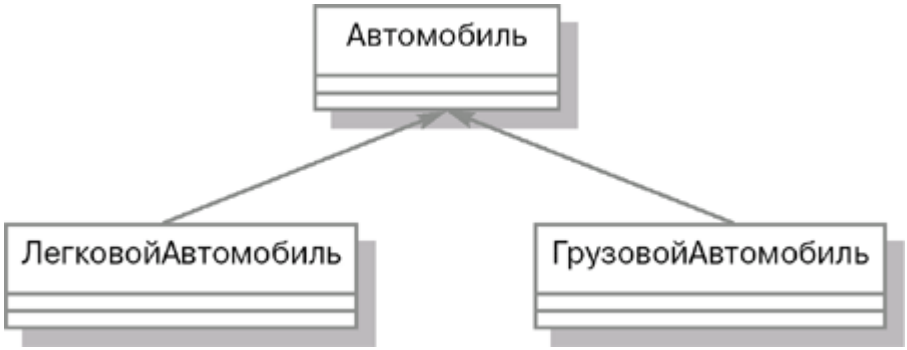
Лекция 3.
Наследование.
Виртуальные функции.
Модификаторы видимости

Наследование

- **Наследование** – процесс создания на основе старого класса нового класса, который может переопределять члены класса-родителя и добавлять новые члены
- Класс-родитель называют **базовым классом** или **суперклассом**
- Класс-наследник называют **подклассом** или **производным классом**



Пример наследования из жизни

- Пример – класс **Млекопитающие** и класс **Человек**
- Можно сказать что класс **Человек** наследуется от класса **Млекопитающие**. Человек имеет все свойства, присущие млекопитающим – например, кормление детей молоком. И при этом добавляет новые свойства – например, прямохождение, отсутствие хвоста, развитый мозг и т.д.
- Каждый человек является млекопитающим, но не каждое млекопитающее – человек. Везде, где требуется некоторое млекопитающее, можно использовать человека

Наследование

- Объекты производных классов обычно являются более узко-специализированными
- Пример из жизни: общий класс – **Число**. Конкретный класс – **Целое число**.
- Или: общий класс – **Хищник**. Производный класс – **Кошка**
- Каждый объект производного класса является и объектом базового класса и может использоваться везде, где ожидается ссылка на базовый класс

Наследование

- Объекты производных классов обычно являются более узко-специализированными

- Все объекты класса **ЛегковойАвтомобиль** являются и объектами класса **Автомобиль**
- Но не все объекта класса **Автомобиль** являются объектами класса **ЛегковойАвтомобиль** (они могут быть объектами класса **ГрузовойАвтомобиль**)

Наследование

```
public class Robot {
    private int power;
    public void move() {
     //…
    }
}
```

```
public class RobotCleaner extends Robot {
    public void clean() {
        // ..
    }
}
```

Класс RobotCleaner наследуется от Robot и добавляет новый метод clean()

```
public static void main(String[] args) {
    RobotCleaner robot = new RobotCleaner();
    robot.clean();
    robot.move();
}
```

Экземпляры класса RobotCleaner наследуют (получают) члены своих родителей – метод move и поле power

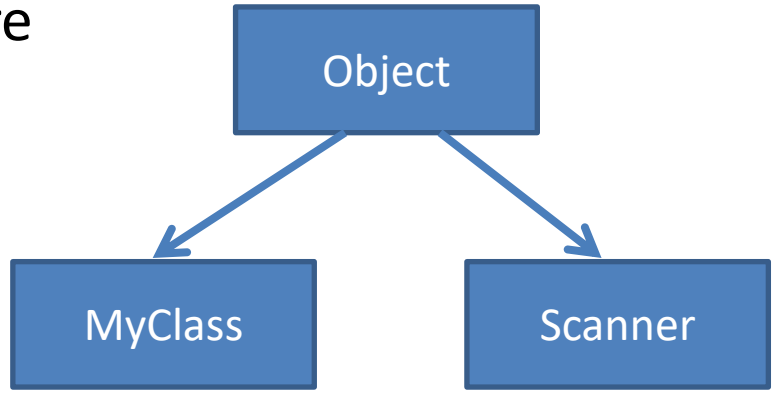
Важные моменты при наследовании

- При наследовании наследуется все, кроме конструкторов

- В классах-наследниках нет доступа к private членам родителей несмотря на то, что они наследуются
- Потому что модификатор private для членов класса означает, что «доступно внутри класса», а класс-наследник – это уже другой класс

Наследование в Java

- В Java каждый класс может непосредственно наследоваться только от одного класса-родителя (это называется **одиночное наследование**)
- Все классы в Java в конечном итоге наследуются от java.lang.Object и наследуют его методы
- Если у класса при объявлении не указан родитель, то неявно родителем считается Object



- Примитивные типы лежат вне этой иерархии, и не являются объектами

Наследование от Object

- ```
  public class A {
      private int a;

      public void f() {
          //…
      }
  }
  ```

Класс A неявно наследуется от Object
Это то же самое, что написать
A extends Object
  ```

Конструкторы при наследовании

- При наследовании каждый конструктор дочернего класса **обязан** вызвать какой-либо конструктор родительского класса
- Это нужно, т.к. у класса родителя могут быть поля, которые нужно заполнить
- Это делается при помощи ключевого слова super
-

```
public class A {
    private int x;

    public A(int x) {
        // неявно super();
        this.x = x;
    }
}
```

```
public class B extends A {
    private int y;

    public B(int x, int y) {
        super(x);
        this.y = y;
    }
}
```

Конструкторы при наследовании

- super обязан быть первой командой в конструкторе
- В круглых скобках передаются аргументы конструктору класса-родителя
- Если не писать super, то неявно пишется super(); – вызов конструктора родителя без аргументов (а его может не быть, тогда будет ошибка компиляции)
-

```
public class A {
    private int x;

    public A(int x) {
        // неявно super();
        this.x = x;
    }
}
```

```
public class B extends A {
    private int y;

    public B(int x, int y) {
        super(x);
        this.y = y;
    }
}
```

Наследование

```
public class Person {
    private String name;
    public Person(String name) { this.name = name; }
    public String getName() { return name; }
}

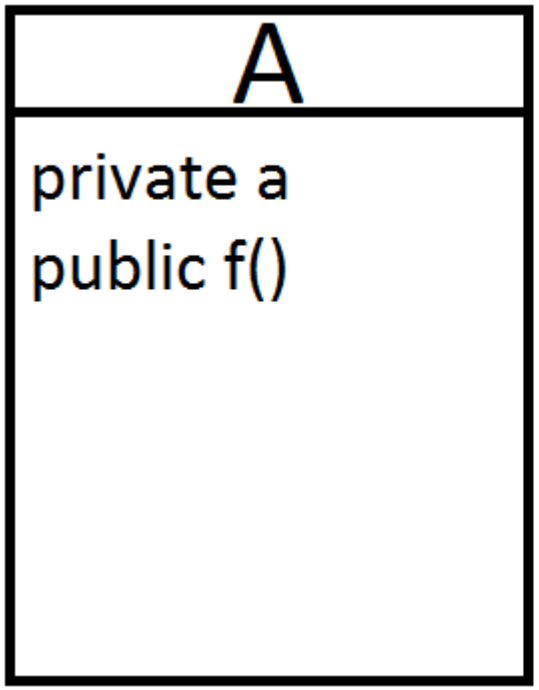
public class Employee extends Person {
    private int salary;

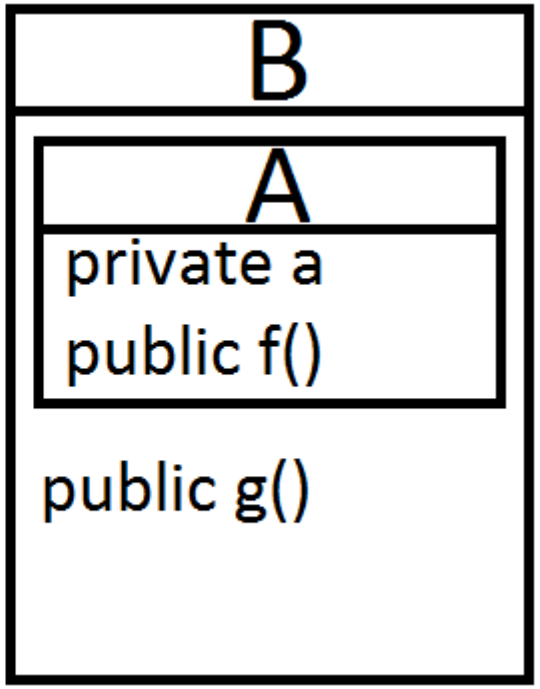
    public Employee(String name, int salary) {
        super(name);
        this.salary = salary;
    }
    public int getSalary() { return salary; }
}
```

Наследование

```
public class A {
    private int a;
    public void f() {
        //…
    }
}
```

```
public class B extends A {
    public void g() {
        // ..
    }
}
```





Приведение ссылочных типов

Приведение ссылочных типов

- Объекты классов-наследников можно использовать везде, где требуется класс-родитель
- Пусть класс RobotCleaner наследуется от класса Robot
- ```
  Robot a = new Robot();
  RobotCleaner b = new RobotCleaner();
  Robot c = new RobotCleaner();  // неявное приведение к
                                 // базовому типу
  ```
- Для переменной **c** – сам объект принадлежит классу RobotCleaner, но переменная – ссылка на класс Robot
  ```

Приведение ссылочных типов

- Robot c = new RobotCleaner(); // неявное приведение к
  // базовому типу
- При приведении типа от наследника к родителю сам объект никак не изменяется – он не усекается, поля не удаляются, новый объект не создается и т.д.
- Переменная **c** ссылается на тот же самый объект, но т.к. тип переменной теперь Robot, а не RobotCleaner, то компилятор разрешает обращаться только к тем членам класса, которые объявлены в Robot
- c.clean(); // ошибка компиляции
  c.move(); // нормально работает

Приведение ссылочных типов

- ```
  public class Utils {
      public static void useRobot(Robot a) {
      }
  }
  ```
- В этот метод можно передавать и наследников Robot

- ```
  Robot a = new Robot();
  Utils.useRobot(a);   // OK
  ```
- ```
  RobotCleaner b = new RobotCleaner();
  Utils.useRobot(b);   // OK, неявное приведение к
                       // базовому типу Robot
  ```
- ```
  Robot c = new RobotCleaner();
  Utils.useRobot(c);   // OK
  ```

Приведение ссылочных типов

- ```
  public class Utils {
      public static void useRobot(RobotCleaner b) {
      }
  }
  ```
- ```
  Robot a = new Robot();
  Utils.useRobot(a);  // ошибка компиляции – не каждый
                      // Robot является RobotCleaner
  ```
- ```
  RobotCleaner b = new RobotCleaner();
  Utils.useRobot(b);  // OK
  ```
- ```
  Robot c = new RobotCleaner();
  Utils.useRobot(c);  // ошибка компиляции – переменная
                      // типа Robot, компилятор не понимает,
                      // что это в самом деле RobotCleaner
  ```

Явное приведение типов

- Если мы точно знаем, что хоть и переменная ссылается на базовый тип, а сам объект принадлежит производному типу, то мы можем выполнить явное приведение типа

- ```
  Robot a = new Robot();
  RobotCleaner b = new RobotCleaner();
  Robot c = new RobotCleaner();
  ```

- ```
  RobotCleaner d = (RobotCleaner)c;  // OK
  RobotCleaner e = (RobotCleaner)a;  // ошибка во время
                                     // исполнения, нельзя
                                     // преобразовать объект
                                     // Robot в RobotCleaner
  ```

Проверка принадлежности классу

- Оператор проверки, что объект принадлежит классу или является потомком этого класса – instanceof

- Пример:
- if (“abc” instanceof String) {
  // выполнится – строка принадлежит типу String
  }
- if (“abc” instanceof Object) {
  // выполнится – все классы наслед-ся от Object
  }

Проверка принадлежности классу

- Примеры:
- ```
  if (null instanceof <ЛюбойТип>) {
      // не выполнится - всегда false
  }
  ```
- ```
  if (“abc” instanceof Scanner) {
      // не выполнится – строка не наследуется от
      // Scanner’a
  }
  ```

Виртуальные функции

Переопределение методов

- Кроме добавления новых полей и методов, производные классы могут переопределять методы своих классов-предков
- Это называется **переопределением метода (overriding)**
- Функции, которые можно переопределить в классах-потомках, называются **виртуальными (virtual functions)**

- **Виртуальная функция –** функция, которую можно переопределить в классах-наследниках так, что при ее вызове будет использоваться реализация, соответствующая настоящему типу объекта

Переопределение методов

```
public class A {
    public void f() {
        System.out.println(1);
    }
}
```

```
public class B extends A {
    public void f() {
        System.out.println(2);
    }
}
```

- В классах A и B есть функция с одинаковой сигнатурой
- ```
  A a = new A();
  a.f(); // 1
  ```
- ```
  B b = new B();
  b.f(); // 2
  ```
- ```
  A c = new B();
  c.f(); // 2
  ```

Для виртуальной функции вызывается та реализация, которая определена для фактического типа **объекта**, а не для типа ссылки

Для полей ничего такого нет
  ```

Виртуальные функции

- **Виртуальные функции** – являются еще одним примером **полиморфизма**
- ```
  public static void workWithA(A a) {
      a.f();
  }
  ```
- Функция **workWithA** ничего не знает о том, какой именно класс у объекта **a**
  - Но она точно знает что этот класс либо A, либо наследник класса A
  - Поэтому **workWithA** может вызвать метод **f** класса A
- Если передать объект класса A, то вызовется реализация класса A. Если передать объект класса B, и в нем переопределен метод **f**, то вызовется метод **f** класса B
  ```

Пример – геометрические фигуры

- Пусть есть базовый класс Shape (фигура)
- ```
  public class Shape {
      public double getArea() {
          return 0;  // нет разумной реализации, поэтому пока так
      }
  }
  ```
- А дальше создаем классы-наследники для прямоугольника, треугольника, круга и т.д., и в них правильно реализуем этот метод
- ```
  public class Rectangle extends Shape {
      // опущен код полей и конструктора
      public double getArea() { return width * height; }
  }
  ```

Пример – геометрические фигуры

- И тогда эти примеры будут работать правильно
- ```
  Shape s1 = new Rectangle(10, 2);
  System.out.println(s1.getArea());  // 20
  // вызывается реализация для прямоугольника
  ```
- ```
  Shape s2 = new Triangle(0, 0, 3, 0, 0, 4);
  System.out.println(s2.getArea());  // 6
  // вызывается реализация для треугольника
  ```

Пример использования полиморфизма

- ```
  public class Employee {
      public int getSalary() {
          return 20000;
      }
  }
  ```
- ```
  public class Director extends Employee {
      public int getSalary() {
          return 50000;
      }
  }
  ```
- ```
  public class Manager extends Employee {
      public int getSalary() {
          return 30000;
      }
  }
  ```
  ```

Пример использования полиморфизма

```
public static int getTotalSalary(Employee[] employees) {
    // выдает суммарную зарплату по всем переданным
    int result = 0;
    for (Employee e : employees) {
        result += e.getSalary();
    }
    return result;
}
```

Очень простая реализация за счет полиморфизма и виртуальной функции **getSalary**

Код метода **getTotalSalary** никогда не изменится и будет работать и для новых типов сотрудников

```
public static void main() {
    Employee[] e = { new Director(), new Employee (),
      new Manager (), new Manager (), new Employee () };

    System.out.println(getTotalSalary(e));
}
```

Пример отсутствия полиморфизма

```
public enum EmployeeType {
    EMPLOYEE, DIRECTOR, MANAGER
}
```

```
public class Employee {
    private EmployeeType type;
    public int getSalary() {
        if (type == EmployeeType.EMPLOYEE) {
            return 20000;
        }
        if (type == EmployeeType.DIRECTOR) {
            return 50000;
        }
        return 30000;
    }
}
```

Код не будет работать для новых типов сотрудников, нужно дописывать ветку в if

Таблица виртуальных функций

- Как реализованы виртуальные функции?
- Для каждого класса в Java есть **таблица виртуальных функций**, она одна на весь класс
- В этой таблице хранятся адреса функций (функции тоже хранятся в памяти, у них есть адрес)
- Внутри каждого объекта хранится ссылка на таблицу, которая соответствует типу объекта

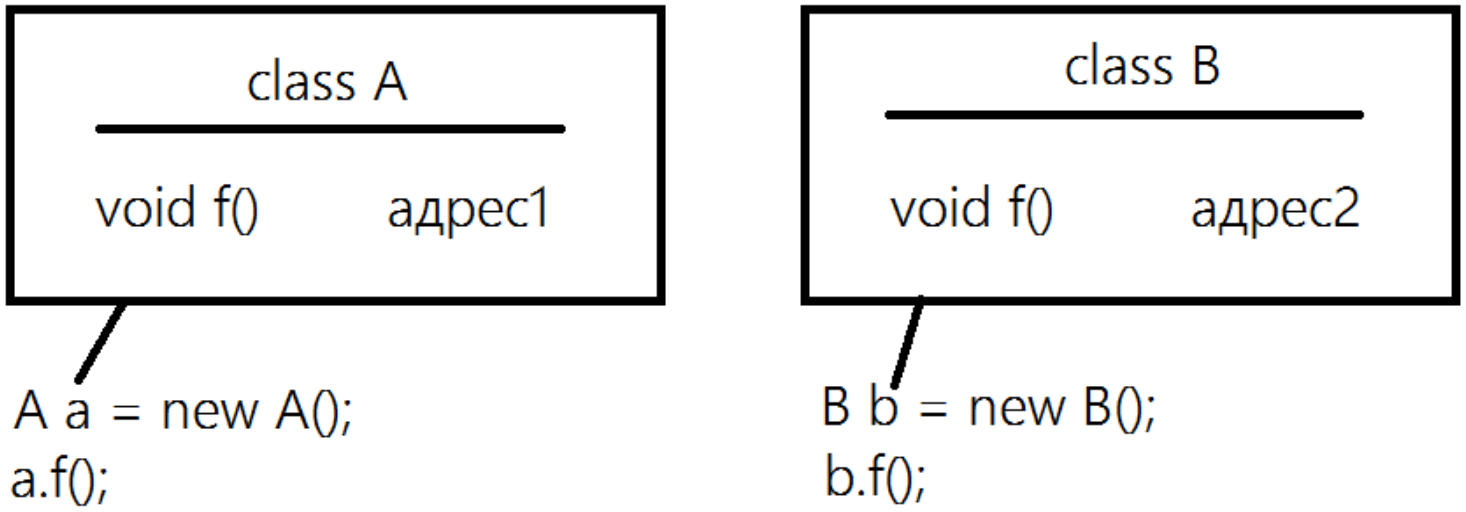
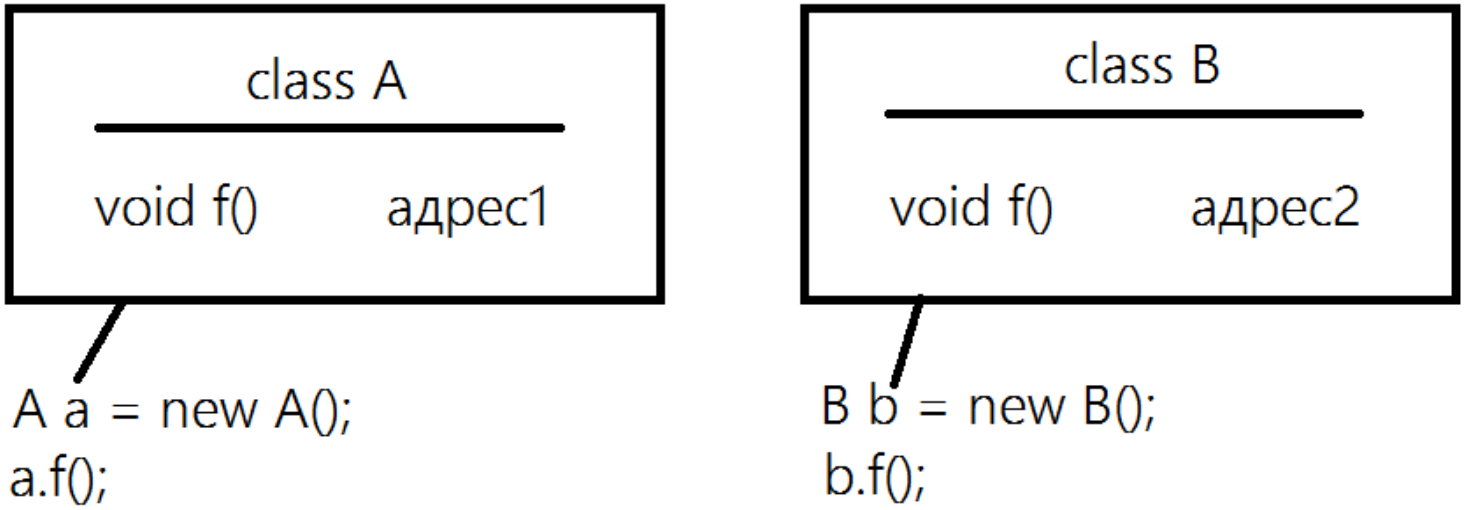


Таблица виртуальных функций

- При вызове метода **f**, Java смотрит в таблицу виртуальных функций для текущего объекта, и вызывает функцию по адресу, который указан в таблице
- Если объект по факту принадлежит классу A, то вызовется реализация по адресу1, если классу B – то по адресу2



Аннотация @Override

- При переопределении виртуального метода можно случайно опечататься или указать не ту сигнатуру, тогда это не будет считаться переопределением
- Чтобы защититься от этого, можно при переопределении добавить к методу аннотацию @Override
- Тогда компилятор выдаст ошибку, если этот метод ничего не переопределяет из родительских классов
-
```
public class A {
    public void f() {
        System.out.println(1);
    }
}
```

```
public class B extends A {
    @Override
    public void f() {
        System.out.println(2);
    }
}
```

Модификаторы видимости

- В Java есть 4 варианта модификаторов видимости для полей и методов классов
- package ru.academits.vm;

```
public class A {
    private int x;        // доступен только внутри класса A
    public int y;         // доступен везде
    int z;                // доступен только внутри пакета
    protected int w;  // доступен внутри пакета
                      // и всем подклассам, независимо от пакета
}
```

Модификаторы видимости

строгость

Модификатор	Пояснение
private	Видимость только внутри текущего класса
без модификатора	Видимость внутри всего пакета, где объявлен класс
protected	Видимость внутри всего пакета, где объявлен класс. Видимость внутри всех классов-наследниках, независимо от пакета
public	Видимость везде

Модификаторы видимости

- private члены недоступны даже классам-наследникам
- Если в классе-наследнике нужно обратиться к private членам классов-предков, то можно применить следующие варианты:
  - поменять этим членам класса модификатор на protected
  - если это поля, то поля можно оставить private, но сделать protected getter и/или setter

Какие функции являются виртуальными?

- В Java виртуальными являются только public, protected, (без модификатора видимости) нестатические функции
- То есть не являются виртуальными:
  - Все static функции
  - Все private функции
- Эти функции можно переопределить в классах-потомках, но они не будут виртуальными

Ограничения при переопределении

- При переопределении метода в производном классе нужно использовать либо тот же модификатор видимости, что в родителе, либо слабее
- ```
  public class A {
      protected void f() {
          // код
      }
  }
  ```

```
public class B extends A {
    public void f() {
        // код
    }
}
```
```

Переопределение методов

```
public class A {
    public static void f() {
        System.out.println(1);
    }
}
```

```
public class B extends A {
    public static void f() {
        System.out.println(2);
    }
}
```

- В классах A и B есть статическая функция с одинаковой сигнатурой
- `A a = new A();` `a.f(); // 1`
- `B b = new B();` `b.f(); // 2`
- `A c = new B();` `c.f(); // 1`

static функции рекомендуется всегда вызывать через имя класса: A.f();

Для статических функций вызванная реализация определяется типом ссылки, а не объекта

Перекрытие полей (hiding)

```
public class A {
    public int x = 1;
}
```

```
public class B extends A {
    public int x = 2;
}
```

```
B b1 = new B();
A b2 = b1;

System.out.println(b1.x);  // 2
System.out.println(b2.x);  // 1
```

Создание невиртуального члена с таким же именем и сигнатурой в потомке называется перекрытием (hiding)

Поля не являются виртуальными. К какому полю пойдет обращение, зависит от типа ссылки

Слово super для методов и полей

- Из дочернего класса можно обратиться к полям и методам непосредственного родителя при помощи слова super

```
public class A {
    public void f() {
        System.out.println(1);
    }
}
```

```
public class B extends A {
    public void f() {
        super.f();
        System.out.println(2);
    }
}
```

- ```
  A b = new B();
  b.f();
  // 1
  // 2
  ```

Здесь слово super не обязано быть первой командой в методе

Полезно когда мы хотим «дополнить» реализацию метода
  ```

Слово super для методов и полей

- Слово super позволяет классам-наследникам обращаться только к членам непосредственного родителя
- К членам родителя родителя и т.д. обратиться таким образом нельзя

Слово super для методов и полей

- Из дочернего класса можно обратиться к полям и методам непосредственного родителя при помощи слова super

```
public class A {
    protected int x = 1;
    public void f() {
        System.out.println(x);
    }
}
```

```
public class B extends A {
    protected int x = 2;
    public void f() {
        System.out.println(x + super.x);
    }
}
```

super.x указывает именно на поле в классе-родителе

```
A b = new B();
b.f(); // 3
```

Слово final для классов

- Можно запретить наследоваться от класса, добавив к его объявлению ключевое слово final

- ```
  public final class A {
      // код
  }
  ```
- ```
  public class B extends A {
      // ошибка компиляции, нельзя наследоваться
      // от final класса
  }
  ```

Зачем запрещать наследование?

- Из соображений безопасности – ведь наследники могут переопределять методы как хотят и обращаться к protected членам, а хочется запретить менять реализацию класса
- Это улучшает производительность
  - Виртуальные функции замедляют работу программы (это связано с тем, как они реализованы, – требуется дополнительный переход к таблице виртуальных функций)
  - final методы, так как не могут быть переопределены, реализованы как не виртуальные, поэтому их вызов быстрее

Слово final для методов

- Можно запретить наследникам переопределять метод, указав для него модификатор final

```
public class A {
    public final String getName() {
        return “1”;
    }
}
```

```
public class B extends A {
    public String getName() {
        return “2”; // ошибка компиляции
       // нельзя переопределять final метод
    }
}
```

Зачем final для методов?

- Аналогично мотивам использования final для классов – запретить изменение реализации или повысить производительность, только мы не хотим запрещать наследоваться от класса, а запрещаем переопределять только некоторые методы

Зачем нужно наследование?

- Помогает избавиться от дублирования кода:
  для иерархии классов можно создать базовый класс, который реализует основную логику, а от него будут наследоваться классы-наследники и переопределять лишь некоторую часть методов
- Например, если мы наследуемся от класса JFrame – который представляет в Java окно, то мы автоматически получаем все его методы, и чтобы создать своё окно таким как хочется, нужно лишь переопределить и добавить некоторые методы

Зачем нужно наследование?

- Но главное, ради чего стоит наследоваться – это полиморфизм
- Он позволяет создавать свои классы, которые можно использовать в уже существующем коде **библиотек** и **фреймворков**
- Например, библиотека представляет набор базовых классов, от которых можно создать наследников, чтобы решать свои задачи

Когда не нужно наследоваться

- Если мы просто хотим использовать некоторые методы класса-родителя, чтобы выполнить свою работу
- При этом наш класс логически не сильно связан с классом-родителем, либо бОльшая часть методов класса-родителя ему вообще не нужна

Когда не нужно наследоваться

- Допустим, есть класс «Окно операционной системы». Он очень сложно устроен и много чего умеет, например, отрисовываться на экране, получать события от пользователя о нажатиях мыши и клавиатуры и т.д. И еще у него есть ширина и высота и методы для работы с ними
- Допустим, мы хотим создать свой класс для геометрических фигур, и нам тоже надо уметь работать с шириной и высотой. Отнаследовавшись от окна, мы бы получили реализацию этих методов
- Но тем самым мы:
  - Получили много лишнего кода
  - Наш класс может использоваться везде, где нужны окна, а это не нужно

Когда не нужно наследоваться

- Общее правило такое – если вы при наследовании не переопределили ни один виртуальный метод, то наследование не нужно
- Тогда можно просто обойтись полем-ссылкой на нужный объект