

Обектно-ориентирано програмиране с **Java**

Предната лекция говорихме за...

Вградените типове данни

Условия и разклонения

Итерация / Цикли

Низове

Масиви

Функции

Днес продължаваме с...

Класове и обекти

Абстрактни класове и интерфейси

Фундаменталните ООП принципи

- Енкапсулация
- Наследяване
- Полиморфизъм
- Абстракция

Обектно-ориентирано програмиране

Парадигма за дизайн и имплементация на приложения като съвкупност от обекти, които

- съдържат състояние (член-променливи)
- съдържат поведение (методи)
- комуникират чрез съобщения

ООП се основава на разбирането, че всяка програма е симулация на реален или виртуален свят.

Клас

Клас – дефиниция на обект

- описва състояние – член-променливи
- описва поведение – методи

Конструктор(и)

Метод – функция за манипулиране на член-променливите на класа

- сигнатура и тяло
- параметри и локални променливи
- тип на връщаната стойност

Ключовата дума `this`

Референция към конкретния обект

Неявно се подава като параметър на всеки конструктор и метод на класа

Употребява се за

- Достъпване на член-променливи, „скрити“ от едноименни параметри на метод или локални променливи
- извикване от конструктор на друг `overloaded` конструктор в същия клас
- извикване на произволен метод на класа

```
public class Human {
```

```
    private String name;
```

```
    public Human() {  
        this("Unknown");  
    }
```

```
    public Human(String name) {  
        this.name = name;  
    }
```

```
    public void whoAmI() {  
        System.out.println("My name is " + name);  
    }
```

```
}
```

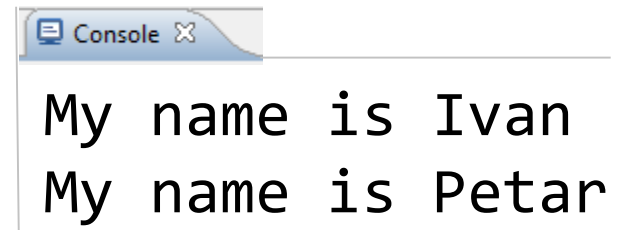
Извикване на overload-натия
конструктор със String
параметър

Достъпване на член-
променлива, скрита от
едноименен параметър

Обекти

Обектът е конкретна инстанция на даден клас.

```
public class MainApp {  
  
    public static void main(String[] args) {  
        Human ivan = new Human("Ivan");  
        ivan.whoAmI();  
        Human petar = new Human("Petar");  
        petar.whoAmI();  
    }  
}
```



Console X

My name is Ivan
My name is Petar

Пакети

Именувани групи от семантично свързани класове

Служат за йерархично организиране на кода

Съответстват на директорно дърво на файловата система

Конвенция за именуване

- само малки букви, точка за разделител
- компаниите използват обърнат домейн адрес

`mail.google.com` → `com.google.mail`

Достъп до клас от друг пакет

Всеки клас има достъп по подразбиране (имплицитно) до

- Класовете от собствения си пакет
- Класовете в пакета `java.lang`

Ако искаме клас да има достъп до клас в някой друг пакет, трябва експлицитно да го заявим с `import` декларация която поставяме над декларацията на класа.

Прието е да се подреждат в сортиран ред по *пакет.клас*

По-чисто е да се изброят конкретните класове от пакета вместо

import java.util.*;

```
import java.util.Arrays;  
import java.util.Scanner;
```

```
public class StringUtilsils {
```

```
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        char[] ca = sc.nextLine().toCharArray();  
        Arrays.sort(ca);  
        System.out.println(ca);
```

```
    }
```

```
}
```

Console X

```
coffee  
ceeffo
```

Модификатори за достъп

За клас*

`public`

Достъпен за всеки клас във всеки пакет

без модификатор (“`package-private`”, “*default*”)

Достъпен само за класовете в собствения си пакет

* `top-level` клас, т.е. невложен в друг

За член-променливи и методи на клас

Модификатор	Същия клас	Друг клас в същия пакет	Клас-наследник	Всеки друг клас, виждащ дадения
<code>public</code>	да	да	да	да
<code>protected</code>	да	да	да	не
без модификатор	да	да	не	не
<code>private</code>	да	не	не	не

Енкапсулация (Encapsulation)

Само вътрешните методи на даден обект имат достъп до неговото състояние, като правят невъзможни неочакваните промени на състоянието от външния свят.

В Java се постига чрез модификаторите за достъп.

Енкапсулация – пример за нарушаване

```
public class Human {  
  
    public String name; // no encapsulation:  
  
    public Human(String name) {  
        this.name = name;  
    }  
  
}  
  
public class HumanFake {  
  
    public static void main(String[] args) {  
        Human ivan = new Human("Ivan");  
        ivan.name = "Faked Ivan"; // hmm...  
    }  
  
}
```

Енкапсулация – пример за спазване

```
public class Human {  
  
    private String name; // stays hidden  
  
    public Human(String name) {  
        this.name = name;  
    }  
  
}  
  
public class HumanFake {  
  
    public static void main(String[] args) {  
        Human ivan = new Human("Ivan");  
        ivan.name = "Faked Ivan"; // won't compile  
    }  
  
}
```



Наследяване (Inheritance)

Позволява преизползване и разширяване на поведение и състояние на вече съществуващи класове

В Java се реализира с ключовата дума `extends`

Класът – наследник получава достъп до всички `public` и `protected` член-променливи и методи на родителския клас

Класът – наследник може да предостави собствена дефиниция на (т.е. да предефинира) методи на родителския клас (`method overriding`). Модификаторът за достъп на метод в класа – наследник трябва да съвпада или да разширява модификатора за достъп на метода в родителския клас (но не може да я свива/ограничава)

Java не поддържа множествено наследяване

Ключовата дума `super`

Референция към прекия родител на обекта

Неявно се подава като параметър на всеки конструктор и метод на класа

Употребява се за

- Достъпване на член-променливи на родителя
- извикване от конструктор в текущия клас на конструктор в родителския клас
- извикване на произволен метод на родителския клас

Не нарушава енкапсулацията: през `super` може да достъпим само `public` и `protected` членове на родителския клас

```
public class Student extends Human {
```

Извикване на родителски
конструктор

```
    private int facultyNumber;
```

```
    public Student(String name, int facultyNumber) {  
        super(name);  
        this.facultyNumber = facultyNumber;  
    }
```

```
    public static void main(String[] args) {  
        Student ivan = new Student("Ivan", 61786);  
        ivan.whoAmI();  
    }  
}
```

Наследен от родителя метод

Console X

My name is Ivan

Йерархията от класове в Java

Всички класове в Java са (преки или косвени) наследници на класа `java.lang.Object`

Липсата на множествено наследяване означава, че всеки клас има точно един родител (с изключение на един-единствен клас, `java.lang.Object`, който няма родител).

→ Йерархията от класове е дърво, с `java.lang.Object` в корена

Операторът instanceof

Използва се за type checking: дали даден обект е инстанция на даден клас

```
Student ivan = new Student("Ivan", 61786);  
Human petar = new Human("Petar");
```

```
System.out.println(ivan instanceof Student); // true  
System.out.println(ivan instanceof Human); // true  
System.out.println(petar instanceof Student); // false  
System.out.println(petar instanceof Human); // true
```

```
System.out.println(null instanceof AnyClass); // false for any class: null is  
not an instance of anything  
System.out.println(ref instanceof Object); // true for any non-null ref,  
because any class extends java.lang.Object
```

Ключовата дума `final`

- в декларация на променлива → прави я константа
- в декларация на метод → методът не може да се `override`-ва
- в декларация на клас → класът не може да се наследява

Полиморфизъм (Polymorphism)

От гръцки poly (много) + morphe (форма)

Дефиниция от биологията - съществуване на морфологично различни индивиди в границите на един вид

ООП - наследниците на даден клас споделят поведение от родителския клас, но могат да дефинират и собствено поведение

Всички Java обекти са полиморфични, понеже

всеки обект наследява Object класа



Runtime полиморфизъм чрез method overriding

Overriding: класът - наследник предефинира поведението на класа - родител

```
public class Human {  
  
    private String name;  
  
    public Human(String name) {  
        this.name = name;  
    }  
  
    public void whoAmI() {  
        System.out.println("My name is " + name);  
    }  
}
```

```
public class Student extends Human {  
  
    private int facultyNumber;  
  
    public Student(String name, int facultyNumber) {  
        super(name);  
        this.facultyNumber = facultyNumber;  
    }  
  
    public void whoAmI() {  
        super.whoAmI();  
        System.out.println("My faculty number is "  
            + this.facultyNumber);  
    }  
}
```

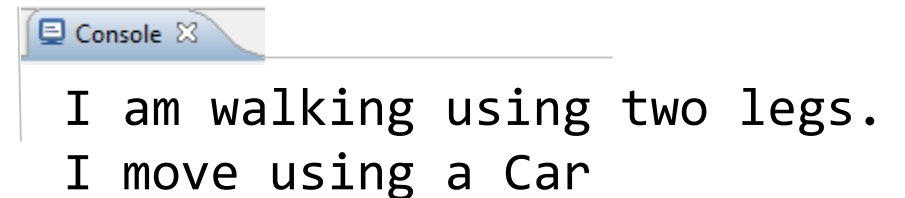
Compile-time полиморфизъм чрез method overloading

Overloading - класът декларира методи с едно и също име и различен брой и/или тип параметри

```
public void move() {  
    System.out.println("I am walking using two legs.");  
}
```

```
public void move(String vehicle) {  
    System.out.println("I move using a " + vehicle);  
}
```

```
public static void main(String[] args) {  
    Human ivan = new Human("Ivan");  
    ivan.move();  
    ivan.move("Car");  
}
```



Console X

I am walking using two legs.
I move using a Car


```
Student ivan = new Student("Ivan", 61786);  
Human petar = new Student("Petar", 74451);
```

```
Object[] objArr = { ivan, petar };  
  
for (Object o : objArr) {  
    // non-polymorphic code  
    // instanceof and explicit casts  
    // are the "red lights"  
    if (o instanceof Student) {  
        ((Student) o).whoAmI();  
    } else if (o instanceof Human) {  
        ((Human) o).whoAmI();  
    }  
}
```

```
Human[] hArr = { ivan, petar };  
  
for (Human h : hArr) {  
    h.whoAmI(); // polymorphic code  
}
```



Полиморфният код е не само по-кратък и четим. Помислете как трябва да се променят двата фрагмента код, ако в бъдеще се появят нови класове – наследници на Human

Абстрактни класове

Дефинират се с модификатора `abstract`

Един клас не може да е едновременно **`abstract`** и **`final`** – защо?

Могат да имат методи без имплементация, които се декларират с модификатора `abstract`

Не са напълно дефинирани (оставят на наследниците си да ги конкретизират/допълнят) → не могат да се създават обекти от тях

```
public abstract class Cat {  
    public void move() {  
        System.out.println("I am walking on 4 toes.");  
    }  
  
    public void communicate() {  
        System.out.println("I mew.");  
    }  
  
    public abstract void eat();  
}
```

```
public class DomesticCat extends Cat {  
    public void eat() {  
        System.out.println("I eat Whiskas.");  
    }  
}  
  
public class Leopard extends Cat {  
    public void eat() {  
        System.out.println("I eat any prey.");  
    }  
}
```

Интерфейси

Съвкупност от декларации на методи *без имплементация**

Описва формално поведение без да го имплементира

Може да съдържа `static final` член-променливи == константи

*от Java 8, може да
съдържат също **default**
и **static** методи с
имплементация. В Java
9 – и **private** методи

Всички методи и
константи са по
подразбиране **public**

```
public interface Animal {  
    public void move();  
    public void communicate();  
}
```

```
public class Human implements Animal{  
    private String name;  
  
    public Human(String name) {  
        this.name = name;  
    }  
  
    public void communicate() {  
        System.out.println("I speak");  
    }  
  
    public void move(){  
        System.out.println("I am walking using two legs");  
    }  
}
```

```
public class Cat implements Animal{  
  
    public void move(){  
        System.out.println("I am walking with 4 toes");  
    }  
  
    public void communicate() {  
        System.out.println("I mew");  
    }  
}
```

Абстракция (Abstraction)

Абстракция означава, моделирайки в обектно-ориентиран език за програмиране обекти от реалния или виртуалния свят, да се ограничим само до съществените им за конкретната задача характеристики и да се абстрахираме (пропуснем) в модела несъществените или нерелевантни за задачата.

Пример: моделирайки студент, да го характеризираме само с име и факултетен номер, абстрахирайки се от всички други характеристики на студента в реалния свят (напр. цвят на очите).

Абстракция също означава да работим с нещо, което знаем как да използваме, без да знаем как работи вътрешно. Всяка конкретна имплементация на поведение е скрита в своя обект, за външния свят е видимо само поведението.

Принципът за абстракция се постига в Java чрез интерфейси и абстрактни класове.

java.lang.Object

Класът `java.lang.Object`

`.equals()`

`.hashCode()`

`.toString()`

`.clone()`

Обекти се сравняват за равенство с `.equals()`, а не с `==`

```
if ("".equals(s)) {  
    System.out.println("Empty string");  
}
```

equals()

Кога трябва да го предефинираме?

Ако сравняваме два обекта за семантична (т.е. смислова) еднаквост, а не по референциите им (т.е. адреса им в паметта).

Например, две инстанции на клас Student смислово са еднакви (отговарят на един и същи студент), ако факултетните им номера са еднакви – без значение дали *референциите към тях*, които участват в сравнението, са еднакви или не.

hashCode()

Кога трябва да го предефинираме?

Когато сме предефинирали equals()

Важна забележка:

При предефинирането на hashCode(), ако equals() връща true, hashCode-ът на съответните обекти трябва да е равен. Ако hashCode-ът на два обекта е равен, не е задължително equals() да връща true.

static

Статични член-променливи и статични методи

Те са част от класа, а не от конкретна негова инстанция (обект).

Могат да се достъпват без да е създаден обект: само с името на класа, точка, името на статичната член-променлива или метод. Например:

```
Math.PI // константата л  
Math.pow(double, double) // вдига първия аргумент на степен втория
```

Статичните член-променливи имат едно-единствено копие, което се споделя от всички инстанции на класа.

- ако са константи, пестим памет (няма смисъл да се мултиплицират във всяка инстанция)
- ако са променливи, всяка инстанция „вижда“ и променя една и съща стойност, което е механизъм за комуникация между всички инстанции на дадения клас

Статичните методи имат достъп само до статичните член-променливи и други статични методи на класа. Нестатичните методи имат достъп както до статичните, така и до нестатичните членове на класа.

```
public class Utils {  
    public static final double PI = 3.14; // constant  
    private static int radius = 10; // static member  
    private String fact5 = "5!"; // non-static member  
  
    public static long fact(int n) { // static method  
        if (n == 1) { return 1; } else { return n*fact(n-1); }  
    }  
    public String getFact() { // non-static method  
        return fact5;  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Perimeter is " + 2 * Utils.PI * radius);  
        System.out.println(new Utils().getFact() + "=" + Utils.fact(5));  
        // Utils.getFact() will not compile  
    }  
}
```

Исключения

Изключения (Exceptions)

Изключение е събитие (проблем), което се случва по време на изпълнение на дадена програма и нарушава нормалната последователност на изпълнение на инструкциите ѝ.

(Изключение е съкратено за изключително събитие)

Още един начин за комуникация на метод с извикващите го: връщана стойност при нормално изпълнение и изключение при проблем.

Например...

- Подали сме невалидни входни данни
- Опитваме се да отворим несъществуващ файл
- Мрежата се е разкачила по време на комуникация
- Свършила е паметта на виртуалната машина
- ...

Как се генерира („хвърля“) изключение?

```
public Object pop() {  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
    ...  
}
```

Как се обработва („лови“) изключение

```
try {  
    // код, който може да хвърли изключение  
} catch (Exception e) {  
    // обработваме изключението (“exception handler”).  
    // Може да има повече от един catch блок  
} finally {  
    // при нужда, някакви заключителни операции  
    // (finally блокът е optional, но ако го има, се изпълнява задължително  
    щом влезем в try-a)  
}
```

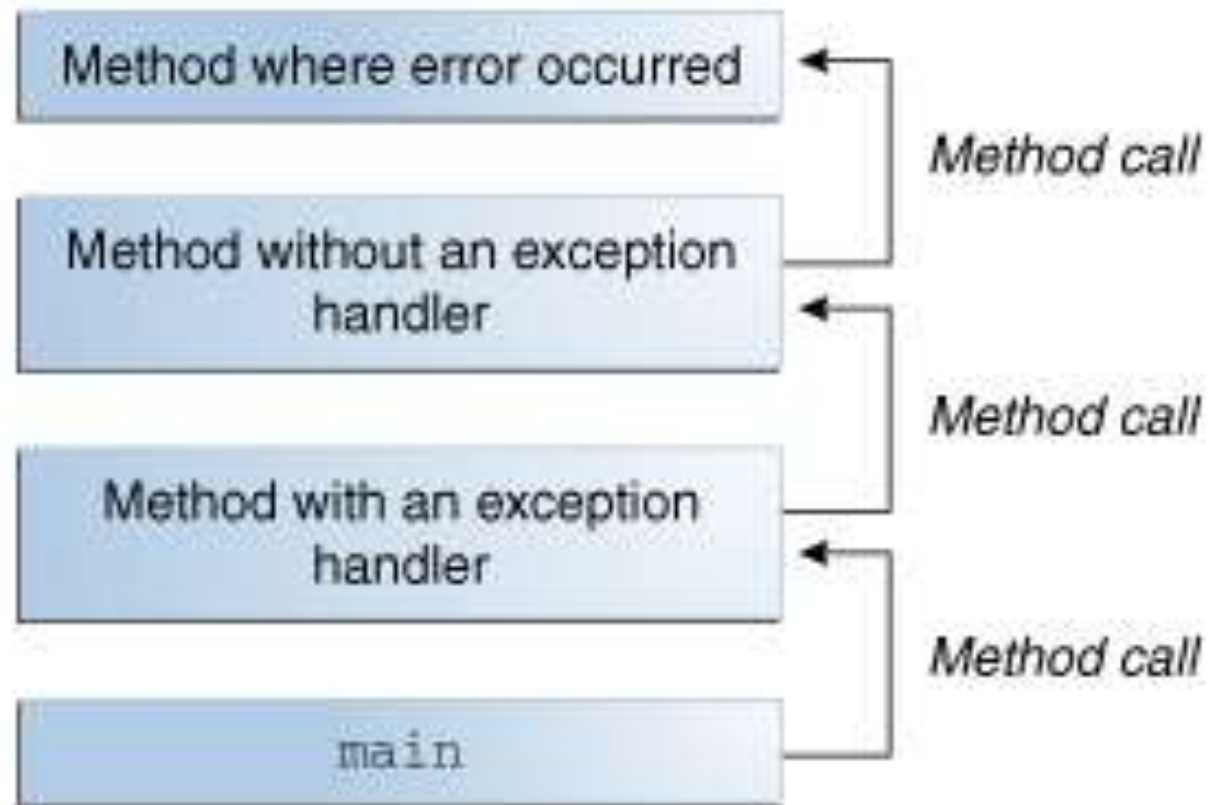

Catch block chain

```
try {  
    ...  
} catch (MostSpecificException mse) {  
} catch (MoreGeneralException mge) {  
} ... {  
} catch (LeastSpecificException lse) {  
}
```

Multi catch block

```
catch (IOException | SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

Стек на извикванията (call stack)



Finally – не само за обработка на изключения

```
try {  
    // тук може да се хвърлят изключения  
    // или да има return/continue/break  
} finally {  
    // някакъв важен cleanup code -  
    // ще се изпълни винаги*, независимо какво се случи в try блока  
}
```

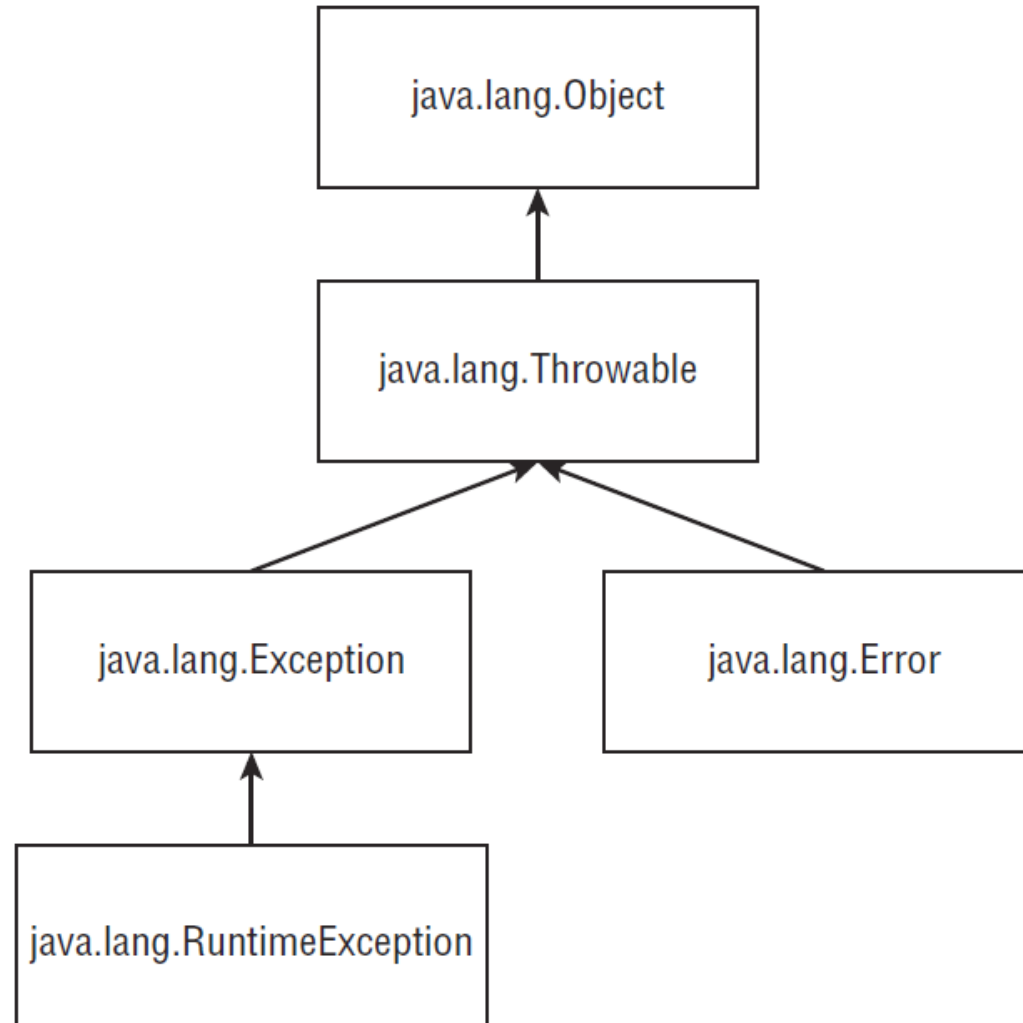
Видове изключения

Изключителните събития могат да се дължат на грешка на потребителя, бърк в кода или физически ресурс, който не е достъпен.

Делят се на три вида:

- Checked exceptions
- Unchecked (Runtime) exceptions
- Errors

Видове изключения



Деклариране на хвърляни изключения

Ако метод не прехваща/обработва даден **checked** exception, който може да се хвърли в тялото му, той трябва да го декларира в прототипа си, за да „предупреди“ тези, които го викат:

```
public void writeList() throws IOException, FileNotFoundException {  
  
    ...  
}
```

Защо да ползваме изключения?

- Отделяме кода за обработка на грешки от останалия → става по-четим
- „Препредаване“ на грешки по стека на извикванията
- Групиране и диференциране на различните типове грешки

Коллекции

Колекции

Java предоставя т.нар. `collections framework`, съдържащ интерфейси, имплементации и алгоритми върху най-използваните структури от данни.

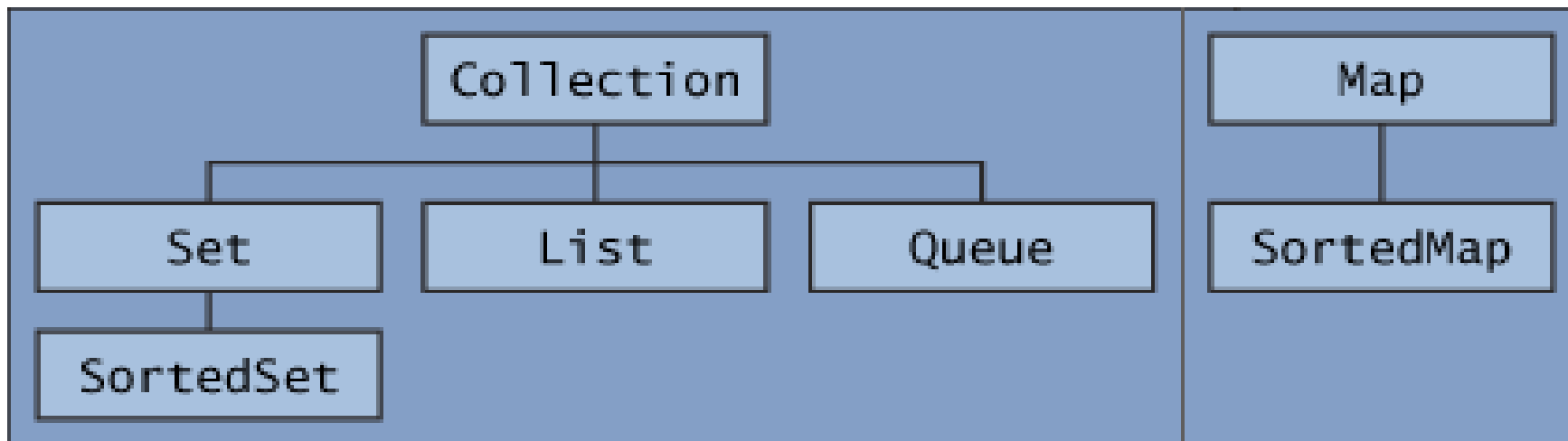
Всички интерфейси и класове се намират се в пакета `java.util`

Ползи:

- Не се налага да преоткриваме топлата вода
- Качество + производителност
- Code reuse

Колекции: интерфейси

...от птичи поглед:



Коллекции: имплементации

List: ArrayList, LinkedList, ...

Map: TreeMap, HashMap, ...

Set: TreeSet, HashSet, ...

Колекции: имплементации

`ArrayList` – списък върху динамичен масив. Константна амортизирана сложност за повечето операции

`LinkedList` – свързан списък

`TreeMap` / `TreeSet` – червено-черни дървета, запазват естествената наредба*. Елементите трябва да имплементират интерфейса `Comparable` и метода `equals()`. Гарантирана логаритмична сложност за повечето операции

`HashMap` / `HashSet` – хеш таблици, нямат наредба. Елементите трябва да имплементират методите `hashCode()` и `equals()`. Константна** сложност за повечето операции

*, ** Повече детайли в документацията на всеки клас

Колекции: често използвани методи

List

- `add()`
- `contains()`
- `get(int index), indexOf()`
- `remove() / remove(int index)`
- `size(), isEmpty()`
- `toArray()`

Set

- `add()`
- `contains()`
- `remove()`
- `size(), isEmpty()`
- `toArray()`

Map

- `put(K, V)`
- `get(K)`
- `remove(K)`
- `size(), isEmpty()`
- `keySet()`
- `values()`

Queue

- `add()`
- `peek()`
- `poll()`
- `remove()`

Колекции: алгоритми

Най-често употребявани

- **Сортиране** – `sort()`
- **Търсене** – `indexOf()`, `binarySearch()`
- **Разбъркване** – `shuffle()`
- **Манипулация** – `copy()`, `fill()`, `reverse()`, `swap()`
- **Статистики** – `min()`, `max()`, `frequency()`

Реализирани са като статични методи в класа `java.util.Collections`, а някои от тях – като методи в съответните класове - имплементации

Колекции: какво ново в Java 9?

Collection Factory методи

```
List<String> l = List.of("Java", "9", "rulez");
```

```
Set<Integer> s = Set.of(1, 2, 3, 5, 8);
```

```
Map<String, Integer> cities = Map.of("Brussels", 1_139_000, "Cardiff", 341_000);
```

```
Map<String, Integer> m = Map.ofEntries(entry("Brussels"), 1_139_000),  
entry("Cardiff", 341_000);
```


Итератори

java.util

```
public interface Iterator<E>
```

```
    boolean hasNext()
```

```
    E next() throws NoSuchElementException
```

```
    void remove()
```

Въпроси?