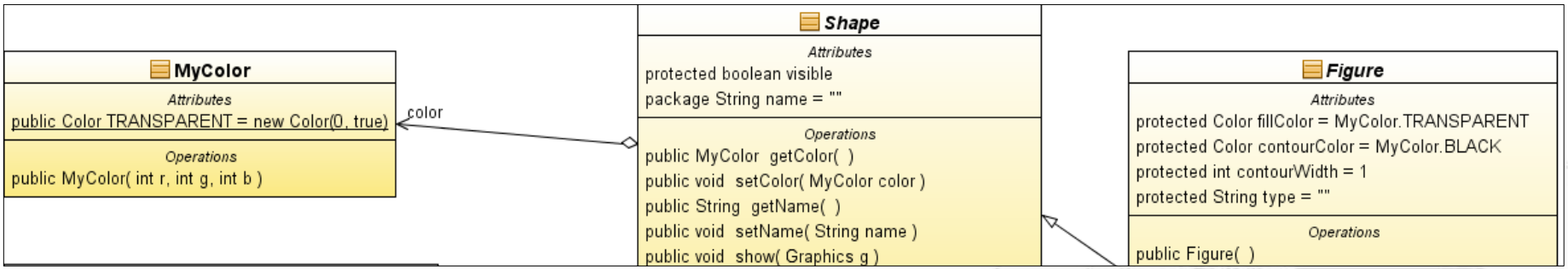
Trayan Iliev
IPT – Intellectual Products & Technologies Ltd.

Параметризирани типове и методи. Контейнери. Обработка на изключения. Новости в езика Java™



Траян Илиев

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: http://www.iproduct.org

Съдържание

1. Структури от данни за работа с множество еднотипни обекти. Масиви - клас Arrays.
2. Контейнерни класове и интерфейси в Java™. Итератори.
3. Параметризирани типове и методи: Generics.
4. Типизирани контейнери в Java™ 5, 6 и 7 – generics.
5. Нов подобрен вид for -цикъл за обхождане на колекции.
6. Списъци. Множества. Асоциативни списъци. Хеширане.
7. Реализиране на структури от данни стек, опашка, дек.
8. Обработка на изключения в Java. Новости в Java 7: multi-catch, try with resources.
9. Новости в Java™ 5, 6, 7 и 8 - enumeration types, static imports, autoboxing, variable argument lists, annotations strings in switch, generics type inference, ламбда изрази и поточно програмиране

Структури от данни за работа с множество еднотипни обекти. Масиви - клас Arrays.

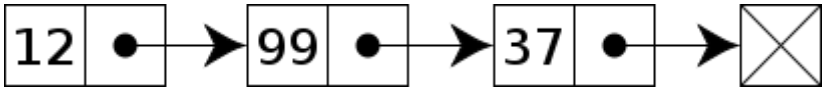
- Масиви от обекти и работа с тях
- Помощни (utility) методи в класа **Arrays**:
  - equals()
  - fill()
  - copyOf() и copyOfRange()
  - binarySearch()
  - sort()
- Сравняване на обекти – интерфейси **Comparable** и **Comparator**

Контейнерни класове и интерфейси в Java. Итератори.

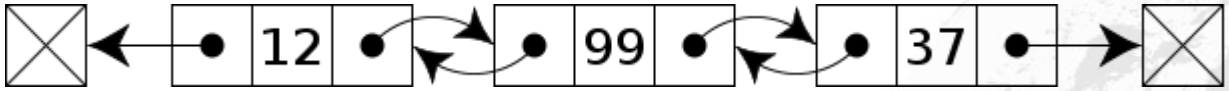
- Колекции – интерфейс **Collection**
- Списъци – интерфейс **List**, реализации – **ArrayList, LinkedList, ...**
- Множества – интерфейс **Set**, реализации – **HashSet, TreeSet, …**
- Асоциативни списъци – интерфейс **Map**, реализации – **HashMap, TreeMap, LinkedHashMap, WeakHashMap, …**
- Обхождане на колекция с итератор.
- Реализиране на структури от данни стек, опашка, дек – интерфейси **Queue** и **Dequeue**. Реализации.

Структури от данни

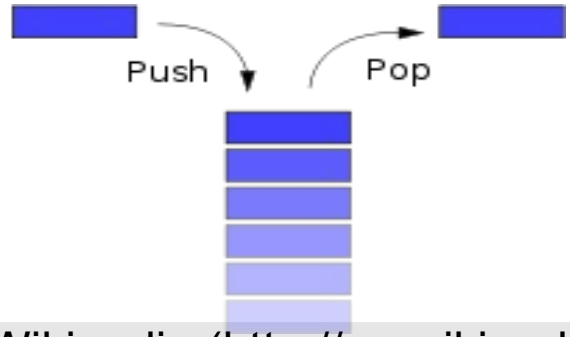
- Едно-свързан списък:



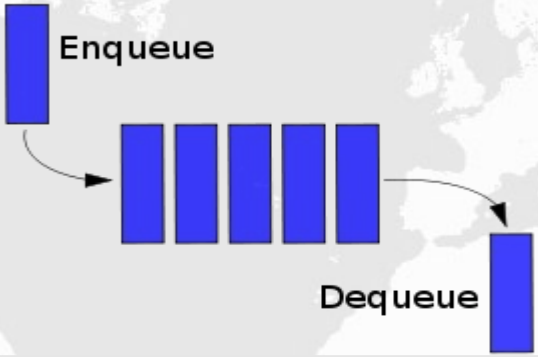
- Дву-свързан списък:



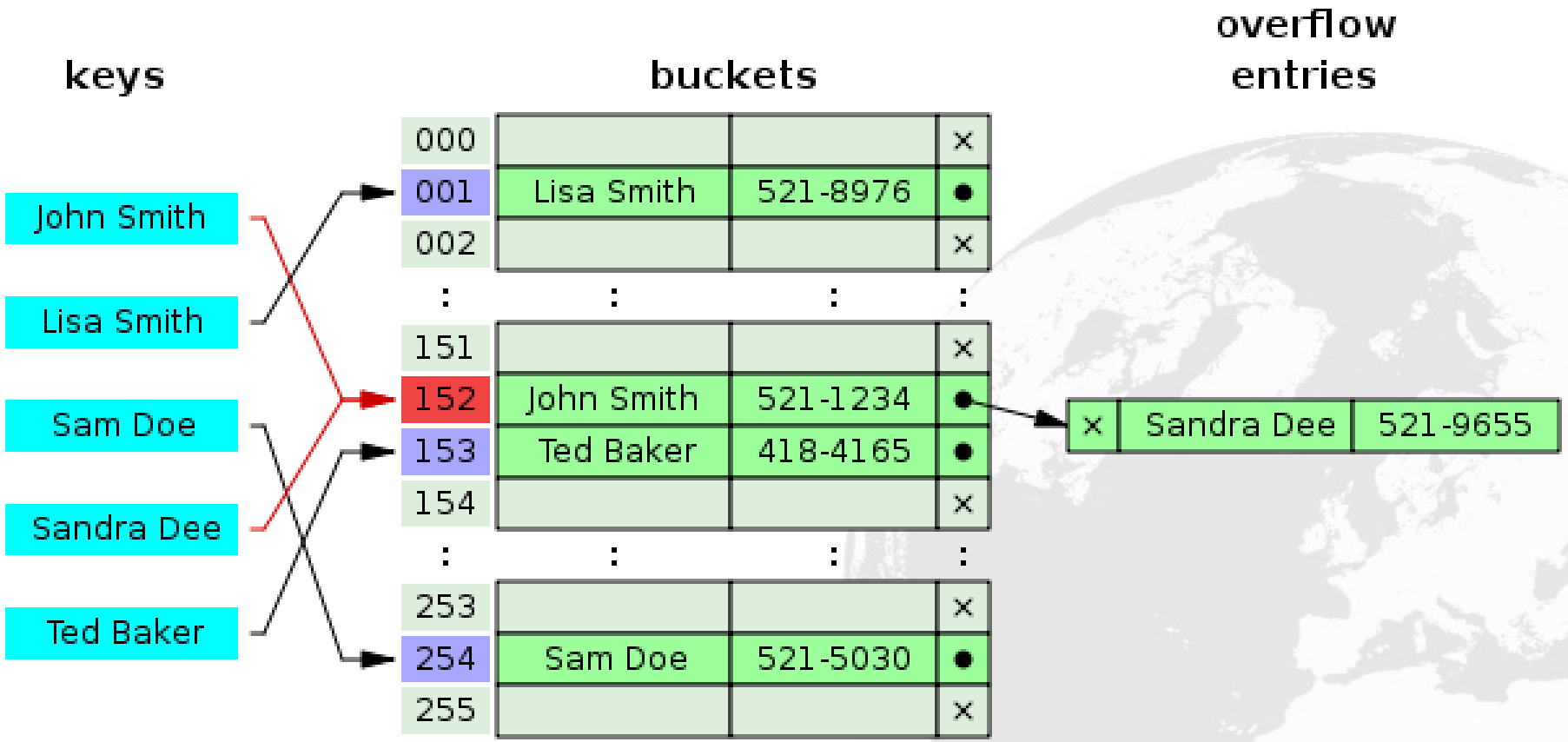
- Стек:



- Опашка:



Хеширане, хеш функции, хеш таблици



Източник: Wikipedia (http://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_0_LL.svg
Автор: Jorge Stolfi, Лиценз: Creative Commons Attribution-Share Alike 3.0 Unported

Параметризирани типове: Generics (1)

- Класовете и техните методи разгледани дотук са ограничени да работят само с един, конкретен тип за всеки от своите атрибути/ параметри.
- По този начин например ако е необходимо да създадем типизирани контейнери за различни по тип обекти, е необходимо да реализираме множество различни контейнерни класове – по един за всеки тип обекти.
- *Пример:* В една електронна книжарница ще продаваме книги и бихме искали списъкът с позиции да съдържа само книги --> трябва да реализираме отделен клас **BookList**, също е необходимо за всяка книга да поддържаме списък от автори --> трябва да реализираме **AuthorList** и т.н.

Параметризирани типове: Generics (2)

- *Решението:* Можем да си спестим писането на множество подобни класове (например контейнери за различни по тип елементи), като използваме **параметризирани класове**:
- *Използване на параметричен тип (Generic type invocation):*

```
List<Book> books = new ArrayList<Book>()
List<Author> authors = new ArrayList<Author>()
```

- **<>** – *Diamond* оператор – нов за Java™ 7, позволява автоматично извеждане на параметричния тип за изряза в дясно:

```
List<Book> books = new ArrayList<>()
List<Author> authors = new ArrayList<>()
```

Параметризирани типове: Generics (3)

- Декларация на параметричен тип *(Generic type declaration)*:

```
public class Position<T extends Product> {
    private T product;
    ...
    public Position(T product, double quantity) {
        this.product = product;
        this.quantity = quantity;
        price = product.getPrice();
    }
    public T getProduct() {
        return product;
    }
...
```

Формален параметър за типа на данните

Конвенции за именуване на параметрите

- T – параметър на типа (ако са повече – S, U, V, W ...)
- E – елемент на колекция – пр.: List<E>
- K – ключ на асоциативна двойка – пр.: Map<K,V>
- V – стойност асоциирана към ключа – пр.: Map<K,V>
- N – числова стойност

- *Пример:*

```
public class Invoice <T extends Product> {
    ...
    private List<Position<T>> positions = new ArrayList<>();
    ...
}
```

Параметризирани методи (Generic Methods) (1)

- Можем да параметризираме също методи и конструктори:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) {
    StringBuilder posStr = new StringBuilder();
    int n = 0;
    for(Position<U> p: positions){
        posStr.append( String.format(
    "\n| %1$3s | %2$30s | %3$6s | %4$4s | %5$6s |  %6$8s |",
        ++n, p.getProduct().getName(), p.getQuantity(),
        p.getProduct().getMeasure(), p.getPrice(), p.getTotal()));
        }
        return posStr.toString();
    } ...
```

Параметризирани методи (Generic Methods) (2)

- Извикване на параметризиран метод / конструктор:

```
result += Invoice.<T> getPositionsAsString(positions);
```

или можем да оставим Java автоматично да изведе типа:

```
result += Invoice.getPositionsAsString(positions);
```

Ограничени параметри за типа (Bounded Type Parameters)

- Можем да зададем горна граница на възможните типове, които могат да бъдат подавани като фактически параметри на класа/ метода/ конструктора:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) { ... }
```

или

```
public static <U extends Product & Printable> String
getPositionsAsString (List<Position<U>> positions) {
    ...
    p.getProduct().print();
    ...
}
```

Наследяване на типа и параметризирани класове (Subtyping)

- Ако **Product** наследява **Item** можем ли да кажем, че **List<Product>** също наследява **List<Item>** и може да се подава на негово място?
- Отговорът е „НЕ“, защото в **List<Product>** можем да добавяме само продукти, докато в **List<Item>** можем да добавяме всякакви наследници на **Item**, които не са продукти (напр. **Service** обекти).
- С други думи ако имаме:

```
interface Service extends Item; Service s = new Service( ...);
List<Service> services =  ...; services.add(s); // OK
interface Product extends Item; Product p = new Product( ...);
List<Product> products =  ...; products.add(p); // OK
List<Item> items = ...; items.add(s); items.add(p); // OK
items = products; // NOT OK
items = services; // NOT OK
```

Използване на ? като спецификатор на типа (Wildcards)

- Ако искаме да декларираме че очакваме конкретен, но неизвестен предварително тип, който разширява например класа **Item**, то можем да го обозначим с **?** :

```
Collection<? extends Item> items; // Upper bound is Item
items = products; // OK
items = services; // OK
Items.add(p); // NOT OK – Can not write into it – it is not safe!
Items.add(s); // NOT OK – Can not write into it – it is not safe!
for(Item i: items) { // OK – Can read it – it is known to be at least Item.
    System.out.println( i.getName() + „:“ + i.getPrice() );
}
List<? super Product> products; // Lower bound is Product
products.add(p); // OK – Can write into it – it is now safe.
Product p = products.get(0); //NOT OK may be superclass of Product
```

Type Erasure & Reification

- **Type Erasure** – начинът избран в Java за реализиране на параметризирани типове (с цел обратна съвместимост), при който информацията за типовите параметрите се ползва само за проверки при компилация и след това се изтрива изцяло, така че по време на изпълнение на програмата (runtime) параметризираният тип се свежда до първичния си (raw) тип:

  Collection<Product> products; --(runtime)--> Collection products;

  При този вариант за реализация има проблеми ако решим да създадем обект от параметричния тип с **new**, да конвертираме към него или да го проверим с **instanceof**.

- **Reification** – по-добра алтернативна стратегия реализирана в други езици като C++, Ada и Eiffel, при която информацията за параметричния тип е достъпна и по време на изпълнение.

Типизирани контейнери в Java 5 и 6 – Generics

- Позволяват проверката за тип на обектите да става по време на компилация – по ранно откриване на грешките
- Спестяват ни ненужни конверсии на типовете по време на изпълнение – чест източник на грешки
- Примери:

```
Collection <String> s = new ArrayList <String>();
Map <Integer, String> table = new HashMap <Integer, String>()
```

- Нов for -цикъл за обхождане на колекции:

```
for(String i: s) { System.out.println(i) }
```

Основни реализации и методи. Примери

- Асоциативни списъци – интерфейс **Map**
- Сравнение на основните реализации:
  - **HashMap**
  - **TreeMap**
  - **LinkedHashMap**
  - **WeakHashMap**
- Хеширане.
- Реализация на кешове – **Reference, SoftReference, WeakReference и PhantomReference**
- Избор на реализация.

Обработка на изключения в Java

- Задължителна обработка на изключенията в езика Java → сигурен и надежден код
- Разделяне на бизнес логиката на програмата от кода за обработка на грешки
- Клас **Throwable** → класове Error и Exception
- Генериране на изключения – ключова дума **throw**
- Обработка на изключения:
  - **try – catch – finally** блок
  - прехвърляне към извикващия метод - **throws**

Обработка на изключения в Java - 2

- Реализация на собствени изключения
- Конструктори с допълнителни аргументи
- Влагане и повторно генериране на изключения – причина Cause
- Специфика при обработката на **RuntimeException** и неговите наследници
- Завършване чрез **finally**

Новости при обработката на изключения в Java 7

- Обработка на множество изключения от в една и съща **catch** клауза:

```
catch (Exception1|Exception2 ex) {
    ex.printStackTrace();
}
```

- Програмен блок **try-with-resources**

```
String readInvoiceNumber(String myfile) throws IOException {
    try (BufferedReader input = new BufferedReader(new
        FileReader(myfile))) {
            return input.readLine();
        }
}
```

Новости в Java™ 5, 6 и 7

- Новости в Java 5, 6 и 7
  - enumeration types
  - static imports
  - autoboxing
  - variable argument lists
  - annotations
  - strings in switch
  - generics type inference

Enumeration Types

- Преди: public static final int INVOICE_SIMPLE = 0;
  public static final int INVOICE_VAT = 1;
- Проблеми:
  - Не се прави проверка на типа при компилация (възможна е невалидна стойност, не е typesafe)
  - Няма уникално пространство от имена и трябва да добавяме префикс INVOICE_ за да осигурим уникалност на името
  - При разпечатване стойностите на съответното поле са просто цели числа, които не дават реална информация за вида на фактурата
- Решение: `enum InvoiceType {SIMPLE,VAT};`

Enumeration Types - пример:

```
public class MyEnumeration {
   public enum InvoiceType { SIMPLE, VAT }
   public static void main(String[] args) {
      for(InvoiceType it : InvoiceType.values())
      System.out.println(it);
   }
}
```

Резултат: SIMPLE
VAT

Новости в Java™ 8

- Ламбда изрази и поточно програмиране – пакети **java.util.function** и **java.util.stream**)
- Референции към методи
- Методи по подразбиране и статични методи в интерфейси – множествено наследяване на поведение в Java 8
- Java 8 Data and Time API (JSR 310)
- Разширени възможности за използване на анотации върху Java типове (JSR 308)
- Функционално програмиране в Java 8 с използване на монади (напр. Optional, Stream) – предимства, начин на реализация, основни езикови идиоми, примери

Функционални интерфейси в Java™ 8

- Функционален интерфейс = интерфейс с един абстрактен метод SAM (Single Abstract Method) – @FunctionalInterface
- Примери за функционални интерфейси в Java 8:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
public interface Runnable {
    public void run();
}
public interface Callable<V> {
    V call() throws Exception;
}
```

Ламбда изрази – пакет **java.util.function**

Примери:

```
(int x, int y) -> x + y
() -> 42
(a, b) -> a * a + b * b;
(String s) -> { System.out.println(s); }
book -> book.getAuthor().fullName()
voter -> voter.getAge() >= legalAgeOfVoting
(person1, person2) -> person1.getAge() - person2.getAge()
(song1, song2) -> song1.getArtist().compareTo(song2.getArtist())
```

Правила за форматирне на ламбда изрази

- Ламбда изразите (функциите) могат да имат произволен брой параметри, които се ограждат в скоби, разделят се със запетаи и могат да имат или не деклариран тип (ако нямат - типът им се извежда от контекста на използване = target typing). Ако са само с един параметър, то скобите не са задължителни.
- Тялото на ламбда изразите се състои от произволен езикови конструкции (statements), разделени с **;** и заградени във фигурни скоби. Ако имаме само една езикова конструкция – израз то използването на фигурни скоби не е необходимо – в този случай стойността на израза автоматично се връща като стойност на функцията.

Пакет **java.util.function**

- **Predicate<T>** – предикат = булев израз представящ свойство на обекта подавано като аргумент
- **Function<A,R>**: функция която приема като аргумент A и го трансформира в резултат R
- **Supplier<T>** – с помощта на **get()** метод всеки път връща инстанция (обект) – фабрика за обекти
- **Consumer<T>** – приема аргумент (метод **accept()**) и изпълнява действие върху него
- **UnaryOperator<T>** – оператор с един аргумент T -> T
- **BinaryOperator<T>** – бинарен оператор (T, T) -> T

Поточно програмиране (1)

Примери:

```
books.stream().map(book ->
    book.getTitle()).collect(Collectors.toList());
books.stream()
       .filter(w -> w.getDomain() == PROGRAMMING)
       .mapToDouble(w -> w.getPrice()) .sum();
document.getPages().stream()
     .map(doc -> Documents.characterCount(doc))
     .collect(Collectors.toList());
document.getPages().stream()
     .map(p -> pagePrinter.printPage(p))
     .forEach(s -> output.append(s));
```

Поточно програмиране (2)

Примери:

```
document.getPages().stream()
        .map(page -> page.getContent())
        .map(content -> translator.translate(content))
        .map(translated -> new Page(translated))
        .collect(Collectors.collectingAndThen(
                Collectors.toList(),
                pages -> new
        Document(translator.translate(document.getTitle()), pages)));
```

Референции към методи

- Статични методи на клас – Class::staticMethod
- Методи на конкретни обектни инстанции – object::instanceMethod
- Методи на инстанции реферирани чрез класа – Class::instanceMethod
- Конструктори на обекти от даден клас – Class::new

```
Comparator<Person> namecomp =
        Comparator.comparing(Person::getName);

Arrays.stream(pageNumbers).map(doc::getPageContent)
        .forEach(Printers::print);

pages.stream().map(Page::getContent).forEach(Printers::print);
```

Статични и Default методи в интерфейси

- Методите с реализация по подразбиране в интерфейс са известни още като virtual extension methods или defender methods, защото дават възможност интерфейсите да бъдат разширявани, без това да води до невъзможност за компилация на вече съществуващи реализации на тези интерфейси (което би се получило ако старите реализации не имплементират новите абстрактни методи).
- Статичните методи дават възможност за добавяне на помощни (utility) методи – например factory методи директно в интерфейсите които ги ползват, вместо в отделни помощни класове (напр. Arrays, Collections).

Пример за default и static методи в интерфейс

```
@FunctionalInterface
interface Event {
    Date getDate();
    default String getDateFormatted() {
        return String.format("%1$td.%1$tm.%1$tY", getDate());
    }
    public static <T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> getKey) {
        return (c1, c2) -> getKey.apply(c1).compareTo(getKey.apply(c2));
    }
}
Event current = () -> new Date();
System.out.println(current.getDateFormatted());
```

Функционално програмиране и монади

- Понятие за **монада** във функционалното програмиране (теория на категориите) – **Монадата** е множество от три елемента:
  1) Параметризиран тип **M<T>**
  2) "**unit**" функция: **T -> M<T>**
  3) "**bind**" операция: **bind(M<T>, f:T -> M<U>) -> M<U>**
- В Java 8 пример за монада е класът **java.util.Optional<T>**
  1) Параметризиран тип: **Optional<T>**
  2) "**unit**" функции: **Optional<T> of(T value) , Optional<T> ofNullable(T value)**
  3) "**bind**" операция: **Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**

Литература и интернет ресурси

- Eckel, B., Thinking in Java, 4th edition, Prentice Hall, 2006, http://mindview.net/Books/TIJ4
- Oracle® Java™ Technologies webpage – http://www.oracle.com/technetwork/java/index.html
- Новости в Java 5.0 - http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- Новости в Java 6 - http://java.sun.com/javase/6/webnotes/features.html
- Методи за сортиране в Wikipedia - http://en.wikipedia.org/wiki/Sorting_algorithm

Литература и интернет ресурси

- Oracle tutorial – lambda expressions - http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Java SE 8: Lambda Quick Start - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html#
- OpenJDK Lambda Tutorial - https://github.com/AdoptOpenJDK/lambda-tutorial

Благодаря Ви за вниманието!

Въпроси?