**Модификаторы доступа, переопределение методов, реализация абстрактных методов**

[Java Core](https://javarush.ru/quests/QUEST_JAVA_CORE)

[Уровень 5](https://javarush.ru/quests/lectures?quest=QUEST_JAVA_CORE&level=5), Лекция 1

— Я расскажу тебе про «модификаторы доступа». Когда-то я уже рассказывал про них, но повторение – мать учения.

Ты можешь управлять доступом (видимостью) методов и переменных твоего класса из других классов. Модификатор доступа отвечает на вопрос «Кто может обращаться к данному методу/переменной?». Каждому методу или переменной можно указывать только один модификатор.

**1)** Модификатор «**public**».

К переменной, методу или классу, помеченному модификатором **public**, можно обращаться из любого места программы. Это самая высокая степень открытости – никаких ограничений нет.

**2)** Модификатор «**private**».

К переменной, методу или классу, помеченному модификатором **private**, можно обращаться только из того же класса, где он объявлен. Для всех остальных классов помеченный метод или переменная – невидимы. Это самая высокая степень закрытости – только свой класс. Такие методы не наследуются и не переопределяются. Доступ к ним из класса-наследника также невозможен.

**3) «Модификатор** **«по умолчанию».**

Если переменная или метод не помечены никаким модификатором, то считается, что они помечены «модификатором по умолчанию». Переменные и методы с таким модификатором видны всем классам пакета, в котором они объявлены, и только им. Этот модификатор еще называют «**package**» или «**package private**», намекая, что доступ к переменным и методам открыт для всего пакета, в котором находится их класс

**4)** Модификатор «**protected**».

Этот уровень доступа чуть шире, чем **package**. К переменной, методу или классу, помеченному модификатором **protected**, можно обращаться из его же пакета (как package), но еще из всех классов, унаследованных от текущего.

Таблица с пояснением:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип видимости** | **Ключевое слово** | **Доступ** | | | |
| **Свой класс** | **Свой пакет** | **Класс — наследник** | **Все классы** |
| **Закрытый** | **private** | Есть | Нет | Нет | Нет |
| **Пакет** | (нет модификатора) | Есть | Есть | Нет | Нет |
| **Защищенный** | **protected** | Есть | Есть | Есть | Нет |
| **Открытый** | **public** | Есть | Есть | Есть | Есть |

Есть способ, чтобы легко запомнить эту таблицу. Представь себе, что ты составляешь завещание и делишь все вещи на четыре категории. Кто может пользоваться твоими вещами?

|  |  |  |
| --- | --- | --- |
| **Кто имеет доступ** | **Модификатор** | **Пример** |
| Только **я сам** | **private** | Личный дневник |
| Семья | (нет модификатора) | Семейные фотографии |
| Семья и наследники | **protected** | Фамильное поместье |
| Все | **public** | Мемуары |

— Если представить, что классы, лежащие в одном пакете, – это одна семья, то очень даже похоже.

— Хочу также рассказать тебе несколько интересных нюансов насчет переопределения методов.

**1) Неявная реализация абстрактного метода.**

Допустим, у тебя есть код:

Код

class Cat

{

public String getName()

{

return "Васька";

}

}

И ты решил унаследовать от него класс тигр и добавить новому классу интерфейс

Код

class Cat

{

public String getName()

{

return "Васька";

}

}

interface HasName

{

String getName();

int getWeight();

}

class Tiger extends Cat implements HasName

{

public int getWeight()

{

return 115;

}

}

Если ты просто реализуешь все недостающие методы, которые тебе подскажет Intellij IDEA, то можешь потом долго искать ошибку.

Оказывается, что в классе Tiger есть унаследованный от Cat метод getName, который и будет считаться реализацией метода getName для интерфейса HasName.

— Не вижу в этом ничего страшного.

— Это не очень плохо, это скорее потенциальное место для ошибок.

Но может быть еще хуже:

Код

interface HasWeight

{

int getValue();

}

interface HasSize

{

int getValue();

}

class Tiger extends Cat implements HasWeight, HasSize

{

public int getValue()

{

return 115;

}

}

Оказывается, ты не всегда можешь унаследоваться от нескольких интерфейсов. Вернее унаследоваться можешь, а вот корректно их реализовать – нет. Посмотри на пример, оба интерфейса требуют, чтобы ты реализовал метод getValue(), и не ясно, что он должен возвращать: вес(weight) или размер(size). Это довольно-таки неприятная вещь, если тебе придется с ней столкнуться.

— Да, согласен. Хочешь реализовать метод, а не можешь. Вдруг ты уже унаследовал метод с таким же именем от базового класса. Обломись.

— Но есть и приятные новости.

**2) Расширение видимости.** При переопределении типа разрешается расширить видимость метода. Вот как это выглядит:

|  |  |  |
| --- | --- | --- |
| **Код на Java** | | **Описание** |
| class Cat  {  protected String getName()  {  return "Васька";  }  } | |  |
| class Tiger extends Cat  {  public String getName()  {  return "Василий Тигранович";  }  } | | Мы расширили видимость метода с protected до public. |
| **Использование** | **Почему это «законно»** | |
| public static void main(String[] args)  {  Cat cat = new Cat();  cat.getName();  } | Все отлично. Тут мы даже не знаем, что в классе-наследнике видимость метода была расширена. | |
| public static void main(String[] args)  {  Tiger tiger = new Tiger();  tiger.getName();  } | Тут вызывается метод, у которого расширили область видимости.  Если бы этого сделать было нельзя, всегда можно было бы объявить метод в Tiger: **public** String getPublicName() { super.getName(); //вызов protected метода }  Т.е. ни о каком нарушении безопасности и речи нет. | |
| public static void main(String[] args)  {  Cat catTiger = new Tiger();  catTiger.getName();  } | Если все условия подходят для вызова метода базового типа (**Cat**), то они уж точно подойдут для вызова типа наследника (**Tiger**) . Т.к. ограничения на вызов метода были ослаблены, а не усилены. | |

— Не уверен, что понял полностью, но то, что так можно делать, запомню.

**3)** **Сужение типа результата.**

В переопределенном методе мы можем **поменять тип результата**, **сузив** его.

|  |  |  |
| --- | --- | --- |
| **Код на Java** | **Описание** | |
| class Cat  {  public Cat parent;  public Cat getMyParent()  {  return this.parent;  }  public void setMyParent(Cat cat)  {  this.parent = cat;  }  } |  | |
| class Tiger extends Cat  {  public Tiger getMyParent()  {  return (Tiger) this.parent;  }  } | Мы переопределили метод getMyParent, теперь он возвращает объект типа Tiger. | |
| **Использование** | | **Почему это «законно»** |
| public static void main(String[] args)  {  Cat parent = new Cat();  Cat me = new Cat();  me.setMyParent(parent);  Cat myParent = me.getMyParent();  } | | Все отлично. Тут мы даже не знаем, что в классе наследнике тип результата метода getMyParent был сужен.  «Старый код» как работал так и работает. |
| public static void main(String[] args)  {  Tiger parent = new Tiger();  Tiger me = new Tiger();  me.setMyParent(parent);  Tiger myParent = me.getMyParent();  } | | Тут вызывается метод, у которого сузили тип результата.  Если бы этого сделать было нельзя, всегда можно было бы объявить метод в Tiger: **public Tiger** getMyTigerParent() { return (Tiger) this.parent; }  Т.е. ни о каком нарушении безопасности и/или контроля приведения типов нет речи. |
| public static void main(String[] args)  {  Tiger parent = new Tiger();  Cat me = new Tiger();  me.setMyParent(parent);  Cat myParent = me.getMyParent();  } | | И тут все отлично работает, хотя мы расширили тип переменных до базового класса (Cat).  Нет ничего страшного при вызове метода getMyParent, т.к. его результат, хоть и класса Tiger, все равно сможет отлично присвоиться в переменную myParent базового класса (Cat).  Объекты Tiger можно смело хранить как в переменных класса Tiger, так и в переменных класса Cat. |

— Ага. Я понял. Надо **при переопределении методов беспокоиться о том, как все это будет работать, если мы передадим наши объекты в код, который умеет обращаться только с базовым классом, и ничего о нашем классе не знает.**

— Именно! Тогда вопрос на засыпку, почему нельзя расширить тип результата при переопределении метода?

— Это же очевидно, тогда перестанет работать код в базовом классе:

|  |  |
| --- | --- |
| **Код на Java** | **Пояснение проблемы** |
| class Cat  {  public Cat parent;  public Cat getMyParent()  {  return this.parent;  }  public void setMyParent(Cat cat)  {  this.parent = cat;  }  } |  |
| class Tiger extends Cat  {  public Object getMyParent()  {  if (this.parent != null)  return this.parent;  else  return "я - сирота";  }  } | Мы переопределили метод getMyParent и расширили тип его результата.  Тут все отлично. |
| public static void main(String[] args)  {  Tiger parent = new Tiger();  Cat me = new Tiger();  Cat myParent = me.getMyParent();  } | Тогда у нас перестанет работать этот код.  Метод getMyParent может вернуть любой объект типа Object, т.к. на самом деле он вызывается у объекта типа Tiger.  А у нас нет проверки перед присваиванием. Тогда вполне возможно, что переменная myParent типа Cat будет хранить ссылку на строку. |

— Отличный пример, Амиго!

В Java **перед вызовом метода не проверяется, есть ли такой метод у объекта или нет. Все проверки происходят во время выполнения.** И [гипотетический] вызов отсутствующего метода, скорее всего, приведет к тому, что программа начнет выполнять байт-код там, где его нет. Это, в конце концов, приведет к фатальной ошибке, и операционная система принудительно закроет программу.

— Ничего себе. Буду знать.