, задается на уровне кода и всегда проверяется во время компиляции. Попытка обратиться к закрытому элементу напрямую вызовет ошибку.

В Java модификаторы доступа указываются для:

- типов (классов и интерфейсов) объявления верхнего уровня;
- элементов ссылочных типов (полей, методов, внутренних типов);
- конструкторов классов.

Как следствие, массив также может быть недоступен в том случае, если недоступен тип, на основе которого он объявлен.

Все четыре уровня доступа имеют только элементы типов и конструкторы. Это:

- public;
- private;
- protected;
- если не указан ни один из этих трех типов, то уровень доступа определяется по умолчанию (default).

Первые два из них уже были рассмотрены. Последний уровень (доступ по умолчанию) – он допускает обращения из того же пакета, где объявлен и сам этот класс. По этой причине пакеты в Java являются не просто набором типов, а более структурированной единицей, так как типы внутри одного пакета могут больше взаимодействовать друг с другом, чем с типами из других пакетов.

Наконец, protected дает доступ наследникам класса. Понятно, что наследникам может потребоваться доступ к некоторым элементам родителя, с которыми не приходится иметь дело внешним классам.

Однако описанная структура не позволяет упорядочить модификаторы доступа так, чтобы каждый следующий строго расширял предыдущий. Модификатор protected может быть указан для наследника из другого пакета, а доступ по умолчанию допускает обращения из классов-ненаследников, если они находятся в том же пакете. По этой причине возможности protected были расширены таким образом, что он включает в себя доступ внутри пакета. Итак, модификаторы доступа упорядочиваются следующим образом (от менее открытых – к более открытым):

```
private
(none) default
protected
public
```

Эта последовательность будет использована далее при изучении деталей наследования классов.

Теперь рассмотрим, какие модификаторы доступа возможны для различных элементов языка.

- Пакеты доступны всегда, поэтому у них нет модификаторов доступа (можно сказать, что все они public, то есть любой существующий в системе пакет может использоваться из любой точки программы).
- Типы (классы и интерфейсы) верхнего уровня объявления. При их объявлении существует всего две возможности: указать модификатор public или не указывать его. Если доступ к типу является public, то это означает, что он доступен из любой точки кода. Если же он не public, то уровень доступа назначается по умолчанию: тип доступен только внутри того пакета, где он объявлен.
- Массив имеет тот же уровень доступа, что и тип, на основе которого он объявлен (естественно, все примитивные типы являются полностью доступными).
- Элементы и конструкторы объектных типов. Обладают всеми четырьмя возможными значениями уровня доступа. Все элементы интерфейсов являются public.

Для типов объявления верхнего уровня нет необходимости во всех четырех уровнях доступа. Private -типы образовывали бы закрытую мини-программу, никто не мог бы их использовать. Типы, доступные только для наследников, также не были признаны полезными.

Разграничения доступа сказываются не только на обращении к элементам объектных типов или пакетов (через составное имя или прямое обращение), но также при вызове конструкторов, наследовании, приведении типов. Импортировать недоступные типы запрещается.

Проверка уровня доступа проводится компилятором. Обратите внимание на следующие примеры:

```
public class Wheel {
   private double radius;
   public double getRadius() {
      return radius;
   }
}
```

Значение поля radius недоступно снаружи класса, однако открытый метод getRadius() корректно возвращает его.

Рассмотрим следующие два модуля компиляции:

```
package first;

// Некоторый класс Parent
public class Parent {
}
```

```
package first;

// Класс Child наследуется от класса Parent,
// но имеет ограничение доступа по умолчанию
class Child extends Parent {
}

public class Provider {
   public Parent getValue() {
      return new Child();
   }
}
```

К методу getValue() класса Provider можно обратиться и из другого пакета, не только из пакета first, поскольку метод объявлен как public. Данный метод возвращает экземпляр класса Child, который недоступен из других пакетов. Однако следующий вызов является корректным:

```
package second;

import first.*;
```

```
public class Test {
  public static void main(String s[])
  {
   Provider pr = new Provider();
   Parent p = pr.getValue();
   System.out.println(p.getClass().getName());
   // (Child)p - приведет к ошибке компиляции!
  }
}
```

Результатом будет:

```
first.Child
```

То есть на самом деле в классе Test работа идет с экземпляром недоступного класса Child, что возможно, поскольку обращение к нему делается через открытый класс Parent. Попытка же выполнить явное приведение вызовет ошибку. Да, тип объекта "угадан" верно, но доступ к закрытому типу всегда запрещен.

Следующий пример:

```
public class Point {
  private int x, y;

  public boolean equals(Object o) {
    if (o instanceof Point) {
      Point p = (Point)o;
      return p.x==x && p.y==y;
    }
    return false;
  }
}
```

В этом примере объявляется класс Point с двумя полями, описывающими координаты точки. Обратите внимание, что поля полностью закрыты – private. Далее попытаемся переопределить стандартныйметод equals() таким образом, чтобы для аргументов, являющихся экземплярами класса Point, или его наследников (логика работы оператора instanceof ), в случае равенства координат возвращалось истинное значение. Обратите внимание на строку, где делается сравнение координат,– для этого приходится обращаться к private -полям другого объекта!

Тем не менее, такое действие корректно, поскольку private допускает обращения из любой точки класса, независимо от того, к какому именно объекту оно производится.

В заключение рассмотрим применение модификаторов доступа для конструкторов. Может вызвать удивление возможность объявлять конструкторы как private. Ведь они нужны для генерации объектов, а к таким конструкторам ни у кого не будет доступа. Однако в ряде случаев модификатор private может быть полезен. Например:

- private -конструктор может содержать инициализирующие действия, а остальные конструкторы будут использовать его с помощью this, причем прямое обращение к этому конструктору по каким-то причинам нежелательно;
- запрет на создание объектов этого класса, например, невозможно создать экземпляр класса Math ;
- реализация специального шаблона проектирования из ООП Singleton, для работы которого требуется контролировать создание объектов, что невозможно в случае наличия не- private конструкторов.

## Дополнительные свойства классов

### 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 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 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. 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
   }
}
```

**Метод main**

Итак, виртуальная машина реализуется приложением операционной системы и запускается по обычным правилам. Программа, написанная на Java, является набором классов. Понятно, что требуется некая входная точка, с которой должно начинаться выполнение приложения.

Такой входной точкой, по аналогии с языками C/C++, является метод main(). Пример его объявления:

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

Модификатор static в этой лекции не рассматривался и будет изучен позже. Он позволяет вызвать метод main(), не создавая объектов. Метод не возвращает никакого значения, хотя в C есть возможность указать код возврата из программы. В Java для этой цели существует метод System.exit(), который закрывает виртуальную машину и имеет аргумент типа int.

Аргументом метода main() является массив строк. Он заполняется дополнительными параметрами, которые были указаны при вызове метода.

```
package test.first;

public class Test {
   public static void main(String[] args) {
      for (int i=0; i<args.length; i++) {
         System.out.print(args[i]+" ");
      }
      System.out.println();
   }
}
```

Для вызова программы виртуальной машине передается в качестве параметра имя класса, у которого объявлен метод main(). Поскольку это имя класса, а не имя файла, то не должно указываться никакого расширения ( .class или .java ) и расположение класса записывается через точку (разделитель имен пакетов), а не с помощью файлового разделителя. Компилятору же, напротив, передается имя и путь к файлу.

Если приведенный выше модуль компиляции сохранен в файле Test.java, который лежит в каталоге test\first, то вызов компилятора записывается следующим образом:

```
javac test\first\Test.java
```

А вызов виртуальной машины:

```
java test.first.Test
```

Чтобы проиллюстрировать работу с параметрами, изменим строку запуска приложения:

```
java test.first.Test Hello, World!
```

Результатом работы программы будет:

```
Hello, World!
```

**Параметры методов**

Для лучшего понимания работы с параметрами методов в Java необходимо рассмотреть несколько вопросов.

Как передаются аргументы в методы – по значению или по ссылке? С точки зрения программы вопрос формулируется, например, следующим образом. Пусть есть переменная и она в качестве аргумента передается в некоторый метод. Могут ли произойти какие-либо изменения с этой переменной после завершения работы метода?

```
int x=3;
process(x);
print(x);
```

Предположим, используемый метод объявлен следующим образом:

```
public void process(int x) {
   x=5;
}
```

Какое значение появится на консоли после выполнения примера? Чтобы ответить на этот вопрос, необходимо вспомнить, как переменные разных типов хранят свои значения в Java.

Напомним, что примитивные переменные являются истинными хранилищами своих значений и изменение значения одной переменной никогда не скажется на значении другой. Параметр метода process(), хоть и имеет такое же имя x, на самом деле является полноценным хранилищем целочисленной величины. А потому присвоение ему значения 5 не скажется на внешних переменных. То есть результатом примера будет 3 и аргументы примитивного типа передаются в методы по значению. Единственный способ изменить такую переменную в результате работы метода – возвращать нужные величины из метода и использовать их при присвоении:

```
public int doubler(int x) {
   return x+x;
```

```
}

public void test() {
   int x=3;
   x=doubler(x);
}
```

Перейдем к ссылочным типам.

```
public void process(Point p) {
   p.x=3;
}

public void test() {
   Point p = new Point(1,2);
   process(p);
   print(p.x);
}
```

Ссылочная переменная хранит ссылку на объект, находящийся в памяти виртуальной машины. Поэтому аргумент метода process() будет иметь в качестве значения ту же самую ссылку и, стало быть, ссылаться на тот же самый объект. Изменения состояния объекта, осуществленные с помощью одной ссылки, всегда видны при обращении к этому объекту с помощью другой. Поэтому результатом примера будет значение3. Объектные значения передаются в Java по ссылке. Однако если изменять не состояние объекта, а саму ссылку, то результат будет другим:

```
public void process(Point p) {
   p = new Point(4,5);
}

public void test() {
   Point p = new Point(1,2);
   process(p);
   print(p.x);
}
```

В этом примере аргумент метода process() после присвоения начинает ссылаться на другой объект, нежели исходная переменная p, а значит, результатом примера станет значение 1. Можно сказать, что ссылочные величины передаются по значению, но значением является именно ссылка на объект.

Теперь можно уточнить, что означает возможность объявлять параметры методов и конструкторов как final. Поскольку изменения значений параметров (но не объектов, на которые они ссылаются) никак не

сказываются на переменных вне метода, модификатор final говорит лишь о том, что значение этого параметра не будет меняться на протяжении работы метода. Разумеется, для аргумента final Point p выражение p.x=5 является допустимым (запрещается p=new Point(5, 5)).

**Перегруженные методы**

Перегруженными (overloaded) методами называются методы одного класса с одинаковыми именами. Сигнатуры у них должны быть различными и различие может быть только в наборе аргументов.

Если в классе параметры перегруженных методов заметно различаются: например, у одного метода один параметр, у другого – два, то для Java это совершенно независимые методы и совпадение их имен может служить только для повышения наглядности работы класса. Каждый вызов, в зависимости от количества параметров, однозначно адресуется тому или иному методу.

Однако если количество параметров одинаковое, а типы их различаются незначительно, при вызове может сложиться двойственная ситуация, когда несколько перегруженных методов одинаково хорошо подходят для использования. Например, если объявлены типы Parent и Child, где Child расширяет Parent, то для следующих двух методов:

```
void process(Parent p, Child c) {}
void process(Child c, Parent p) {}
```

можно сказать, что они допустимы, их сигнатуры различаются. Однако при вызове

```
process(new Child(), new Child());
```

обнаруживается, что оба метода одинаково годятся для использования. Другой пример, методы:

```
process(Object o) {}
process(String s) {}
```

и примеры вызовов:

```
process(new Object());
process(new Point(4,5));
process("abc");
```

Очевидно, что для первых двух вызовов подходит только первый метод, и именно он будет вызван. Для последнего же вызова подходят оба перегруженных метода, однако класс String является более "специфичным",

или узким, чем класс Object. Действительно, значения типа String можно передавать в качестве аргументов типа Object, обратное же неверно. Компилятор попытается отыскать наиболее специфичный метод, подходящий для указанных параметров, и вызовет именно его. Поэтому при третьем вызове будет использован второй метод.

Однако для предыдущего примера такой подход не дает однозначного ответа. Оба метода одинаково специфичны для указанного вызова, поэтому возникнет ошибка компиляции. Необходимо, используя явное приведение, указать компилятору, какой метод следует применить:

```
process((Parent)(new Child()), new Child());
// или
process(new Child(),(Parent)(new Child()));
```

Это верно и в случае использования значения null:

```
process((Parent)null, null);
// или
process(null,(Parent)null);
```