Лекция 9

Основы ООП: перегрузка, полиморфизм, абстракция, интерфейсы.

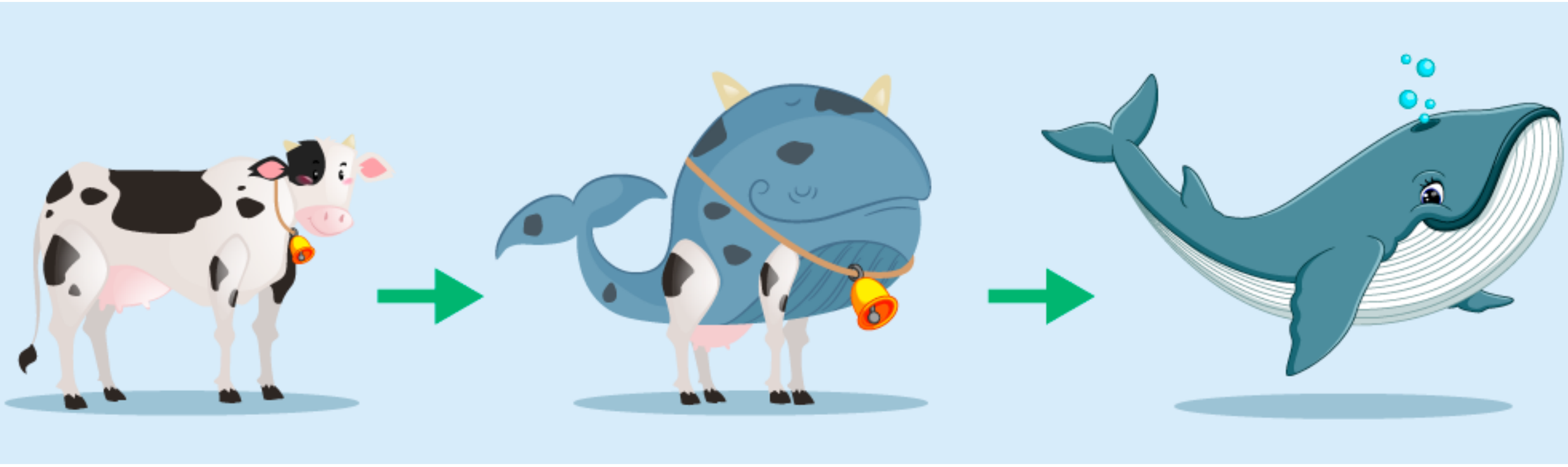
Полиморфизм и его особенности

Переопределение метода

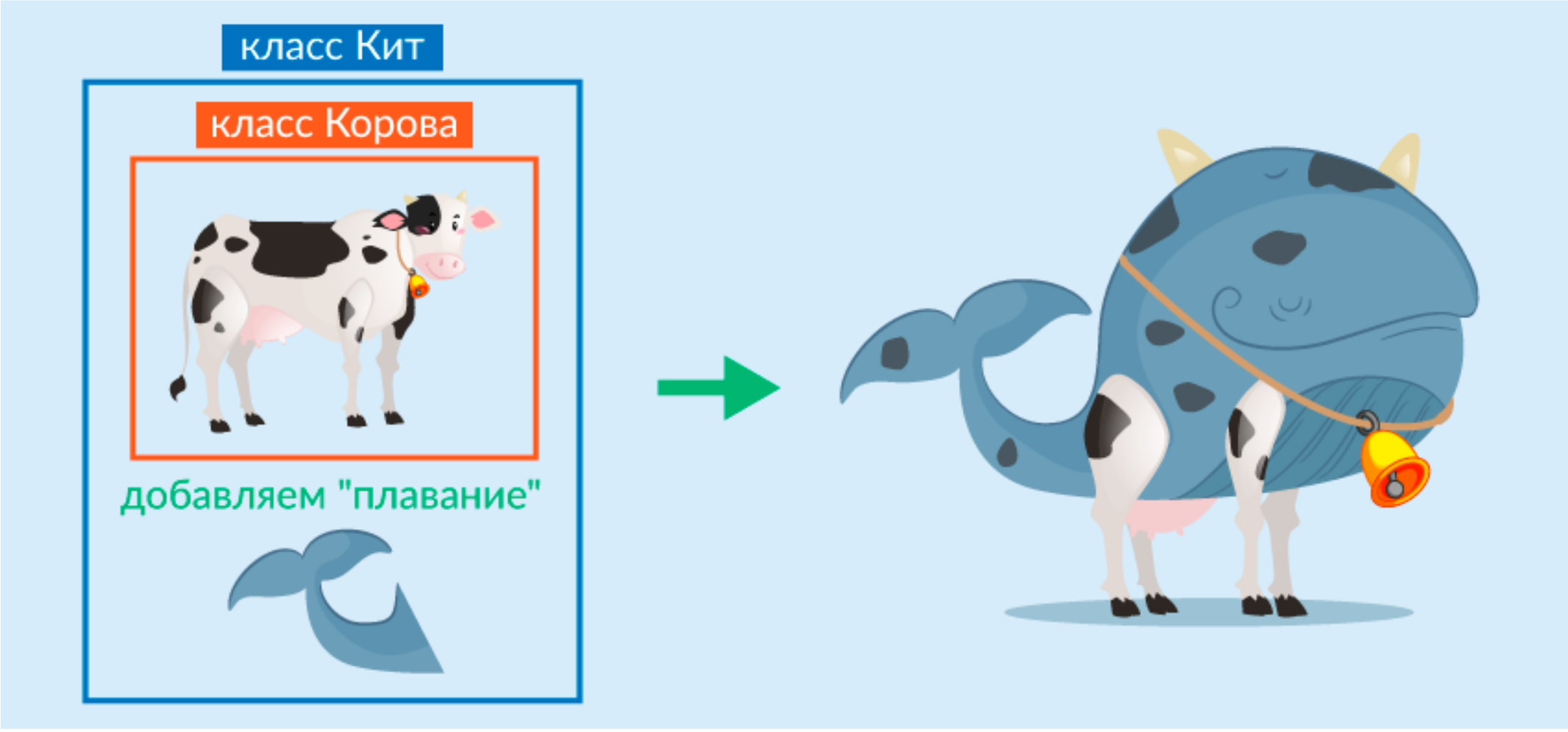
- Представьте, что вы для игры написали класс «Корова». В нем есть много полей и методов. Объекты этого класса могут делать разные вещи: идти, есть, спать. Еще коровы звонят в колокольчик, когда ходят. Допустим, вы реализовали в классе все до мелочей.

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

- Вы начали проектировать класс «Кит» и поняли, что он лишь немного отличается от класса «Корова». Логика работы обоих классов очень похожа, и вы решили использовать наследование.

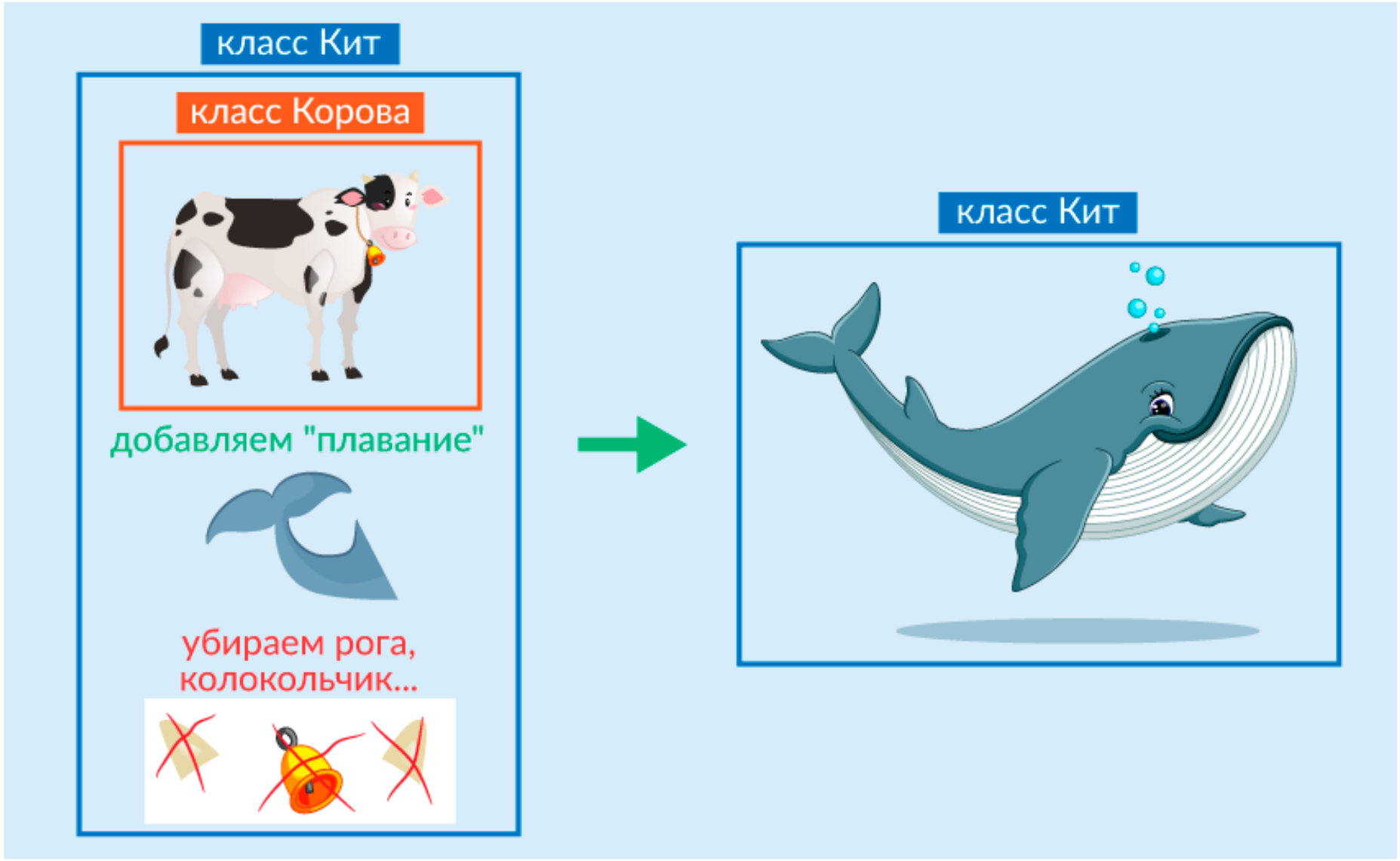


- Класс «Корова» идеально подходит на роль класса-родителя, там есть все необходимые переменные и методы. Достаточно только добавить киту возможность плавать. Но есть проблема: у вашего кита есть ноги, рога и колокольчик. Ведь эта функциональность реализована внутри класса «Корова». Что тут можно сделать?



- 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.

- 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.



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.

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

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

- Обратите 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.

- 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:

```
class Whale
{
 public void printAll()
 {
  printColor();
  printName();
 }
 public void printColor()
 {
  System.out.println("Я - белая»);
 }
 public void printName()
 {
  System.out.println("Я - кит");
 }
}
```

Приведение типов

- 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).

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.

Вызов метода объекта (динамическая диспетчеризация методов)

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.

- Обратите 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.

- 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.

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

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.

Вызов оригинального метода

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");
}
}
	Иногда хочется не заменить унаследованный метод на свой при переопределении метода, а лишь немного дополнить его.
	В этом случае можно 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().

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

Перегрузка методов

- Перегрузка методов – это не операция над методами, хотя иногда ее называют страшным словом – **параметрический полиморфизм.**

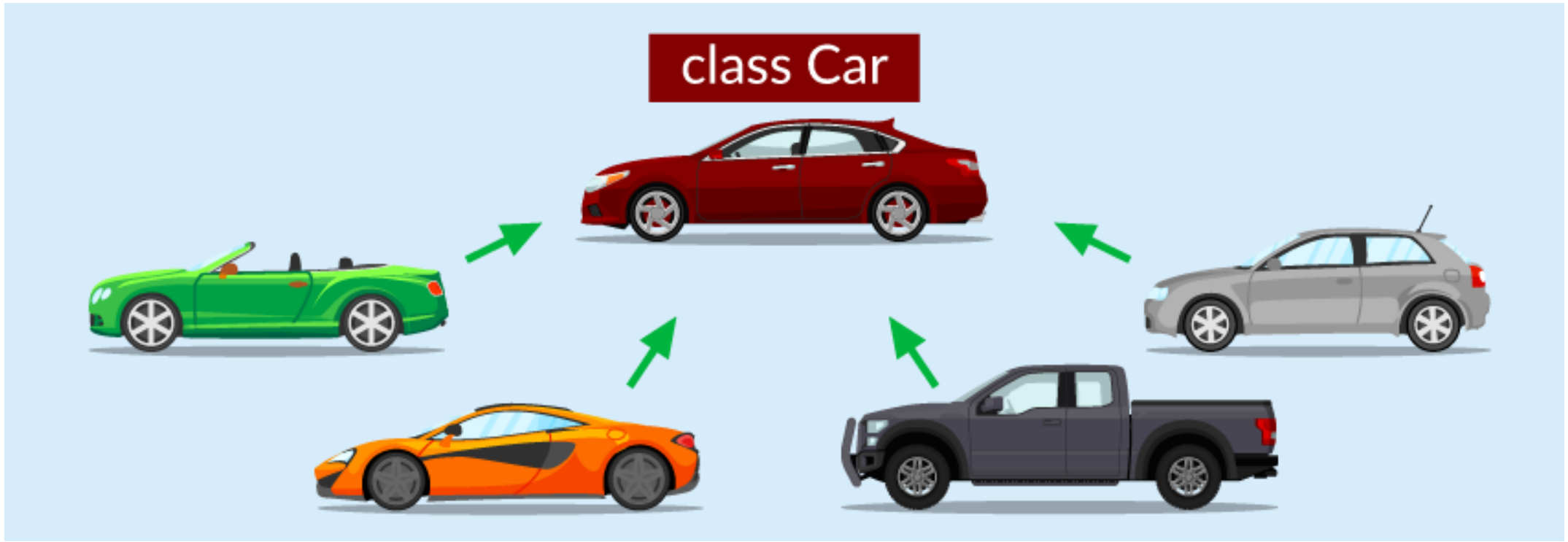
- Дело в том, что все методы внутри класса должны иметь уникальные имена.
  Но это совсем не так. Метод не должен иметь уникальное имя. Уникальным должно быть объединение из имени и типов параметров этого метода. Их еще называют сигнатурами методов.

- Так же обращаю внимание, что имена параметров роли не играют – они теряются при компиляции.
  После компиляции о методе известно только его имя и типы параметров.

Код	Описание
public void print(); **public void** print2();	Так можно. Два метода имеют уникальные имена.
public void print(); **public void** print(**int** n);	И так можно. Два метода имеют уникальные имена (сигнатуры).
public void print(**int** n, **int** n2); **public void** print(**int** n);	Все еще уникальные методы.
public int print(**int** a); **public void** print(**int** n);	А так нельзя. Методы не уникальные, хоть и возвращают разные типы.
public int print(**int** a, **long** b); **public long** print(**long** b, **int** a);	А так – можно. Параметры методов уникальные.

Абстрактные классы

- в Java есть специальный тип классов – **абстрактные классы**. Вот четыре вещи, которые стоит помнить об абстрактных классах.



- **1)** Абстрактный класс может содержать объявление метода без его реализации. Такой метод называется абстрактным.

Пример

```
public abstract class ChessItem
{
 public int x, y; //координаты
 private int value; // «ценность» фигуры

 public int getValue() //обычный метод, возвращает значение value
 {
   return value;
 }

 public abstract void draw(); //абстрактный метод. Реализация отсутствует.

}
```

- **2)** Абстрактный метод помечается специальным ключевым словом **abstract**.
  Если в классе есть хоть один абстрактный метод, класс тоже помечается ключевым словом abstract.

- **3)** Создавать объекты абстрактного класса нельзя. Такой код просто не скомпилирует.

Код на Java	Описание
`ChessItem item = new ChessItem();` `item.draw();`	Этот код не скомпилируется
`ChessItem item = new Queen();` `item.draw();`	А так можно.

- **4)** Если вы наследовали свой класс от абстрактного класса, то нужно переопределить все унаследованные абстрактные методы — написать для них реализацию. Иначе такой класс тоже придется объявить абстрактным. Если в классе есть хотя-бы один нереализованный метод, объявленный прямо в нем или унаследованный от класса-родителя, то класс считается абстрактным.

Интерфейсы

- Интерфейс очень напоминает абстрактный класс, у которого все методы абстрактные. Он объявляется так же, как и класс, только используется ключевое слово **interface**.

- У интерфейсов есть два сильных преимущества по сравнению с классами

Код	Описание и Факты

interface Drawable
{
void draw();
}
interface HasValue
{
int getValue();
}
	1) Вместо слова class пишем interface.
	
	2) Содержит только абстрактные методы (слово abstract писать не нужно).
	3) На самом деле у интерфейсов все методы — public.

interface Element **extends** Drawable, HasValue
{
int getX();
int getY();
}
	Интерфейс может наследоваться только от интерфейсов.
	
	Интерфейсов-родителей может быть много.

abstract class ChessItem **implements** Drawable, HasValue
{
private int x, y, value;
 	
public int getValue()
{
return value;
}
 	
public int getX()
{
return x;
}
 	
public int getY()
{
return y;
}
 	
}
	Класс может наследоваться от нескольких интерфейсов (и только от одного класса). При этом используется ключевое слово implements.
	
	Класс ChessItem объявлен абстрактным: он реализовал все унаследованные методы, кроме draw.
	Т.е. класс ChessItem содержит один абстрактный метод: draw().

1) Отделение «описания методов» от их реализации.

- Если вы хотите разрешить вызывать методы своего класса из других классов, то их нужно пометить ключевым словом public. Если же хотите, чтобы какие-то методы можно было вызывать только из вашего же класса, их нужно помечать ключевым словом private. Другими словами мы делим методы класса на две категории: «для всех» и «только для своих».

- С помощью интерфейсов, это деление можно усилить еще больше. Мы сделаем специальный «класс для всех», и второй «класс для своих», который унаследуем от первого. Вот как это примерно будет:

Было	Стало

class Student
{
private String name;
 	
public Student(String name)
{
this.name = name;
}
 	
public String getName()
{
return this.name;
}
 	
private void setName(String name)
{
this.name = name;
}
	interface Student
	{
	public String getName();
	}
	
	class StudentImpl **implements** Student
	{
	private String name;
	public StudentImpl(String name)
	{
	this.name = name;
	}
	public String getName()
	{
	return this.name;
	}
	private void setName(String name)
	{
	this.name = name;
	}
	}

public static void main(String[] args)
{
Student student =
new Student("Alibaba");
System.out.println(student.getName());
}
	public static void main(String[] args)
	{
	Student student =
	new StudentImpl("Ali");
	System.out.println(student.getName());
	}

- Мы разбили наш класс на два: интерфейс и класс, унаследованный от интерфейса.

- Один и тот же интерфейс могут реализовывать (наследовать) различные классы. И у каждого может быть свое собственное поведение. Так же как **ArrayList** и **LinkedList** – это две различные реализации интерфейса List.

- Таким образом, мы скрываем не только различные реализации, но и даже сам класс, который ее содержит (везде в коде может фигурировать только интерфейс). Это позволяет очень гибко, прямо в процессе исполнения программы, подменять одни объекты на другие, меняя поведение объекта скрытно от всех классов, которые его используют.

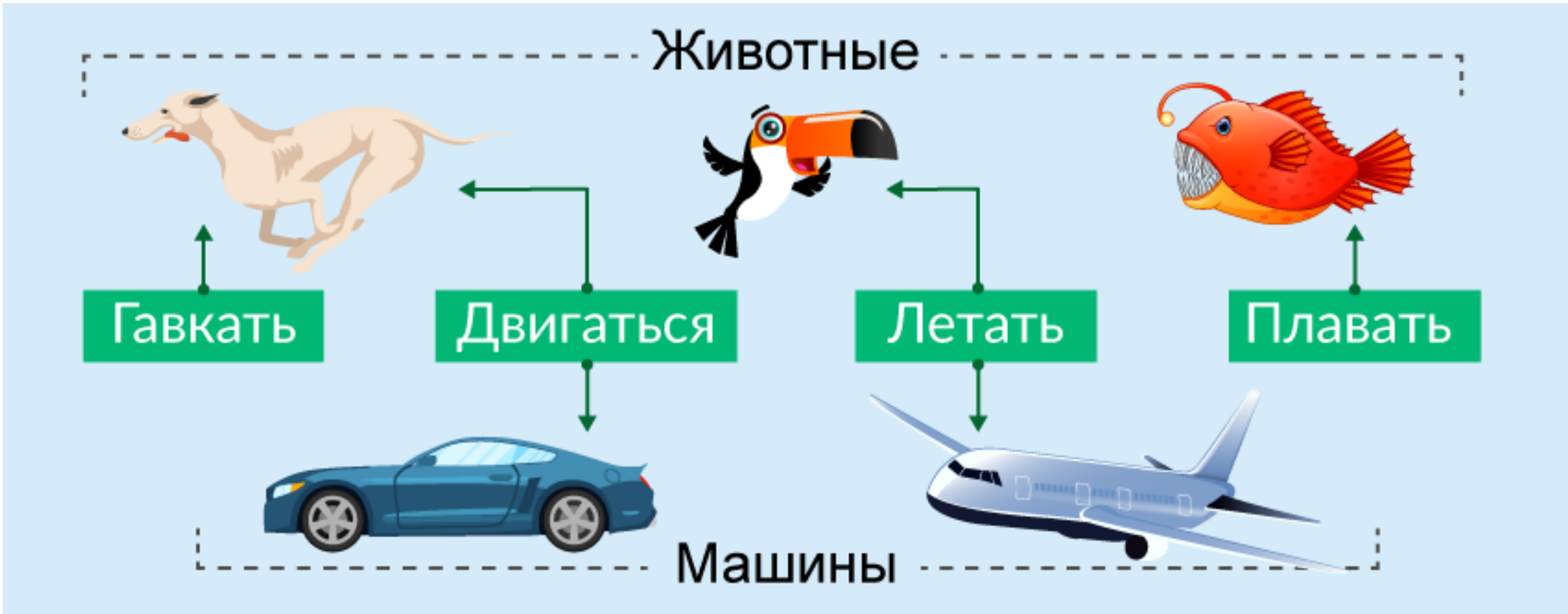
2) Множественное наследование.

- В Java все классы могут иметь только одного класса-родителя. В других языках программирования, классы часто могут иметь несколько классов-родителей. Это очень удобно, но приносит так же много проблем.

- В Java пришли к компромиссу – запретили множественное наследование классов, но разрешили множественное наследование интерфейсов. Интерфейс может иметь несколько интерфейсов-родителей. Класс может иметь несколько интерфейсов-родителей и только один класс-родитель.

Интерфейсы

- Класс – это, чаще всего модель какого-то конкретного объекта. Интерфейс же больше соответствует не объектам, а их способностям или ролям.



- Например, такие вещи, как машина, велосипед, мотоцикл и колесо лучше всего представить в виде классов и объектов. А такие их способности как «могу ездить», «могу перевозить людей», «могу стоять» — лучше представить в виде интерфейсов.

Код на Java	Описание

interface Moveable
{
void move(String newAddress);
}
	— соответствует способности передвигаться.

interface Driveable
{
void drive(Driver driver);
}
	— соответствует способности управляться водителем.

interface Transport
{
void addStaff (Object staff);
Object removeStaff();
}
	— соответствует способности перевозить грузы.

class Wheel **implements** Moveable
{
...
}
	— класс «колесо». Обладает способностью передвигаться.

class Car **implements** Moveable, Drivable, Transport
{
...
}
	— класс «машина». Обладает способностью передвигаться, управляться человеком и перевозить грузы.

class Skateboard **implements** Moveable, Driveable
{
...
}
	— класс «скейтборд». Обладает способностью передвигаться и управляться человеком.

- Интерфейсы сильно упрощают жизнь программиста. Очень часто в программе тысячи объектов, сотни классов и всего пара десятков интерфейсов — ролей. Ролей мало, а их комбинаций — классов — очень много.

- Весь смысл в том, что вам не нужно писать код для взаимодействия со всеми классами. Вам достаточно взаимодействовать с их ролями (интерфейсами).

- Представьте, что вы — робот-строитель и у вас в подчинении есть десятки роботов, каждый из которых может иметь несколько профессий. Вам нужно срочно достроить стену. Вы просто берете всех роботов, у которых есть способность «строитель» и говорите им строить стену. Вам все равно, что это за роботы. Хоть робот-поливалка. Если он умеет строить — пусть идет строить.

- Вот как это выглядело бы в коде:

Код на Java	Описание

static interface WallBuilder
{
void buildWall();
}
	— способность «строитель стен». Понимает команду «(по)строить стену» — имеет соответствующий метод.

static class РабочийРобот **implements** WallBuilder
{
void buildWall()
{
…
}
}
static class РоботСторож **implements** WallBuilder
{
void buildWall()
{
…
}
}
static class Поливалка
{
…
	— роботы у которых есть эта профессия/ особенность.
	
	— для удобства сделали классам имена на русском. Такое допускается в java, но крайне нежелательно.
	— поливалка не обладает способностью строить стены (не реализует интерфейс WallBuilder).

public static void main(String[] args)
{
//добавляем всех роботов в список
ArrayList robots = **new** ArrayList();
robots.add(**new** РабочийРобот());
robots.add(**new** РоботСторож());
robots.add(**new** Поливалка());
//строить стену, если есть такая способность
for (Object robot: robots)
{
if (robot **instanceof** WallBuilder)
{
WallBuilder builder = (WallBuilder) robot;
builder.buildWall();
}
}
}
	— как дать им команду – построить стену?

Задачи

1. Переопределить метод getName в классе **Whale**(**Кит**), чтобы программа выдавала: Я не корова, Я - кит.

```
public class Solution {
   public static void main(String[] args) {
      Cow cow = new Whale();
      System.out.println(cow.getName());
   }

   public static class Cow {
      public String getName() {
         return "Я - корова";
      }
   }

   public static class Whale extends Cow {
   }
}
```

2. Переопределить метод getName в классе **Whale**(**Кит**), чтобы программа ничего не выводила на экран.

```
public class Solution {
   public static void main(String[] args) {
      Cow cow = new Whale();
      System.out.println(cow.getName());
   }

   public static class Cow {
      public String getName() {
         return "Я - корова";
      }
   }

   public static class Whale extends Cow {
   }
}
```

3. Написать метод, который определяет, объект какого класса ему передали, и возвращает результат - одно значение из: "**Корова**", "**Кит**", "**Собака**", "**Неизвестное животное**».

```
public class Solution {
   public static void main(String[] args) {
      System.out.println(getObjectType(new Cow()));
      System.out.println(getObjectType(new Dog()));
      System.out.println(getObjectType(new Whale()));
      System.out.println(getObjectType(new Pig()));
   }

   public static String getObjectType(Object o) {
      //Напишите тут ваше решение

      return "Неизвестное животное";
   }

   public static class Cow {
   }

   public static class Dog {
   }

   public static class Whale {
   }

   public static class Pig {
   }
}
```

4. Сделать класс **Pet** абстрактным.

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

   public static class Pet {
      public String getName() {
         return "Я - котенок";
      }
   }
}
```

5. Исправьте код, чтобы программа компилировалась.

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

   public static class Pet {
      public String getName() {
         return "Я - котенок";
      }

      public Pet getChild();
   }
}
```