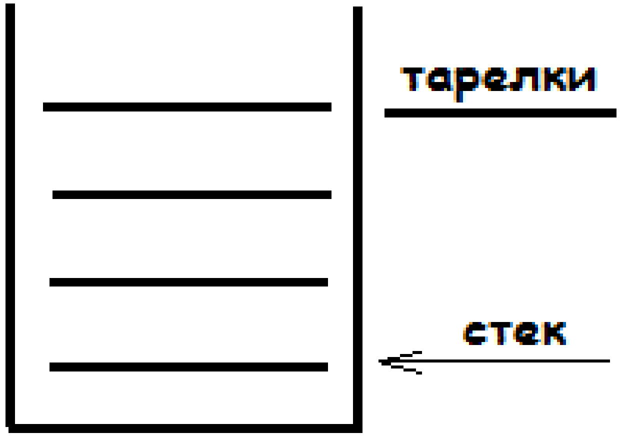
Лекция 10.
Рекурсия.
Бинарный поиск

Стек

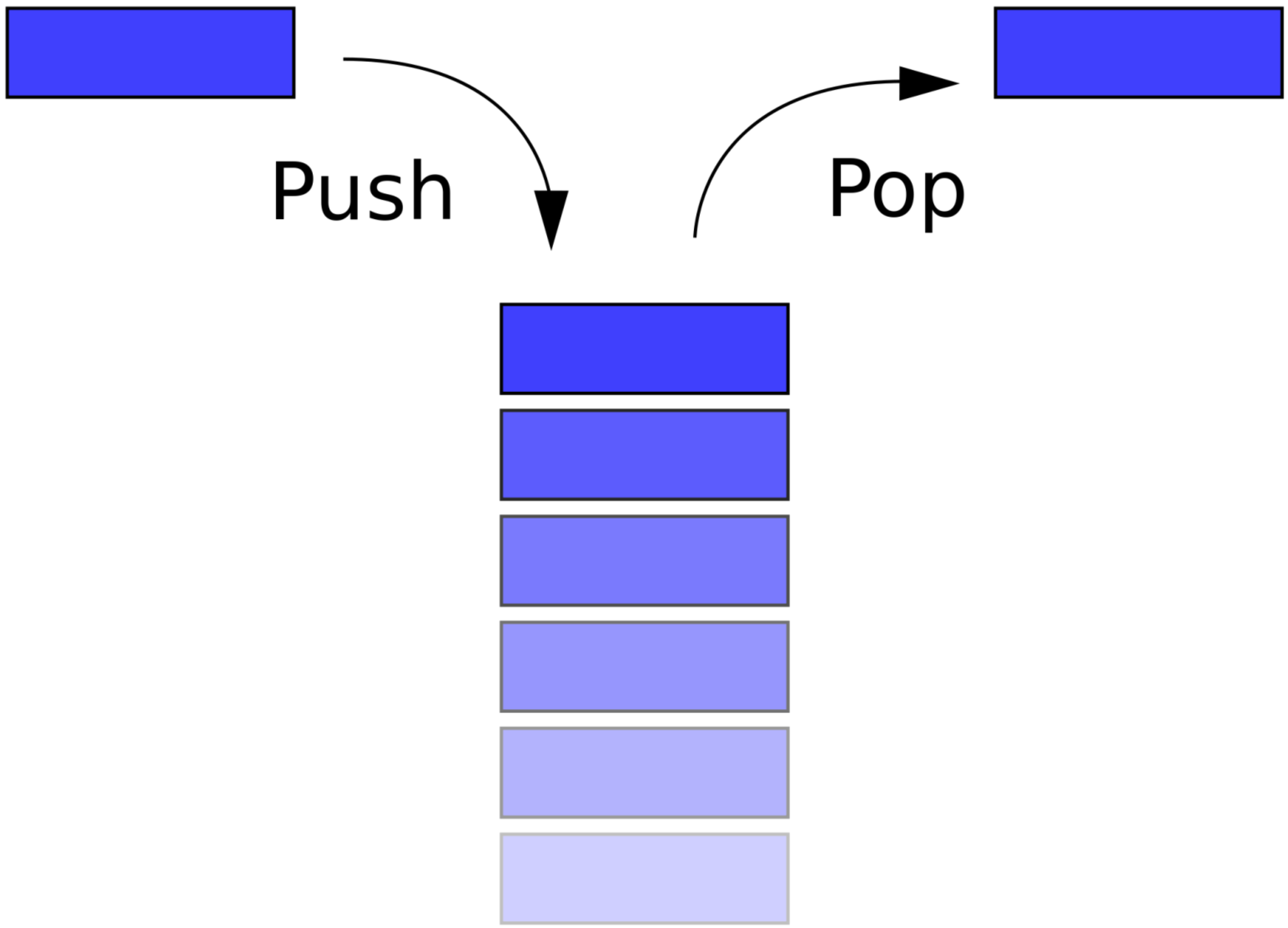
- **Стек** – это структура данных, которая работает по принципу «последний вошел – первый вышел» (**LIFO** - Last in, first out)
- **Пример в жизни:** стопка тарелок
- Верхнюю тарелку мы положили последней, но забираем первой



Стек

- Если рассматривать стек как класс, то это будет класс с тремя основными функциями:
- void push(Element e);
  вставляет элемент в верхний конец стека
- Element pop();
  вытаскивает верхний элемент из стека и возвращает его
- int size();
  выдает количество элементов в стеке

Стек



Стек

- Структура стека широко используется во многих известных алгоритмах
- Кроме того, последовательность вызова функций друг из друга тоже представляет собой стек, который называется **стеком вызовов**

Стек вызовов

- **Стек вызовов** – стек, который хранит порядок вызова функций друг из друга
- Каждый элемент стека соответствует одному вызову функции и хранит в себе:
  - Переданные в функцию аргументы
  - Локальные переменные
  - Адрес возврата в вызывающую функцию
- **Адрес возврата** – это адрес памяти, где находится код, который должен исполняться после завершения функции

Адрес возврата

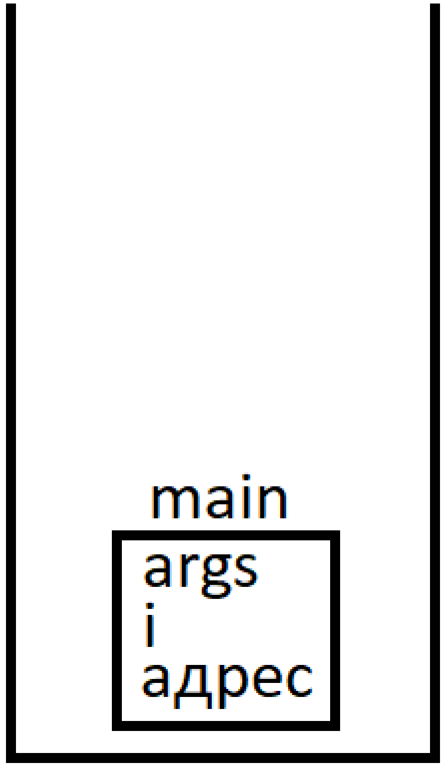
- **Адрес возврата** – это адрес памяти, где находится код, который должен исполняться после завершения функции

Адрес возврата указывает сразу за вызов функции

- ```
  public static int g() {
    return 3;
  }
  public static void main(String[] args) {
    int i = 5;
    g();
  }
  ```
  ```

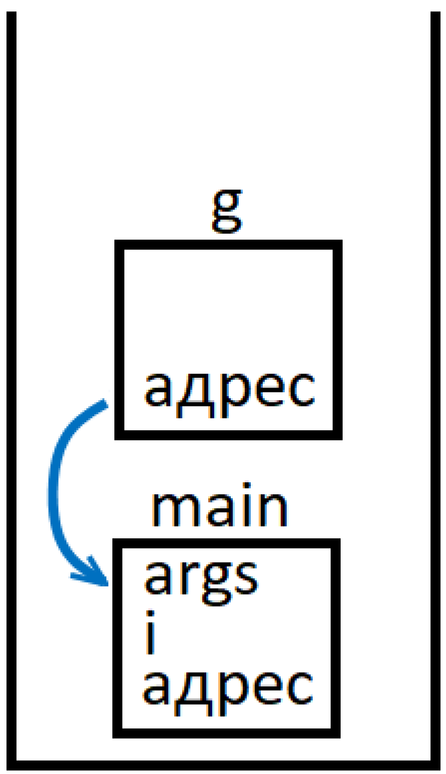
Стек

```java
public static int f() {
  return g() + 1;
}
public static int g() {
  return 3;
}
public static void main(String[] args) {
  int i = 5;
  g();
  f();
}
```



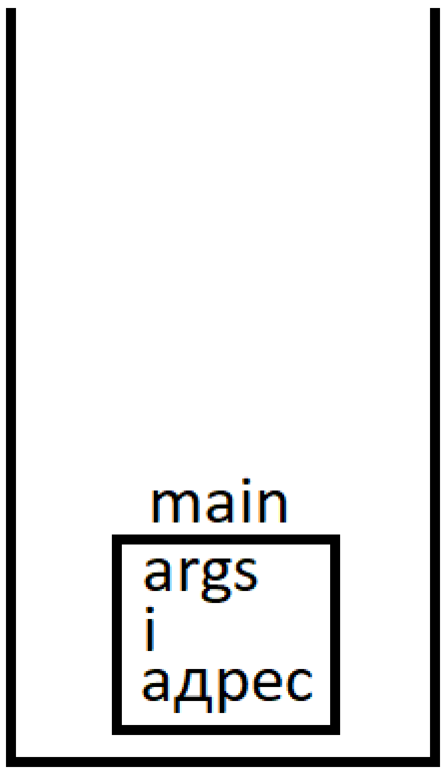
Стек

```
public static int f() {
    return g() + 1;
}
public static int g() {
    return 3;
}
public static void main(String[] args) {
    int i = 5;
    g();
    f();
}
```



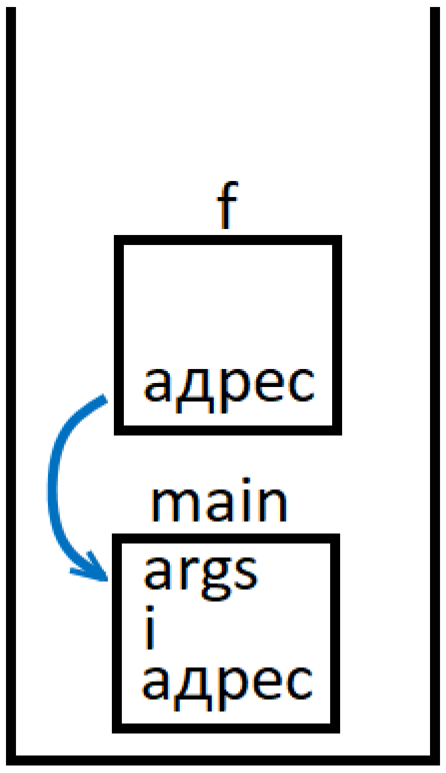
Стек

```
public static int f() {
    return g() + 1;
}
public static int g() {
    return 3;
}
public static void main(String[] args) {
    int i = 5;
    g();
    f();
}
```



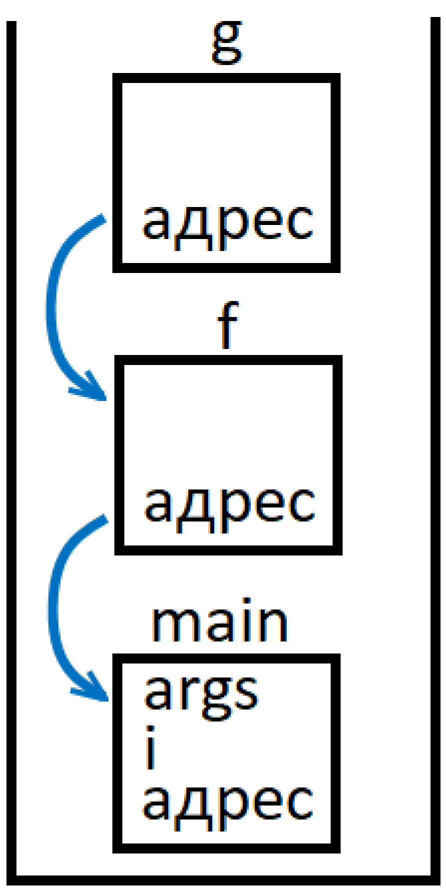
Стек

- ```
  public static int f() {
     return g() + 1;
  }
  public static int g() {
     return 3;
  }
  public static void main(String[] args) {
     int i = 5;
     g();
     f();
  }
  ```


  ```

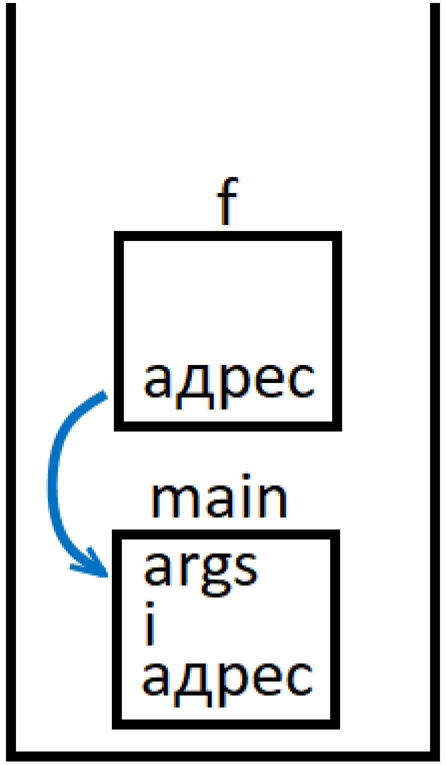
Стек

```
public static int f() {
   return g() + 1;
}
public static int g() {
   return 3;
}
public static void main(String[] args) {
   int i = 5;
   g();
   f();
}
```



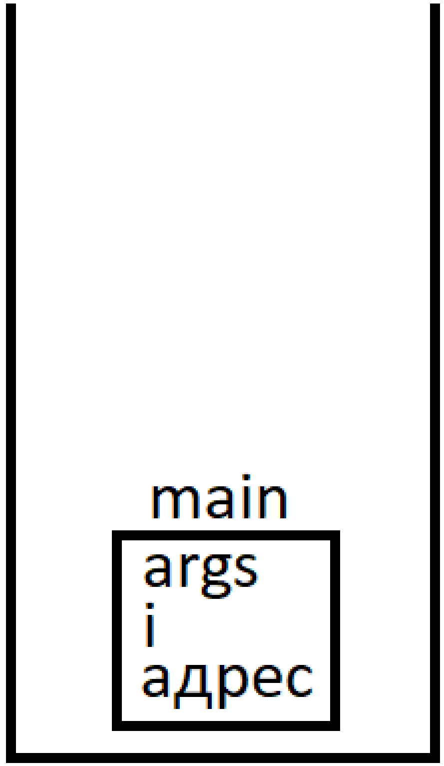
Стек

```
public static int f() {
   return g() + 1;
}
public static int g() {
   return 3;
}
public static void main(String[] args) {
   int i = 5;
   g();
   f();
}
```



Стек

```
public static int f() {
    return g() + 1;
}
public static int g() {
    return 3;
}
public static void main(String[] args) {
    int i = 5;
    g();
    f();
}
```



Рекурсия

- **Рекурсия** – это вызов функцией самой себя
- Просто в стек добавляется еще один вызов той же самой функции (с точки зрения программы это ничем не отличается от вызова другой функции)
- Пример:
-
```
public static void f(int i) {
    System.out.println(i);
    f(i + 1);
}
```
- Эта функция упадет из-за переполнения стека (размер стека ограничен)

Рекурсия

- При написании рекурсивных функций важно, чтобы в какой-то момент рекурсия завершалась
-
```
public static void f(int i) {
  if (i >= 10) {
    return;
  }
  System.out.println(i);
  f(i + 1);
}
```
- Такая функция уже не упадет

Рекурсия

- Предыдущий пример работает также как цикл от начального значения i до 10, но рекурсия значительно медленнее и требует больше памяти, чем цикл
- Поэтому при возможности следует избегать рекурсию

Зачем нужна рекурсия?

- Некоторые алгоритмы при помощи нее реализовать проще и быстрее
- Если производительность неважна, и глубина вызовов небольшая, то это вполне подходящий вариант

Вычисление факториала

- Рекурсия часто встречается и в математике

- **Факториал**: $n! = 1 \cdot 2 \cdot 3 \cdots \cdot n$
- Считается что $0! = 1$
- Можно заметить, что $n! = (n - 1)! \cdot n$,
  $(n - 1)! = (n - 2)! \cdot (n - 1)$ и т.д.
- Это и есть рекурсивная формула

Вычисление факториала

- $n! = (n-1)! \cdot n$ $0! = 1$

-
```java
public static int getFactorial(int n) {
    if (n == 0) {
        return 1;     // завершение рекурсии
    }

    return getFactorial(n - 1) * n;
}
```

Хвостовая рекурсия

- **Хвостовая рекурсия** – вариант рекурсии, при котором рекурсивный вызов функции происходит последней операцией в функции
- Такую рекурсию всегда можно преобразовать в цикл
- В некоторых языках это делается автоматически (не в Java)

Пример хвостовой рекурсии

- Пример вычисления факториала с хвостовой рекурсией:
- ```
  private static int getFactorialTail(int n, int result) {
      if (n == 0) {
          return result;
      }
      return getFactorialTail(n – 1, result * n);
  }
  ```
- Предполагается, что первую функцию никто напрямую вызывать не будет, а будут вызывать эту функцию:
- ```
  public static int getFactorial(int n) {
      return getFactorialTail(n, 1);
  }
  ```
- А эта функция передает правильное начальное значение 1

Нехвостовая рекурсия

- ```
  public static int getFactorial(int n) {
      if (n == 0) {
          return 1;
      }
      return getFactorial(n - 1) * n;
  }
  ```
- Здесь рекурсия не хвостовая, т.к. еще в конце идет умножение на n
- Но этот код тоже легко заменяется на цикл
- Вообще, рекурсию обычно можно заменить на цикл, если рекурсивный вызов делается 1 раз за функцию
  ```

Вычисление факториала без рекурсии

- ```
  public static int getFactorial(int n) {
      int result = 1;  // переменная для накопления
                      // результата
      for (int i = 2; i <= n; ++i) {
          result *= i;
      }

      return result;
  }
  ```
- Работает быстрее и не падает при больших n
  ```

Замена любой рекурсии

- Во многих случаях нехвостовую рекурсию не удастся заменить обычным циклом
- Но любой код с рекурсией можно переписать без рекурсии, если использовать стек
- Т.е. мы имитируем стек вызовов обычным стеком с данными

Задача на дом «Возведение в степень»

- Написать рекурсивную функцию возведения целого числа в целую неотрицательную степень (упрощенный аналог Math.pow)
- Нельзя использовать Math.pow
- Когда закончите – напишите тут же нерекурсивную функцию

Задача на дом «Алгоритм Евклида»

- Задача про НОД
- Необходимо реализовать рекурсивную версию в виде функции

$$\text{НОД}(a, b) = \begin{cases} a, \text{если } b = 0 \\ \text{НОД}(b, a \,\%\, b) \text{ иначе,} \end{cases}$$

- где a % b – остаток от деления a на b
- Если оба числа равны 0, то НОД искать нельзя

Бинарный поиск – постановка задачи

- Есть массив, отсортированный по возрастанию (или убыванию)

0	1	2	3	4	5
1	2	5	7	10	12

- Хотим найти индекс, по которому лежит заданное число x. Если числа нет в массиве, то выдать -1

Бинарный поиск – пример решения

- Пусть есть отсортированный массив целых чисел int[] array

0	1	2	3	4	5
1	2	5	7	10	12

- Обозначим индексы границ массива:
  int left = 0;
  int right = array.length – 1;

- В ходе алгоритма эти границы left и right будут изменяться

Бинарный поиск – пример решения

0	1	2	3	4	5
1	2	5	7	10	12

- int left = 0;
  int right = array.length – 1; // 5
- Допустим, ищем x = 7
- Вычисляем средний индекс:
  int middle = (left + right) / 2; // (0 + 5) / 2 = 2
- Проверяем элемент по этому индексу.
  Получилось, что там 5 – оно меньше, чем x = 7
- Так как массив отсортирован, то правее среднего элемента лежат большие числа, а значит x надо искать там

Бинарный поиск – пример решения

0	1	2	3	4	5
1	2	5	7	10	12

- Значит, ищем правее, чем индекс middle = 2. Меняем left:
  left = middle + 1; // 3
- Опять вычисляем средний индекс:
  middle = (left + right) / 2; // (3 + 5) / 2 = 4
- Смотрим на элемент по этому индексу, там 10 – оно больше, чем x = 7
- Значит, надо искать левее, чем middle – меняем правую границу поиска:
  right = middle – 1; // 3

Бинарный поиск – пример решения

0	1	2	3	4	5
1	2	5	7	10	12

- Опять вычисляем средний индекс:
  middle = (left + right) / 2; // (3 + 3) / 2 = 3
- Смотрим на элемент по этому индексу, там 7
- Всё, нашли, результат алгоритма: 3

Бинарный поиск – идея алгоритма

- Сам алгоритм многошаговый, каждый шаг выглядит одинаково
- На каждом шаге у нас будут текущая левая и правая границы массива (left и right), внутри которых мы будем искать решение
- На каждом шаге вычисляем средний индекс:
```
int middle = (left + right) / 2;    // деление целочисленное
```
- Далее смотрим на элемент array[middle]:
    - Если он – искомый, то всё, нашли
    - Если средний элемент больше искомого, то на следующем шаге ищем слева от среднего элемента (меняем right), иначе – справа от среднего (меняем left)

Бинарный поиск – алгоритм

1. Вычисляем средний индекс middle через left и right
2. Смотрим элемент по индексу middle. Если он равен x, то всё, нашли
3. Если x > a[middle], то ищем в правой части, присваиваем left = middle + 1, и на шаг 1
4. Иначе (если x < a[middle]), то ищем в левой части, присваиваем right = middle – 1, и на шаг 1
5. Если left > right, то заканчиваем алгоритм, выдаем -1

Бинарный поиск

- Данный алгоритм очень эффективен – для массива из 1023 элементов достаточно 10 шагов, чтобы найти элемент
- Уже после первого шага остается 511 элементов, после второго – только 255

Рекурсивный вариант

```
public static int binarySearch(int[] a, int left, int right, int x) {
    if (left > right) {
        return -1;
    }

    int middle = (left + right) / 2;
    //… допишите, используя описание алгоритма
}
```

Задача на курс «Бинарный поиск»

1. Доделать бинарный поиск с рекурсией
2. Реализовать вариант бинарного поиска без рекурсии