



**Lambda изрази. Stream API.**

# Съдържание

Функционално програмиране

Lambda изрази

Stream API

**Функционално програмиране**

# Какво е ФП?

In computer science, **functional programming** is a **programming paradigm**, a style of building the structure and elements of computer programs, that treats computation as the evaluation of **mathematical functions** and **avoids changing-state and mutable data**. It is a **declarative programming** paradigm, which means programming is done with **expressions**.

- Уикипедия, ноември 2018

# ФП е парадигма

Стил на програмиране, следващ определен дизайн, правила и принципи.

Може да се пише „функционално“ на императивен език, но на функционален е много по-лесно и интуитивно.

Основни програмни парадигми:

- императивно (ООП, структурно)
- функционално
- логическо

# Промяна на данни

При императивните езици дадена променлива може да променя стойността си.

```
x = x + 1; // Няма математически смисъл!
```

При функционалните езици, изчислителният модел е подобен на математиката: щом веднъж е зададена стойност на дадена променлива, тя не може да се изменя.

Обект, чието състояние не може да се измени никога, се нарича *immutable*.

Immutable обектите имат няколко предимства като thread safety и caching

# Statement vs Expression

В езиците за програмиране има условно два типа контролни инструкции:

*Statement* – не връщат резултат

*Expression* – връщат резултат

В императивните езици се ползват основно statements, докато във ФП expressions.

Statements имат смисъл да бъдат изпълнени единствено, ако извършат страничен ефект.

# Странични ефекти

Страничен ефект е всяко действие във функция, което променя външно за нея състояние:

- промяна на данни (външни за функцията)
- I/O операция
- хвърляне на изключение
- извикване на друга функция, която предизвиква side-effect

Всяка функция в Java, която „връща“ void, извършва страничен ефект.



# Чисти функции

Чистите функции не извършват странични ефекти:

- винаги когато извикаме функцията с даден параметър е гарантиран един и същ резултат => лесни за анализ на проблеми
- резултатът им може да се запази и преизползва
- извикването на функцията може да бъде заменено с резултата (referential transparency)
- чистите функции са лесно композируеми

# Чисти функции - пример

Събирането на две числа е чиста функция.

```
int add(int a, int b) {  
    return a + b;  
}  
add(3, 5); // == 8  
add(3, 5); // == 8
```

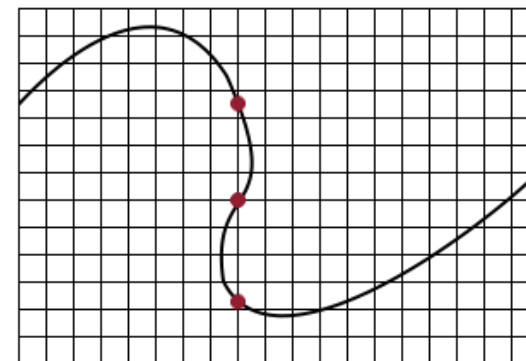
Трансфер на пари в банкова сметка не е!

*// външно за функцията състояние*

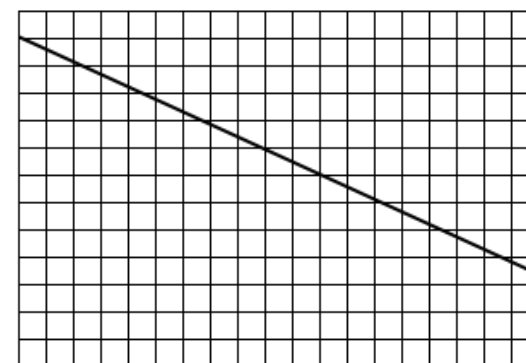
```
int balance = 100;  
int transfer(int amount) {  
    balance += amount;  
    return balance;  
}
```

```
transfer(10); // == 110  
transfer(10); // == 120
```

Тест на вертикалната линия



Not a function



Function

# Декларативен стил

Java използва основно императивен стил на програмиране.

При декларативния стил имаме акцент върху това **какво** трябва да се направи, а при императивния - **как** трябва да се направи.

Концепцията на декларативния стил на програмиране е, че задаваме само логиката за изчисление, а не контролните инструкции (if/else, for и т.н.).

# Декларативен стил - пример

```
List<Integer> result = new ArrayList<>();  
for (int i = 0; i < data.size(); i++) {  
    if (data[i] % 3 == 0) {  
        result.add(data[i]);  
    }  
}
```

```
data.stream().filter(new Predicate<Integer>() {  
    public boolean test(Integer value) {  
        return (value % 3 == 0);  
    }  
});
```

```
data.stream().filter(i -> i % 3 == 0);
```

Императивен стил - програмистът е отговорен **как** трябва да се обходи структурата и построи резултата

Декларативен стил – библиотеката, която предоставя структурата, е отговорна за итерацията. Ние само задаваме **какво** да се направи на всяка стъпка.

Отново декларативен стил – този път с много по-удобен синтаксис.

# Декларативен стил



Ясно разграничение между изчислителната логика и control-flow командите.

Възможност за генерална оптимизации от библиотеката/компилятора/езика



Губи се контрол над изпълнението => възможност за конкретна оптимизация.

В общия случай по-ниска ефективност.

# Функции от по-висок ред

Функционалните езици третират функциите като първостепенни конструкции, т.е. може да правим всичко, което може да правим с обекти в ООП:

- декларираме/инициализираме функция
- подаваме функция като параметър
- връщаме функция като резултат от друга функция

Функциите от по-висок ред ни предоставят средство, с което да композираме логика, комбинирайки функции.

# Предимства на ФП

По-удобен и смислен синтаксис в дадени ситуации;

Чистите функции са композируеми;

(По-)лесен за анализиране на проблеми код;

Декларативният стил позволява част от инфраструктурния код да се премести от програмиста към библиотеката.

Възможност за редица оптимизации като:

- Memoization
- Lazy-evaluation
- Паралелизация

# Ламбда изрази



# Функцията като тип

В Java (< 8) няма възможност за използването на функция като отделен тип. Функциите са винаги асоциирани или с клас (статични методи) или с инстанция на клас (методи).

Предаването на функция като параметър на друга функция е много удобно, в случай че само *поведението* е значимо.

```
// Имплементираме клас, само за да подадем поведение (при натискане) като
// параметър, анонимният вътрешен клас не е асоцииран със състояние (state)
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed!"); // Поведение!
    }
});
```

# Функцията като тип

Най-често използваният похват за подаване на *поведение* на метод е чрез т.нар. „SAM (Single Abstract Method) интерфейси“, чиято имплементация се представя чрез анонимен вътрешен клас.

SAM интерфейсите съдържат един-единствен метод (пр. `Runnable`, `Comparable`, `ActionListener`). В Java 8 се наричат *Functional Interface*.

Недостатък на този подход е прекалено разтегленият синтаксис, който води до т.нар. „вертикален проблем“.

# Вертикален проблем

```
list.stream()
    .filter(
        new Predicate<String>() {
            public boolean test(String str) {
                return (str.length() > 5);
            }
        })
    .forEach(
        new Consumer<String>() {
            public void accept(String s) {
                System.out.println(s);
            }
        });
```



значим код



ЛОГИЧЕСКИ ИЗЛИШЕН,  
НО НЕОБХОДИМ КОД

# Ламбда изрази

Добавен в Java 8 с т. нар. „проект Ламбда“. Счита се за най-голямата промяна, правена някога в езика.

Позволяват използване на функционален стил на програмиране в Java като:

- добавя нов логически функционален тип;
- въвежда лесен и удобен начин за представяне на анонимна функция чрез нововъведения литерал на функция, който решава „вертикалния проблем“;
- улеснява писането и поддръжката на код чрез възможности за декларативен стил на програмиране.

# Какво е Lambda израз?

Анонимна функция;

Литерал, който представлява много удобен и лесен начин за предоставяне имплементация на функционален интерфейс;

// По-удобен и интуитивен начин да подадем "поведение"

```
button.addActionListener(  
    (ActionEvent e) -> System.out.println("Button pressed"));
```

Може да бъде:

- Captured (closure) – използва в тялото си променливи, които са дефинирани извън ламбда израза;
- Non-captured – не използва външни променливи.

# Lambda литерал - синтаксис

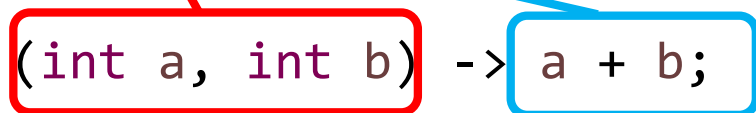
Литерал – нотация за създаване и инициализация на обект без използване на оператора **new**, а чрез дефиниран в езика синтаксис:

```
int i = 5; String s = "123";  
float f = 1.1f; int[] a = {1, 2, 3};
```

Функционалният литерал предоставя възможност за удобна имплементация на lambda израз

Състои се условно от две части, разделени с оператора стрелка (->):

- дефиниция на функция (списък на входящи параметри)
- тяло на функция (логиката, извършвана в метода)



The diagram shows a lambda expression: `(int a, int b) -> a + b;`. The parameter list `(int a, int b)` is enclosed in a red rounded rectangle, and the body `a + b;` is enclosed in a blue rounded rectangle. A red arrow points from the first bullet point of the list above to the red box, and a blue arrow points from the second bullet point to the blue box.

```
(int a, int b) -> a + b;
```

# Lambda – синтаксис дефиниция

Съдържа списък с входните параметри. Ламбда израз може да има нула, един или повече параметри. Параметрите се поставят в скоби () и се изреждат със запетая;

Празни скоби () означават, че функцията не приема входни параметри;

Типът на параметрите може да бъде експлицитно зададен или изпуснат („type inference”);

Когато има един-единствен параметър и неговият тип може да бъде изпуснат, може да се изпуснат скобите;

# Примери синтаксис дефиниция

`(String name) -> System.out.println("Hello " + name);` // 1 параметър

`(name) -> System.out.println("Hello " + name);` // изпускаме типа

`name -> System.out.println("Hello " + name);` // изпускаме скобите

`(int a, int b, double c) -> a + b + c;` // 3 параметъра

`(a, b, c) -> a + b + c;` // Изпускаме типа на параметрите



# Lambda – синтаксис тяло

Тялото на ламбда израз може да има нула, един или повече изрази;

Когато има единствен израз в тялото, къдравите скоби {} могат да се пропуснат. В противен случай, тялото се поставя в къдрави скоби и отделните изрази се разделят чрез “;” или вложени къдрави скоби;

Когато има единствен израз в тялото, return клаузата може да се изпусне;

Когато имаме повече от един израз, трябва експлицитно да предоставим return клауза;

NB! Препоръчва се lambda изразите да са one-liners!

# Примери – синтаксис тяло

```
() -> {}; // празно тяло на функция
(a, b) -> a + b; // изпуснали сме return, защото има само 1 израз
(a, b) -> {a = 5; b = 2; return a + b;}; // 3 израза в тялото
(a, b) -> { // Тялото може
    a = 5; // да бъде разположено
    b = 2; // на няколко
    return a + b; // реда (Лош стил!)
};
```

# Тип на Lambda израз

Java е статично типизиран език => Lambda изразите си имат тип!

По време на компилация се знае типът на ламбда израза (даден функционален интерфейс), както и типът на параметрите на функцията и на връщания резултат.

```
ActionListener al = (ActionEvent e) -> System.out.println(e.getSource());
```

# Type inference

С цел по-удобен дизайн на ламбда израза, може да се изпусне типа на променливите, но това е в случай че няма двусмислие и Java компилаторът може да е сигурен какъв е типът на дадена променлива (“type inference”)

```
ActionListener al = (e) -> System.out.println(e.getSource());
```

# Функционални интерфейси

В Java 8 SAM интерфейсите се наричат „функционални интерфейси“. Всеки SAM интерфейс е ФИ.

Опционално може да се анотира с `@FunctionalInterface`. Сама по себе си тази анотация не прави нищо, освен че ще даде компилационна грешка, ако някой добави нов метод към този интерфейс (подобно на `@Override`)

Java 8 добавя много нови функционални интерфейси. Представяват “форма” за основните типове ламбда изрази, но не за всички. Намират се в пакета `java.util.function` и са предназначени за ползване както от Java SDK, така и от програмистите.

# Функционални интерфейси

Основните предефинирани форми на функция са:

| <code>java.util.function</code>        | Ламбда нотация                 |
|--|--------------------------------|
| <code>Consumer&lt;T&gt;</code>         | <code>(T) -&gt; void</code>    |
| <code>Predicate&lt;T&gt;</code>        | <code>(T) -&gt; boolean</code> |
| <code>Supplier&lt;R&gt;</code>         | <code>() -&gt; R</code>        |
| <code>Function&lt;T, R&gt;</code>      | <code>(T) -&gt; R</code>       |
| <code>BiFunction&lt;T, U, R&gt;</code> | <code>(T, U) -&gt; R</code>    |

# Контекст на Lambda израз

Един и същ ламбда литерал, може да има различен тип в зависимост от контекста:

```
Supplier s = () -> "Hello World"; // Supplier  
Callable c = () -> "Hello World"; // Callable
```

Възможни контексти за ламбда израз:

- декларация на променлива;
- присвояване;
- връщане от метод;
- инициализация на масив;
- параметър на метод или конструктор;
- в тялото на ламбда израз;
- тринарен оператор;
- „cast“ операции.

# Captured Lambda

Когато ламбда изразите не използват локални променливи, дефинирани извън самия израз, ламбда изразът е „non-captured“, а в противен случай се нарича “*captured*” или „*closure*”.

- За да се използва променлива извън контекста на ламбда израза, тя трябва да бъде **final** или *ефективно final* (т.е. никога да не ѝ се променя стойността след инициализация).
- **this** в тяло на lambda израз „сочи“ към обграждащия клас



# Метод референции

Много често в ламбда изразите просто делегираме параметрите на някой съществуващ метод:

```
list.forEach(s -> System.out.println(s));  
list.filter(s -> s.equalsIgnoreCase("ivan"));
```

Метод референциите са удобен синтаксис, с който може да се създаде ламбда израз от вече съществуващ метод:

```
list.forEach(System.out::println);  
list.filter("ivan"::equalsIgnoreCase);
```

Дефиницията на метод референцията трябва да е същата като метода на функционалния интерфейс, който методът приема.

# Метод референции - видове

Статичен метод (**ClassName::methName**)

`String::valueOf`

Метод на конкретна инстанция от даден клас (**instanceRef::methName**)

`“FPRocks!”::equals`

Метод на произволна инстанция на даден клас (**ClassName::methName**)

`String::toLowerCase`

Конструктор (**ClassName::new**)

`String::new`, `String[]::new`

Метод на суперкласа (**super::methName**)

`super::equals`

# Композиция на функции

`Java.util.function.Function` предоставя помощни методи, с които може да „композираме“ функция от вече съществуващи.

```
// z1(x) = f(g(x))
```

```
Function<Integer, Integer> z1 = f.compose(g);
```

```
// z2(x) = g(f(x))
```

```
Function<Integer, Integer> z2 = f.andThen(g);
```

# Stream API

# Еволюция на Collection API

Съществуващият Collections API би изглеждал много различно, ако ламбда изразите съществуваха от началото.

Въвеждането на нов API (Collections II, New Collections API и т.н.) би решило проблема, но ще отнеме години да измести от употреба старата версия.

Решението е:

- добавяне на нови default методи към съществуващите интерфейси (`Iterable.forEach`, `Collections.removeIf` и т.н.), позволяващи работа с ламбда изрази;
- добавяне на нова „stream“ абстракция, позволяващата верижни операции;
- лесна взаимозаменяемост на линейна и паралелна обработка (`stream()` или `parallelStream()`).

# Stream API

`java.util.stream` – въведен в Java 8

Stream представлява абстракция на последователност от стойности и възможност за „функционални“ операции върху тях;

Collections API предоставят удобен начин за създаване на Stream от съществуваща колекция.  
(`java.util.Collection.stream()`)

Мотивация:

- декларативен стил на работа с колекции (и потоци от данни като цяло);
- възможност за верижно изпълнение.
- лесна миграция от последователно към паралелно изпълнение;

# Stream vs Collections

Потоците не са алтернатива на колекциите!

Колекциите са отговорни за съхранение и достъп до техните данни.

Особености на Stream:

- няма собствено хранилище, а използва източник на данни (колекция, I/O и т.н.);
- функционална същност – не модифицират вътрешните данни;
- колекциите винаги имат краен брой елементи, докато Stream може да бъде безкраен;
- позволяват lazy операции;
- елементите могат да се „консумират“ само веднъж подобно на итератор.

# Създаване на потоци I

```
// създава поток от Колекция ('names' е ArrayList<String>)
Stream<String> namesStream = names.stream();

// създава поток от масив
IntStream primesStream = Arrays.stream(primes);

// създава поток от краен брой обекти.
Stream<Object> objects = Stream.of("A string", 123, 3.14f);

// създава поток от целочислена редица в даден интервал
IntStream zeroToNine = IntStream.range(0, 10);

// Празен поток
Stream<Stream> emptyStream = Stream.empty();
```



## Създаване на потоци II

// Конкатенация на два потока (left и right са Stream<Object>)

```
Stream<Object> union = Stream.concat(left, right);
```

// Редовете на файл могат да бъдат взети като поток с:

```
Stream<String> lines = new BufferedReader(new FileReader(file)).lines();
```

// Поток, съдържащ всички файлове в директория:

```
Stream<Path> files = Files.list(Paths.get(".")). // не е рекурсивно!
```

# Създаване на потоци III

От генератор-функция – полезно за създаване на безкраен поток.

```
// Безкраен поток от случайни числа в интервала [0 - 100)
```

```
Stream<Integer> randNums = Stream.generate(() -> new Random().nextInt(100));
```

```
// Безкраен поток от четни числа
```

```
Stream<Integer> evens = Stream.iterate(0, (x) -> x + 2);
```

# Операции върху Stream

Stream API предоставя набор от функции от по-висок ред, чрез които декларативно можем да обработим данните.

Препоръчва се ламбда изразите да не модифицират:

- данните на източника (non-interference)
- външни променливи (stateless)

Операциите с потоци биват два вида:

- Междинни – връщат “Stream” обект като резултат.
- Терминални – връщат обект, различен от Stream, или не връщат резултат.

Междинните операции се изпълняват lazy.

# Stream API pipeline

След като междинните операции връщат като резултат Stream, е възможно да извикаме нова операция върху Stream.

// Верижно изпълнение

```
List<Person> result = people.stream()  
    .filter(p -> p.getFirstName().equals("Nikolay"))  
    .filter(p -> p.getAge() > 25)  
    .sorted(Comparator.comparing(Person::getLastName))  
    .collect(Collectors.toList());
```

Легенда:

Източник

Междинна оп

Терминираща оп

# filter

Междинна операция, приема функция ( $T \rightarrow \text{Boolean}$ ) и връща поток само с елементите, за които резултатът е true.

```
// Връща Stream<Employee> с всички служители, отговарящи на критерия  
// Stream<T> -> filter -> Stream<T>  
employees.filter(e -> e.getAge() > 25);
```

# map

Междинна операция, приема функция  $(T \rightarrow V)$  и връща поток със същия брой елементи, но от новия тип  $(V)$

```
// Връща Stream<String>, съдържащ само имената на служителите  
// Stream<T> -> map -> Stream<V>  
employees.map(e -> e.getName());
```

Специализирани map операции:

mapToInt, mapToLong, mapToDouble

# flatMap

Междинна операция, приема функция `(T -> Stream[V])` и връща поток със същия брой елементи с линейна структура (flat)

```
// Връща 1-мерен поток с всички възнаграждения на служителите.
```

```
// Stream<T> -> flatMap -> Stream<V>
```

```
employees.flatMap(e -> Stream.of(e.getSalary(), e.getBonus()));
```

# forEach

Терминална операция, приема функция ( $T \rightarrow \text{void}$ ) и не връща резултат! (т.е. няма функционална природа).

```
// Stream<T> -> forEach -> void
```

```
// Изписва първото име и възрастта на всеки служител
```

```
employees.forEach(e -> System.out.println(e.getName() + " " + e.getAge()));
```



# reduce

Терминална операция, приема функция  $((T, T) \rightarrow T)$  и връща единичен резултат.

```
// Връща сумата на заплатите на всички служители
```

```
// Stream<T> -> reduce -> T
```

```
employees.mapToDouble(Employee::getSalary).reduce((res, el) -> res + el);
```

Специализирани reduce операции (само за Int/Long/DoubleStream):

```
sum(), min(), max(), average()
```

# reduce – генерална форма

Reduce има и по-обща форма, която дава възможност да се върне резултат, различен от типа на потока  $((V, T) \rightarrow V)$

```
// Генерална форма на reduce, обикновено по-чисто може да се  
// опише чрез комбинация 'map-reduce'  
// Stream<T> -> reduce -> V
```

```
double result = employees.reduce(  
    0.0,                                // начална стойност  
    (res, el) -> res + el.getSalary(), // accumulator  
    (left, right) -> left + right);    // combiner
```

# collect

Възможно е “reduce” операция да върне резултат, който не е единичен обект, а колекция.

`collect` предоставя възможност да акумулиране резултата във външен контейнер (колекция)

Класът `Collectors` предоставя помощни методи за най-често използваните колекции:

`toCollection` / `toList` / `toSet` / `toMap`

// Връща `List<Employees>`

```
employees.filter(e -> e.getAge() > 25).collect(Collectors.toList());
```

# Колектори

Помощният клас `java.util.stream.Collectors` също така предоставя полезни помощни функции за `reduce` операции.

`groupBy` – групира елементите по даден признак

```
// Резултатът е от тип Map<Department, List<Employee>>  
employees.collect(Collectors.groupingBy(Employee::getDepartment));
```

`joining` – конкатенира `String` обекти в един цял низ.

```
employees.map(Employee::getName).collect(Collectors.joining(", "));
```

# Short Circuiting операции

Операции, които прекратяват обхождането на потока преждевременно. Полезни са при безкрайни потоци.

`findFirst()` – първия елемент на потока

`findAny()` – връща произволен елемент от потока

`allMatch(T -> boolean)` – дали всички елементи отговарят на дадено условие

`anyMatch(T -> boolean)` – дали някой елемент отговаря на дадено условие

`limit(int n)` – връща `n` елементи (междинна операция)

# Други операции

`count()` – връща броя на елементите (терминална)

`distinct()` – връща поток с уникални елементи (на базата на `equals`)

`sorted()` – подрежда елементите

`sorted(Comparator)` – сортира по зададен признак

# Optional API

Контейнер обект, който може да съдържа или не дадена стойност.

Някои от методите на Stream API връщат като резултат `Optional<T>`.

Начин за избягване на `null` проверки или `NullPointerException`, предоставя възможности за:

```
// Проверка дали има стойност или не
```

```
boolean isPresent = optionalEmployee.isPresent();
```

```
// Изпълнява действие само, ако има налична стойност
```

```
optionalEmployee.ifPresent(System.out::println);
```

# Optional API

// Връща стойността на контейнера или хвърля NoSuchElementException

```
Employee e = optionalEmployee.get();
```

// Връща default стойност, ако няма обект

```
optionalEmployee.orElse(Employee.UNKNOWN);
```

// Хвърля определена грешка, ако няма стойност

```
optionalEmployee.orElseThrow(NoSuchElementException::new);
```



# Паралелно изпълнение

Както вече знаем, потоците „итерират“ вътрешно източника на данни, което дава възможност за редица оптимизации – като например да изпълним pipeline от задачи паралелно.

Целта на паралелното изпълнение е да подобрим бързодействието на дадено изчисление, възползвайки се от мултипроцесорната архитектура.

Можем да превърнем всеки поток в паралелен, използвайки метода `parallel()`

Паралелният поток има същия API като серийния.

NB! Паралелното програмиране крие опасности:

- не всички задачи са подходящи за паралелизация
- не всички структури от данни са подходящи за паралелна обработка.
- не винаги постигаме оптимизацията, която целим

# Заключение

ФП има редица предимства пред императивното програмиране, но и редица недостатъци – избирайте разумно кой подход ще ви даде по-добро (елегантно/бързо) решение на конкретната задача.

Достатъчно кратки и ясни ламбда изрази могат да подобрят значително четимостта на кода.

Когато използвайте Lambda изрази и Stream API, стремете се към функционален стил, избягвайте да мутирате данни или извършвате странични ефекти.

Ако изпълнявате Stream API pipeline паралелно, винаги изпълнявайте релевантни performance тестове.

**Въпроси?**

## ИЗПОЛЗВАНА ЛИТЕРАТУРА

<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>