**Полиморфизм и переопределение**

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

[Уровень 2](https://javarush.ru/quests/lectures?quest=QUEST_JAVA_CORE&level=2), Лекция 1

— Амиго, ты любишь китов?

— Китов? Не, не слышал.

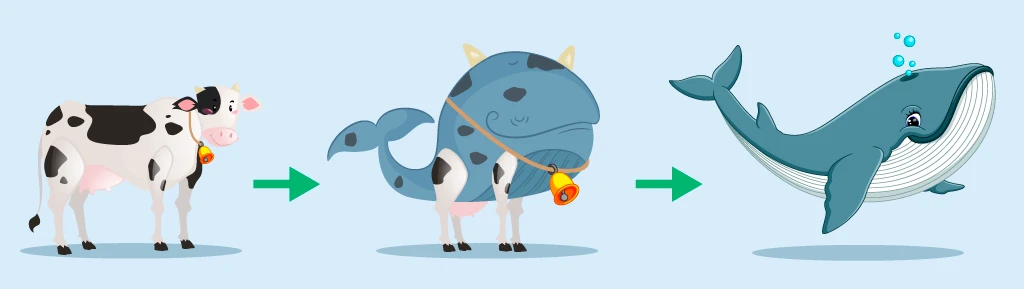
— Этот как корова, только больше и плавает. Кстати, киты произошли от коров. Ну, или имели общего с ними предка. Не столь важно.

[](https://cdn.javarush.ru/images/article/2b5bd5fe-f3b4-4380-907e-4d377084ac19/original.jpeg)

— Так вот. Хочу рассказать тебе об еще одном очень мощном инструменте ООП – это **полиморфизм**. У него есть четыре особенности.

**1) Переопределение метода.**

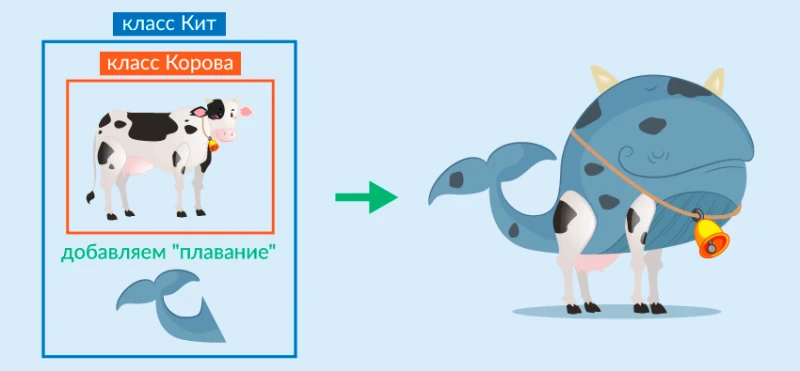
Представь, что ты для игры написал класс «Корова». В нем есть много полей и методов. Объекты этого класса могут делать разные вещи: идти, есть, спать. Еще коровы звонят в колокольчик, когда ходят. Допустим, ты реализовал в классе все до мелочей.

[](https://cdn.javarush.ru/images/article/ff910c8e-eff0-40db-af8b-e67da08b1713/original.jpeg)

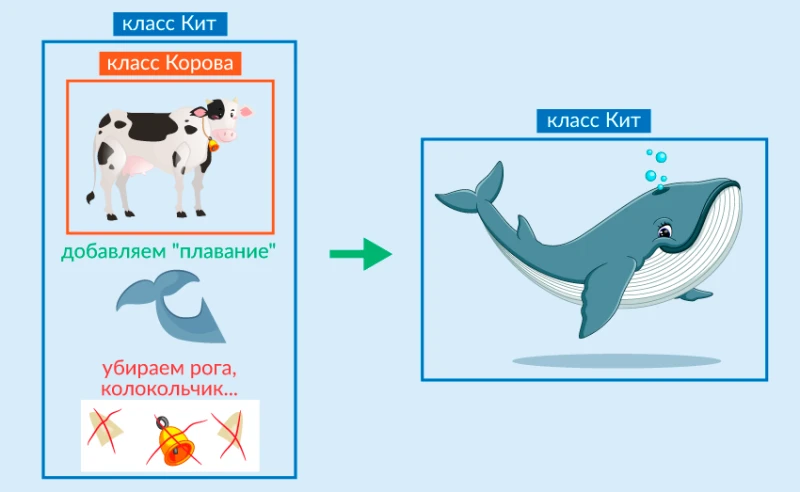
А тут приходит заказчик проекта и говорит, что хочет выпустить новый уровень игры, где все действия происходят в море, а главным героем будет кит.

Ты начал проектировать класс «Кит» и понял, что он лишь немного отличается от класса «Корова». Логика работы обоих классов очень похожа, и ты решил использовать наследование.

Класс «Корова» идеально подходит на роль класса-родителя, там есть все необходимые переменные и методы. Достаточно только добавить киту возможность плавать. Но есть проблема: у твоего кита есть ноги, рога и колокольчик. Ведь эта функциональность реализована внутри класса «Корова». Что тут можно сделать?

[](https://cdn.javarush.ru/images/article/409e89ba-fbca-4e8b-842e-bef26b21b0a0/original.jpeg)

К нам на помощь приходит переопределение (замена) методов. Если мы унаследовали метод, который делает не совсем то, что нужно нам в нашем новом классе, мы можем заменить этот метод на другой.

[](https://cdn.javarush.ru/images/article/7eb5ef82-34a9-49df-bba4-7026ff207dcc/original.jpeg)

Как же это делается? В нашем классе-потомке мы объявляем **такой же метод, как и метод класса родителя, который хотим изменить.** Пишем в нем новый код. И все – как будто старого метода в классе-родителе и не было.

Вот как это работает:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| class Cow {  public void printColor() {  System.out.println("Я - белая");  }  public void printName() {  System.out.println("Я - корова");  }  }  class Whale extends Cow {  public void printName() {  System.out.println("Я - кит");  }  } | Тут определены два класса Cow и Whale. Whale унаследован от Cow.    В классе Whale переопределен метод printName(); |
| public static void main(String[] args) {  Cow cow = new Cow();  cow.printName();  } | Данный код выведет на экран надпись «Я – корова» |
| public static void main(String[] args) {  Whale whale = new Whale();  whale.printName();  } | Данный код выведет на экран «Я – кит» |

После наследования класса Cow и переопределения метода printName, класс Whale фактически содержит такие данные и методы:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| class Whale {  public void printColor() {  System.out.println("Я - белая");  }  public void printName() {  System.out.println("Я - кит");  }  } | Ни о каком старом методе мы и не знаем. |

— Честно говоря, ожидаемо.

**2) Но это еще не все.**

— Предположим в классе Cow есть метод printAll, который вызывает два других метода, тогда код будет работать так:

На экран будет выведена надпись Я – белая Я – кит

|  |  |
| --- | --- |
| **Код** | **Описание** |
| class Cow {  public void printAll() {  printColor();  printName();  }  public void printColor() {  System.out.println("Я - белая");  }  public void printName() {  System.out.println("Я - корова");  }  }  class Whale extends Cow {  public void printName() {  System.out.println("Я - кит");  }  } |  |
| public static void main(String[] args) {  Whale whale = new Whale();  whale.printAll();  } | На экран будет выведена надпись Я – белая Я – кит |

Обрати внимание, когда вызываются метод printAll() написанный в классе Cow, у объекта типа Whale, то будет использован метод printName класса Whale, а не Cow.

Главное, не в каком классе написан метод, а какой тип (класс) объекта, у которого этот метод вызван.

— Ясно.

— Наследовать и переопределять можно только нестатические методы. Статические методы не наследуются и, следовательно, не переопределяются.

Вот как выглядит класс Whale после применения наследования и переопределения методов:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| class Whale {  public void printAll() {  printColor();  printName();  }  public void printColor() {  System.out.println("Я - белая");  }  public void printName() {  System.out.println("Я - кит");  }  } | Вот как выглядит класс Whale, после применения наследования и переопределения метода. Ни о каком старом методе printName мы и не знаем. |

**3) Приведение типов.**

Тут есть еще более интересный момент. Т.к. класс при наследовании получает все методы и данные класса родителя, то объект этого класса разрешается сохранять (присваивать) в переменные класса родителя (и родителя родителя, и т.д., вплоть до Object). Пример:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| public static void main(String[] args) {  Whale whale = new Whale();  whale.printColor();  } | На экран будет выведена надпись Я – белая |
| public static void main(String[] args) {  Cow cow = new Whale();  cow.printColor();  } | На экран будет выведена надпись Я – белая |
| public static void main(String[] args) {  Object o = new Whale();  System.out.println(o.toString());  } | На экран будет выведена надпись Whale@da435a Метод toString() унаследован от класса Object. |

— Очень интересно. А зачем это может понадобиться?

— Это ценное свойство. Позже ты поймешь, что очень, очень ценное.

**4) Вызов метода объекта (динамическая диспетчеризация методов).**

Вот как это выглядит:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| public static void main(String[] args) {  Whale whale = new Whale();  whale.printName();  } | На экран будет выведена надпись Я – кит. |
| public static void main(String[] args) {  Cow cow = new Whale();  cow.printName();  } | На экран будет выведена надпись Я – кит. |

Обрати внимание, что на то, какой именно метод printName вызовется, от класса Cow или Whale, влияет не тип переменной, а тип – объекта, на который она ссылается.

В переменной типа Cow сохранена ссылка на объект типа Whale, и будет вызван метод printName, описанный в классе Whale.

— Это не просто для понимания.

— Да, это не очень очевидно. Запомни главное правило:

Набор методов, которые можно вызвать у переменной, определяется типом переменной. А какой именно метод/какая реализация вызовется, определяется типом/классом объекта, ссылку на который хранит переменная.

— Попробую.

— Ты будешь постоянно сталкиваться с этим, так что скоро поймешь и больше никогда не забудешь.

**5) Расширение и сужение типов.**

Для ссылочных типов, т.е. классов, приведение типов работает не так, как для примитивных типов. Хотя у ссылочных типов тоже есть расширение и сужение типа. Пример:

|  |  |
| --- | --- |
| **Расширение типа** | **Описание** |
| Cow cow = new Whale(); | Классическое расширение типа. Теперь кита обобщили (расширили) до коровы, но у объекта типа Whale можно вызывать только методы, описанные в классе Cow.  Компилятор разрешит вызвать у переменной cow только те методы, которые есть у ее типа — класса Cow. |
| **Сужение типа** | **Описание** |
| Cow cow = new Whale();  if (cow instanceof Whale) {  Whale whale = (Whale) cow;  } | Классическое сужение типа с проверкой. Переменная cow типа Cow, хранит ссылку на объект класса Whale. Мы проверяем, что это так и есть, и затем выполняем операцию преобразования (сужения) типа. Или как ее еще называют – downcast. |
| Cow cow = new Cow();  Whale whale = (Whale) cow; //exception | Ссылочное сужение типа можно провести и без проверки типа объекта. При этом, если в переменной cow хранился объект не класса Whale, будет сгенерировано исключение – InvalidClassCastException. |

**6) А теперь еще на закуску. Вызов оригинального метода**

Иногда тебе хочется не заменить унаследованный метод на свой при переопределении метода, а лишь немного дополнить его.

В этом случае очень хочется исполнить в новом методе свой код и вызвать этот же метод, но базового класса. И такая возможность в Java есть. Делается это так: super.method().

Примеры:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| class Cow {  public void printAll() {  printColor();  printName();  }  public void printColor() {  System.out.println("Я – белый");  }  public void printName() {  System.out.println("Я – корова");  }  }  class Whale extends Cow {  public void printName() {  System.out.print("Это неправда: ");  super.printName();  System.out.println("Я – кит");  }  } |  |
| public static void main(String[] args){  Whale whale = new Whale();  whale.printAll();  } | На экран будет выведена надпись Я – белый Это неправда: Я – корова Я – кит |

— Гм. Ничего себе лекция. Мои робо-уши чуть не расплавились.

— Да, это не простой материал, он один из самых сложных. Профессор обещал подкинуть ссылок на материалы других авторов, чтобы ты, если все-таки что-то не поймешь, мог устранить этот пробел.