Программирование на языке Java

Тема 23. Рекурсия

Рекурсия

Рекурсия – способ организации обработки данных, при котором программа вызывает сама себя непосредственно, либо с помощью других программ.

Примеры:

- **вычисление факториала числа,**
- **вычисление чисел Фибоначчи,**
- **геометрические фракталы**

Рекурсия

Зачем?

- для общего определения объекта с использованием ранее заданных частных определений
- мощный принцип программирования

Во многих вычислениях используется самоповторение

- Сортировка слиянием, быстрое преобразование Фурье, алгоритм Эвклида, поиск в глубину.
- Обработка папок, содержащих файлы и другие папки.

Рекурсия тесно связана с понятием математической индукции

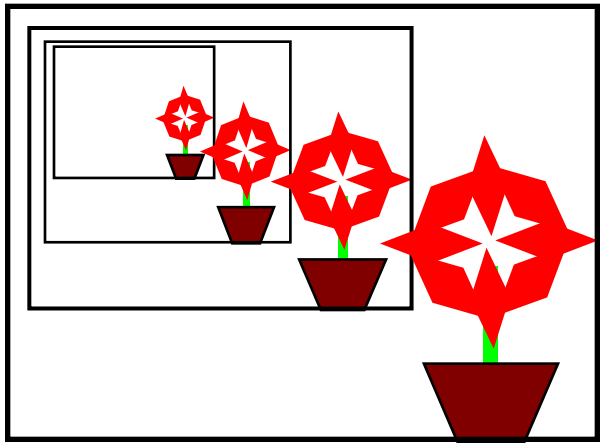
Рекурсивные объекты – 1

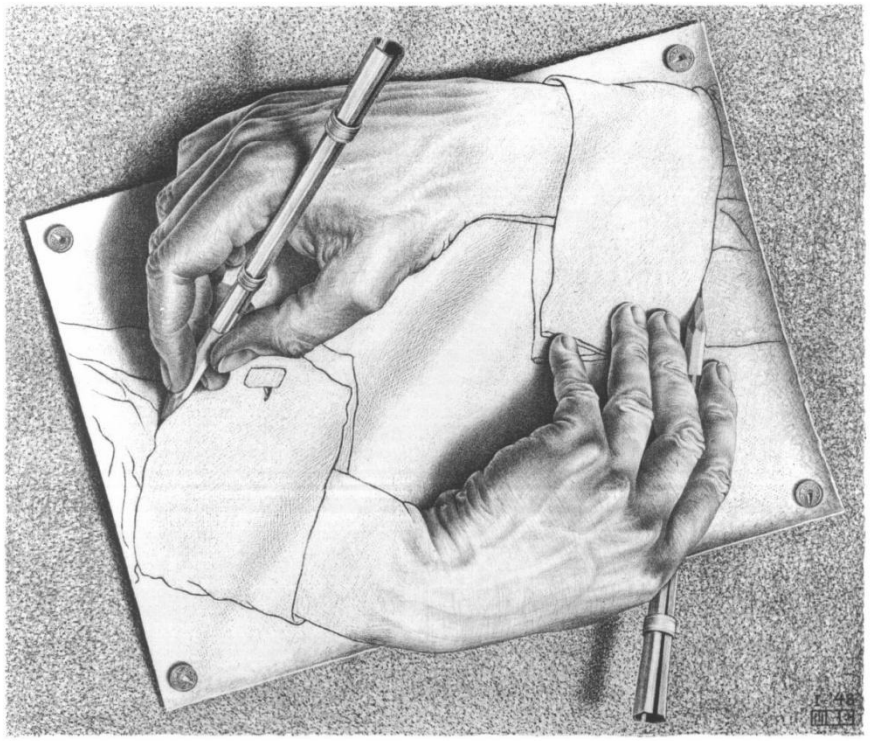
Сказка о попе и собаке:

У попа была собака, он ее любил.
Она съела кусок мяса, он ее убил.
В ямку закопал, надпись написал:

Сказка о попе и собаке

Рисунок с рекурсией:

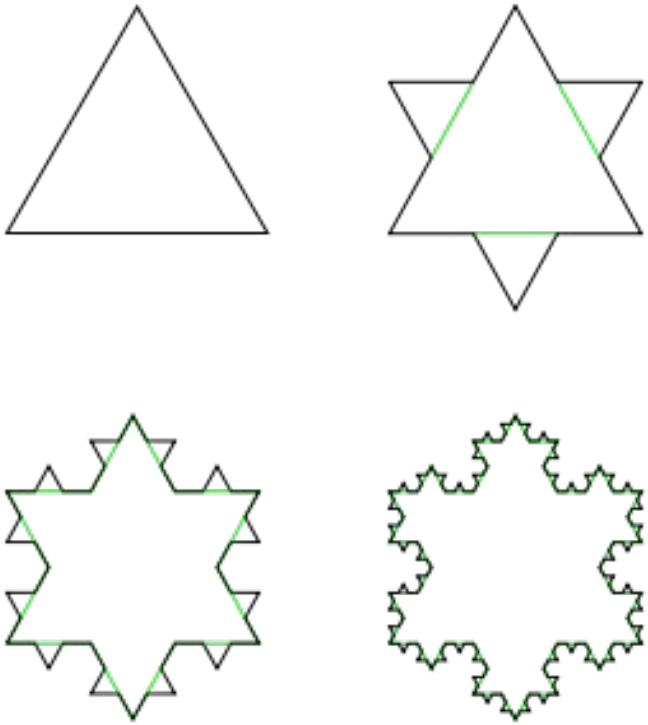




Рекурсивные объекты – 2

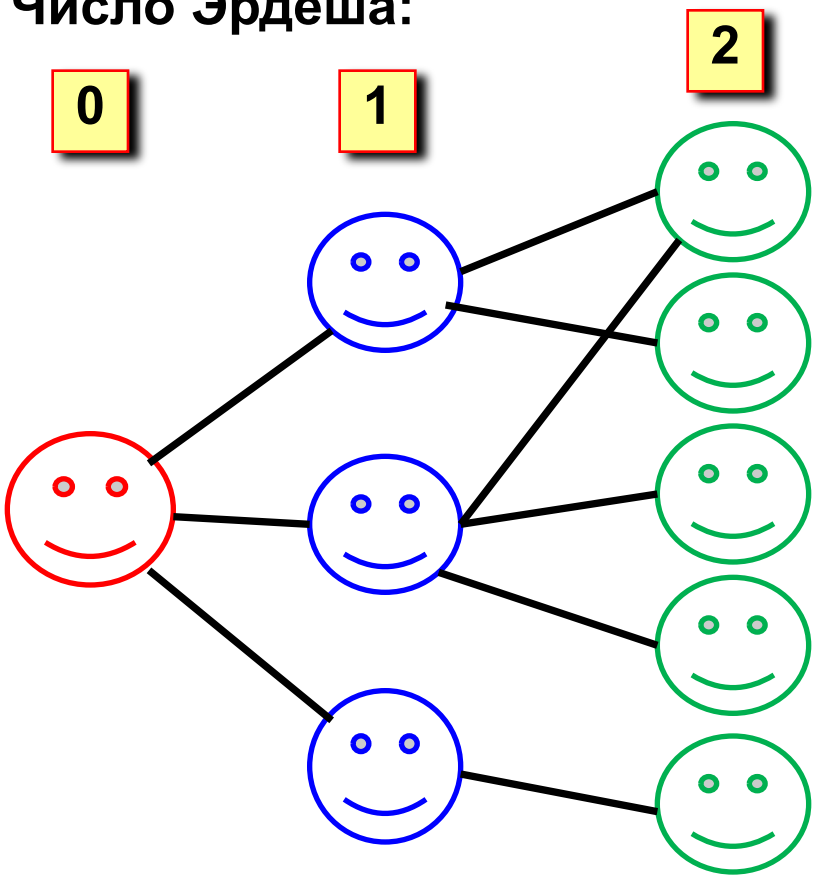
Треугольник Серпинского

Снежинка Коха



Рекурсивные объекты – 3

Число Эрдёша:



- у самого Эрдёша число равно 0;
- у соавторов Эрдёша число равно 1;
- соавторы людей с числом Эрдёша, равным n, имеют число Эрдёша n+1.

Рекурсивные объекты – 4

Факториал:

$$N! = \begin{cases} 1, & \text{если } N = 1, \\ N \cdot (N-1)!, & \text{если } N > 1. \end{cases}$$

$$1! = 1, \quad 2! = 2 \cdot 1! = 2 \cdot 1, \quad 3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1$$

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2 \cdot 1$$

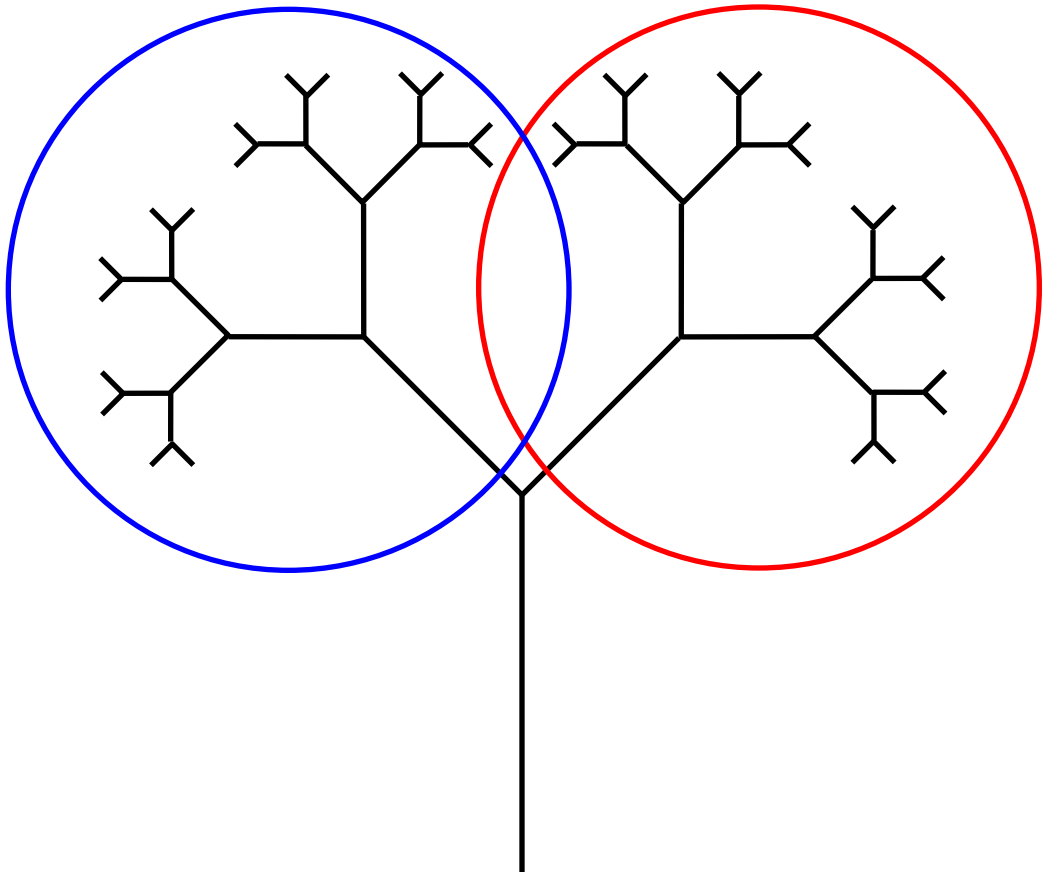
$$N! = N \cdot (N-1) \cdot \boxtimes \cdot 2 \cdot 1$$

Рекурсивный объект – это объект, определяемый через один или несколько таких же объектов.

Дерево Пифагора

Дерево Пифагора из N уровней – это ствол и отходящие от него симметрично **два дерева Пифагора из N-1 уровней**, такие что длина их стволов в 2 раза меньше и угол между ними равен 90º.

6 уровней:



Рекурсия и математическая индукция – 1

Рекурсия используется в методе мат. индукции.

Метод математической индукции:

1. Имеется бесконечный ряд утверждений P_i, где i – натуральное число.
2. Надо доказать верность утверждения P_N, где N – натуральное число.
3. Надо доказать следующее утверждение: если истинно утверждение P_i, то истинно и утверждение P_{i+1}.
4. Тогда истинны все утверждения P_i для i >= N.

Рекурсия и математическая индукция – 2

Пример:

1. Предположим P_1 верно, а нужно выяснить истинность утверждения P_3.
2. P_3 верно если верно P_2.
3. P_2 верно если верно P_1.
4. P_1 верно.
5. Отсюда, P_2 верно.
6. Отсюда, P_3 верно. Что и требовалось доказать.

Пример: Доказать, что $1 + 2 + \ldots + n = n(n+1)/2$

Рекурсия и математическая индукция – 3

Цель индукции: установить истинность выражения для больших задач, основываясь на истинности выражения на малых значениях.

Цель рекурсии: осуществить вычисление, сведя его к задаче меньшей размерности.

Основное правило рекурсии

В любой рекурсивной функции **обязательно должна быть нерекурсивная ветвь**, которая обеспечивает выход из рекурсии.

Рекурсия и итерация

Рекурсия – способ организации обработки данных, при котором программа вызывает сама себя непосредственно, либо с помощью других программ.

Итерация – способ организации обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ.

Рекурсия и итерация взаимосвязаны: любой алгоритм, реализованный в рекурсивной форме, может быть переписан в итеративном виде, и наоборот.

Внимание! Время выполнения и количество используемой памяти этими программами могут не совпадать.

Вычисление факториала числа – 1

Задача: составить рекурсивную функцию, которая вычисляет факториал числа n.

Рекурсивная функция:

выход из рекурсии

```
public static long f ( int n ) {
   if (n == 1) return 1;
   return n * f (n - 1);
}
public static void main(…) {
   ...
   int x = in.nextInt();
   System.out.print(f(x));
}
```

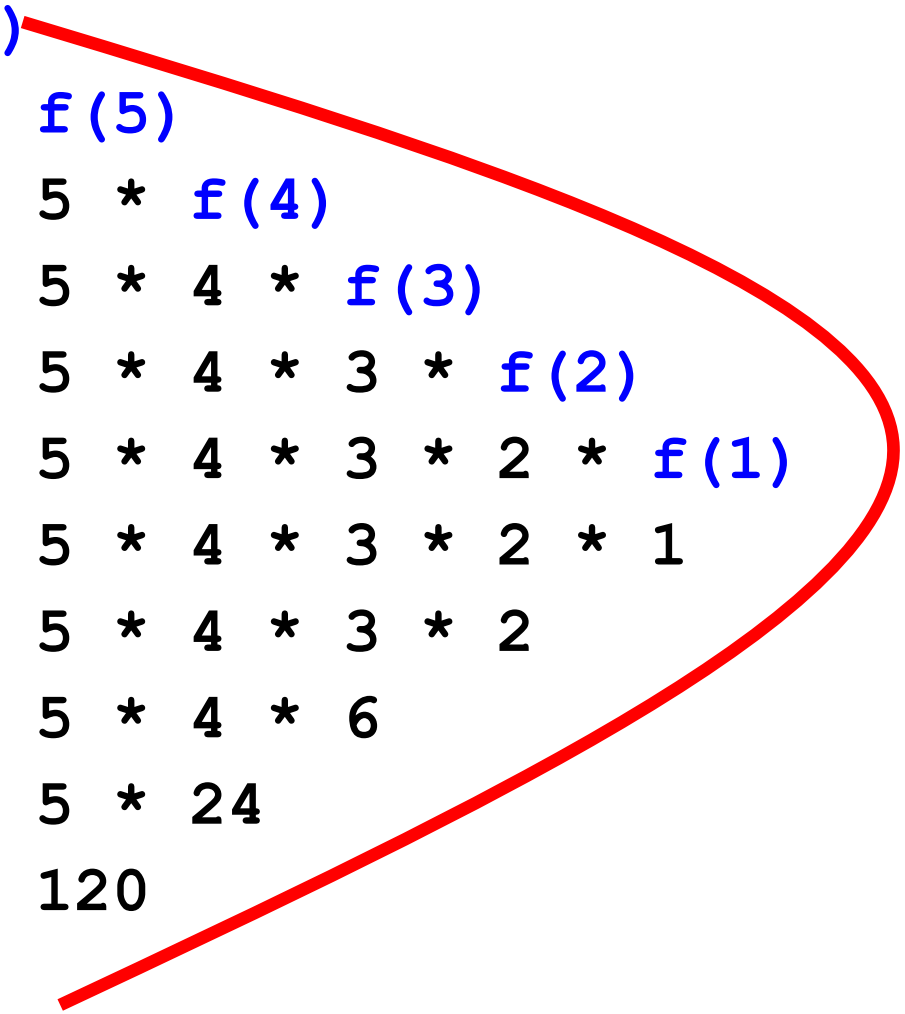
Вызов функции f с меньшим параметром

Вызов функции f с параметром x

Вычисление факториала числа – 2

Вызов рекурсивной функции:

```
f(6)
6 * f(5)
6 * 5 * f(4)
6 * 5 * 4 * f(3)
6 * 5 * 4 * 3 * f(2)
6 * 5 * 4 * 3 * 2 * f(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```



Вычисление факториала числа – 3

Итеративная функция:

```
public static long f_iter ( int n ) {
 int f = 1;
 for(int i = 1; i <= n; i++)
  f *= i;
 return f;
}
public static void main(…) {
...
int x = in.nextInt();
System.out.print(f_iter(x));
}
```

Вычисление факториала числа – 4

Вызов итеративной функции:

```
f_iter(6)
```

i	f(i)
1	1
2	2
3	6
4	24
5	120
6	720

Задание

Что будет выведено на экран?

```
public static void main(String[] args) {
    xMethod(1234567);
}

public static void xMethod(int n) {
    if (n > 0) {
        System.out.print(n % 10);
        xMethod(n / 10);
    }
}
```

7654321

Задание

Что будет выведено на экран?

```
public static void main(String[] args) {
    xMethod(5);
}

public static void xMethod(int n) {
    if (n > 0) {
        System.out.printf("%d ", n);
        xMethod(n - 1);
    }
}
```

5 4 3 2 1

А если поменять эти две строки местами?

1 2 3 4 5

Задание

Что не так?

```
public static void main(String[] args) {
   xMethod(1234567);
}

public static void xMethod(double n) {
   if (n !=0) {
      System.out.println(n);
      xMethod(n / 10);
   }
}
```

n – вещественная переменная, поэтому деление будет продолжаться до 1.0E-323

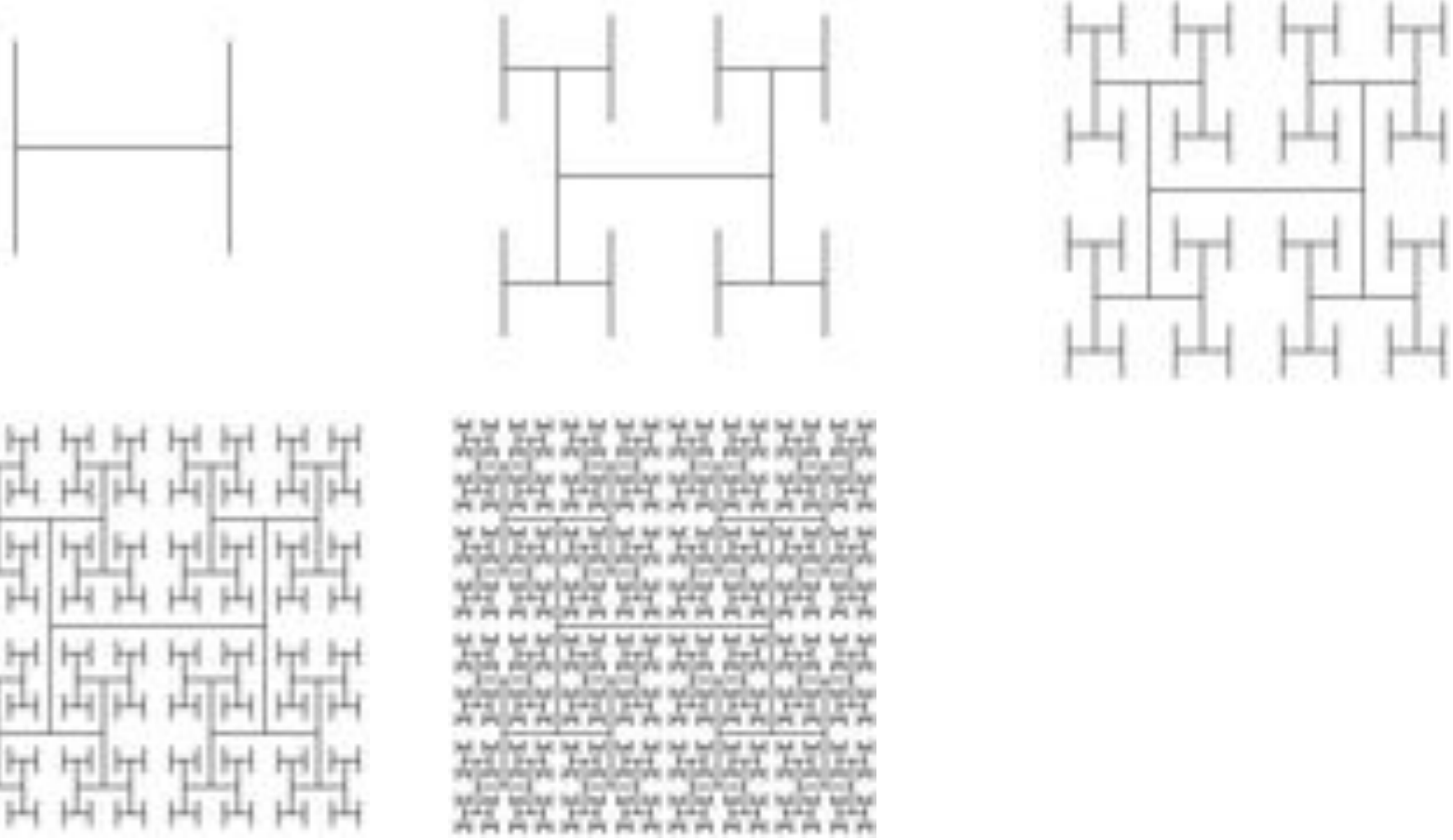
Особенности рекурсивных методов

- В теле метода обязательно есть блок **`if-else`** или **`switch`** с несколькими case
- Обязательно должна быть хотя бы одна ветка, которая останавливает рекурсию.
- Каждый рекурсивный вызов «уменьшает» исходную задачу, делая ее все ближе и ближе к нерекурсивному случаю до тех пор пока не станет равен ему.

```
public static void drinkCoffee(Cup cup) {
  if (!cup.isEmpty()) { // чашка пуста?
    cup.takeOneSip(); // выпить глоток
    drinkCoffee(cup);
  }
}
```

Рекурсивная графика

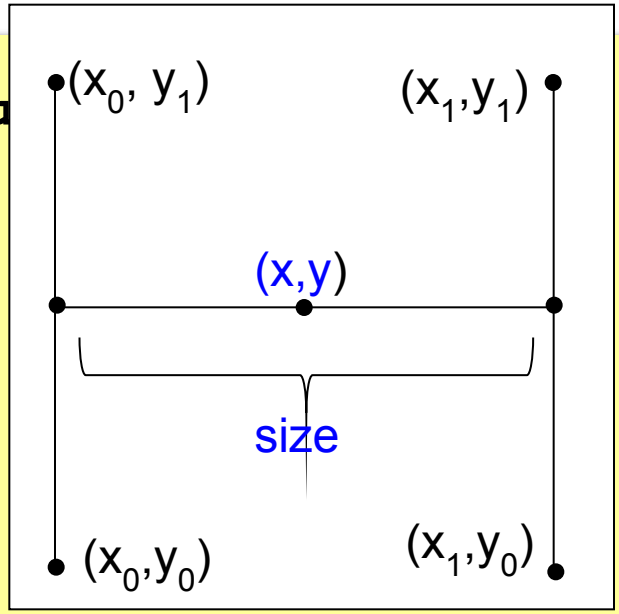
H-дерево порядка n в единичном квадрате



Рекурсивная графика. H-дерево

```
draw(int n, double x, double y, dou
 if (n == 0) return;
 double x0 = x - size/2;
 double x1 = x + size/2;
 double y0 = y - size/2;
 double y1 = y + size/2;
 StdDraw.line(x0, y0, x0, y1);
 StdDraw.line(x1, y0, x1, y1);
 StdDraw.line(x0, y,  x1, y);

 draw(n-1, x0, y0, size/2); // нижнее левое дерево
 draw(n-1, x0, y1, size/2); // верхнее левое дерево
 draw(n-1, x1, y0, size/2); // нижнее правое дерево
 draw(n-1, x1, y1, size/2); // верхнее правое дерево
}
```



Стек вызовов

В один момент времени может исполняться один метод из всей программы.

```
public static void a() {
   print("вызов метода b()");
   b();
}
public static void b() {
   print("выполнение метода b()");
}
public static void main(String[] args) {
   a();
}
```

Стек вызовов

Стек вызовов (call stack) – стек, хранящий информацию для возврата управления из подпрограммы в программу (или подпрограмму при вложенных или рекурсивных вызовах).

Элементы добавляются в стек вызовов по следующему принципу: последний добавленный элемент должен быть извлечён первым.

Переполнение стека

В процессе рекурсии существует опасность переполнения стека вызовов, ошибка **Stack overflow**

```
public static void a() {
    a();
}
public static void main(String[] args) {
    a();
}
```

Результат выполнения

```
Exception in thread "main"
        java.lang.StackOverflowError
```

«Ловушки» рекурсии – 1

Отсутствие конечного случая

Если написать рекурсивный метод без нерекурсивной ветви, то программа выдаст ошибку StackOverflow.

```
public static void a() {
    a();
}
public static void main(String[] args) {
    a();
}
```

«Ловушки» рекурсии – 2

Нет гарантии достижения нерекурсивного случая

```
public static int fact ( int n ) {
 if (n == 1)
  return 1;
 return n * fact(n);
}
```

«Ловушки» рекурсии – 3

Перерасход памяти

Если рекурсивные вызовы функцией самой себя занимают большие промежутки времени до завершения работы, Java может исчерпать доступный объем памяти для хранения промежуточных результатов.

```
public static double H(int n) {
  if (n == 1) return 1.0;
  return H(n - 1) + 1.0 / n;
}
main() {
  H(1000);   // 7.485470860550343
  H(10000);  // 9.787606036044348
  H(100000); // Exception in thread "main"
                java.lang.StackOverflowError
```

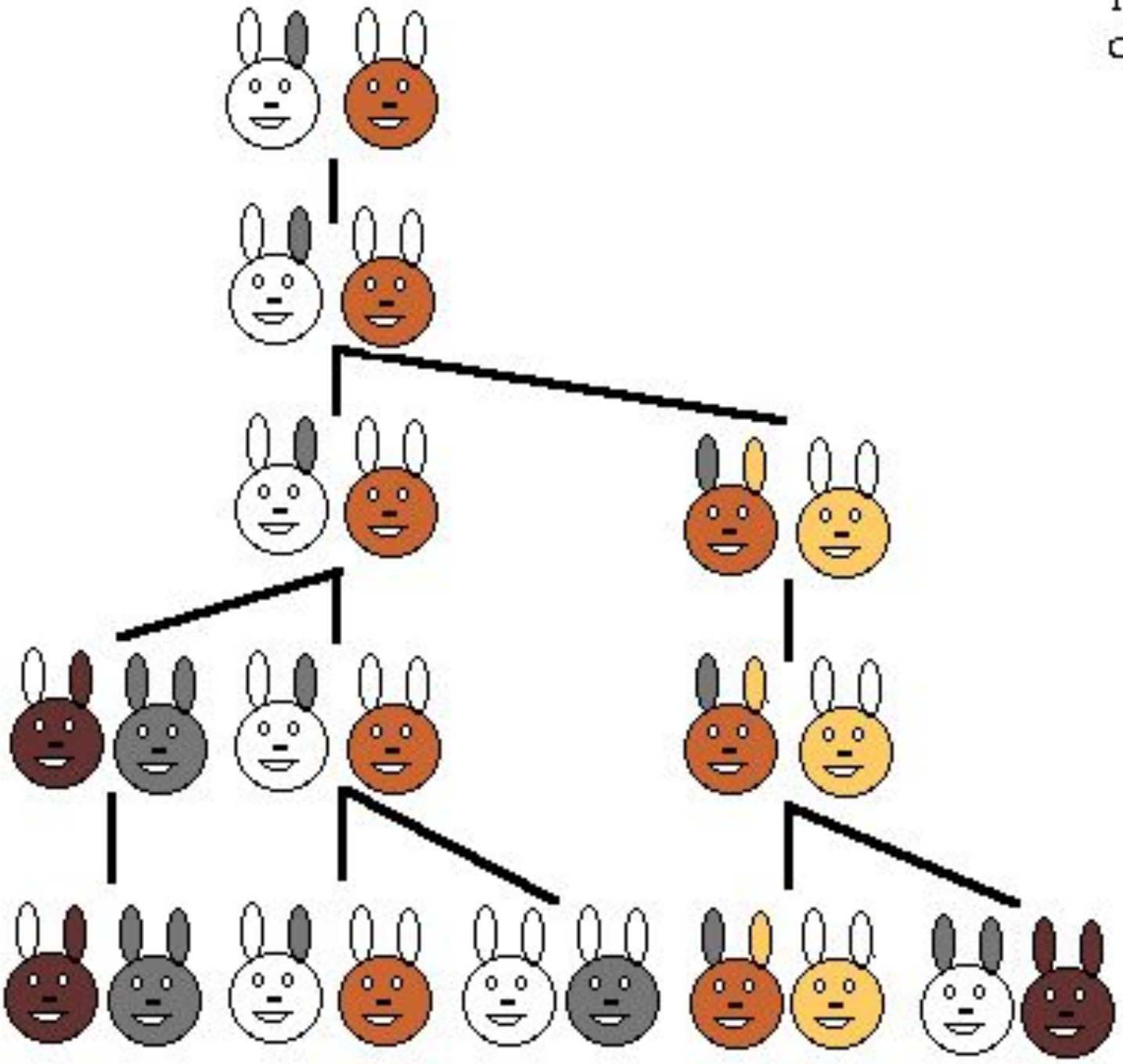
«Ловушки» рекурсии – 4

Чрезмерный объем повторных вычислений

Даже довольно простая функция способна создать очень большой объем повторных вычислений.

Таких ситуаций нужно избегать.

Рассмотрим на примере чисел Фибоначчи.



М
О
,
ую
у

Числа Фибоначчи

$$F(0) = 0,$$
$$F(1) = 1,$$
$$F(2) = F(0) + F(2) = 0 + 1 = 1,$$
$$F(3) = F(1) + F(2) = 1 + 1 = 2,$$
$$F(4) = F(2) + F(3) = 2 + 1 = 3,$$
$$F(5) = F(3) + F(4) = 3 + 2 = 5,$$
$$F(6) = F(4) + F(5) = 5 + 3 = 8,$$
$$F(7) = F(5) + F(6) = 8 + 5 = 13,$$
$$F(8) = F(6) + F(7) = 13 + 8 = 21,$$
$$F(9) = F(7) + F(8) = 21 + 13 = 34, \ldots$$

Числа Фибоначчи

Числа Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$F(n)=\begin{cases} 0, \text{если } n=0 \\ 1, \text{если } n=1 \\ F(n-1)+F(n-2), \text{иначе} \end{cases}$$

Рекурсивное вычисление числа Фибоначчи

Задача: составить рекурсивную функцию, которая вычисляет n-ое число Фибоначчи.

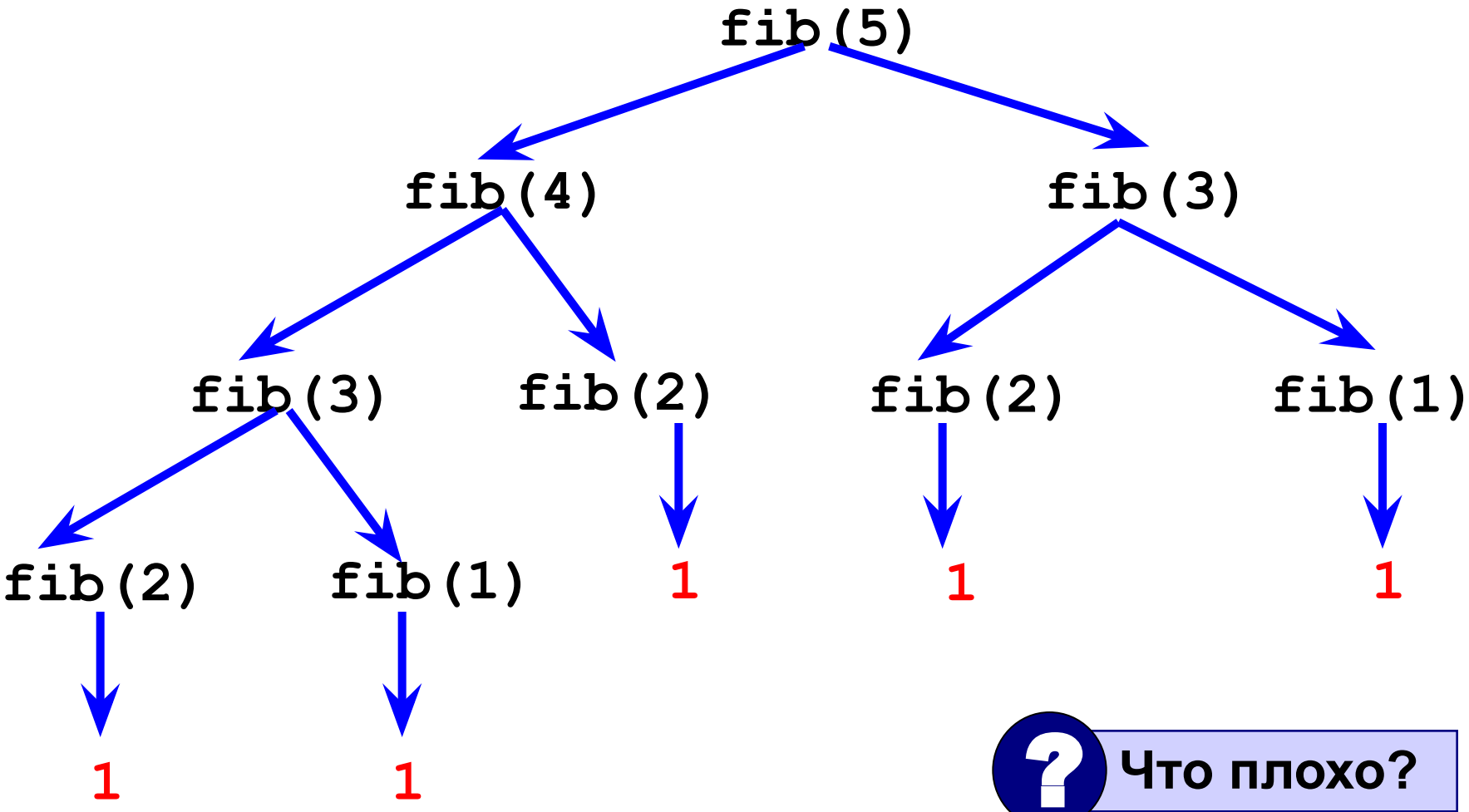
Рекурсивная функция:

```
public static long fib ( int n ) {
  if (n < 2)
    return n;
  return fib (n - 1) + fib (n - 2);
}
```

выход из рекурсии

вызов функции fib с меньшими параметрами

Рекурсивное вычисление числа Фибоначчи



Что плохо?

Древовидная рекурсия: на каждом уровне (кроме дна) ветви разделяются надвое

Рекурсивное вычисление числа Фибоначчи

А если вычислять F(50)?

F(50) вызовется 1 раз (1 минута 16 секунд)

F(49) вызовется 1 раз (47 секунд)

F(48) вызовется 2 раз (29 секунд)

F(47) вызовется 3 раз (18 секунд)

F(46) вызовется 5 раз (11 секунд)

F(45) вызовется 8 раз (7 секунд)

…

F(1) вызовется 12 586 269 025 раз

Вывод: в данном случае рекурсия является неэффективным решением.

Как решать данную задачу? Можно воспользоваться итерацией.

Итеративное вычисление числа Фибоначчи

```
public static long fib_iter ( int n ) {
  if (n == 0) return 0;
  long[] F = new long[n+1];
  F[0] = 0;
  F[1] = 1;
  for (int i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
  return F[n];
}
```

Такой подход называется **динамическое программирование**, когда каждая вспомогательная задача решается только один раз, после чего ответ сохраняется в таблице, что приводит к сокращению количества вычислений.

Задача. Палиндром

Задача. Написать логический рекурсивный метод, который проверяет является ли строка из параметра палиндромом.

```
public static boolean isPalindrome(String s) {
  if (s.length() <= 1)
    return true;
  else if (s.charAt(0) != s.charAt(s.length() - 1))
    return false;
  else
  return isPalindrome(s.substring(1, s.length() - 1));
}
```

Создание новой строки

Сколько нерекурсивных веток?

Что плохо? Как улучшить?

Вспомогательные методы

Воспользуемся вспомогательным методом, в который кроме строки будем передавать индексы начала и окончания подстроки.

```
boolean isPalindrome(String s) {
  return isPalindrome(s, 0, s.length() - 1);
}

boolean isPalindrome(String s, int low, int high) {
  if (high <= low)
    return true;
  else if (s.charAt(low) != s.charAt(high))
    return false;
  else
   return isPalindrome(s, low + 1, high - 1);
}
```

Что выведет программа?

```
public static void main(String[] args) {
  ex(6);
}

public static void ex(int n) {
  if (n<=0)
    return;
  System.out.println(n);
  ex(n-2);
  ex(n-3);
  System.out.println(n);
}
```

Что не так?

```
public static void main(String[] args) {
  ex(6);
}

public static void ex(int n) {
  System.out.println(n);
  ex(n-2);
  ex(n-3);
  System.out.println(n);
   if (n<=0)
    return;
}
```

Итоги

1. Что такое рекурсивный метод?
2. Что такое бесконечная рекурсия?
3. Что означает ошибка Stack Overflow и при каких обстоятельствах она может возникнуть?
4. Бывают ли ситуации, когда итерация является единственно возможным решением задачи?
5. Бывают ли ситуации, когда рекурсия является единственно возможным решением задачи?
6. Что выбрать: рекурсию или итерацию?

Итоги

Я предупрежден о чрезмерном расходе памяти и чрезмерном объеме вычислений, которые могут возникнуть в результате выполнения рекурсивного кода.

Дата _____ Подпись _____

Задания

1. Составить рекурсивную функцию, которая возводит число a в n-ую степень. Воспользуйтесь тем, что

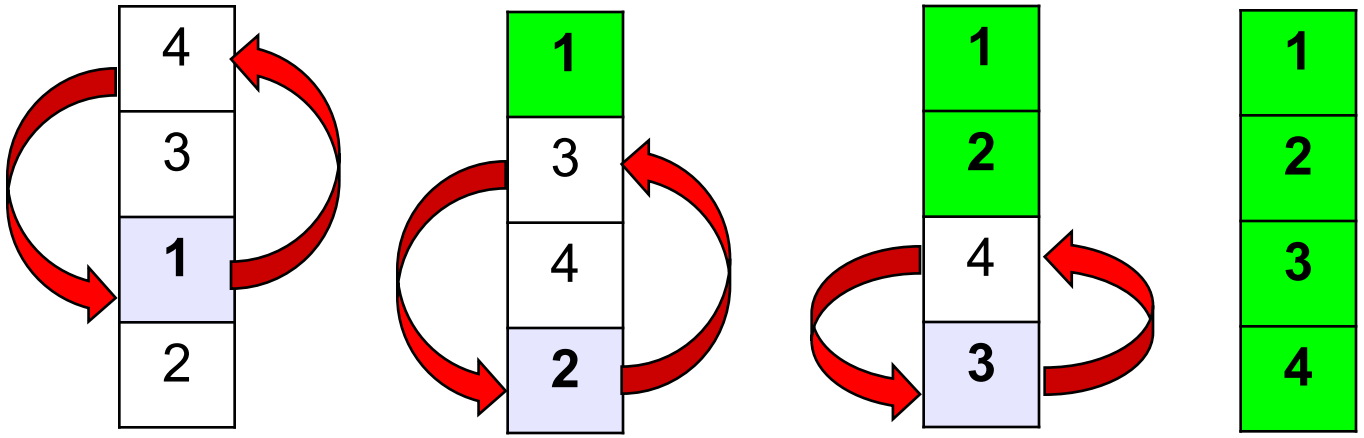
$$a^n = a \cdot a^{n-1},$$

$$a^0 = 1.$$

Сортировка методом выбора

Идея:

- найти минимальный элемент и поставить на первое место (поменять местами с **`A[0]`**)
- **из оставшихся** найти минимальный элемент и поставить на второе место (поменять местами с **`A[1]`**), и т.д.



Рекурсивный подход. Сортировка

2 подзадачи

- Найти минимальный элемент в массиве и поменять его местами с первым;
- Выполнить дальнейшую сортировку рекурсивно, игнорируя первый элемент.

Рекурсивный подход. Сортировка

```
void sort(int[] list, int low, int high) {
  if (low < high) {
    int indexOfMin = low;
    int min = list[low];
    for (int i = low + 1; i <= high; i++)
      if (list[i] < min) {
        min = list[i];
        indexOfMin = i;
      }

    list[indexOfMin] = list[low];
    list[low] = min;
    sort(list, low + 1, high);
  }
}
```

Рекурсивный подход. Сортировка

```
void sort(int[] list) {
  sort(list, 0, list.length - 1);
}
```

Рекурсивный подход. Бинарный поиск

NB! Для того, чтобы осуществить бинарный поиск, элементы массива должны быть упорядочены.

Что нужно учесть?

- Если x меньше элемента в середине массива, рекурсивно ищем в первой половине массива.
- Если x равен элементу в середине массива, то можно завершить поиск.
- Если x больше элемента в середине массива, рекурсивно ищем во второй половине массива.

Рекурсивный подход. Бинарный поиск

```
int binarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return binarySearch(list, key, low, high);
}

int binarySearch(int[] list, int key,
                 int low, int high) {
  // напишите рекурсивный метод поиска
}
```