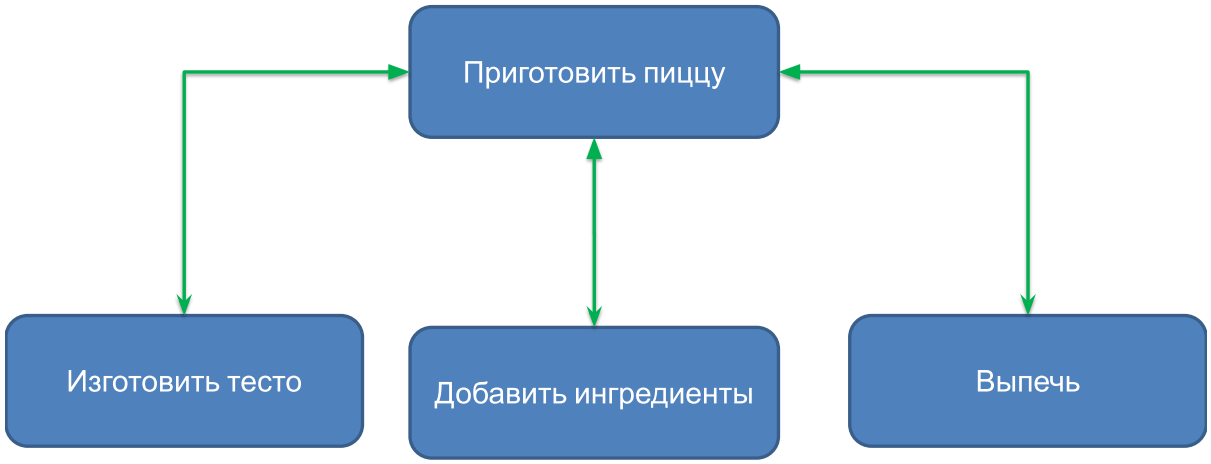
OOP. Classes and Objects

Intro

Problem

Структурное программирование

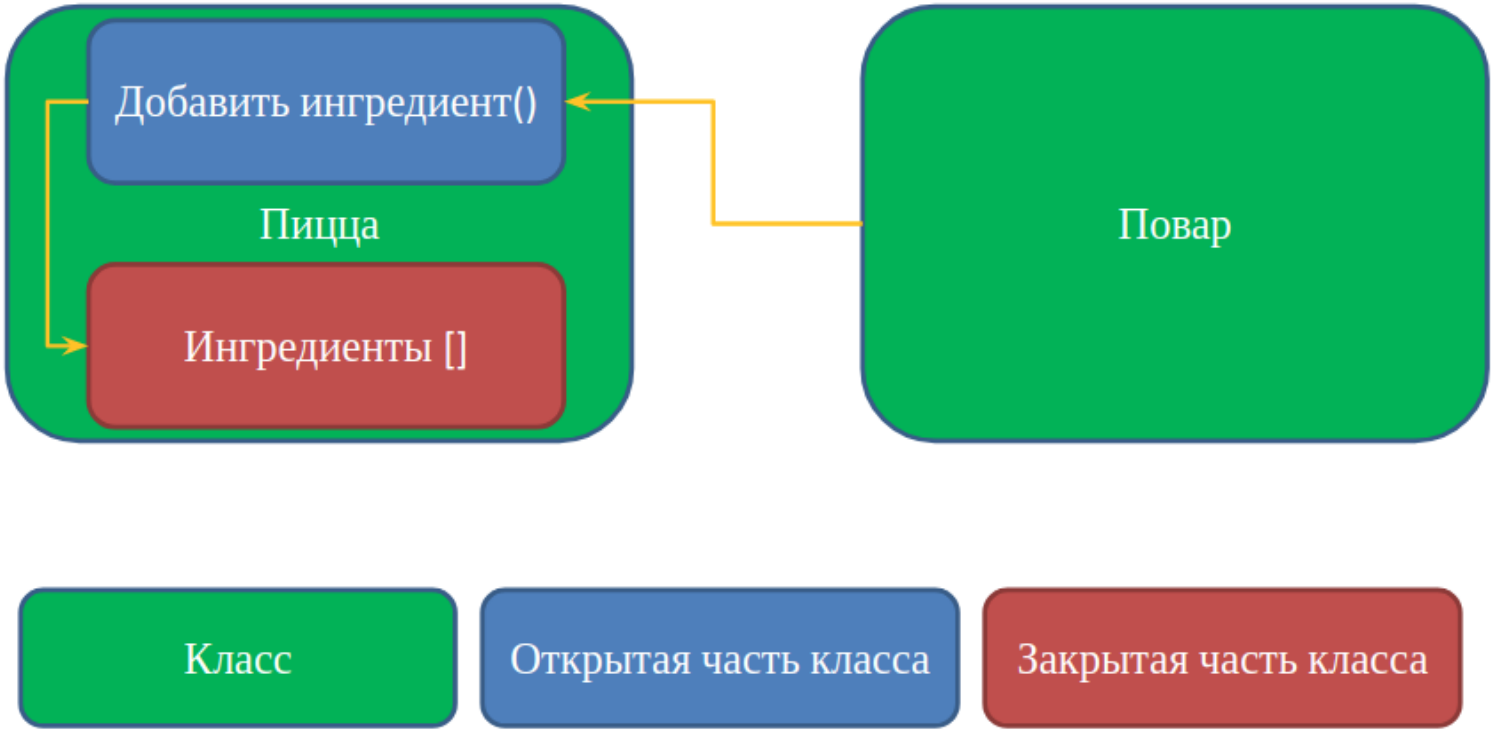
Принцип: код воздействует на данные



Solution

Объектно-ориентированное программирование

Принцип: данные управляют доступом к коду



Classes and objects

OOP concepts

- **ООП —** это парадигма программирования, в которой главные понятия **объекты** и **классы**.
- **ООП** возникло в результате развития идей процедурного программирования, где **данные** и **функции** (**методы**) их обработки формально не связаны.

Classes and objects

- **Класс** (**Class**)- это *шаблон* для создания **объектов**
- **Объект** (**Object**) - это *экземпляр* **класса**
- *Функция*, созданная внутри класса, называется **метод** (**method**)
- *Переменная*, созданная внутри класса, называется **поле** (**field**)

Class definition: syntax

Класс определяет *структуру* и *поведение* **объектов**

```
class Person {
    // class content
}
```

Objects

- **Объект –** любой предмет, четко структурированный и имеющий смысл в контексте решаемой предметной области.
- Все **объекты** имеют одинаковые **наборы полей** данных (**атрибуты объекта**), но с независимыми значениями этих данных для каждого объекта.
- Значения **полей** данных объекта задают его **состояние**.
- **Методы** объекта задают его **поведение**.

Objects

- Объекты безымянны, и доступ к ним осуществляется только через **ссылочные переменные**.

```
User tom = new User();
```

Понятие данных

Данные – члены класса, которые называются **полями** или **переменными класса**, объявляются в классе следующим образом:

```
модификатор тип имя;
```

```
public int age;
```

Модификаторы

- Модификаторы доступа:
  - `public`
  - `private`
  - `protected`
- Специализированные модификаторы:
  - `final`
  - `static`
  - `synchronized`
  - `native`

Метод

- **Метод** – это обособленный блок кода.

```
спецификатор возвращаемый-тип идентификатор-метода(параметры) {
    тело-метода
}
```

```
public class Human {
    private double temperature; // поле класса

    public boolean isIll() { // метод
        return temperature > 37.0 || temperature < 36.2;
    }
}
```

Области видимости переменных

- **Область видимости** — часть текста программы, на протяжении которого к объекту можно обращаться по его имени.
- В языках программирования выделяют:
  - **глобальную** области видимости, если объявление происходит вне любой функции или блока кода и доступно в любой точке программы.
  - **локальную** области видимости, если оно объявлено в теле функции или в пространстве имен. **Область видимости**: от объявления и до окончания блока кода.

Области видимости переменных

- Основные **области видимости** в Java:
  - класс (глобальная)
  - метод (локальная)
  - блоки кода (локальная)

Class definition: example

```
class Person {
    String name;
    int age;

    void displayInfo() {
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

```
class Program {
    public static void main(String[] args) {
        Person tom;
    }
}
```

Constructor

Constructor (Конструктор)

- **Конструктор** и **метод** внешне похожи.
- **Конструктор** имеет имя как у **класса**.
- В **конструкторе** не должно быть лишней логики.
- У конструкторов нет типа возвращаемого результата.
- Если **конструктор** не указан – компилятор создаст **конструктор по умолчанию**.
- Если создали свой **конструктор – конструктор по умолчанию** не создаётся.

Default Constructor: example

```
class Person {
    String name;
    int age;

    void displayInfo() {
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Default Constructor: example

```
public class Program {
    public static void main(String[] args) {
        Person tom = new Person();
        tom.displayInfo();
        tom.name = "Tom";
        tom.age = 34;
        tom.displayInfo();
    }
}
```

Constructors: example

```
class Person {
    String name;
    int age;

    Person() {
        name = "Undefined";
        age = 18;
    }

    Person(String n) {
        name = n;
        age = 18;
    }

    Person(String n, int a) {
        name = n;
        age = a;
    }

    void displayInfo() {
```

Constructors: example

```
public class Program {
    public static void main(String[] args) {
        Person bob = new Person();
        bob.displayInfo();
        Person tom = new Person("Tom");
        tom.displayInfo();
        Person sam = new Person("Sam", 25);
        sam.displayInfo();
    }
}
```

Keyword `this`

Keyword `this`: example

```
class Person {
    String name;
    int age;

    Person() {
        this("Undefined", 18);
    }

    Person(String name) {
        this(name, 18);
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void displayInfo() {
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
```

Keyword `this`: example

```
public class Program {
    public static void main(String[] args) {
        Person undef = new Person();
        undef.displayInfo();
        Person tom = new Person("Tom");
        tom.displayInfo();
        Person sam = new Person("Sam", 25);
        sam.displayInfo();
    }
}
```

Initialization Block (блок инициализации)

Initialization Block

- При описании класса могут быть использованы **блоки инициализации**.
- **Блоком инициализации** называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса.

```
{ /* код */ }
```

Initialization Block

- **Блоки инициализации** чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов как текущего класса, так и не принадлежащих ему.
- При создании объекта класса **блоки инициализации** вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего **блока инициализации** будет вызван **конструктор** класса.

Initialization Block

- Операции с полями класса внутри **блока инициализации** до явного объявления этого поля возможны только при использовании ссылки `this`, представляющую собой ссылку на текущий объект.
- **Блок инициализации** может быть объявлен со спецификатором `static`. В этом случае он вызывается только один раз в жизненном цикле приложения при создании объекта или при обращении к статическому методу (полю) данного класса.

Example

```
class Person {
    String name;
    int age;

    {
        this.name = "Undefined";
        this.age = 18;
    }

    Person() {
    }

    Person(String name) {
        this.name = name;
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

Example

```
public class Program {
    public static void main(String[] args) {
        Person undef = new Person();
        undef.displayInfo();
        Person tom = new Person("Tom");
        tom.displayInfo();
    }
}
```

How does it work?

```
public class Init {
    {
        System.out.println("initializer (1), id=" + this.id);
    }

    private int id = 42;

    public Init(int d) {
        id = d;
        System.out.println("constructor, id=" + id);
    }

    {
        System.out.println("initializer (2), id=" + this.id);
    }

    static {
        System.out.println("static initializer");
    }
```

How does it work?

```
public class Example1 {
    public static void main(String[] args) {
        Init obj = new Init(7);
        System.out.println("value for id=" + obj.getId());
    }
}
```

```
static initializer
initializer (1), id=0
initializer (2), id=42
initializer (3), id=10
constructor, id=7
value for id=7
```

Objects as parameters of methods

Objects as parameters of methods: example

```
class Person {
    private String name;

    Person(String name) {
        this.name = name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

Objects as parameters of methods: example

```
public class Program {
    public static void main(String[] args) {
        Person kate = new Person("Kate");
        System.out.println(kate.getName());
        changeName(kate);
        System.out.println(kate.getName());
    }

    static void changeName(Person p) {
        p.setName("Alice");
    }
}
```

Objects as parameters of methods: example 2

```
class Person {
    private String name;

    Person(String name) {
        this.name = name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

Objects as parameters of methods: example 2

```
public class Program {
    public static void main(String[] args) {
        Person kate = new Person("Kate");
        System.out.println(kate.getName());
        changePerson(kate);
        System.out.println(kate.getName());
    }

    static void changePerson(Person p) {
        p = new Person("Alice");
        p.setName("Ann");
    }

    static void changeName(Person p) {
        p.setName("Alice");
    }
}
```

Packages

Packages

- Для логического группирования множеств классов в связанные группы в Java применяется понятие **пакета** (**package**).
- **Пакеты** обеспечивают:
  - независимые пространства имён (**namespaces**)
  - ограничение доступа к классам
- **Пакеты** — это фактически обычная директория.

Package definition: syntax

```
package your.package.which.can.has.any.name;
```

Package definition: example

```java
package com.rakovets;

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void displayInfo() {
        System.out.printf("Name: %s \t Age: %d \n", name, age);
    }
}
```

Package definition: example

```java
package com.rakovets;

public class Program {
    public static void main(String[] args) {
        Person kate = new Person("Kate", 32);
        kate.displayInfo();
    }
}
```

Packages and Terminal: example

```
cd D:\home\rakovets\dev
javac com\rakovets\Program.java
java com.rakovets.Program
```

`import` Packages and Classes: example

```
package com.rakovets;

import java.util.Scanner;
import java.util.*;

public class Program {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
    }
}
```

`import` Packages and Classes: example

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date();
```

Access modifiers (Модификаторы доступа)

Access modifiers

- `public` - доступно из любого места, но чаще всего для внешнего интерфейса.
- `protected` - внутри пакета и в дочерних классах.
- *friendly/default/package* - доступно внутри пакета.
- `private` - доступно только внутри класса – для скрытия реализации (инкапсуляции).

Access modifiers

	`private`	*friendly*	`protected`	`public`
same class	+	+	+	+
same package subclass	-	+	+	+
same package non-subclass	-	+	+	+
different package subclass	-	-	+	+
different package non-subclass	-	-	-	+

Access modifiers

```
class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Bad practice.

Access modifiers

```
public class Program {
    public static void main(String[] args) {
        Person kate = new Person("Kate", 30);
        System.out.println(kate.age);
        kate.age = 33;
        System.out.println(kate.age);
    }
}
```

Bad practice.

Access modifiers

Good practice.

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return this.age;
    }
```

Access modifiers

Good practice.

```
public class Program {
    public static void main(String[] args) {
        Person kate = new Person("Kate", 30);
        System.out.println(kate.getAge());
        kate.setAge(33);
        System.out.println(kate.getAge());
    }
}
```

Modifiers (Модификаторы)

Modifiers `final`

Модификатор `final` (*неизменяемый*) может применяться к классам, методам и переменным.

```
final double PI = 3.14; // константы
```

```
final void run() {} // запрещено переопределение метода
```

```
final class Example {} // запрещено наследование
```

Modifiers `native`

- Приложение на языке Java может вызывать методы, написанные на языке C++. Такие методы объявляются с ключевым словом `native`, которое сообщает компилятору, что метод реализован в другом месте.

```
public native int loadCripto(int num);
```

- Методы, помеченные `native`, можно переопределять обычными методами в подклассах.

Modifiers `synchronized`

- При использовании нескольких потоков управления в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным.
- Когда интерпретатор обнаруживает `synchronized`, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении.

Modifiers `static`

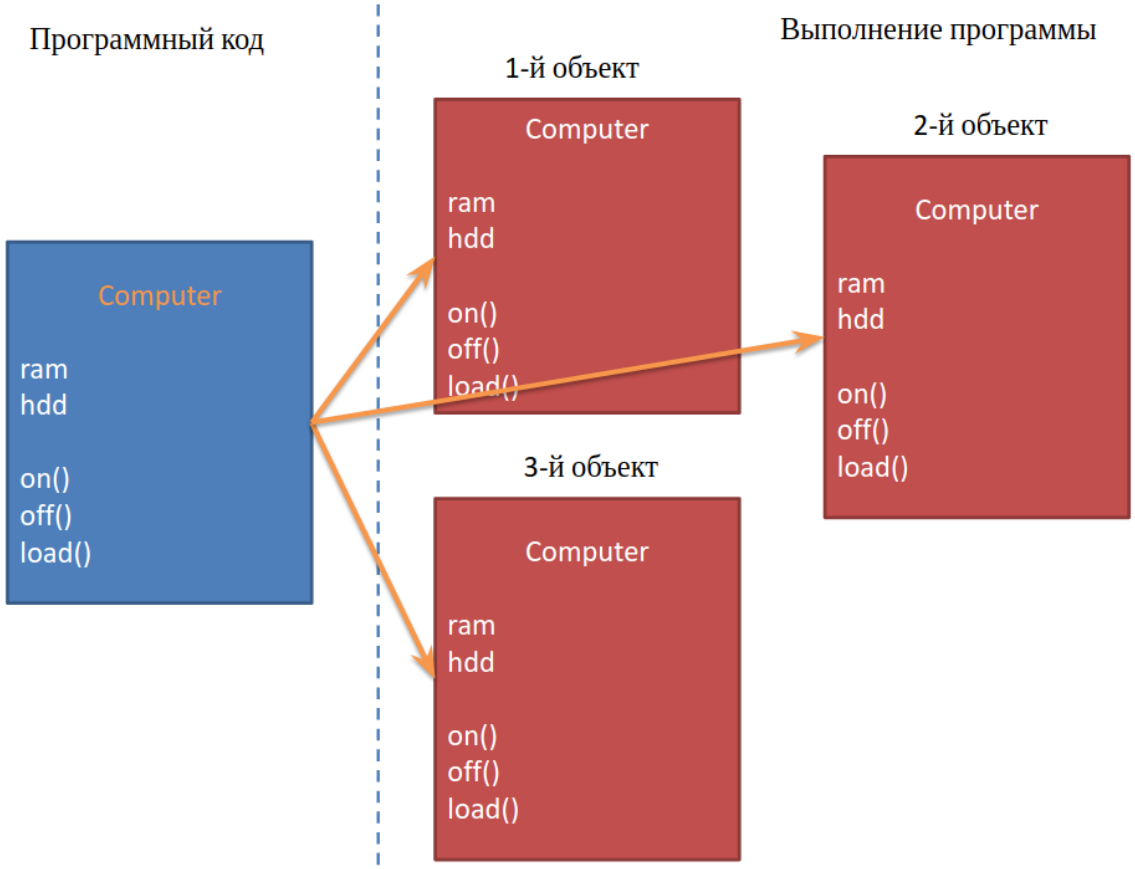
Ключевое слово `static`

- Модификатор `static` (единственный) применяется к методам, переменным и логическим блокам.
- Поля данных, объявленные в классе как `static`, являются общими для всех объектов класса и называются **переменными класса**.
- Для работы со статическими атрибутами используются статические методы, объявленные со спецификатором `static` являются **методами класса**.
- Не привязаны ни к какому объекту.

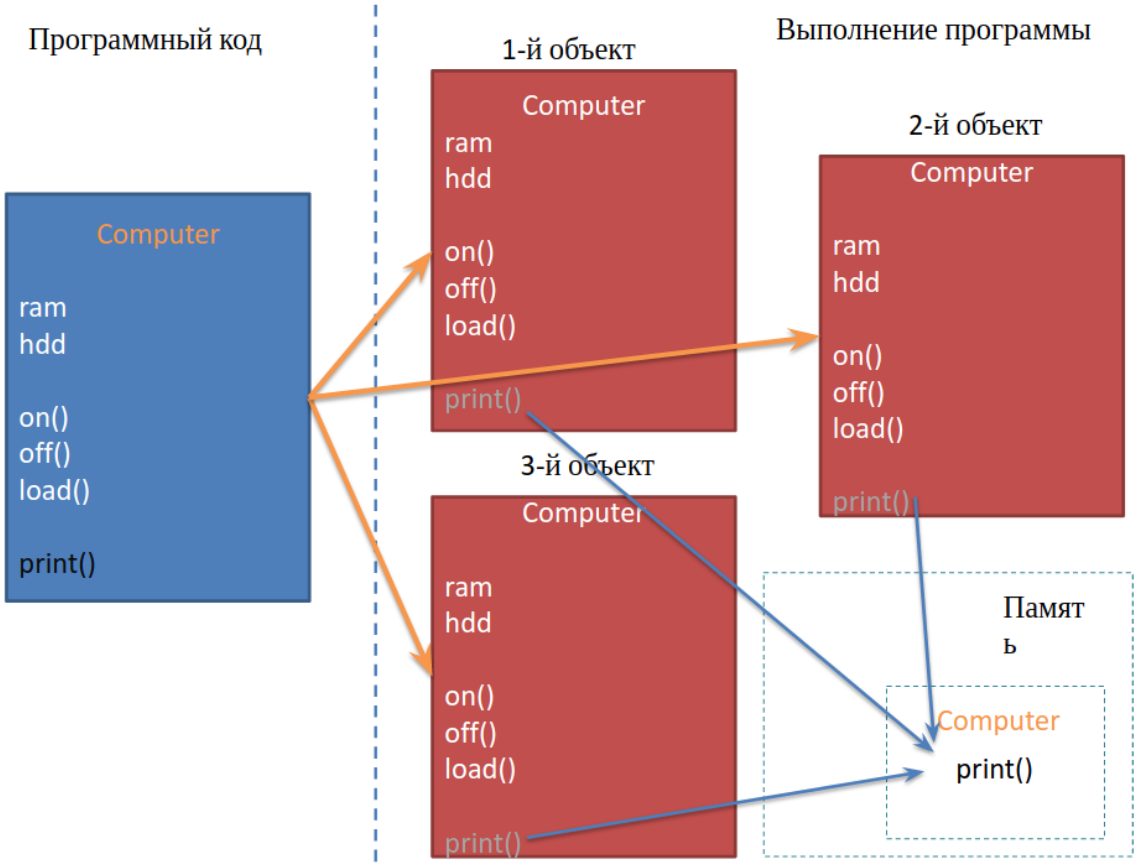
Ключевое слово `static`

- Не содержат указателя `this` на конкретный объект, вызвавший метод.
- Реализует парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции.
- Статические поля и методы не могут обращаться к нестатическим полям и методам напрямую по причине недоступности указателя this.
- Для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

non static


Программный код
Выполнение программы
1-й объект
2-й объект
3-й объект
Computer
ram
hdd
on()
off()
load()
Computer
ram
hdd
on()
off()
load()
Computer
ram
hdd
on()
off()
load()
Computer
ram
hdd
on()
off()
load()

static


Программный код
1-й объект
Выполнение программы
Computer
ram
hdd
on()
off()
load()
print()
Computer
ram
hdd
on()
off()
load()
print()
2-й объект
Computer
ram
hdd
on()
off()
load()
print()
3-й объект
Computer
ram
hdd
on()
off()
load()
print()
Память
Computer
print()

Example

```
public static void main(String[] args) {
    // statements
}
```

`static` fields: example

```
class Person {
    private int id;
    static int counter = 1;

    Person() {
        id = counter++;
    }

    public void displayId() {
        System.out.printf("Id: %d \n", id);
    }
}
```

`static` fields: example

```
public class Program {
    public static void main(String[] args) {
        Person tom = new Person();
        Person bob = new Person();
        tom.displayId();
        bob.displayId();
        System.out.println(Person.counter);
        Person.counter = 8;
        Person sam = new Person();
        sam.displayId();
    }
}
```

`static` constants: example

```
public class Program {
    public static void main(String[] args) {
        double radius = 60;
        System.out.printf("Radisu: %f \n", radius);
        System.out.printf("Area: %f \n", Math.PI * radius);
    }
}

public class Math {
    public static final double PI = 3.14;
}
```

`static` methods: example

```
public class Operation {
    static int sum(int x, int y) {
        return x + y;
    }

    static int subtract(int x, int y) {
        return x - y;
    }

    static int multiply(int x, int y) {
        return x * y;
    }
}
```

`static` methods: example

```
public class Program {
    public static void main(String[] args) {
        System.out.println(Operation.sum(45, 23));
        System.out.println(Operation.subtract(45, 23));
        System.out.println(Operation.multiply(4, 23));
    }
}
```

`static` initializers: example

```
class Person {
    private int id;
    static int counter;

    static {
        counter = 105;
        System.out.println("Static initializer");
    }

    Person() {
        id = counter++;
        System.out.println("Constructor");
    }

    public void displayId() {
        System.out.printf("Id: %d \n", id);
    }
}
```

`static` initializers: example

```
public class Program {
    public static void main(String[] args) {
        Person tom = new Person();
        Person bob = new Person();
        tom.displayId();
        bob.displayId();
    }
}
```

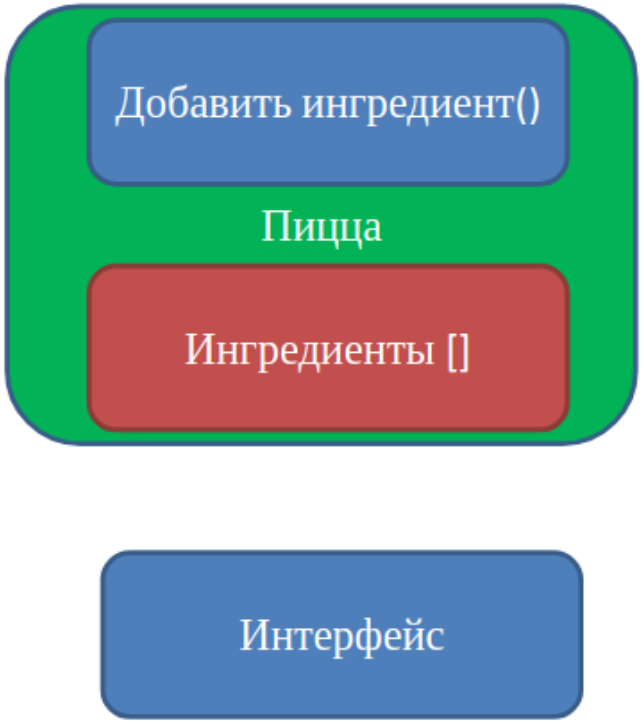
Static `import`: example

```
package study;

import static java.lang.System.*;
import static java.lang.Math.*;

public class Program {
    public static void main(String[] args) {
        double result = sqrt(20);
        out.println(result);
    }
}
```

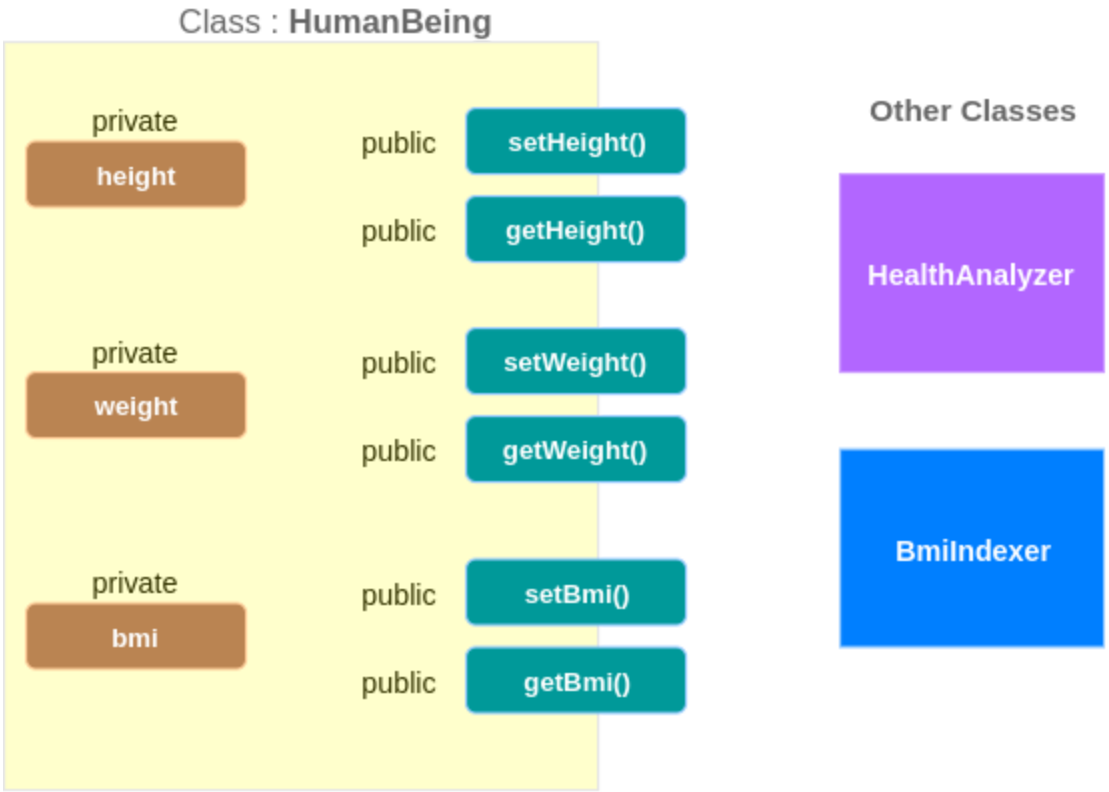
Interface

Interface (Интерфейс)

Открытая часть класса, с помощью которой другие классы могут с ним взаимодействовать

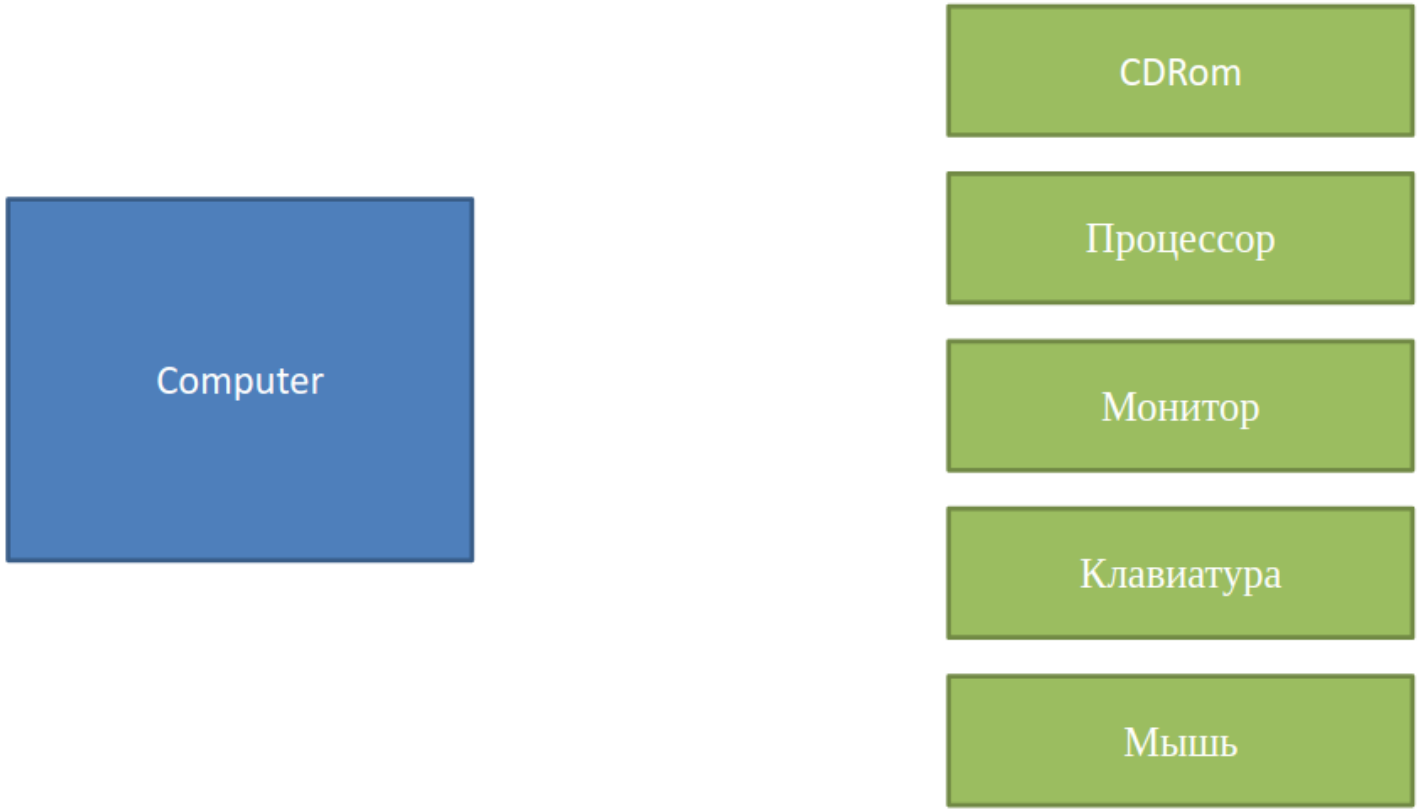


Interface

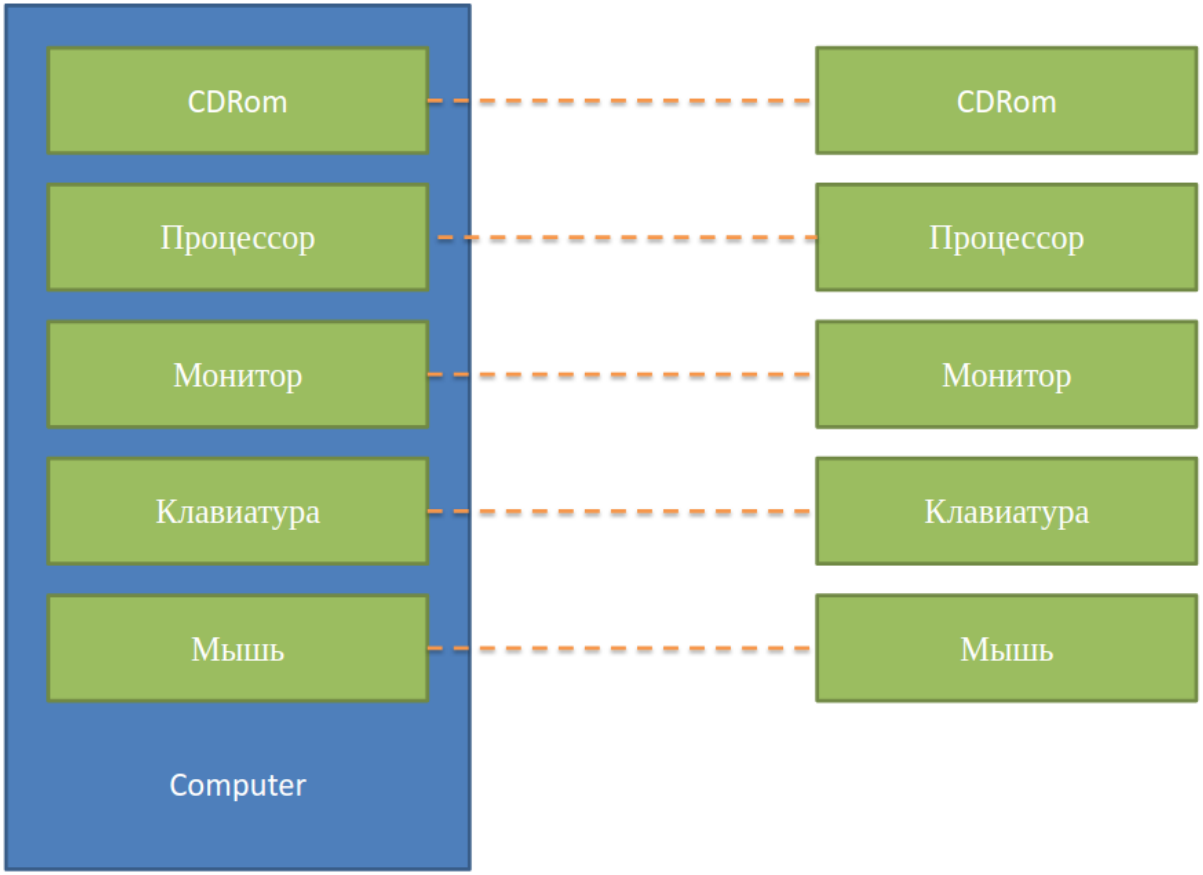

Class : HumanBeing
private
height
public
setHeight()
public
getHeight()
private
weight
public
setWeight()
public
getWeight()
private
bmi
public
setBmi()
public
getBmi()
Other Classes
HealthAnalyzer
BmiIndexer

Composition

Composition (Композиция)



Composition



Total

Нужно ли всегда создавать объекты?

- Даже если программа простейшая – всегда нужно создавать объекты и писать код в стиле ООП
- Это должно быть привычкой
- В программе не должно быть лишних объектов
- Никогда не давайте объекту чужие понятия и действия