Введение в программирование на Java

Лекция 7. ООП (часть 2). Отношения между классами.

Виталий Олегович Афанасьев

24 февраля 2025

Повторное использование классов

Одна из краегоульных вещей ООП (и разработки в целом) — повторное использование уже написанного кода.

В Java это реализуется посредством использования одних классов другими. При помощи:

- Агрегации
- Композиции
- Наследования

Агрегация и композиция

Агрегация и композиция (1)

Суть агрегации и композиции проста — объект может состоять из составных частей:

- Автомобиль состоит из двигателя, колёс, ...
- Стул состоит из ножек, сиденья и спинки.
- ПК состоит материнской платы, процессора, ...

Агрегация и композиция (2)

При **композиции** объект контролирует жизненный цикл составных частей. Части не могут существовать сами по себе — при уничтожении объекта уничтожаются и составные части.

При **агрегации** равноправные объекты ссылаются друг на друга и могут существовать независимо.

Композиция и агрегация задают отношение "содержит"/"состоит из" ("has-a"/"part-of").

Агрегация и композиция (3)

- Колёса у машины можно снять и повесить на другую машину — агрегация.
- Некоторые стулья можно разобрать, сняв с них ножки — агрегация.
- Но у некоторых стульев ножки не снимаются — композиция.
- Факультеты не могут существовать без университета — композиция.

Агрегация

При агрегации объект хранит ссылку на другой объект.

```
1  public class Job {
2      public final String companyName;
3      ...
4  }
5  public class Employee {
6      public Job job;
7      public Employee(Job job) {
8          this.job = job;
9      }
10     public void setJob(Job newJob) {
11         job = newJob;
12     }
13 }
```

Композиция

При композиции объект сам создаёт составные части и хранит ссылки на них.

```
1  public class Department {
2      public final String name;
3      ...
4  }
5  public class University {
6      public Department[] departments;
7      ...
8      public void addDepartment(String name) {
9          Department newDepartment = new Department(name);
10         ... // Расширяем массив
11         departments[...] = newDepartment;
12     }
13 }
```

Наследование

Наследование (1)

Наследование (inheritance) — концепция ООП, позволяющая классу наследовать данные и поведение другого класса, при этом расширяя или изменяя их.

Наследование (1)

Наследование (inheritance) — концепция ООП, позволяющая классу наследовать данные и поведение другого класса, при этом расширяя или изменяя их.

Примеры:

- Птицы — это животные. Для птиц характерно оперение, крылья и клюв.
  - Ласточка — птица.
  - Пингвин — тоже птица, но летать не умеет.
    - Императорский пингвин и галапагосский пингвин — тоже пингвины, но разные.
- Автомобиль — средство передвижения.
  - Существуют грузовые автомобили.
  - Велосипед — не автомобиль, но средство передвижения.

Наследование (2)

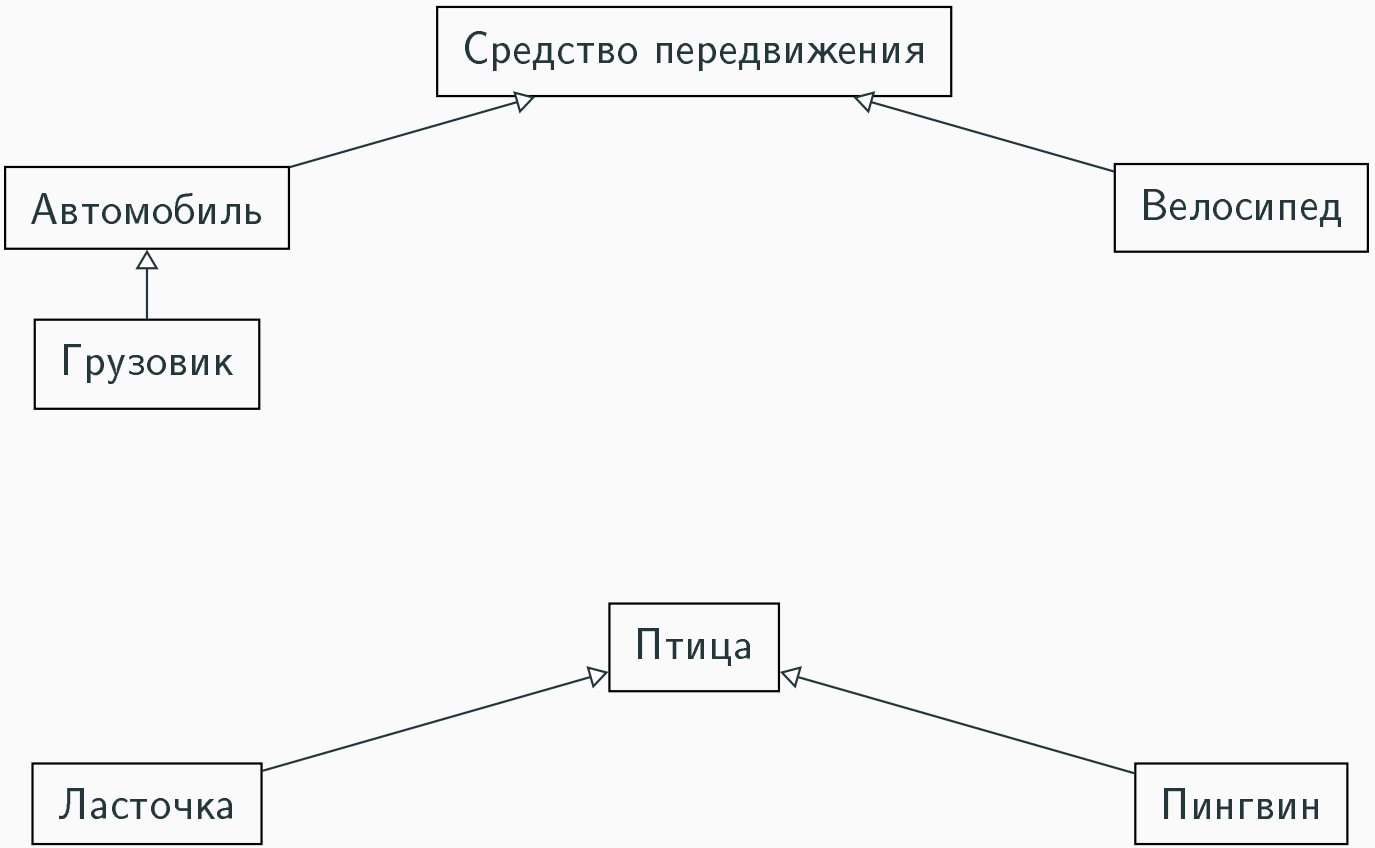
Логически правильнее называть наследование **расширением**:

- Грузовик расширяет поведение и свойства автомобиля.

Один из способов понять, что один класс наследуется от другого — определить, состоят ли они в отношении "является" ("is-a"):

- Грузовик является автомобилем.
- Пингвин является птицей.
- Велосипед является средством передвижения.
- Велосипед не является автомобилем.

Наследование (3)



Наследование (4)

```
1  public class Car {
2      public String model;
3      public int speed = 0;
4      public int maxSpeed;
5
6      public void accelerate() {
7          if (speed < maxSpeed) {
8              ++speed;
9          }
10     }
11
12     public void brake() {
13         if (speed > 0) {
14             --speed;
15         }
16     }
17 }
```

Наследование (5)

Для наследования в Java используется ключевое слово `extends`.

```
1  public class Truck extends Car {
2      public int cargoWeight;
3
4      public void unload() {
5          cargoWeight = 0;
6      }
7
8      public void load(int addCargoWeight) {
9          cargoWeight += addCargoWeight;
10     }
11 }
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7
8
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11 kamaz.brake();
12
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11 kamaz.brake();
12 kamaz.load(100);
13
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11 kamaz.brake();
12 kamaz.load(100);
13 System.out.println(kamaz.cargoWeight); // 100
14
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11 kamaz.brake();
12 kamaz.load(100);
13 System.out.println(kamaz.cargoWeight); // 100
14 kamaz.unload();
15
16
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11 kamaz.brake();
12 kamaz.load(100);
13 System.out.println(kamaz.cargoWeight); // 100
14 kamaz.unload();
15
16 bmw.load(100); // ERROR
17
```

Наследование (6)

Классы-наследники имеют те же поля и методы, что и родители.

```
1  Car bmw = new Car();
2  bmw.model = "BMW";
3  bmw.maxSpeed = 250;
4  bmw.accelerate();
5  bmw.brake();
6
7  Truck kamaz = new Truck();
8  kamaz.model = "KAMAZ";
9  kamaz.maxSpeed = 100;
10 kamaz.accelerate();
11 kamaz.brake();
12 kamaz.load(100);
13 System.out.println(kamaz.cargoWeight); // 100
14 kamaz.unload();
15
16 bmw.load(100); // ERROR
17 bmw.unload(); // ERROR
```

Делегирование конструкторов при наследовании

При помощи ключевого слова super можно вызвать конструктор родительского класса.

```
1  public class Car {
2      ...
3      public Car(String model, int maxSpeed) {
4          this.model = model;
5          this.maxSpeed = maxSpeed;
6      }
7  }
8
9  public class Truck extends Car {
10     ...
11     public Truck(String model, int maxSpeed, int cargoWeight) {
12         super(model, maxSpeed); // Вызов конструктора Car(String, int)
13         this.cargoWeight = cargoWeight;
14     }
15 }
```

Модификатор protected (1)

Модификатор protected у членов класса разрешает доступ к ним только из самого класса и его наследников.

```
1  public class Car {
2      protected int speed;
3  }
4  public class Truck extends Car {
5      public void accessInChild() {
6          speed = 100; // OK: speed имеет protected доступ
7      }
8  }
9  public class Main {
10     public static void main(String[] args){
11         Car car = new Car();
12         car.speed = 100; // ERROR: speed имеет protected доступ
13     }
14 }
```

У protected немного более сложная логика — подробнее узнаем на лекции про пакеты.

Модификатор `protected` (2)

`Protected`-доступ стоит использовать с осторожностью (а лучше — не использовать вовсе).

- `protected` стоит использовать только в тех случаях, если класс специально предназначен, чтобы от него наследовались. Но при этом нужно предвидеть все способы дальнейшего использования этого класса.
- `protected` увеличивает связность между классами. При изменении `protected`-члена в родительском классе требуется проверить, что дочерние классы при этом не сломались.
- При использовании `protected` легко нарушить инкапсуляцию. "Зловредный" класс-наследник может организовать публичный доступ к таким полям и методам.

Переопределение методов (1)

Наследование позволяет переопределять поведение родительского класса.

```java
public class Car {
    public void makeSound() {
        System.out.println("Beep");
    }
}
public class Truck extends Car {
    public void makeSound() {
        System.out.println("BEEEEEEEEEEEEEEEEEEP!!!");
    }
}
public class Main {
    public static void main(String[] args){
        Car car = new Car("BMW", 250);
        car.makeSound(); // Beep
        Truck truck = new Truck("KAMAZ", 100, 0);
        truck.makeSound(); // BEEEEEEEEEEEEEEEEEEP!!!
    }
}
```

Переопределение методов (2)

При переопределении метода рекомендуется использовать аннотацию `@Override`.

Она позволят удостовериться (в момент компиляции), что у родительского класса действительно есть такой метод.

```
1  public class Car {
2      public void makeSound() { ... }
3  }
4  public class Truck extends Car {
5      @Override // OK
6      public void makeSound() { ... }
7
8      @Override // ERROR: В родительском классе нет такого метода
9      public void fakeMethod() { ... }
10
11     @Override // ERROR: В родительском классе нет такого метода
12     public void makeSound(String sound) { ... }
13 }
```

Переопределение методов (3)

При помощи ключевого слова `super` можно явно обратиться к членам родительского класса. Например, вызвать родительский метод.

```
1  public class Car {
2      public void accelerate() {
3          if (speed < maxSpeed) {
4              ++speed;
5          }
6      }
7  }
8  public class Truck extends Car {
9      @Override
10     public void accelerate() {
11         System.out.println("WROOM-WROOM!");
12         super.accelerate();
13     }
14 }
```

Переопределение методов (4)

Переопределение доступно только для методов, которые доступны из дочернего класса. То есть, для `public` и `protected`.

```
1  public class Car {
2      public void makeSound() { ... }
3      protected void makeSoundProtected() { ... }
4      private void makeSoundPrivate() { ... }
5      private void anotherPrivateMethod() { ... }
6  }
7  public class Truck extends Car {
8      @Override // OK
9      public void makeSound() { ... }
10     @Override // OK
11     protected void makeSoundProtected() { ... }
12     @Override // ERROR
13     private void makeSoundPrivate() { ... }
14     // OK, но этот метод не связан с родительским
15     private void anotherPrivateMethod() { ... }
16 }
```

Переопределение методов (5)

При переопределении методов можно делать их более доступными.

Но менее доступными — нельзя.

```
1  public class Car {
2      public void publicMethodOne() { ... }
3      public void publicMethodTwo() { ... }
4      protected void protectedMethodOne() { ... }
5      protected void protectedMethodTwo() { ... }
6  }
7  public class Truck extends Car {
8      @Override // OK
9      public void publicMethodOne() { ... }
10     @Override // ERROR: public -> protected (менее доступен)
11     protected void publicMethodTwo() { ... }
12     @Override // OK
13     protected void protectedMethodOne() { ... }
14     @Override // OK: protected -> public (более доступен)
15     public void protectedMethodOne() { ... }
16 }
```

final-методы

Модификатором `final` можно пометить метод. В таком случае, его нельзя будет переопределить в дочерних классах.

```
1  public class Car {
2      public void makeSound() { ... }
3      public final void accelerate() { ... }
4  }
5  public class Truck extends Car {
6      @Override // OK
7      public void makeSound() { ... }
8      @Override // ERROR: нельзя переопределить final-метод
9      public void accelerate() { ... }
10 }
```

final-классы

Модификатором `final` можно пометить целый класс.

От `final`-классов нельзя наследовать другие классы.

```
1 public final class Car {
2     ...
3 }
4
5 // ERROR: нельзя наследоваться от final-класса
6 public class Truck extends Car {
7     ...
8 }
```

Полиморфизм

Полиморфизм — способность программных систем обрабатывать данные разных типов.

В Java выделяют три основных вида полиморфизма:

- **Специальный (ad-hoc) полиморфизм**.
- **Полиморфизм подтипов**.
- **Параметрический полиморфизм**.

Ad-hoc полиморфизм

Ad-hoc полиморфизм в Java достигается посредством перегрузки функций.

```
1 public void add(int a, int b) { return a + b; }
2 public void add(int a, int b, int c) { return a + b + c; }
3 public void add(String s1, String s2) { return s1 + s2; }
```

При вызове это выглядит так, будто вызывается одна и та же функция.

```
1 add(1, 2); // 3
2 add(4, 5, 6); // 15
3 add("Hello, ", "World!"); // "Hello, World!"
```

Полиморфизм подтипов (1)

Методы, принимающие родительский тип, могут принимать и объекты всех его подтипов.

```
1  public static void main(String[] args) {
2      Car car = new Car("BMW", 250);
3      Truck truck = new Truck("KAMAZ", 100, 0);
4      int halfSpeedOfCar = halfSpeed(car); // 125
5      int halfSpeedOfTruck = halfSpeed(truck); // 50
6  }
7
8  public static int halfSpeed(Car car) {
9      return car.speed / 2;
10 }
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2
3
4
5
6
7
8
9
10
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4
5
6
7
8
9
10
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4  Car truckAsCar = truck; // OK: Car - родитель Truck
5
6
7
8
9
10
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4  Car truckAsCar = truck; // OK: Car - родитель Truck
5  Truck carAsTruck = car; // ERROR: Не каждый Car является Truck
6
7
8
9
10
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4  Car truckAsCar = truck; // OK: Car - родитель Truck
5  Truck carAsTruck = car; // ERROR: Не каждый Car является Truck
6
7  truck = truckAsCar;
8
9
10
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4  Car truckAsCar = truck; // OK: Car - родитель Truck
5  Truck carAsTruck = car; // ERROR: Не каждый Car является Truck
6
7  truck = truckAsCar;
8      // ERROR, хоть мы и знаем, что реальный тип - Truck
9
10
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4  Car truckAsCar = truck; // OK: Car - родитель Truck
5  Truck carAsTruck = car; // ERROR: Не каждый Car является Truck
6
7  truck = truckAsCar;
8      // ERROR, хоть мы и знаем, что реальный тип - Truck
9
10 truck = (Truck) truckAsCar; // OK: Явно приведён тип
11
12
```

Полиморфизм подтипов (2)

Допустимо присваивать объекты дочерних типов в переменные родительских типов.

В обратную сторону — нельзя.

```
1  Car car = new Car("BMW", 250);
2  Truck truck = new Truck("KAMAZ", 100, 0);
3
4  Car truckAsCar = truck; // OK: Car - родитель Truck
5  Truck carAsTruck = car; // ERROR: Не каждый Car является Truck
6
7  truck = truckAsCar;
8      // ERROR, хоть мы и знаем, что реальный тип - Truck
9
10 truck = (Truck) truckAsCar; // OK: Явно приведён тип
11
12 Truck carAsTruck = (Truck) car; // ERROR: Ошибка при ВЫПОЛНЕНИИ
```

Полиморфизм подтипов (3)

При вызове переопределённого метода на объекте он вызывается на **реальном** типе объекта, а не объявленном.

Из-за этого полиморфизм подтипов ещё называют **"динамическим полиморфизмом"**.

```
1 Car car = new Car("BMW", 250);
2 car.makeSound(); // Beep
3 Truck truck = new Truck("KAMAZ", 100, 0);
4 truck.makeSound(); // BEEEEEEEEEEEEEEEEEEEP!!!
5
6 Car truckAsCar = truck;
7 truckAsCar.makeSound(); // BEEEEEEEEEEEEEEEEEEEP!!!
```

Полиморфизм подтипов (4)

При вызове перегруженных методов выбирается наиболее подходящий по типу **переменной**, а не реальному типу объекта.

```
1  public static void main(String[] args) {
2      Car car = new Car("BMW", 250);
3      Truck truck = new Truck("KAMAZ", 100, 0);
4      doSomething(car); // Первый метод: Beep Beep
5      doSomething(truck); // Второй метод: BEEEEEEEEEEEEEEEEEEP!!!
6
7      Car truckAsCar = truck;
8      doSomething(truckAsCar);
9        // Первый метод: BEEEEEEEEEEEEEEEEEEP!!! BEEEEEEEEEEEEEEEEEEP!!!
10 }
11 public static void doSomething(Car car) {
12     car.makeSound();
13     car.makeSound();
14 }
15 public static void doSomething(Truck truck) {
16     truck.makeSound();
17 }
```

Полиморфизм подтипов (5)

Полиморфизм подтипов позволяет абстрагироваться от конкретной реализации.

Код, взаимодействующий с объектом через методы родительского класса, не думает о том, как реализованы методы в его подтипах.

Это снижает зависимость между разными участками кода.

```
public class Shape { public int perimeter() { ... } }
public class Triangle extends Shape { public int perimeter() { ... } }
public class Circle extends Shape { public int perimeter() { ... } }
```

```
1 public void totalPerimeter(Shape[] shapes) {
2     int result = 0;
3     for(int i = 0; i < shapes.length; ++i) {
4         result += shapes[i].perimeter();
5     }
6     return result;
7 }
```