

OOP. Principles.

Intro

Problem

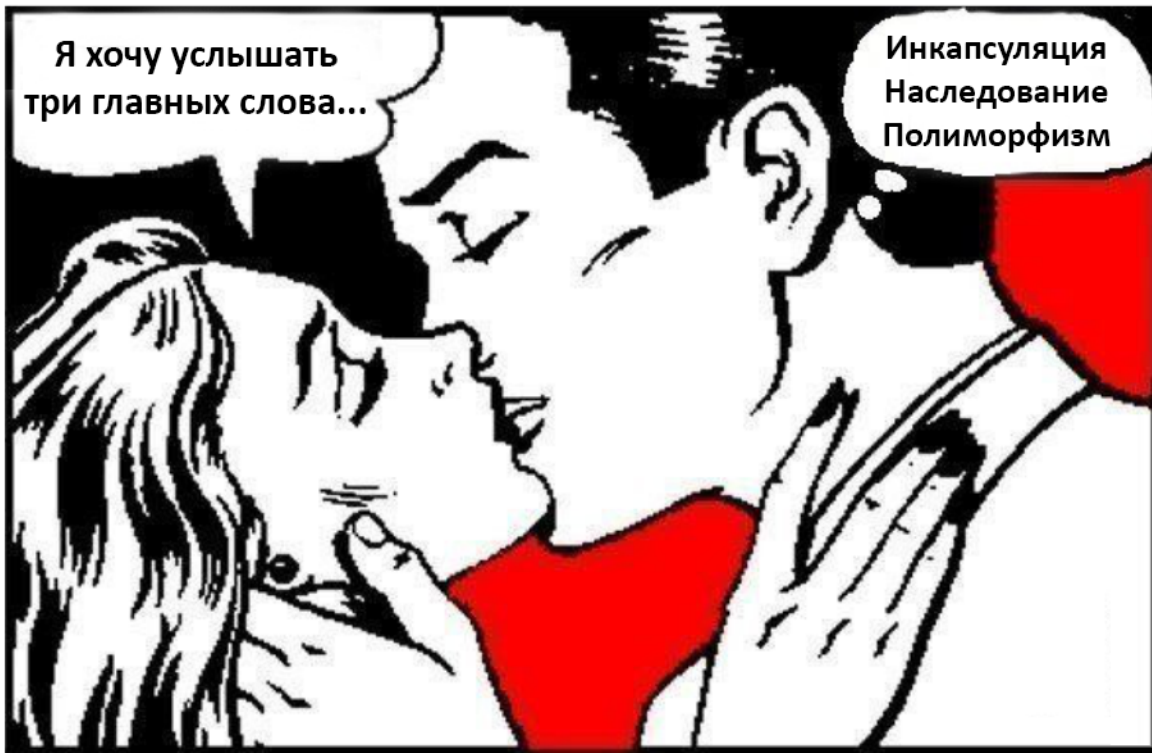
- Возникает необходимость создания классов, которые отличаются несколькими полями/методами.
- Как писать меньше кода?

Solution

- OOP Principles.

OOP principles

OOP principles

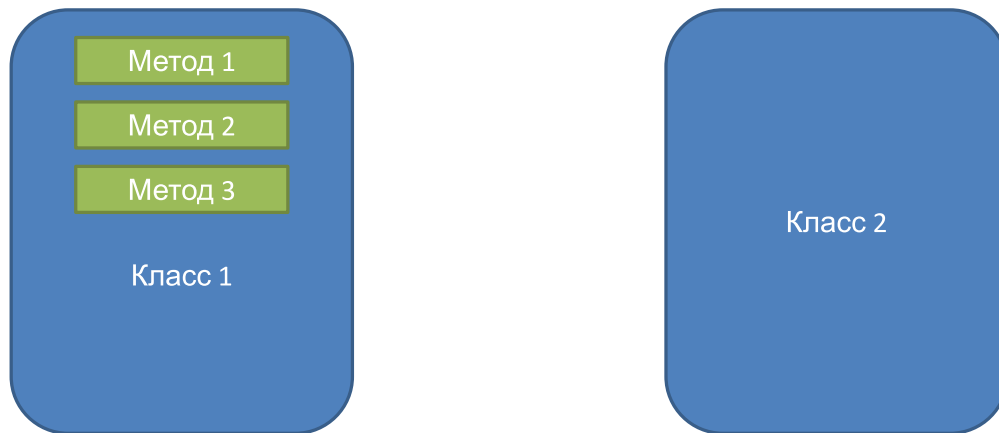


OOP principles

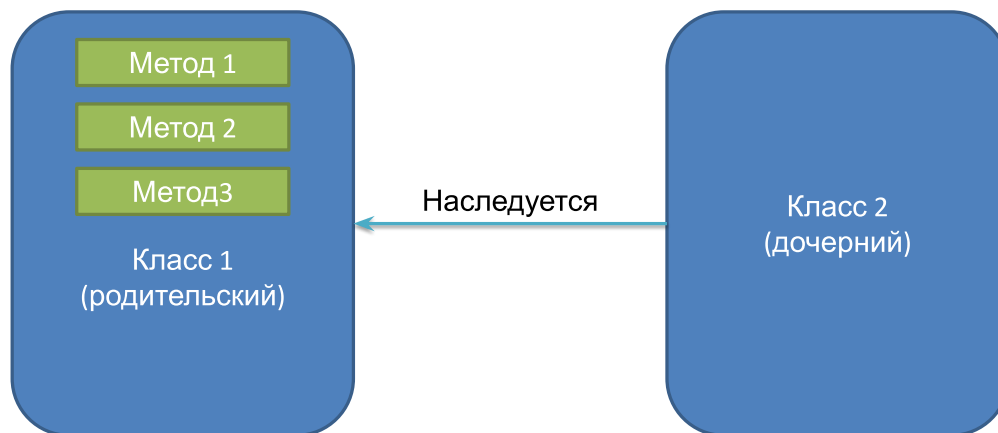
- **Inheritance (Наследование)**
- **Encapsulation (Инкапсуляция)**
- **Polymorphism (Полиморфизм)**
- **Abstraction (Абстракция)**

Inheritance

Inheritance



Inheritance



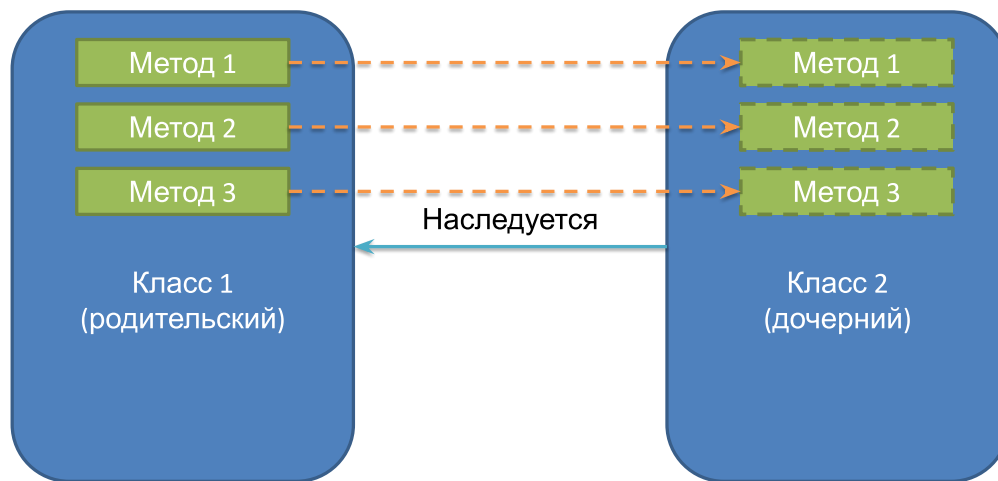
Superclass: example

```
public class Person {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

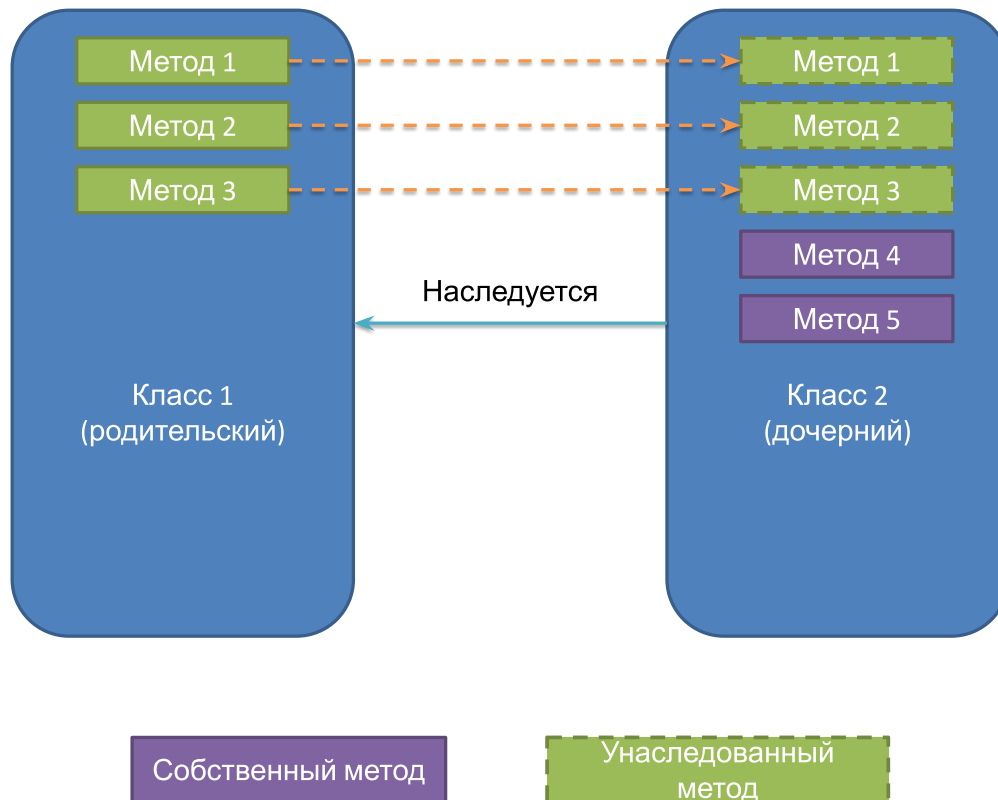
Subclass: example

```
class Employee extends Person {  
}
```

Inheritance



Inheritance



Extends subclass: example

```
class Person {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

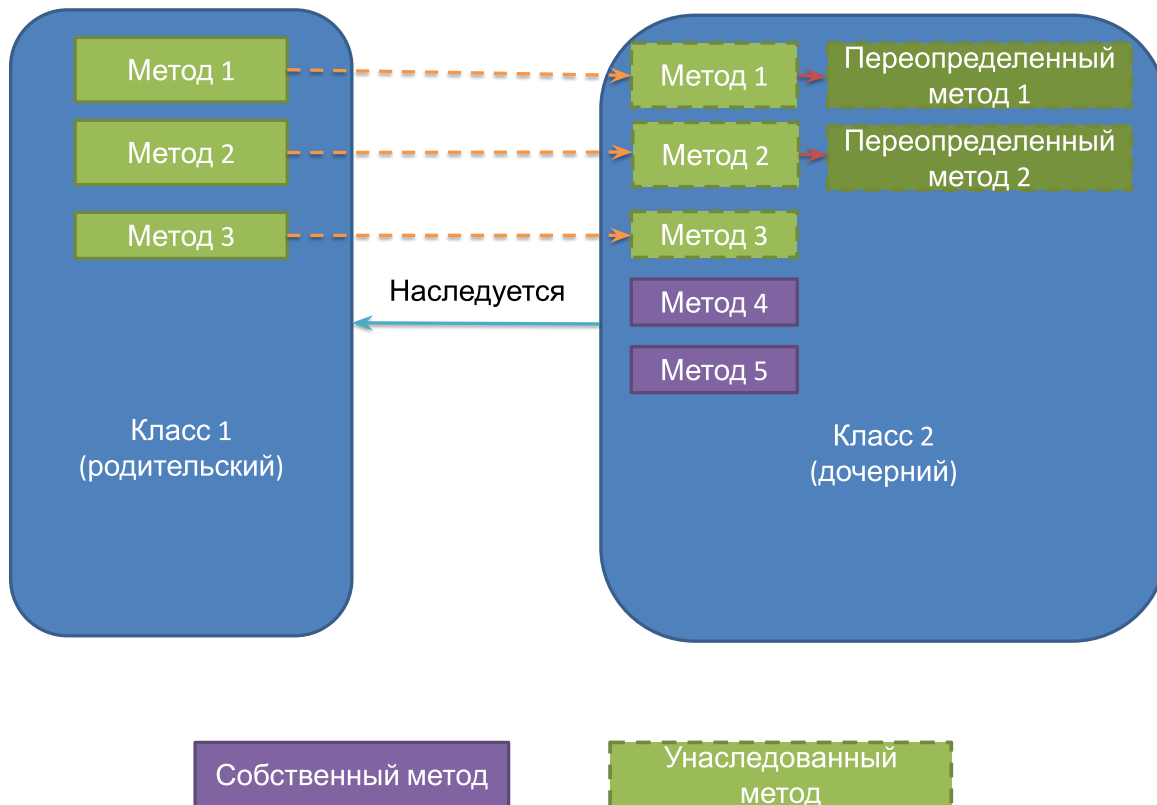
Extends subclass: example

```
class Employee extends Person {  
    private String company;  
  
    public Employee(String name, String company) {  
        super(name);  
        this.company = company;  
    }  
  
    public void work() {  
        System.out.printf("%s works in %s \n",  
            name, company);  
    }  
}
```


Extends subclass: example

```
public class Program {  
    public static void main(String[] args) {  
        Employee sam = new Employee("Sam", "Re  
        sam.display(); // Sam  
        sam.work(); // Sam works in Red Hat  
    }  
}
```

Inheritance



@Override: example

```
class Person {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

super keyword

- Для вызова **methods superclass**, **override** в **subclass**.
- Для доступа к **fields superclass**, если и **superclass**, и **subclass** имеют **fields** с одинаковыми именами.
- Чтобы явно вызвать **superclass** no-args (по умолчанию) или параметризованный **constructor** из **constructor subclass**.

@Override: example

```
class Employee extends Person {  
    private String company;  
  
    public Employee(String name, String company) {  
        super(name);  
        this.company = company;  
    }  
  
    @Override  
    public void display() {  
        System.out.printf("Name: %s \n", getName());  
        System.out.printf("Works in %s \n", company);  
    }  
}
```

@Override: example

```
public class Program {  
    public static void main(String[] args) {  
        Employee sam = new Employee("Sam", "Re  
        sam.display(); // Sam  
        // Works in Red Hat  
    }  
}
```

Inheritance

Iphone

Методы

- Управлять касанием пальца
- Звонить
- СМС
- Слушать музыку
- Смотреть фильмы
- Работать в интернете
- Отображать информацию на экране
- Фотографировать

Свойства

- Размеры
- Материал
- Вес
- Дизайн
- Процессор
- Размеры камеры

Наследование

Iphone 4S

Переопределённые методы

- Управлять касанием пальца
- Работать в интернете
- Отображать информацию на экране
- Фотографировать

Унаследованные методы

- Звонить
- СМС
- Слушать музыку
- Смотреть фильмы

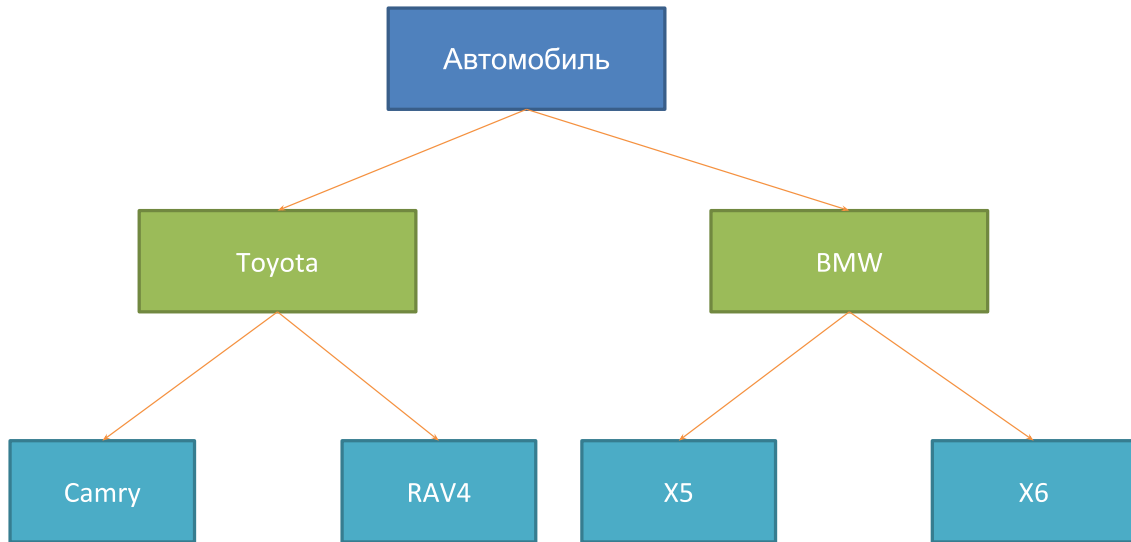
Добавленные методы

- Поддержка сети 3G
- Навигация GPS
- Распознавание речи (Siri)

Inheritance

- Повторное использование кода
- Расширение **superclass**
- **Subclass** будет уметь всё, что умел **superclass** плюс добавляет что-то своё

Inheritance



Subclass

Subclass видит:

- **fields** и **methods** с модификатором **public**.
- **fields** и **methods** с модификатором **protected**.
- **fields** и **methods** без модификатора доступа, если **superclass** в том же **package**, что и **subclass** – так делать нежелательно.

Inheritance

- Все **objects** наследуются от **Object**, даже если не указан *** extends Object**.
- **superclasses** не наследуют **members subclasses!**
- В **subclasses** при наследовании можно расширять **accesses modifier**, но нельзя сужать.
- В Java **НЕТ** множественного наследования, как в C++.

Inheritance

- Когда есть общее поведение для каких-либо **objects** – нужно выносить его в **superclass**.
- Нужно уметь правильно наследоваться, т.е. выделять общие **classes**.
- Наследование избавляет вашу программу от избыточности.

Inheritance

- Если нужно изменить общее поведение, то наследование автоматически передаст это изменение для всех **subclasses**.
- **subclass** наследует доступные **methods** и **fields** от **superclass** и может прибавлять свои собственные **methods** и **fields**.

Inheritance vs Composition

Inheritance vs Composition

- **Inheritance** – не всегда лучший инструмент для повторного использования кода из-за привязки к архитектуре наследования.
- Старайтесь использовать **composition** вместо **inheritance**.
- По времени жизни внутренние объекты зависят от объекта, в котором они созданы.

Inheritance vs Composition

- Если объекты связаны по типу **has a** («содержит»), то нужно применять композицию
- Если объекты связаны по типу **is a** («является»), то нужно применять наследование

Dynamic binding

Example

```
class Person {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public void display() {  
        System.out.printf("Person %s \n", name  
    }  
}
```

Example

```
class Employee extends Person {  
    private String company;  
  
    public Employee(String name, String company) {  
        super(name);  
        this.company = company;  
    }  
  
    @Override  
    public void display() {  
        System.out.printf("Employee %s works i  
    }  
}
```

Example

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

Inheritance Hierarchy and Type Conversion

Upcasting: example

```
class Person {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public void display() {  
        System.out.printf("Person %s \n", name);  
    }  
}
```

Upcasting: example

```
class Employee extends Person {  
    private String company;  
  
    public Employee(String name, String company) {  
        super(name);  
        this.company = company;  
    }  
  
    public String getCompany() {  
        return company;  
    }  
  
    public void display() {  
        System.out.printf("Employee %s works in %s\n", name, company);  
    }  
}
```

Upcasting: example

```
class Client extends Person {  
    private int sum;  
    private String bank;  
  
    public Client(String name, String bank, int sum) {  
        super(name);  
        this.bank = bank;  
        this.sum = sum;  
    }  
  
    public void display() {  
        System.out.printf("Client %s has account %s\n", name, bank);  
    }  
  
    public String getBank() {  
        return bank;  
    }  
  
    public int getSum() {  
        return sum;  
    }  
}
```


Upcasting: example

```
public class Program {  
    public static void main(String[] args) {  
        Person tom = new Person("Tom");  
        tom.display();  
        Person sam = new Employee("Sam", "Oracle");  
        sam.display();  
        Person bob = new Client("Bob", "Deutschland");  
        bob.display();  
    }  
}
```

Upcasting: example

```
Object tom = new Person("Tom");  
Object sam = new Employee("Sam", "Oracle");  
Object kate = new Client("Kate", "DeutscheBank");  
Person bob = new Client("Bob", "DeutscheBank");  
Person alice = new Employee("Alice", "Google");
```

Downcasting: example

```
Object sam = new Employee("Sam", "Oracle");  
Employee emp = (Employee) sam;  
emp.display();  
System.out.println(emp.getCompany());
```

Bad Practice

```
Object kate = new Client("Kate", "DeutscheBank");  
Employee emp = (Employee) kate;  
emp.display();  
((Employee) kate).display();
```

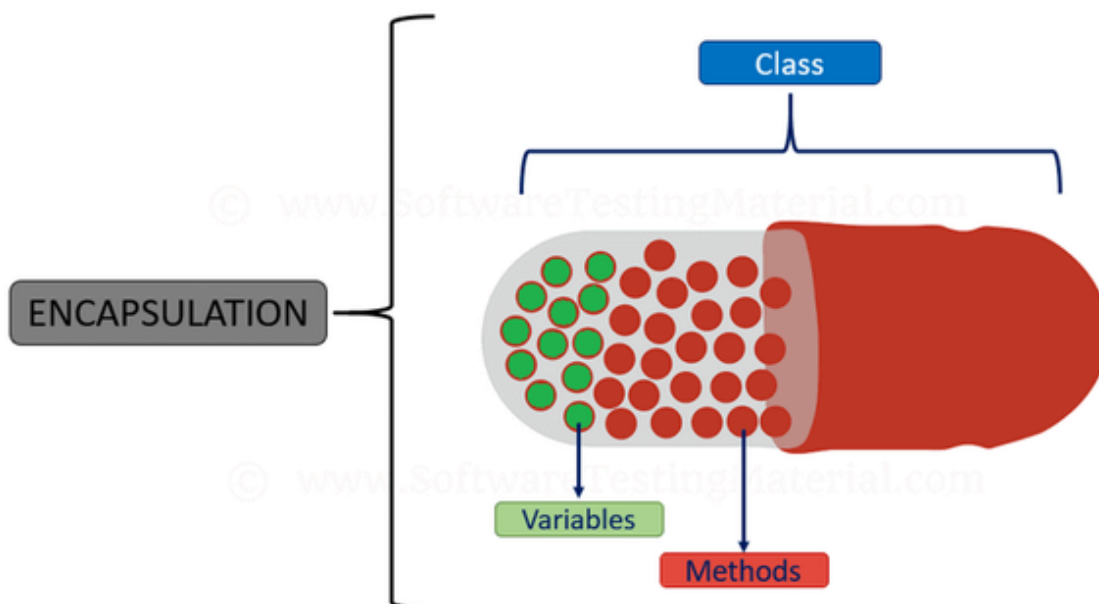
instanceof keyword

```
Object kate = new Client("Kate", "DeutscheBank");
if (kate instanceof Employee) {
    ((Employee) kate).display();
} else {
    System.out.println("Conversion is invalid");
}
```

Encapsulation

Encapsulation

- **Encapsulation (Инкапсуляция)** — это процесс объединения кода и данных в единый блок.
- **Encapsulation** - это ограничение доступа одних компонентов программы к другим.



Encapsulation



Packages

Packages

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

Packages

- **Packages** — это фактически обычная директория.

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

Package definition: example

```
package com.rakovets;

public class User {
    public String name;

    public User(String name) {
        this.name = name;
    }

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

Package definition: example

```
package com.rakovets;  
  
public class Program {  
    public static void main(String[] args) {  
        User dmitry = new User("Dmitry");  
        dmitry.tellAboutYourself();  
    }  
}
```

Packages and Terminal: example

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

Name: Dmitry

import Packages and Classes: example

```
package com.rakovets;  
  
import java.util.Scanner;  
  
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

Access modifiers

(Модификаторы доступа)

- **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	<i>friendly</i>	protected
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;  
    }  
}
```

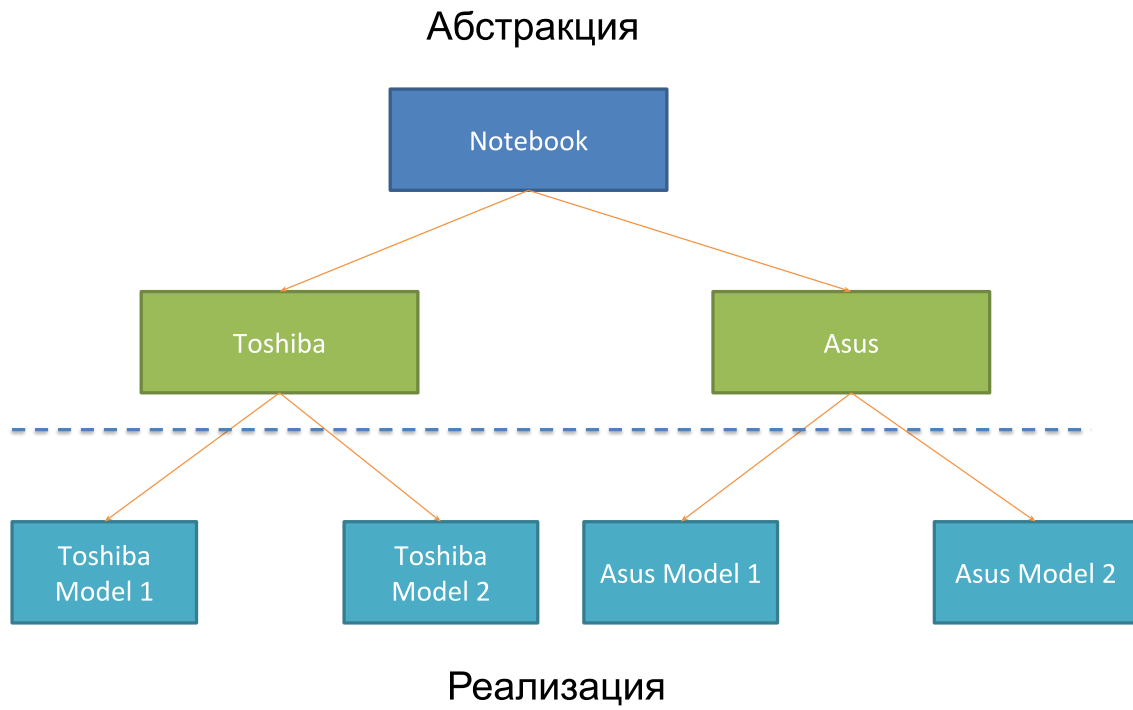
Access modifiers

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());  
    }  
}
```

Abstraction and Polymorphism

Abstraction



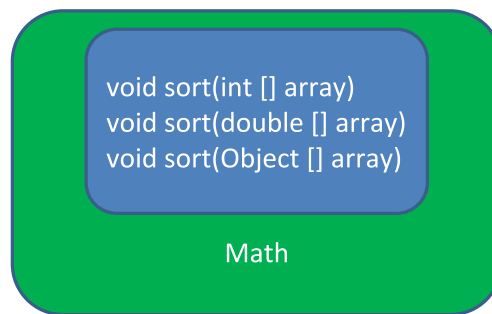
Polymorphism

- Один **interface** – множество **implementations** (реализаций).
- Одно имя – множество вариантов выполнения.

Polymorphism

- **overloading methods**
- **overriding methods**
- **abstract classes**
- **interfaces**

Polymorphism: overloading methods



Abstract classes

Abstract classes

- Абстрактный класс нужен для того, чтобы задать модель поведения для всех дочерних объектов.
- Нельзя создать экземпляр абстрактного класса (через **new**), потому что он ничего не умеет, это просто шаблон поведения для дочерних классов.

Abstract classes

- Если класс имеет хотя бы один абстрактный метод, то он будет абстрактным.
- Любой дочерний класс должен реализовать все абстрактные методы родительского, либо он сам должен быть абстрактным.
- Абстрактный класс может быть абстрактным и при этом не иметь ни одного абстрактного метода.

abstract

```
public abstract class Human {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
}
```


Example

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

Example

```
class Employee extends Person {  
    private String bank;  
  
    public Employee(String name, String compan  
        super(name);  
        this.bank = company;  
    }  
  
    public void display() {  
        System.out.printf("Employee Name: %s \  
    }  
}
```

Example

```
class Client extends Person {  
    private String bank;  
  
    public Client(String name, String company)  
        super(name);  
        this.bank = company;  
    }  
  
    public void display() {  
        System.out.printf("Client Name: %s \t  
                           Bank: %s \n", super.getName(),  
    }  
}
```

Example

```
public class Program {  
    public static void main(String[] args) {  
        Employee sam = new Employee("Sam", "Le  
        sam.display();  
        Client bob = new Client("Bob", "Leman  
        bob.display();  
    }  
}
```

Interfaces

Interfaces

- **Interface** (интерфейс) – более «строгий» вариант **abstract class**. **Methods** могут быть только **abstract**.
- **Interface** задаёт только поведение, без реализации.
- **Interface** может наследоваться от одного или нескольких **interfaces**.

Interfaces definition

```
interface Printable {  
    void print();  
}
```

Interfaces implements

```
class Book implements Printable {  
    String name;  
    String author;  
  
    Book(String name, String author) {  
        this.name = name;  
        this.author = author;  
    }  
  
    public void print() {  
        System.out.printf("%s (%s) \n", name,  
    }  
}
```


Interfaces implements

```
public class Program {  
    public static void main(String[] args) {  
        Printable b1 = new Book("Java. Complet  
        b1.print();  
    }  
}
```

Interfaces and default method: example

```
interface Printable {  
    default void print() {  
        System.out.println("Undefined printabl  
    }  
}
```

Interfaces and default method: example

```
class Journal implements Printable {  
    private String name;  
  
    String getName() {  
        return name;  
    }  
  
    Journal(String name) {  
        this.name = name;  
    }  
}
```

Interfaces and static method: example

```
interface Printable {  
    void print();  
  
    static void read() {  
        System.out.println("Read printable");  
    }  
}  
  
public static void main(String[] args) {  
    Printable.read();  
}
```

Interfaces and private method (@since 9)

```
interface Calculatable {  
    default int sum(int a, int b) {  
        return sumAll(a, b);  
    }  
  
    default int sum(int a, int b, int c) {  
        return sumAll(a, b, c);  
    }  
  
    private int sumAll(int... values) {  
        int result = 0;  
        for (int n : values) {  
            result += n;  
        }  
        return result;  
    }  
}
```

Interfaces and private method: example

```
class Calculation implements Calculatable {  
}
```

Interfaces and private method: example

```
public class Program {  
    public static void main(String[] args) {  
        Calculatable c = new Calculation();  
        System.out.println(c.sum(1, 2));  
        System.out.println(c.sum(1, 2, 4));  
    }  
}
```

Interfaces and constants: example

```
interface Stateable {  
    int OPEN = 1;  
    int CLOSED = 0;  
  
    void printState(int n);  
}
```


Interfaces and constants: example

```
class WaterPipe implements Stateable {  
    public void printState(int n) {  
        if (n == OPEN) {  
            System.out.println("Water is opene  
        } else if (n == CLOSED) {  
            System.out.println("Water is close  
        } else {  
            System.out.println("State is inval  
        }  
    }  
}
```

Interfaces and constants: example


```
public class Program {  
    public static void main(String[] args) {  
        WaterPipe pipe = new WaterPipe();  
        pipe.printState(1);  
    }  
}
```

Multiple implements: example

```
interface Printable {  
}
```

```
interface Searchable {  
}
```

```
class Book implements Printable, Searchable {  
}
```



Interfaces as arguments and result for method: example

```
interface Printable {  
    void print();  
}
```

Interfaces as arguments and result for method: example

```
class Book implements Printable {  
    String name;  
    String author;  
  
    Book(String name, String author) {  
        this.name = name;  
        this.author = author;  
    }  
  
    public void print() {  
        System.out.printf("%s (%s) \n", name,  
    }  
}
```

Interfaces as arguments and result for method: example

```
class Journal implements Printable {  
    private String name;  
  
    String getName() {  
        return name;  
    }  
  
    Journal(String name) {  
        this.name = name;  
    }  
  
    public void print() {  
        System.out.println(name);  
    }  
}
```

Interfaces as arguments and result for method: example

```
public class Program {
    public static void main(String[] args) {
        Printable printable = createPrintable(args);
        printable.print();

        read(new Book("Java for impatient", "O'Reilly"),
            read(new Journal("Java Dayly News"), "O'Reilly"));
    }

    static void read(Printable p) {
        p.print();
    }

    static Printable createPrintable(String name, String option) {
        if (option == "book") {
            return new Book(name, "O'Reilly");
        } else {
            return new Journal(name);
        }
    }
}
```

Abstract classes vs Interfaces

Abstract classes vs Interfaces

- Интерфейс может наследоваться от множества интерфейсов, абстрактный класс — только от одного класса.
- Совет: если есть возможность — используйте интерфейсы.