Программирование
1 семестр

ИТМО

Полиморфизм. Интерфейсы

Базовые принципы ООП

- Абстракция
- Инкапсуляция
- Наследование
- **Полиморфизм**

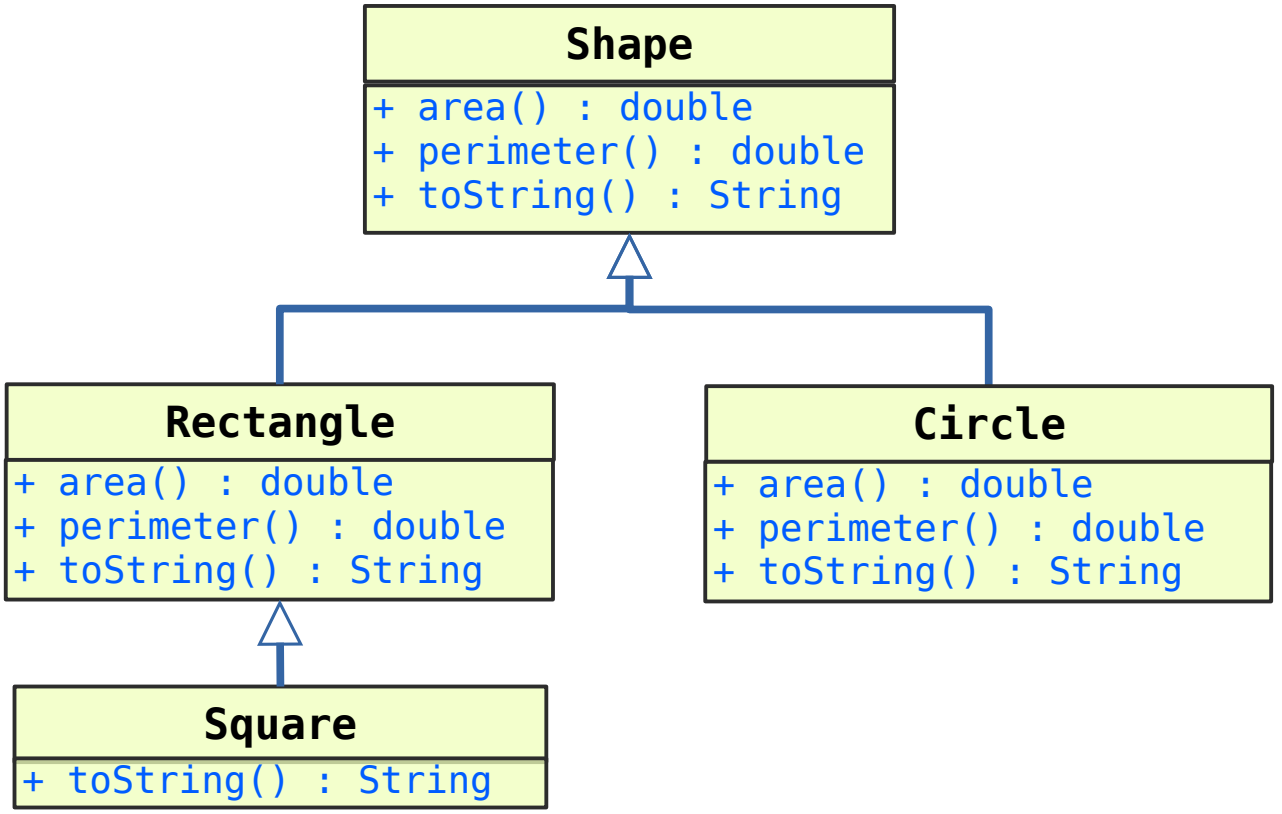
- Polymorphism
- Единообразная обработка с помощью общего интерфейса различных типов данных.
- Виды полиморфизма:
  - ad hoc (специальный) - перегрузка
  - подтипов
  - параметрический

Полиморфизм

ИТМО

- Полиморфизм позволяет:
  - Писать менее специфичный, более общий код
  - Обеспечить расширяемость программ
- Полиморфизм подтипов:
  - Работает с иерархией наследования
  - Позволяет подставлять наследников на место их предков
  - Связывание с конкретным классом - во время выполнения

Пример полиморфизма



Пример полиморфизма

ИТМО

```
class Shape {
   ...
}
```

```
class Rectangle extends Shape {
  public double area() { return width * height; }
  pbulic double perimeter() { return 2 * (width+height); }
  public String toString() {
    return "Rectangle (" + width + " * " + height + ")";
  }
}
```

```
class Square extends Rectangle {
  public String toString() { return "Square ("+side+")"; }
}
```

```
class Circle extends Shape {
  public double area() { return Math.PI * radius*radius; }
  pbulic double perimeter() { return 2*Math.PI * radius; }
  public String toString() { return "Circle ("+radius)"; }
}
```

Пример полиморфизма

ИТМО

```
class Shape {
  ...
}
```

```
Shape[] shapes = {
 new Rectangle(),
 new Circle(),
 new Square() };

for (Shape s : shapes) {
  var res = s + "\n" +
  s.area() + "\n" +
  s.perimeter() + "\n";
  System.out.print(res);
}
```

```
class Rectangle extends Shape {
  public double area() { return width * height; }
  pbulic double perimeter() { return 2 * (width+height); }
  public String toString() {
    return "Rectangle (" + width + " * " + height + ")";
  }
}
```

```
class Square extends Rectangle {
  public String toString() { return "Square ("+side+")"; }
}
```

```
class Circle extends Shape {
  public double area() { return Math.PI * radius*radius; }
  pbulic double perimeter() { return 2*Math.PI * radius; }
  public String toString() { return "Circle ("+radius)"; }
}
```

Плюсы полиморфизма - обобщение кода

ИТМО

```
Shape[] shapes = {
 new Rectangle(),
 new Circle(),
 new Square() };
for (Shape s : shapes) {
  double area = switch (s) {
    case Rectangle -> s.width() * s.height();
    case Square -> s.side() * s.side();
    case Circle -> Math.PI * s.radius() * s.radius();
  }
  System.out.println(area);
}
```

Плюсы полиморфизма - обобщение кода

```
Shape[] shapes = {
 new Rectangle(),
 new Circle(),
 new Square() };
for (Shape s : shapes) {
  double area = switch (s) {
    case Rectangle -> s.width() * s.height();
    case Square -> s.side() * s.side();
    case Circle -> Math.PI * s.radius() * s.radius();
  }
  System.out.println(area);
}
```

```
Shape[] shapes = {
 new Rectangle(),
 new Circle(),
 new Square()
};

for (Shape s : shapes) {
  System.out.print(s.area());
}
```

- Полиморфизм позволяет классам взаимодействовать на уровне более абстрактных суперклассов без привязки к деталям реализации

Плюсы полиморфизма - расширяемость

ИТМО

```
class Triangle extends Shape {
  public double area() { ... }
  public double perimeter() { ... }
  public String toString() { ... }
}
```

```
Shape[] shapes = {
 new Rectangle(),
 new Circle(),
 new Square(),
 };

for (Shape s : shapes) {
  System.out.print(s.area());
}
```

Плюсы полиморфизма - расширяемость

ИТМО

```
class Triangle extends Shape {
  public double area() { ... }
  public double perimeter() { ... }
  public String toString() { ... }
}
```

```
Shape[] shapes = {
 new Rectangle(),
 new Circle(),
 new Square(),
 new Triangle()
 };
for (Shape s : shapes) {
  System.out.print(s.area());
}
```

- Полиморфизм позволяет без изменения кода добавлять новые подклассы, которые реализуют специфичное поведение,

Динамическое связывание

- В классе хранится таблица виртуальных методов vtable
  - При компиляции - в таблицу заносятся методы класса
  - При загрузке класса - методу задается конкретный адрес
  - При выполнении из таблицы берется адрес метода

Ковариантность возвращаемого типа

ИТМО

- При переопределении - тип возвращаемого значения в методе потомка должен соответствовать типу значения метода предка
  - либо должен совпадать

```
class Animal {
  public AnimalSound makeSound() { }
}
```

```
class Cat extends Animal {
  public AnimalSound makeSound() { }
}
```

Ковариантность возвращаемого типа

ИТМО

- При переопределении - тип возвращаемого значения в методе потомка должен соответствовать типу значения метода предка
  - либо должен совпадать
  - либо быть ковариантным типом (подклассом возвращаемого значения метода предка)

```
class Animal {
  public AnimalSound makeSound() { }
}
```

```
class Cat extends Animal {
  public CatSound makeSound() { }
}

class CatSound extends AnimalSound { }
```

Объекты как параметры методов

ИТМО

- Если метод принимает параметр какого-то типа, то в этот метод можно передать:
  - объект указанного типа
  - объект потомка указанного типа

```
class ShapePrinter {
  public printShape(Shape shape) {
    System.out.println(s + "\n" +
                s.area() + "\n" +
            s.perimeter() + "\n");
  }
}
```

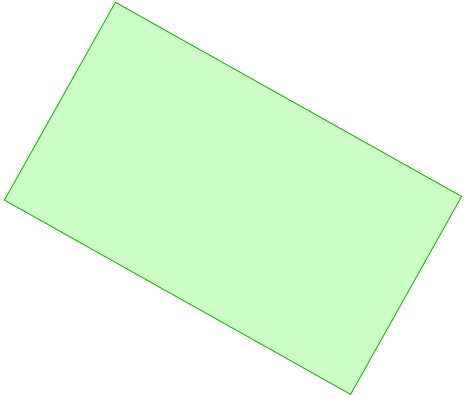
```
ShapePrinter sp = new ShapePrinter();
sp.printShape(new Rectangle(1.0, 2.0));
sp.printShape(new Square(5.0));
sp.printShape(new Circle(4.0));
sp.printShape(new Triangle(4.0,3.0,5.0));
```

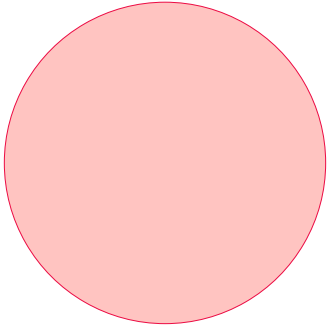
Конкретные классы

- Как выглядит объект типа прямоугольник?
- Как выглядит объект типа окружность?
- Как выглядит объект типа кот?

Конкретные классы

- Как выглядит объект типа прямоугольник?
- Как выглядит объект типа окружность?
- Как выглядит объект типа кот?



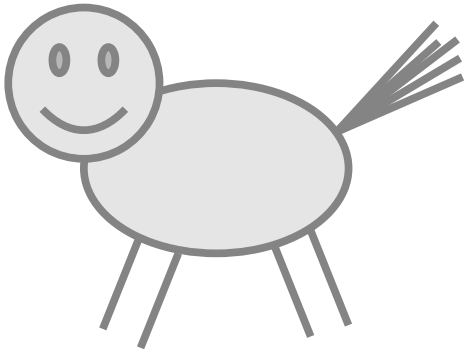


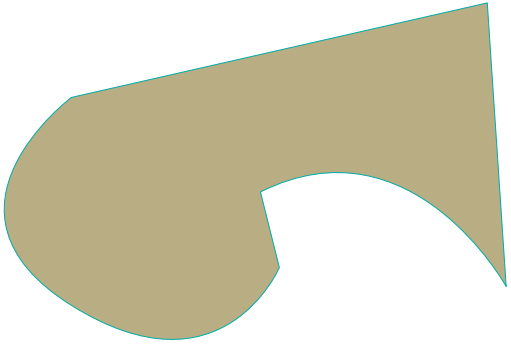
Абстрактные классы

- Как выглядит объект типа геометрическая фигура?
- Как выглядит объект типа животное?

Абстрактные классы

- Как выглядит объект типа геометрическая фигура?
- Как выглядит объект типа животное?





- Объекты абстрактного класса обычно не имеют смысла и напрямую не создаются

Абстрактные классы и методы

- Класс Shape можно (нужно) сделать абстрактным
  - Представляет абстрактную фигуру произвольной формы
  - Абстрактные методы для расчета площади и периметра
  - Абстрактные методы не реализованы в абстрактном классе
  - Метод toString() - из класса Object, но переопределен
  - Метод description() - общее поведение для наследников

Абстрактные классы и методы

ИТМО

```
abstract class Shape {

  public abstract double area();
  public abstract double perimeter();

  @Override
  public String toString() {
    return "Shape";
  }

  public String desciption() {
    return s.toString() + "\nArea:\t" + s.area() + "\nPerimeter:\t" + s.perimeter();
  }
}
```

Особенности абстрактных классов

- Класс - абстрактный, если в нем есть абстрактный метод, но их может и не быть
- Абстрактные методы должны переопределяться в потомках, иначе потомок тоже будет абстрактным
- Абстрактный класс не может быть финальным
- Нельзя создать объект абстрактного класса с помощью new
- Конструктор абстрактного класса обычно protected

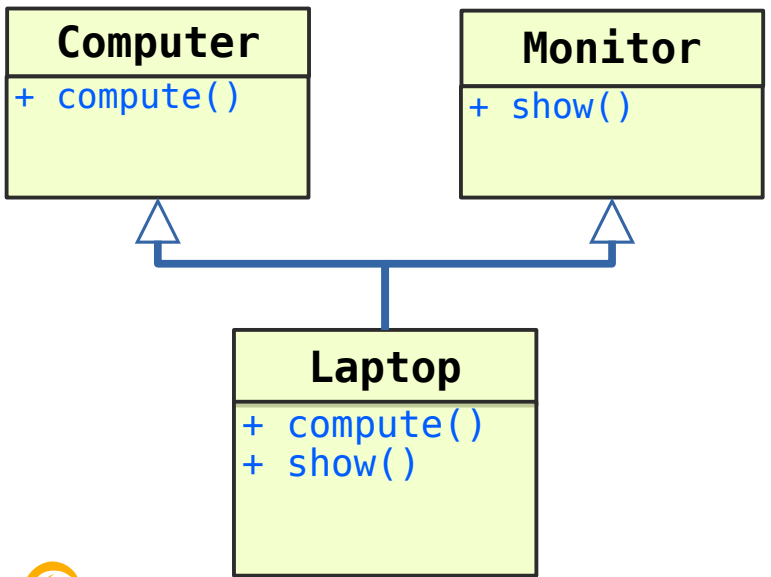
Ссылки на абстрактные классы

ИТМО

- Абстрактный класс может быть типом ссылки.
- Переменной, имеющей тип ссылки на абстрактный класс, можно присваивать ссылки на объекты потомков

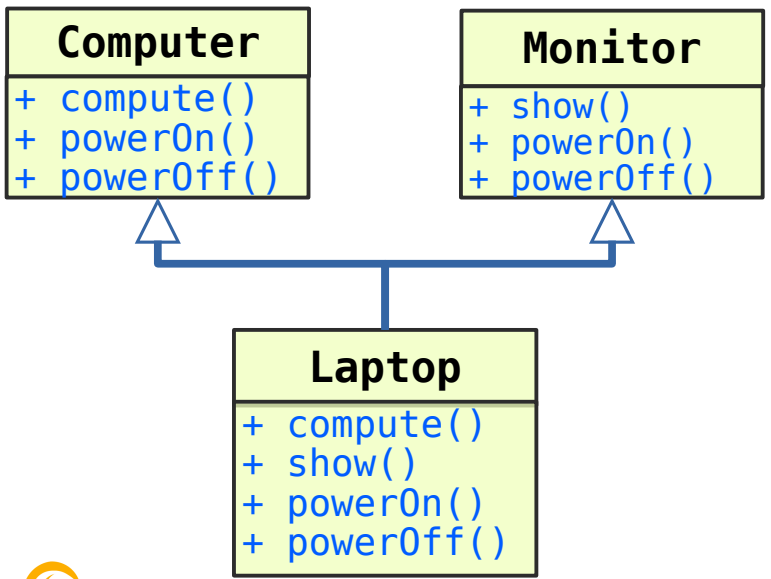
```
Shape s1 = new Shape () // нельзя, если Shape - абстрактный

Shape s2 = new Rectangle(3.4); // можно, если Rectangle - не абстрактный
```

Наследование от нескольких классов



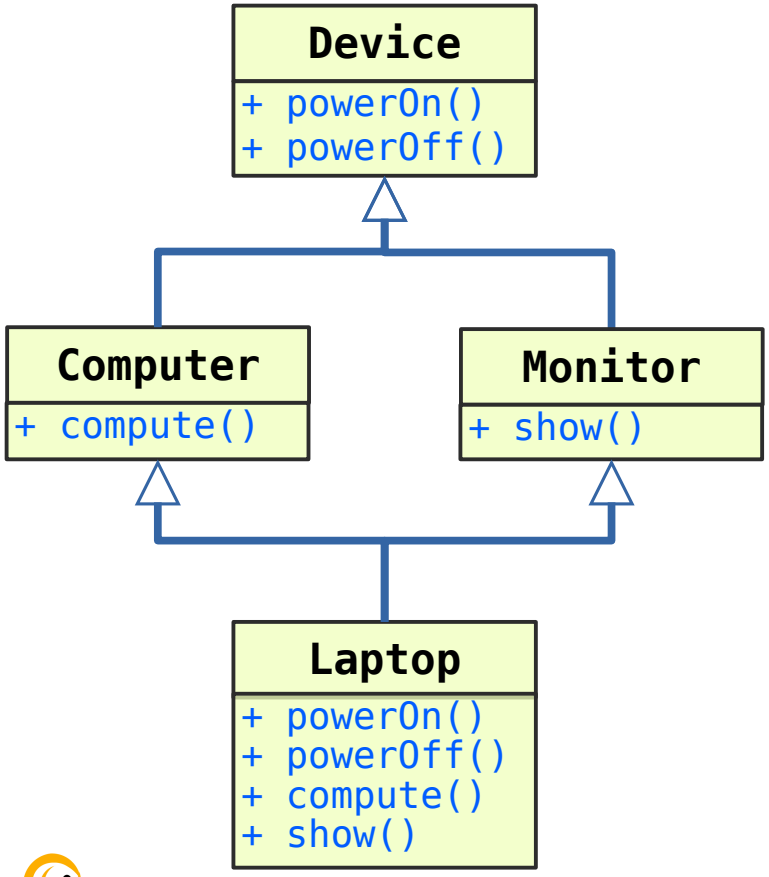
- Класс приобретает свойства обоих родителей
- Состояние от обоих родителей (переменные)
- Поведение от обоих родителей (методы)

Конфликты множественного наследования



- Конфликт имен методов
- Конфликт имен полей
- Сложность разрешения неоднозначностей

"Проблема ромба" (Diamond Problem)



- Класс Laptop наследует от Computer и Monitor
- Computer и Monitor наследуют от Device
- Неоднозначный порядок вызова конструкторов
- Неоднозначность при вызове powerOn() в Laptop

"Проблема ромба" (Diamond Problem)

ИТМО

- Пути решения:
  - MRO (method resolution order) (Python)
  - Виртуальное наследование (C++)
  - Интерфейсы (Java)

Концепция интерфейсов

- Интерфейс - тип данных
- Описывает только поведение без реализации
- Контракт интерфейса определяет, **что** делает класс, но не определяет, **как** он это делает
- У класса может быть единственный класс-предок
- Класс может реализовывать много интерфейсов

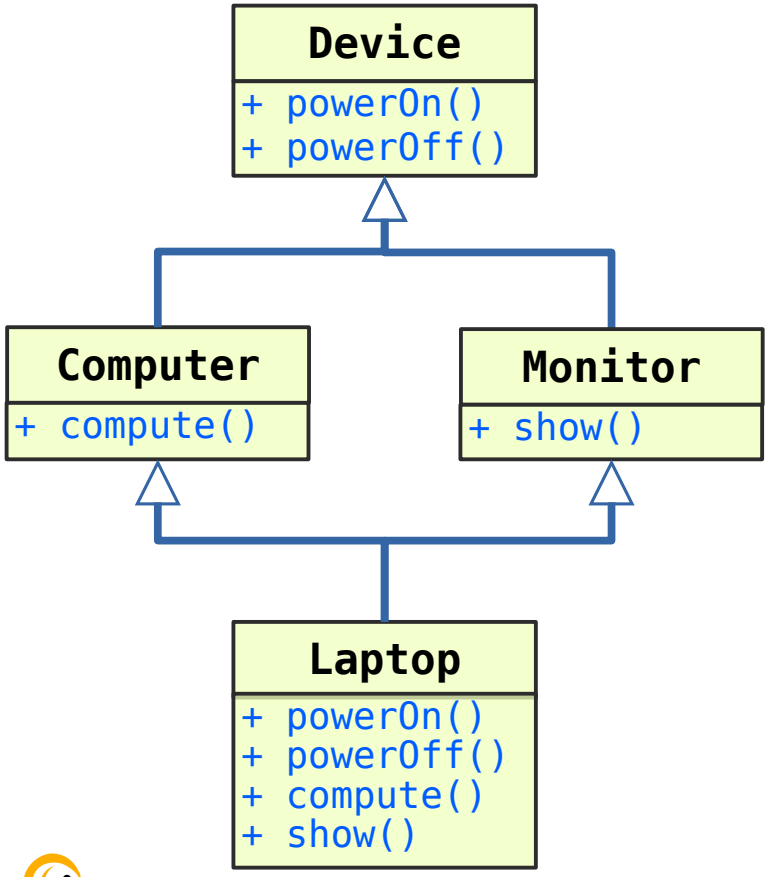
Принципы интерфейсов

- Отсутствие состояния (полей экземпляра)
- Нет конструкторов
- Множественная реализация
- Полная абстракция поведения

Определение интерфейсов

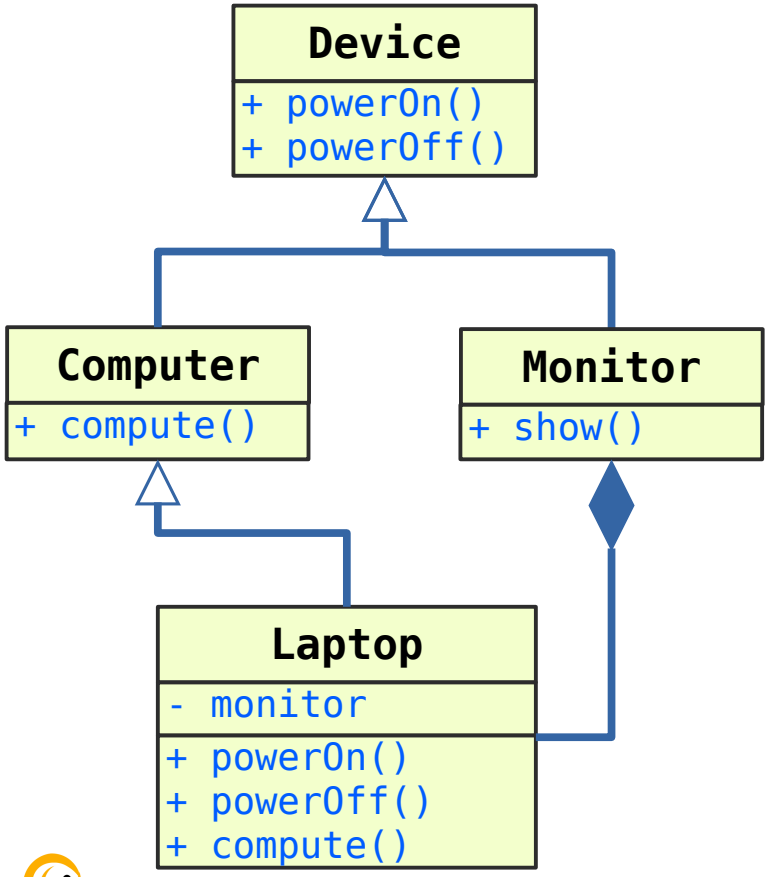
- Ключевое слово interface вместо class
- Все методы по умолчанию public abstract
- Все поля - public static final
- Реализация через implements вместо extends

Разбор ситуации с ноутбуком



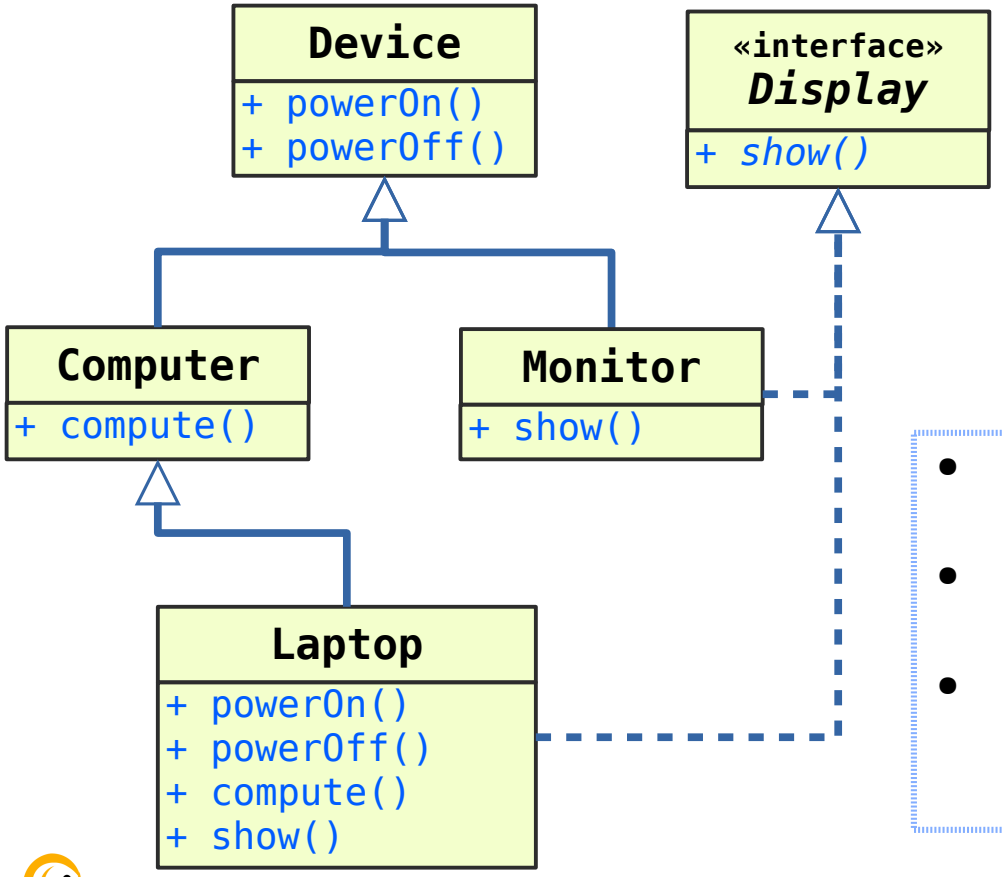
- Компьютер - это устройство?
- Монитор - это устройство?
- Ноутбук - это компьютер?
- Ноутбук - это монитор?

Разбор ситуации с ноутбуком



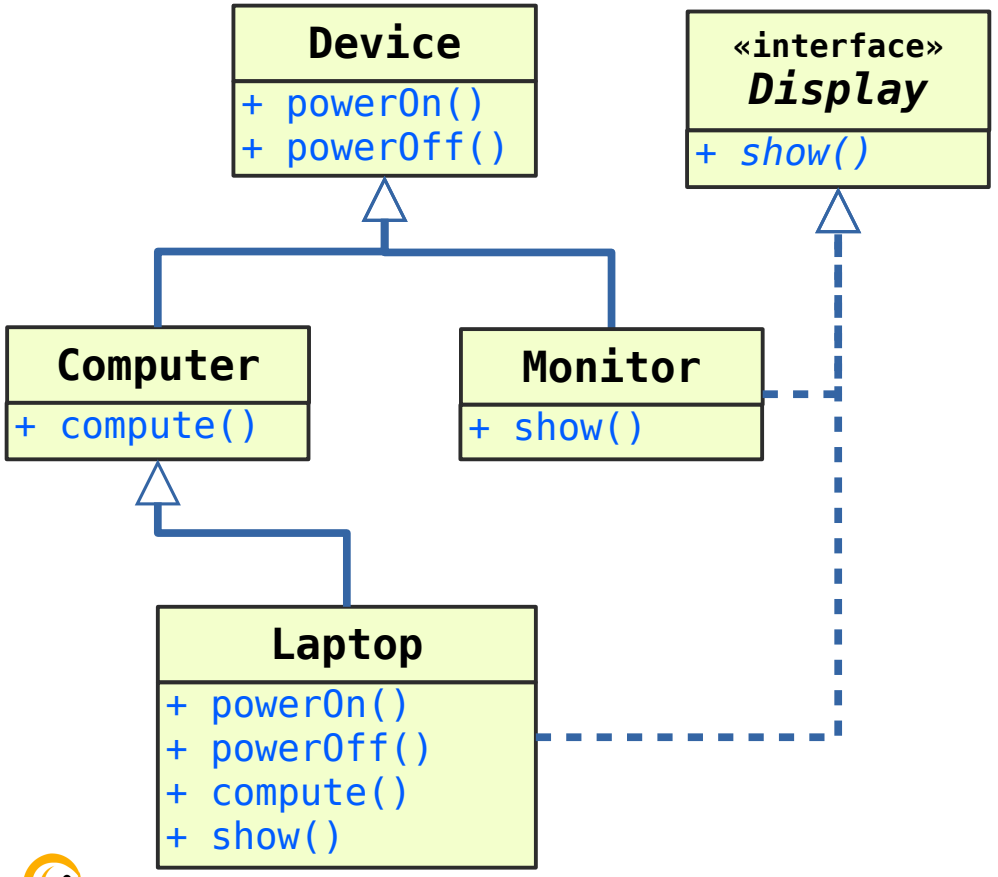
- Компьютер - это устройство?
- Монитор - это устройство?
- Ноутбук - это компьютер?
- Ноутбук - это монитор? - НЕТ!
- Монитор - это часть ноутбука
  - композиция

Реализация через интерфейсы



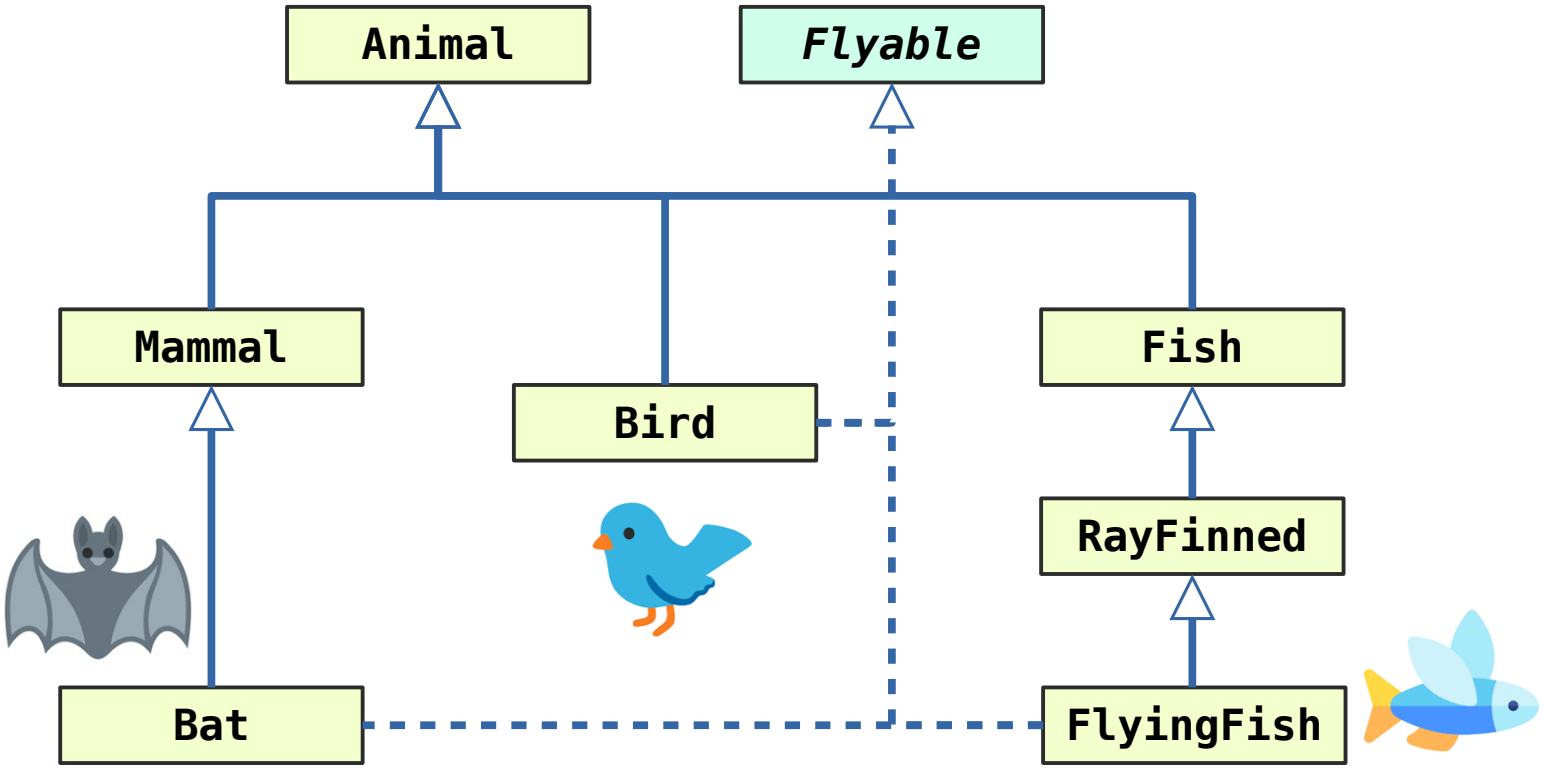
- Монитор отображает картинку
- Ноутбук отображает картинку
- У них общее поведение - интерфейс Display

Реализация через интерфейсы

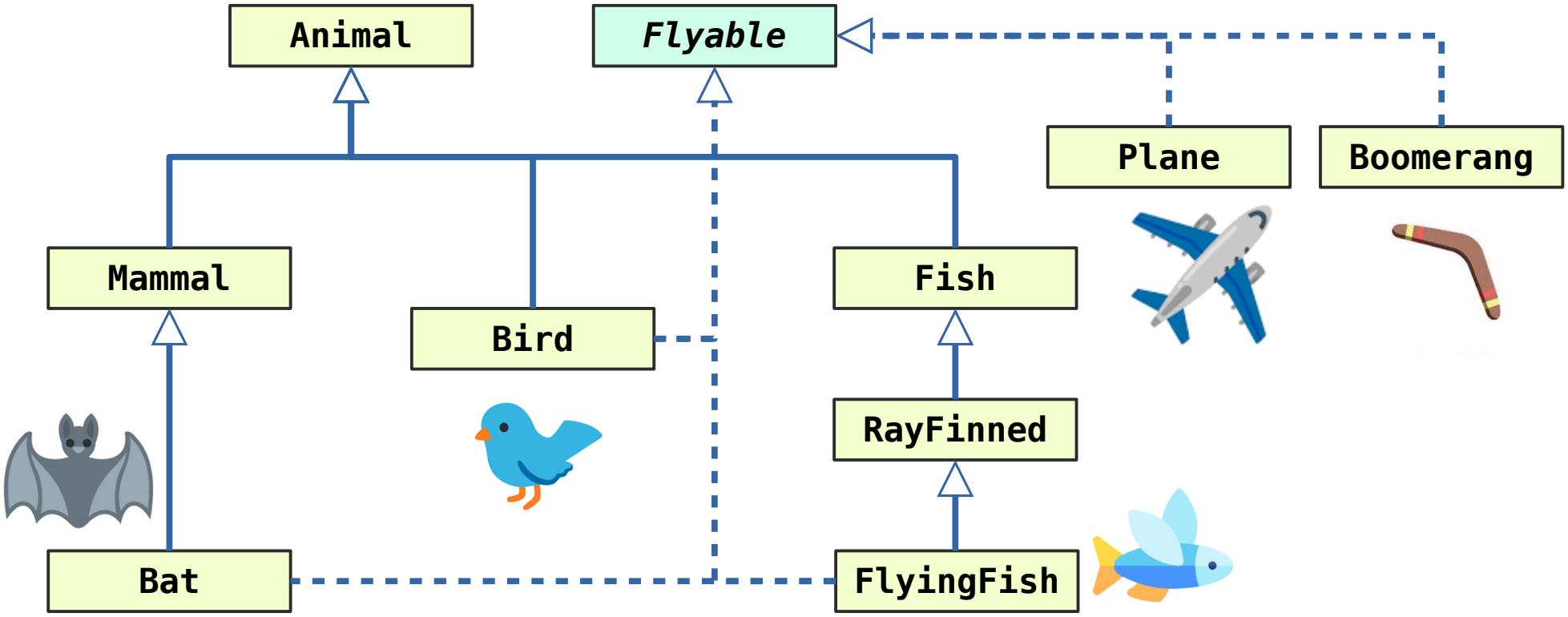



```
public interface Display {
  void show();
}

public class Monitor extends Device
                     implements Display {
 @Override public void show() {
    // код метода
  }
}

public class Laptop extends Computer
                      implements Display {
 @Override public void show() {
    // код метода
  }
}
```

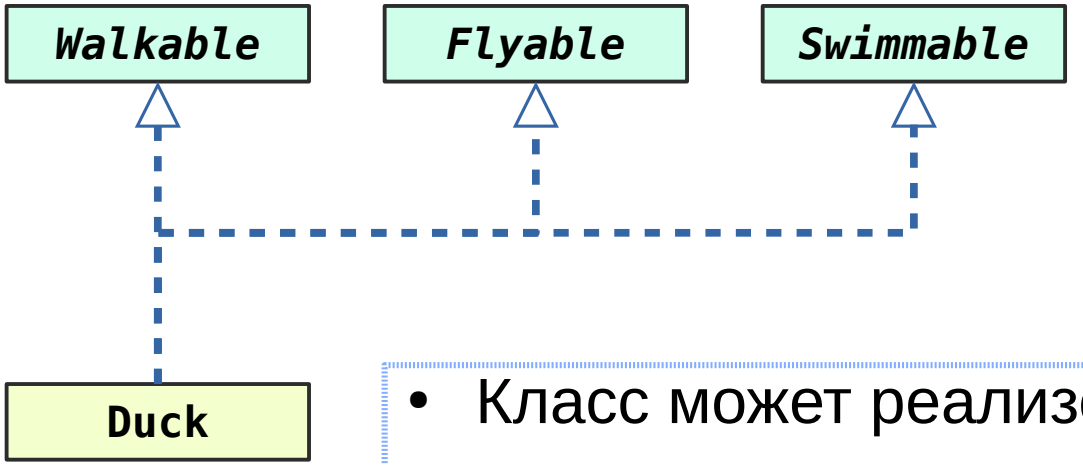
Летающие животные



Летающие животные и не только животные



Множественная реализация интерфейсов



- Класс может реализовать несколько интерфейсов
- interface Swimmable, Flyable, Walkable
- Класс Duck реализует все три интерфейса
- Разные аспекты поведения через разные интерфейсы

Реализация нескольких интерфейсов

- Конфликты в методах
  - Код методов есть только в классе
- Конфликты состояния - переменные экземпляра
  - Состояние есть только в классе
  - В интерфейсе только статические константы

Интерфейс - еще один уровень абстракции

ИТМО

- Program to an interface
  - Меньше зависимости от конкретных классов
  - Проще замена реализации без изменения кода
  - Проще тестирование
  - Проще поддерживать код

Наследование и реализация интерфейсов

ИТМО

- Класс может
  - расширять (extends) один другой класс
  - реализовывать (implements) много интерфейсов
- Интерфейс может расширять (extends) другие интерфейсы

```
class C extends B implements I1, I2, I3 { }

interface I extends I1, I2, I3 { }
```

Методы в интерфейсах

ИТМО

- Абстрактные - основной контракт интерфейса
  - Методы без реализации
  - Обязательны для реализации в классах
  - Неявно public и abstract
- Методы с реализацией по умолчанию (Java 8+)
- Статические методы (Java 8+)
- Приватные методы (Java 9+)

Default методы в интерфейсах

- Ключевое слово default
- Реализация вспомогательного поведения в интерфейсе
- Необязательное переопределение в классах
- Обратная совместимость при изменении интерфейсов
- Если в двух интерфейсах есть default-методы с одинаковым именем - класс обязан их переопределить

Static методы в интерфейсах

- Ключевое слово static
- Принадлежат интерфейсу, а не экземплярам
- Вызов через Interface.method()
- Не наследуются классами реализации
- Реализация утилитарных функций

Private методы в интерфейсах

- Ключевое слово private
- Вспомогательный код для default методов
- Сокрытие внутренней логики
- Устранение дублирования кода

Интерфейсы vs Абстрактные классы

	Интерфейсы	Абстрактные классы
Наследование	Множественное	Единичное
Переменные	только public static final	любые переменные экземпляра
Конструкторы	нет	есть
Методы с кодом	default, static, private	любые
Методы без кода	по умолчанию public abstract	abstract с любым доступом

Когда использовать интерфейсы

- Разные сущности с общим поведением
- Несвязанные иерархии классов
- Несколько аспектов поведения
- Определение контракта для внешних систем

Когда использовать абстрактные классы

ИТМО

- Близкородственные сущности
- Общая базовая реализация
- Общее состояние для всех наследников
- Шаблонный метод (template method pattern)

Динамическое связывание (интерфейсы)

- В классе хранится таблица виртуальных методов vtable
  - При компиляции - в таблицу заносятся методы класса
  - При загрузке класса - методу задается конкретный адрес
  - При выполнении из таблицы берется адрес метода
- У потомков класса ссылка на метод имеет то же смещение
- При реализации интерфейса у ссылки - другое смещение

Инструкции для вызова метода в байткоде

ИТМО

- invokespecial - для конструкторов и приватных методов
- invokestatic - для статических методов
- invokevirtual - для виртуальных методов
- invokeinterface - для методов интерфейса
- invokedynamic - для лямбда-выражений

Стандартные интерфейсы Java

- java.lang.Cloneable для клонирования объктов
- java.lang.Comparable для естественного порядка
- java.util.Comparator для альтернативной сортировки
- java.util.Iterable и Iterator для обхода коллекций

Интерфейс Cloneable

ИТМО

- Не содержит методов - интерфейс-маркер
- Метод protected Object clone() класса Object
  - проверяет, реализован ли интерфейс Cloneable
  - если да, то возвращает копию объекта с копиями полей
  - выполняет "мелкое" (shallow) копирование
  - возвращает копию объекта
  - для непримитивных полей тоже нужно вызывать clone()

Интерфейс Cloneable

ИТМО

```
public class MyClass implements Cloneable {
  private int field;

  public MyClass(int value) {
    field = value;
  }

  public int getField() {
    return field;
  }

  @Override public MyClass clone() {
    return (MyClass) super.clone();
  }
}
```

Интерфейс Comparable

ИТМО

- Естественный порядок сортировки
- Метод x.compareTo(y) возвращает нулевой (x == y), положительный (x > y) или отрицательный (x < y) результат
- Используется для сортировки коллекций
- Определяется разработчиком класса
- Реализован в большинстве стандартных классов
  - String, Integer, Long, Double, LocalDate, LocalTime, ...

Интерфейс Comparable

ИТМО

```
Integer a = 20;
Integer b = 10;

System.out.println(a.compareTo(b));  // 1

String s = "Hello";
String w = "world";

System.out.println(s.compareTo(w));  // -47
```

Сортировка массивов

```
int[] array = { 7, 4, 5, 1, -2, 0};

Arrays.sort(array);
// { -2, 0, 1, 4, 5, 7 }


String[] dow = { "sunday", "monday",
 "tuesday", "wednesday", "thursday",
 "friday", "saturday" };

Arrays.sort(dow);
// { "friday", "monday", "saturday",
// "sunday", "thursday", "tuesday",
// "wednesday" }
```

Интерфейс Comparator

- Альтернативные порядки сортировки
- Метод int compare(T o1, T o2)

Сортировка массивов

ИТМО

```
String[] dow = { "sunday", "monday",
 "tuesday", "wednesday", "thursday",
 "friday", "saturday" };

Arrays.sort(dow);
// { "friday", "monday", "saturday",
// "sunday", "thursday", "tuesday",
// "wednesday" }
```

```
class LenComparator
  implements Comparator<String> {
    public int compare(String s1, String s2) {
      if (s1.length() != s2.length()) {
        return s1.length() - s2.length();
      }
      return s1.compareTo(s2);
    }
}
```

Сортировка массивов

ИТМО

```
String[] dow = { "sunday", "monday",
 "tuesday", "wednesday", "thursday",
 "friday", "saturday" };

Arrays.sort(dow);
// { "friday", "monday", "saturday",
// "sunday", "thursday", "tuesday",
// "wednesday" }

var lc = new LenComparator();
Arrays.sort(dow, lc);
// { "friday", "monday", "sunday",
// "tuesday", "saturday", thursday",
// "wednesday" }
```

```
class LenComparator
  implements Comparator<String> {
    public int compare(String s1, String s2) {
      if (s1.length() != s2.length()) {
        return s1.length() - s2.length();
      }
      return s1.compareTo(s2);
    }
}
```

Вложенные классы

ИТМО

- Объявление класса внутри другого класса
  - Статические вложенные классы
  - Внутренние классы (inner classes)
  - Локальные классы (local classes)
  - Анонимные классы (anonymous classes)

```
class Outer {
  class Nested {
  }
}
```

Статические вложенные классы

ИТМО

- Ключевое слово static
- Доступ только к статическим полям внешнего класса
- Не требует экземпляра внешнего класса
- Логическая группировка классов

```
class Outer {
  static class Static { }
}

var object = new Outer.Static();
System.out.println(object.toString());
```

Внутренние классы (Inner Classes)

ИТМО

- Нестатические вложенные классы
- Доступ ко всем полям внешнего класса
- Связаны с экземпляром внешнего класса

```
class Outer {
  private int field;
  class Inner {
    public int getOuterField() {
      return field;
    }
  }
}
```

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
System.out.println(inner.getOuterField());
```

Локальные классы (Local Classes)

ИТМО

- Объявляются внутри метода
- Видимость только внутри метода
- Не имеет модификаторов

```
class Outer {
  void outerMethod() {
    class Local {
      void innerMethod() { }
    }
  Local local = new Local();
  local.innerMethod();
  }
}
```

Анонимные классы (Anonymous Classes)

ИТМО

- Объявление и создание в одном выражении
- Отсутствие имени класса
- Реализация интерфейсов или наследование классов

```
interface MyInterface { void method(); }

MyInterface intObject = new MyInterface() {
  void method() { System.out.println("Hello!"); }
};

intObject.method();
```

Анонимные классы (Anonymous Classes)

ИТМО

- Объявление и создание в одном выражении
- Отсутствие имени класса
- Реализация интерфейсов или наследование классов

```
System.out.println(new Object() {
  @Override public String toString() {
    return "object of my anonymous class";
  }
});
```

Анонимный компаратор

ИТМО

```
String[] dow = { "sunday", "monday",
 "tuesday", "wednesday", "thursday",
 "friday", "saturday" };


class LenComparator
  implements Comparator<String> {
  public int compare(String s1,
                     String s2) {
    if (s1.length() != s2.length()) {
      return s1.length() - s2.length();
    }
    return s1.compareTo(s2);
  }
}
```

```
var lc = new LenComparator();
Arrays.sort(dow, lc);









// { "friday", "monday", "sunday",
// "tuesday", "saturday", thursday",
// "wednesday" }
```

Анонимный компаратор

ИТМО

```
String[] dow = { "sunday", "monday",
 "tuesday", "wednesday", "thursday",
 "friday", "saturday" };


class LenComparator
  implements Comparator<String> {
  public int compare(String s1,
                     String s2) {
    if (s1.length() != s2.length()) {
      return s1.length() - s2.length();
    }
    return s1.compareTo(s2);
  }
}
```

```
Arrays.sort(dow, new Comparator<String> {
  public int compare(String s1,
                     String s2) {
    if (s1.length() != s2.length()) {
      return s1.length() - s2.length();
    }
    return s1.compareTo(s2);
  }
});

// { "friday", "monday", "sunday",
// "tuesday", "saturday", thursday",
// "wednesday" }
```

Сравнение типов вложенных классов

ИТМО

- Статические
  - доступ к static полям, независимы от экземпляра
- Внутренние
  - доступ ко всем полям, связаны с экземпляром
- Локальные
  - видимость в методе
- Анонимные
  - одноразовое использование, компактный синтаксис

Класс ArrayList

ИТМО

- Массив
  - любые типы данных
    - примитивные, массивы, классы, интерфейсы, ...
  - фиксированный размер
    - задается при создании и больше не меняется
- java.util.ArrayList
  - только ссылочные типы данных (НЕ ПРИМИТИВНЫЕ)
  - динамический размер (может увеличиваться)
  - Быстрый произвольный доступ по индексу
  - Медленные вставки/удаления в середине

Создание и базовые операции ArrayList

- new ArrayList<>() - создание пустого списка
- add(element) - добавление элемента
- get(index) - доступ по индексу
- set(index, element) - замена элемента
- remove(index) - удаление по индексу

Создание и базовые операции ArrayList

```
String[] array = new String[3];

array[0] = "A";
System.out.println(array[0]);

array[2] = "D";




System.out.println(array.length);
System.out.println(array);
```

```
var list = new ArrayList<String>();

list.add("A");
System.out.println(list.get(0));

list.set(2, "D");

list.add("B"); list.add(0,"C");
list.add("D"); list.add(1,"E");
list.remove(0);
System.out.println(list.size());
System.out.println(list);
```

Метод List.of

```
int[] array = {1,2,3};

array[3] = 4;
```

```
var list = List.of(1,2,3);

list.add(4);
```

Метод List.of

```
int[] array = {1,2,3};

array[3] = 4;
```

```
var list = List.of(1,2,3);

list.add(4);

var list2 = new ArrayList(
            List.of(1,2,3));

list2.add(4);
```

Метод List.of

```
int[] array = {1,2,3};

array[3] = 4;




for (var i : array) {
  System.out.println(i);
}
```

```
var list = List.of(1,2,3);

list.add(4);

var list2 = new ArrayList(
            List.of(1,2,3));

list2.add(4);

for (var i : list2) {
  System.out.println(i);
}
```

Продвинутые операции с ArrayList

- addAll(collection) - добавление коллекции
- subList(from, to) - получение представления
- toArray() - преобразование в массив
- Collections.sort() - сортировка элементов

Консольный ввод-вывод

ИТМО

- java.io.Console
  - Console console = System.console()
  - console.readLine()
  - console.readPassword()
  - console.printf()
  - console.format()

Базовое использование Scanner

ИТМО

- Чтение данных из System.in
- Методы nextLine(), nextInt(), nextDouble()
- Проверка наличия данных: hasNext(), hasNextInt()
- Закрытие scanner.close() для освобождения ресурсов

Обработка разных типов данных

- nextLine() - чтение строки до конца линии
- nextInt() - чтение целого числа
- nextDouble() - чтение числа с плавающей точкой
- nextBoolean() - чтение логического значения

Scanner

ИТМО

```
Scanner scanner = new Scanner(System.in);
List<List<Integer>> ints = new ArrayList<>();
List<Integer> intList = null;
while (scanner.hasNext()) {
  String line = scanner.nextLine();
  Scanner intScanner = new Scanner(line);
  intList = new ArrayList<Integer>();
  while (intScanner.hasNextInt()) {
    intList.add(intScanner.nextInt());
  }
  ints.add(intList);
  intScanner.close();
}
```

```
100 200 300 400 500
1 2 3 4 5 6 7 8 9
-1 -2 -3 -4 -5
9 8 7 6 5 4 3 2 1
45 0 -45
```

Чтение из файла с Scanner

ИТМО

- Scanner scanner = new Scanner(new File("file.txt"))
- Построчное чтение с помощью hasNextLine() и nextLine()
- Обработка FileNotFoundException
- Закрытие Scanner для освобождения ресурсов

Классы для даты и времени (Java 8+)

ИТМО

- LocalDate - работа с датами
- LocalTime - работа с временем
- LocalDateTime - дата и время вместе
- Period и Duration - промежутки времени

LocalDate - работа с датами

ИТМО

- now() - текущая дата
- of(year, month, day) - создание конкретной даты
- parse("yyyy-MM-dd") - разбор из строки
- plusDays(), plusMonths(), plusYears() - операции с датами

LocalTime - работа с временем

ИТМО

- now() - текущее время
- of(hour, minute, second) - создание конкретного времени
- plusHours(), plusMinutes(), plusSeconds() - операции со временем
- getHour(), getMinute(), getSecond() - получение компонентов

LocalDateTime - дата и время

ИТМО

- Комбинация LocalDate и LocalTime
- now() - текущие дата и время
- of(date, time) - создание из даты и времени
- toLocalDate(), toLocalTime() - получение компонентов

Period и Duration

ИТМО

- Period - промежуток между датами (дни, месяцы, годы)
- Duration - промежуток между временами (часы, минуты, секунды)
- between() - вычисление промежутка между двумя точками
- ofDays(), ofHours() - создание промежутков

Лаба 3

ИТМО

- Первый этап - создать простой сценарий
- Сказка Репка
- Классы?

Лаба 3

ИТМО

- Первый этап - создать простой сценарий
- Сказка Репка
- Классы
  - Vegetable
  - Farmer
  - TurnipStory

Vegetable

- class Vegetable
  - double weight;
  - boolean inSoil()
  - методы
    - void plant()
    - void grow()
    - void water()
    - double harvest (double strength)

Farmer

- class Farmer
  - String name;
  - double strength;
  - Vegetable[] field;
  - методы
    - void buy()
    - void plant()
    - void water()
    - void harvest()
    - void sell()

- Farmer ded = new Farmer("Кузьма", 20.0);
- Vegetable repka = new Vegetable("репа");
- ...

Вопросы?

Регулярные выражения (Regex)

- Паттерны для поиска и обработки текста
- Синтаксис на основе специальных символов
- Использование в валидации данных
- Применение в поиске и замене текста

Синтаксис регулярных выражений

ИТМО

- Литералы - точное совпадение символов
- Классы символов - [abc], [a-z], [0-9]
- Метасимволы - ., \d, \w, \s
- Квантификаторы - *, +, ?, {n,m}

Регулярные выражения

ИТМО

подстрока символов

```
in
```

начало и конец строки

```
^lo
```

```
ua$
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

символьный класс

```
[bp]or
i[^dtns]
e[l-n]
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

символьный класс -

сокращенное обозначение

```
\w\w\wt\W
```

\d	цифра	\D	не цифра
\w	буква	\W	не буква
\s	пробел	\S	не пробел

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

любой символ

```
...c
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor_
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

альтернатива

```
s(i|ec)t
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

квантификаторы

```
\s.{6}\s

e.{0,2}i

s\S{0,1}m
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

квантификаторы

```
s\S?m    {0,1}
b.+d      {1,}
et[^o]*,   {0,}
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

жадность

```
lor.*it

\si\w*?i
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

группы

```
(\w+).*\1,
```

1 = it

```
\w+@(\w+\.)+\w{2,}

user@se.itmo.ru
```

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Регулярные выражения

ИТМО

группы

```
(\w)..\1(.).{1,6}\2
```

1 = i 2 = d

1 = t 2 = _

```
lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed
do eiusmod tempor
incididunt ut labore
et dolore magna
aliqua
```

Синтаксис регулярных выражений

ИТМО

- Класс Pattern — представляет регулярное выражение
- Класс Matcher — движок, проверяющий соответствие

```
String regex = "a*b+";
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher("aaabbb");
boolean b = m.matches();  // true

boolean b = Pattern.matches(regex, "aaabbb"); // true
```

Поиск и извлечение данных

- Pattern.compile() - компиляция регулярного выражения
- Matcher.find() - поиск совпадений в тексте
- Группы захвата для извлечения частей текста
- Итерация по всем совпадениям

Замена текста с помощью regex

- replaceAll() - замена всех совпадений
- replaceFirst() - замена первого совпадения
- Использование групп в замене
- Форматирование данных