Inheritance and Polymorphism

Problem

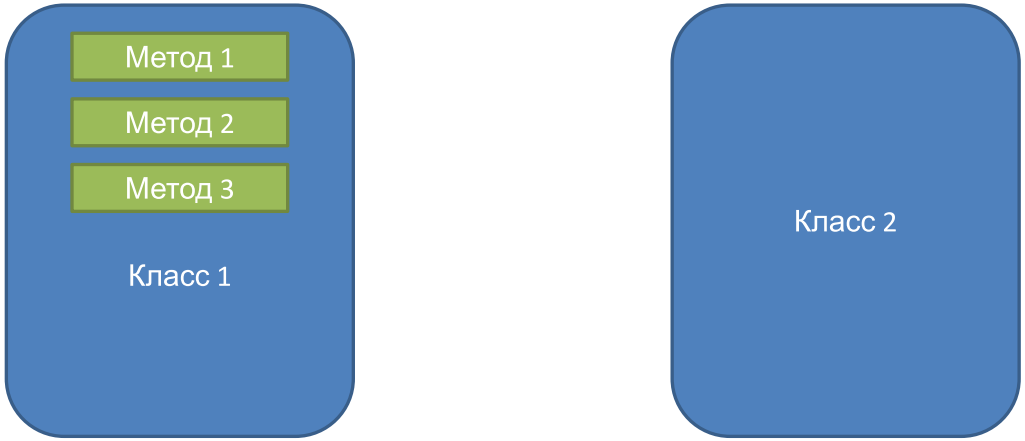
Problem

Возникает необходимость создания классов, которые отличаются несколькими полями/методами.

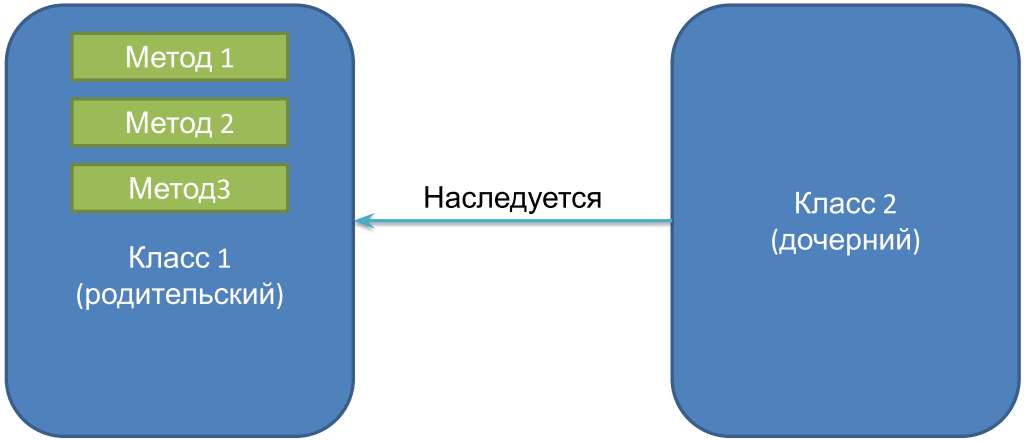
Solution

Inheritance

Inheritance



Inheritance



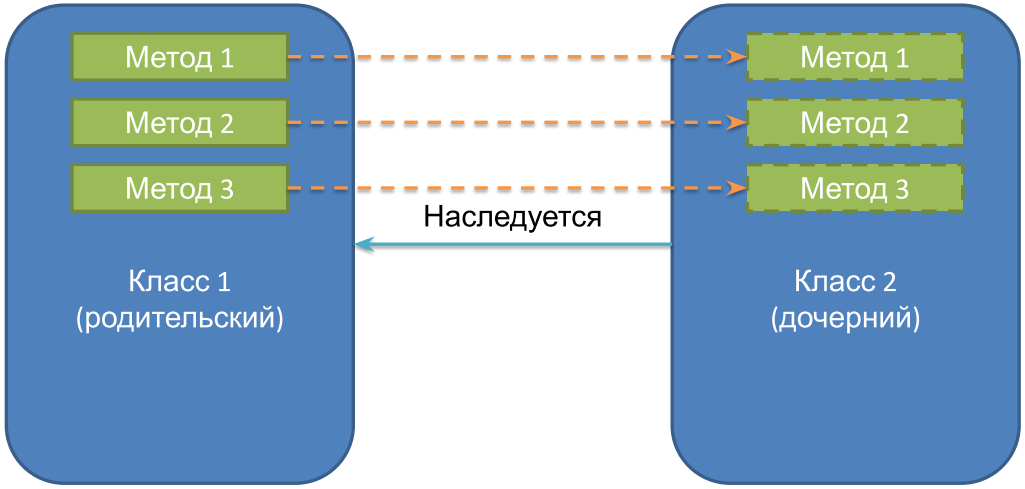
Super class: example

```
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);
    }
}
```

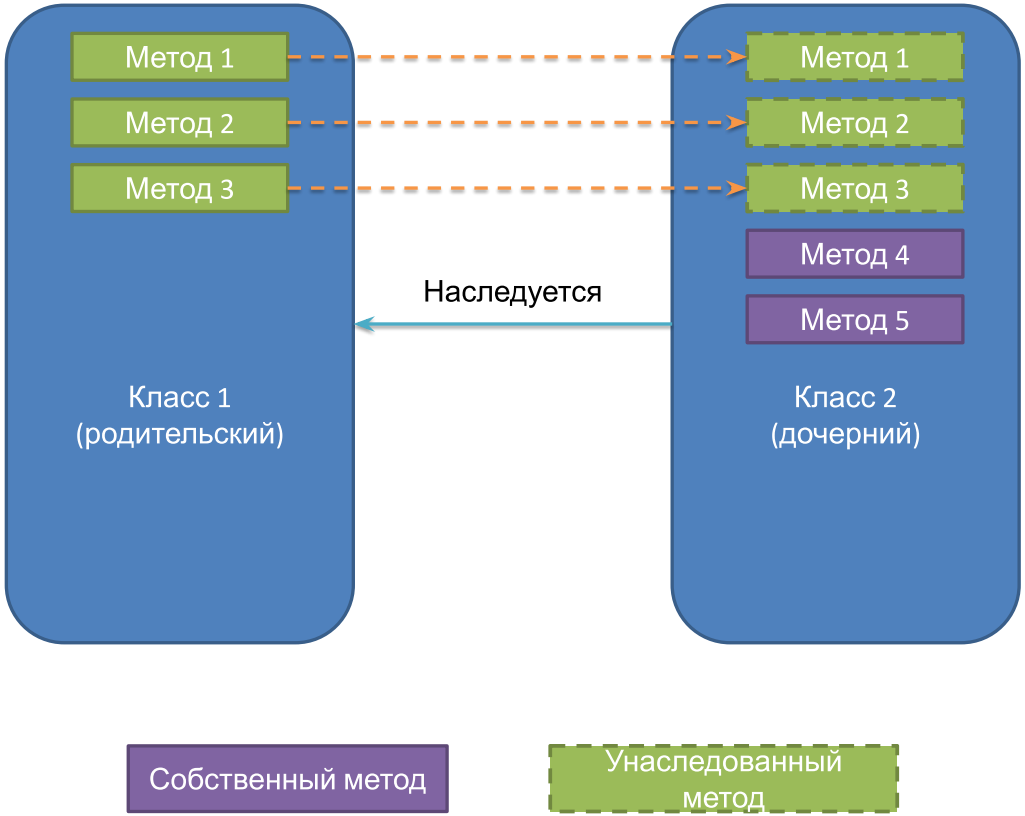
Sub class: example

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

Inheritance



Inheritance



Extends sub class: 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);
    }
}
```

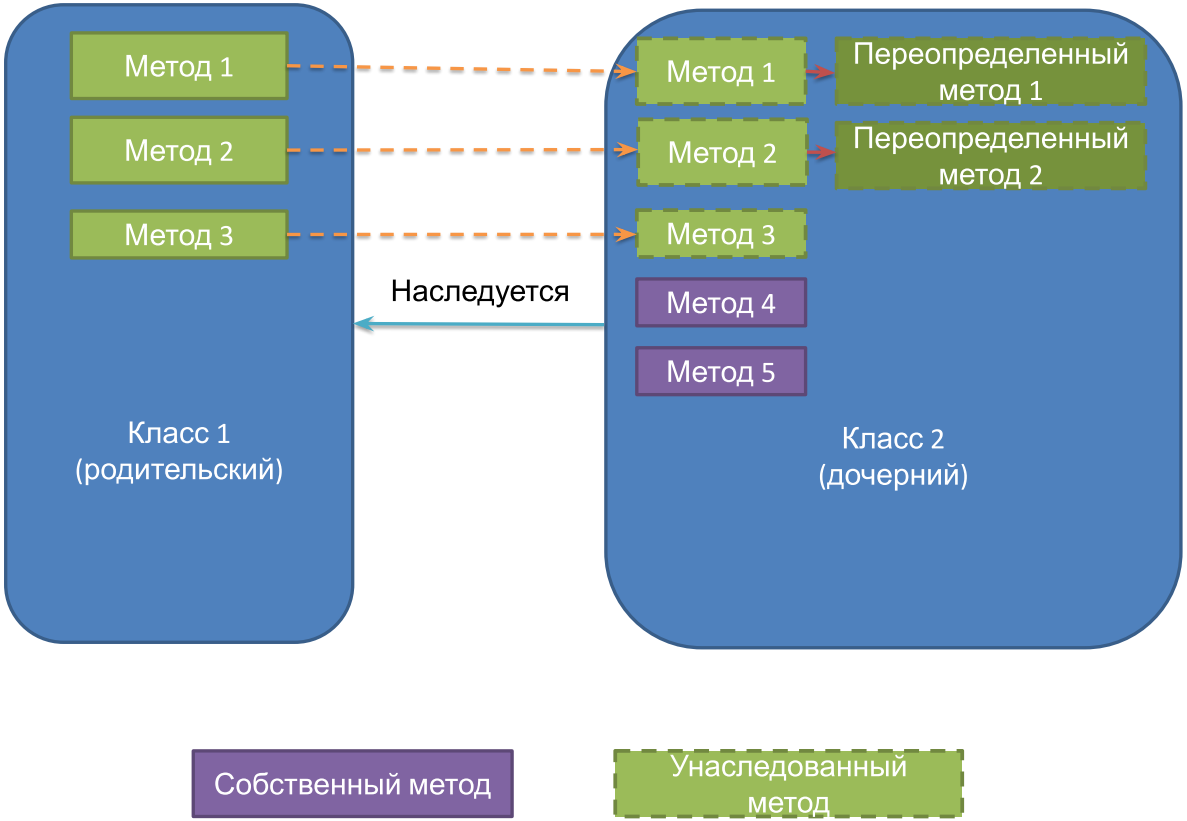
Extends sub class: example

```
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", getName(), company);
    }
}
```

Extends sub class: example

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

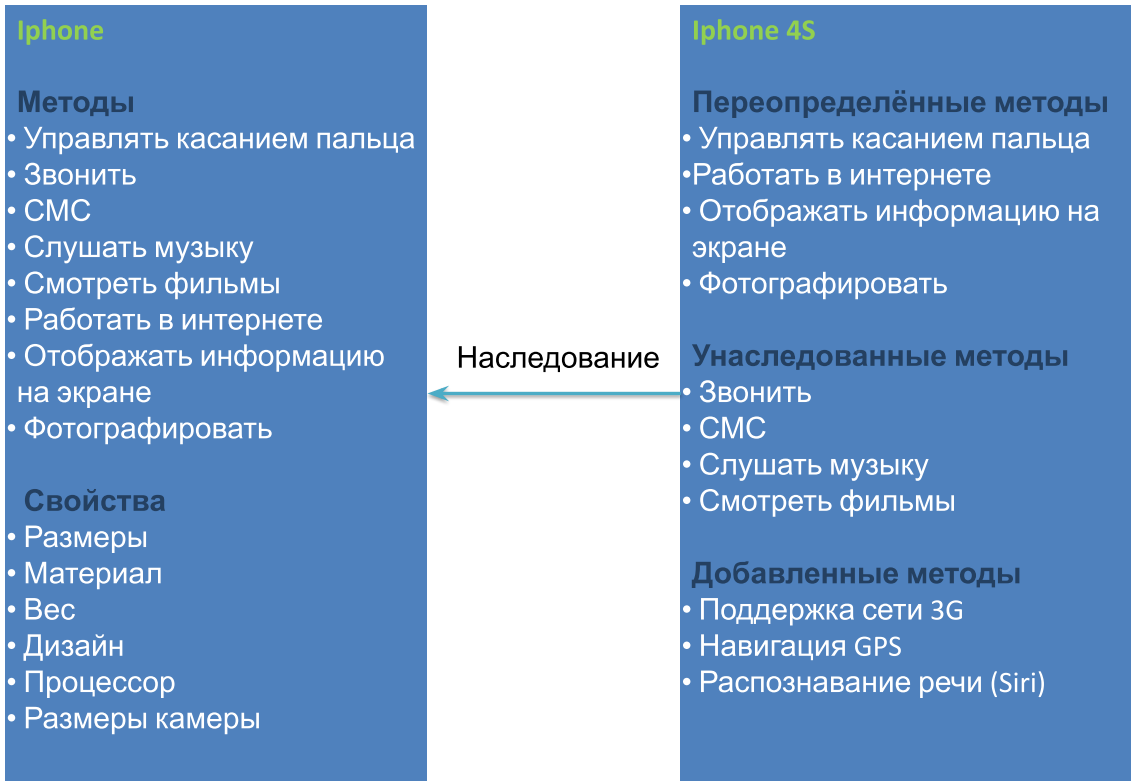
`@Override`: example

```
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", "Red Hat");
        sam.display(); // Sam
        // Works in Red Hat
    }
}
```

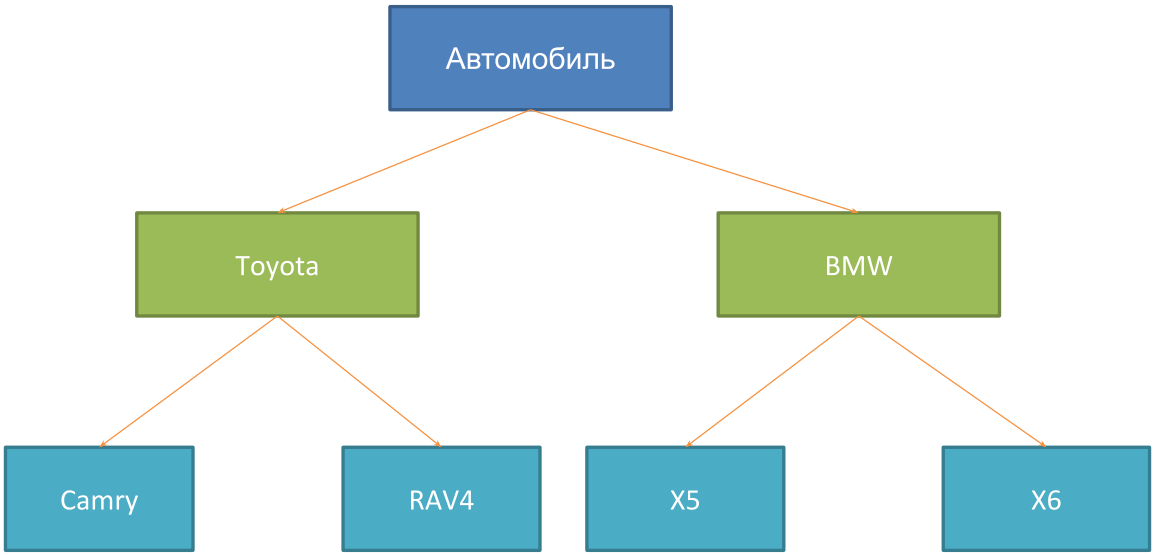
Inheritance



Inheritance

- Повторное использование кода
- Расширение родительского класса
- Дочерний класс будет уметь всё, что умел родительский плюс добавляет что-то своё

Inheritance



Subclass

Дочерний класс видит:

- Открытые методы и переменные с модификатором `public`
- Защищённые (`protected`) методы и переменные
- Методы и переменные на уровне пакета (без модификатора доступа), если суперкласс в том же пакете, что и дочерний – так делать нежелательно

Inheritance

- Все объекты наследуются от `Object`, даже если не указан `extends Object`
- Родительские классы не наследуют элементы дочернего класса!
- В дочерних классах при наследовании можно расширять модификатор доступа, но нельзя сужать
- В Java нет множественного наследования, как в C++

Inheritance

- Когда есть общее поведение для каких-либо объектов – нужно выносить его в родительский класс.
- Нужно уметь правильно наследоваться, т.е. выделять общие классы.
- Наследование избавляет вашу программу от избыточности.

Inheritance

- Если нужно изменить общее поведение, то наследование автоматически передаст это изменение для всех дочерних классов.
- Дочерний класс наследует доступные методы и переменные от родительского класса и может прибавлять свои собственные методы и переменные.

Inheritance vs Composition

Inheritance vs Composition

- **Наследование** – не всегда лучший инструмент для повторного использования кода из-за привязки к архитектуре наследования.
- Старайтесь использовать **композицию** вместо **наследования**.
- 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

Inheritance and `final`

Inheritance and `final`

- Переменная, объявленная `final` не может изменить своё значение.
- Метод, объявленный `final` не может быть переопределён в подклассе.
- Класс, объявленный `final` не может иметь подклассы.

Example

```
public final class Person {
}

class Employee extends Person {
} // Compile error
```

Example

```
public class Person {
    public final void display() {
        System.out.println("Имя: " + name);
    }
}

class Employee extends Person {
    @Override
    public void display() {
        System.out.println("Имя: " + name);
    } // Compile error
}
```

`final`

- Однако, у объектной переменной можно изменить внутреннее состояние (свойства) с помощью вызова методов, даже если она объявлена `final`.
- Объявление `private`-методов `final` не имеет смысла, так как `private`-методы не наследуются

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 in %s \n", super.getName(), cor
    }
}
```

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();
    }
}
```

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", super.getName(), cor
    }
}
```

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 in %s \n", super.getName(
    }

    public String getBank() {
        return bank;
    }

    public int getSum() {
```

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", "DeutscheBank", 3000);
        bob.display();
    }
}
```

Upcasting: example

```
Object tom = new Person("Tom");
Object sam = new Employee("Sam", "Oracle");
Object kate = new Client("Kate", "DeutscheBank", 2000);
Person bob = new Client("Bob", "DeutscheBank", 3000);
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", 2000);
Employee emp = (Employee) kate;
emp.display();
((Employee) kate).display();
```

Good Practice

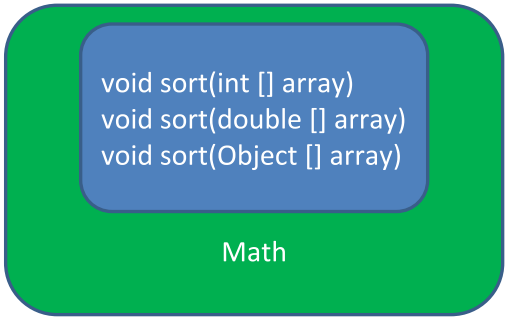
```
Object kate = new Client("Kate", "DeutscheBank", 2000);
if (kate instanceof Employee) {
    ((Employee) kate).display();
} else {
    System.out.println("Conversion is invalid");
}
```

Polymorphism

Polymorphism

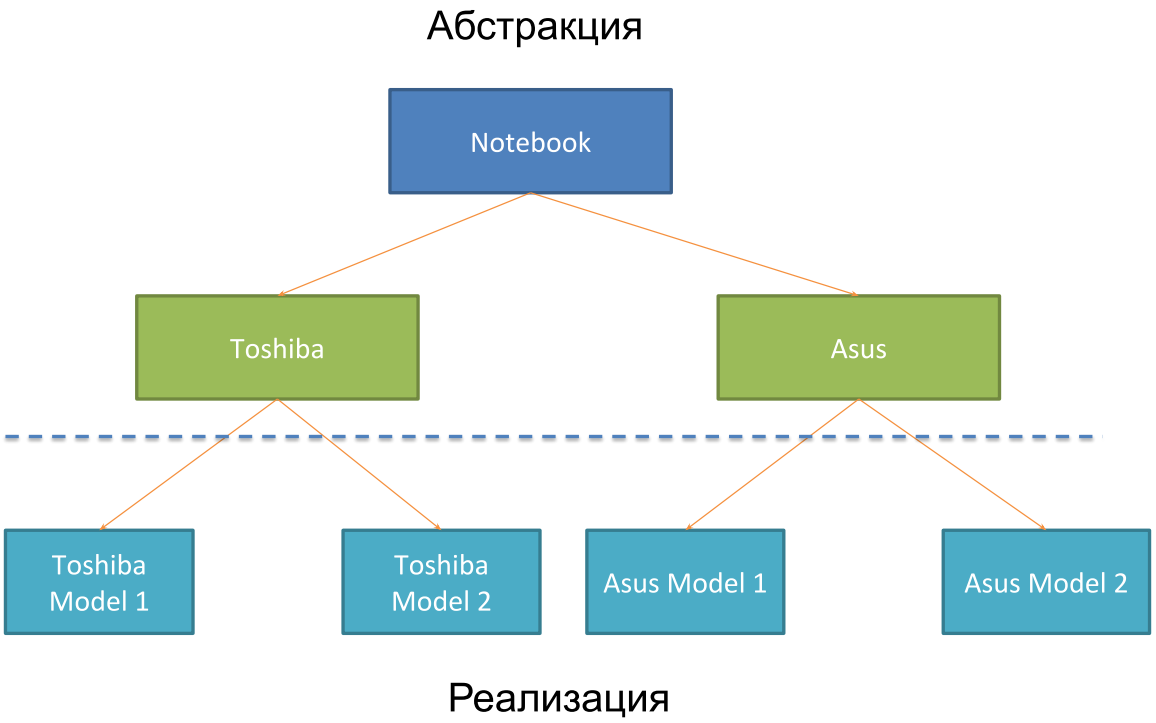
- Один интерфейс – множество реализаций
- Одно имя – множество вариантов выполнения

Polymorphism



Abstracttion

Abstracttion



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

Keyword `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 company) {
        super(name);
        this.bank = company;
    }

    public void display() {
        System.out.printf("Employee Name: %s \t Bank: %s \n", super.getName
    }
}
```

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", "Leman Brothers");
        sam.display();
        Client bob = new Client("Bob", "Leman Brothers");
        bob.display();
    }
}
```

Interfaces

Interfaces

- Интерфейс – более «строгий» вариант абстрактного класса. Методы могут быть только абстрактными.
- Интерфейс задаёт только поведение, без реализации.
- Интерфейс может наследоваться от одного или нескольких интерфейсов.

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, author);
    }
}
```

Interfaces implements

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

`Interfaces` and `default` method: example

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

`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 opened");
        } else if (n == CLOSED) {
            System.out.println("Water is closed");
        } else {
            System.out.println("State is invalid");
        }
    }
}
```

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, author);
    }
}
```

Interfaces as arguments and result for method: example

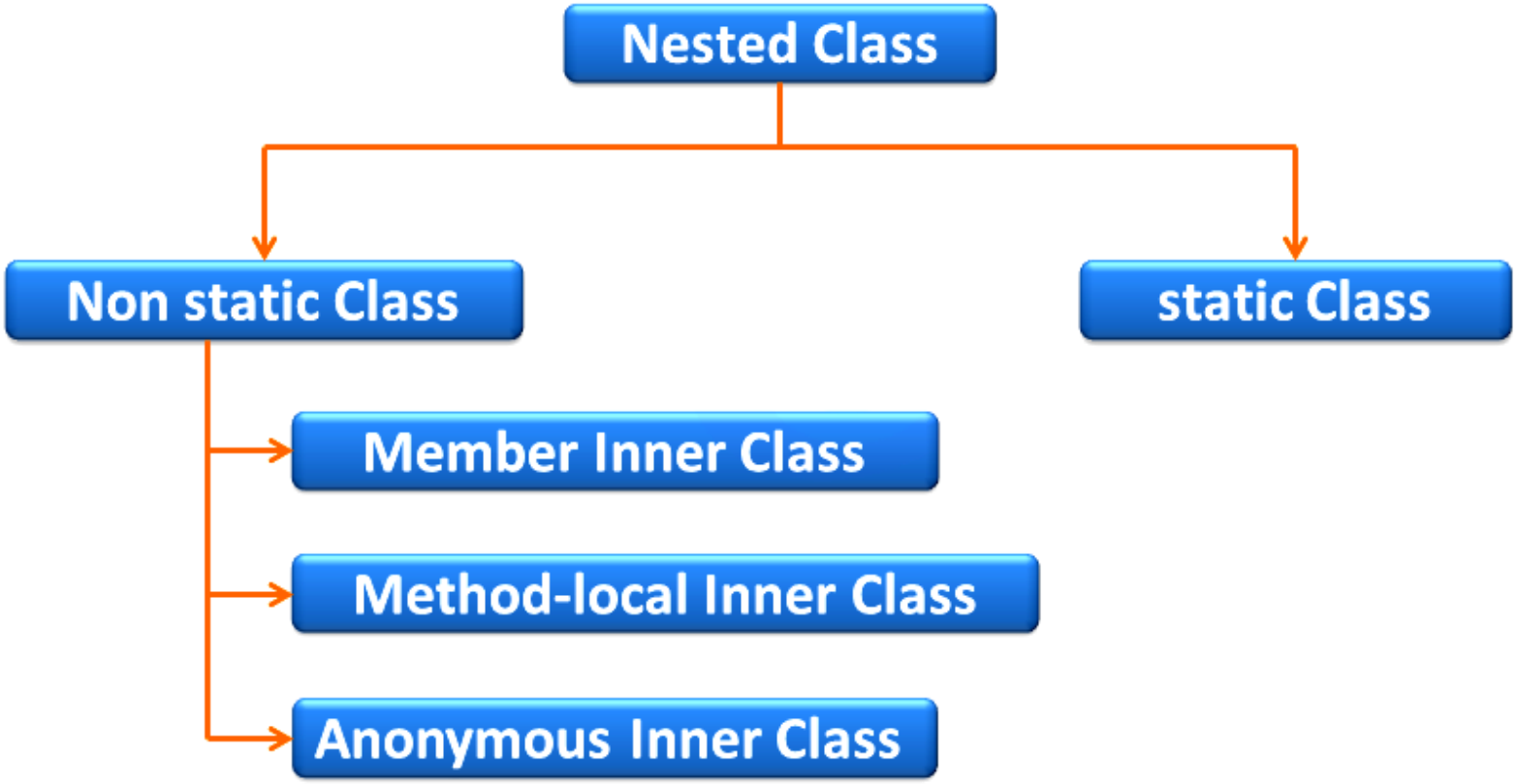
```
class Journal implements Printable {
    private String name;

    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("Foreign Affairs", false);
        printable.print();

        read(new Book("Java for impatients", "Cay Horstmann"));
        read(new Journal("Java Dayly News"));
    }

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

    static Printable createPrintable(String name, boolean option) {
        if (option) {
            return new Book(name, "Undefined");
        } else {
            return new Journal(name);
        }
    }
```

Nested Classes

Nested Classes



Nested Classes

Внутренним классом называют класс, который является членом другого класса. Существует четыре базовых типа внутренних классов в Java:

- **Static Nested classes** or **Member of outer class** (**статические вложенные классы**)
- **Nested Inner classes** (**вложенные внутренние классы**)
- **Method Local Inner classes** (**внутренние классы в локальном методе**)
- **Anonymous Inner classes** (**анонимные классы**)

Static Nested classes

```
class Math {
    public static class Factorial {
        private int result;
        private int key;

        public Factorial(int number, int x) {
            this.result = number;
            this.key = x;
        }

        public int getResult() {
            return result;
        }

        public int getKey() {
            return key;
        }
    }

    public static Factorial getFactorial(int x) {
```

Static Nested classes

```
public class Program {
    public static void main(String[] args) {
        Math.Factorial fact = Math.getFactorial(6);
        System.out.printf("Факториал числа %d равен %d \n", fact.getKey(),
    }
}
```

Nested Inner classes

```
class Person {
    private String name;
    Account account;

    Person(String name, String password) {
        this.name = name;
        account = new Account(password);
    }

    public void displayPerson() {
        System.out.printf("Person \t Name: %s \t Password: %s \n", name, a
    }

    public class Account {
        private String password;

        Account(String password) {
            this.password = password;
        }
```

Nested Inner classes

```
public class Program {
    public static void main(String[] args) {
        Person tom = new Person("Tom", "qwerty");
        tom.displayPerson();
        tom.account.displayAccount();
    }
}
```

Method Local Inner Classes

```
class Person {
    private String name;

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

    public void setAccount(String password) {
        class Account {
            void display() {
                System.out.printf("Account Login: %s \t Password: %s \n", na
            }
        }
        Account account = new Account();
        account.display();
    }
}
```

Method Local Inner Classes

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

Anonymous Inner classes

Extend a Class:

```
new Book("Design Patterns") {
    @Override
    public String description() {
        return "Famous GoF book.";
    }
}
```

Anonymous Inner classes

Implement an Interface:

```
new Runnable() {
    @Override
    public void run() {
        // code
    }
}
```

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.