

02.5. УПРАВЛЯЮЩИЕ

ЛЕКЦИИ ПО ИНФОРМАТИКЕ ДЛЯ СТУДЕНТОВ ВТОРОГО КУРСА
ВЫСШЕЙ ШКОЛЫ ИТИС КФУ
2019

М.М. АБРАМСКИЙ
СТАРШИЙ ПРЕПОДАВАТЕЛЬ
КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ

Управление (кибернетика)

- Принцип обратной связи – управление обработкой информационным процессом должно изменяться под влиянием получаемой информации
 - Термин «обратная связь» – из кибернетики
- Необходимость особых конструкций – *структур управления*

BLOOPER

Целочисленное деление

```
int k = 1;  
int m = 2;  
double x = k / m;
```

Ожидание: 0.5

Реальность: 0.0

Порядок действий:

- Сначала выполняется деление k на m (целого числа на целое число).
Получается тоже целое число 0.
- Целое число 0 присваивается *double* переменной x .
- **В итоге в x лежит 0.0.**

boolean

true, false

Сравнение переменных: \leq , \geq , $==$, \neq (не равно)

! – «НЕ», $\&\&$, $\&$ – «И», $||$, $|$ – «ИЛИ»

Разница:

$\&$, $|$ - считают всё всегда.

$\&\&$, $||$ - вычисление останавливается, если результат уже понятен.

– Ленивые (lazy) выражения.

Оператор вывода (Набросок)

`System.out.print()` – вывел и всё.

`System.out.println()` – вывел и перенес курсор на новую строку (будто enter нажал).

`System.out.println(x, y)`

НЕЛЬЗЯ

`System.out.println(x + " " + y)`

МОЖНО

Образец

Должно быть в файле **Task.java**

```
public class Task {  
  
    public static void main(String[] args) {  
        int x = 100;  
        int y = 200;  
        int z = x + y;  
        System.out.println(z);  
    }  
}
```

Последовательное соединение операторов

- *Далее: команда, инструкция, оператор (не символ, а конструкция) – синонимы. Англ. – statement.*

Команда1; Команда2; ... КомандаN;

– **последовательность команд**

{ Команда1; Команда2; ... КомандаN; }

– **блок**

Условный оператор

С точки зрения теории программирования есть только **if-else**

- все остальное придумано для удобства (*синтаксический сахар*)

```
if (условие) {
```

Последовательность команд

 — В т.ч. может быть и другой условный оператор

```
}
```

```
else {
```

Последовательность команд

```
}
```

Условие

- Выражение типа `boolean`, но принимает, вообще говоря, аргументы всех разных типов

«`if (x > 0)`»

`x` – аргумент, `(x > 0)` – значение

- В математической логике это называется **предикатом**
 - Функция, принимающая аргументы любого типа, но возвращающая `true` или `false`
 - В чем разница с булевой функцией?
 - Приведите пример предикатов!

if без else

Например, нужно выполнить $x \leftarrow |x|$

```
x = ... ;
```

```
if (x < 0)
```

```
    x = -x;
```

; В заголовке нет!

Иначе получается очень интересный оператор.

```
if (x < 0);  
    x = -x;
```

Это работающий код.
Который умножает x на -1 . Всегда.

Dangling else

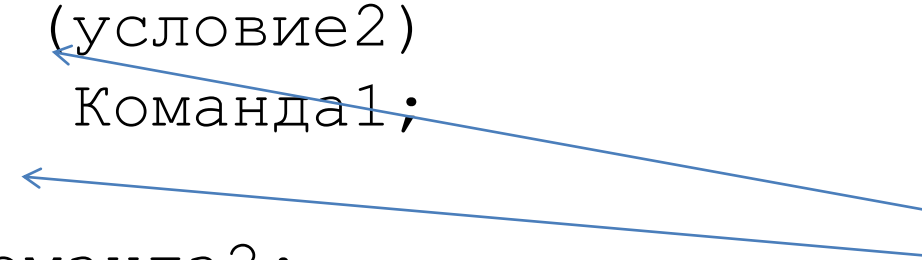
- Если нет операторных скобок, то else приклеивается к ближайшему if
- Может возникнуть вот такая ситуация:

```
if (условие1)
    if (условие2)
        Команда1;
else
    Команда2;
```

Dangling else

- Если нет операторных скобок, то else приклеивается к ближайшему if
- Может возникнуть вот такая ситуация:

```
if (условие1)
    if (условие2)
        Команда1;
else
    Команда2;
```



Java – не Python,
на отступы не
смотрит,
else приклеит ко
второму if

Решение?

Решение Dangling else – операторные скобки

```
if (условие1) {  
    if (условие2)  
        Команда1;  
}  
else  
    Команда2;
```

Рекомендуется вообще всегда их ставить!

?

```
if (x == 0) {  
    P1;  
else if (x == 1) {  
    P2;  
}  
else if (x == 2) {  
    P3;  
}  
...  
else {  
    Q;  
}
```


Switch case

```
switch (x) {  
  case 0:  
    P1; break;  
  case 1:  
    P2; break;  
  case 2:  
    P3; break;  
  default:  
    Q;  
}
```

Switch case

- Можно написать

```
switch (x) {  
  case 0:  
    P1; break;  
  case 1:  
    P2; break;  
  case 2:  
    P3; break;  
  default:  
    Q;  
}
```

break обязателен, если хотите
соответствие ***if***-у!

Без break это будет работать так

- Если 0, то выполнится P1, P2, P3, Q
- Если 1, то выполнится P2, P3, Q
- и т.д.

Это тоже может быть полезно.



Кусочно-заданные функции

Нам часто нужно считать выражения вида

$$y = \begin{cases} A, & \text{если верно } P \\ B, & \text{иначе} \end{cases}$$

Например:

$$\text{abs}(x) = \begin{cases} x, & \text{если } x > 0 \\ -x, & \text{иначе} \end{cases}$$

Как привыкли считать:

```
double y;  
if (P) {  
    y = A;  
}  
else {  
    y = B;  
}
```

Как еще можно считать (*тернарный оператор*)

```
double y;  
if (P) {  
    y = A;  
}  
else {  
    y = B;  
}
```

double $y = P ? A : B;$

присвоить A, если верно P, иначе присвоить B

Примеры

Модуль:

```
double y = x >= 0 ? x : -x;
```

Можно вкладывать один в другой (и даже скобки не нужны)

```
int sgn = x > 0 ? 1 : x < 0 ? -1 : 0;
```

«если x положительный, то 1, а иначе если x отрицательный, то -1, а иначе 0»

Повтор действий

- А что, если не знаем, сколько раз придется повторять то или иное действия?
- В жизни — никаких проблем
 - Просим кого-то «стучи, пока не откроют — не говорим ему «стучи 10 раз, стучи 20 раз»

Цикл

С точки зрения теории языков программирования есть только цикл **while**

- все остальное придумано для удобства (*синтаксический сахар*)

```
while (условие) {  
    Последовательность команд  
}
```

Один проход цикла – *итерация*

do while

```
do {  
    Последовательность команд  
}  
while (условие)
```

Выполнится как минимум 1 раз

Для знающих Pascal – аналог repeat until (но не идентичный):

- Выходим из **repeat until**, если условие – **true**
- В **do while** все как в **while** – выходим, если **false**

Перепишите **do while** в **while**

```
do {  
    p1; p2; p3;  
} while (b);
```

Частый вид `while`

инициализация (чтобы работала проверка условия)

`while` (условие) {

 Последовательность команд;

 Команда, обеспечивающая переход на следующую итерацию (**переход**)

}

Пример:

```
int i = 0;
while (i < 5) {
    s.o.p.(i);
    i++;
}
```

for

В C, C++, Java, C#, JavaScript – сокращение записи while

```
for (инициализация; условие; переход) {  
    Последовательность команд  
}
```

Пример:

```
for (int i = 0; i < 5; i++) {  
    System.out.println(i);  
}
```

*Какой угодно переход
а не как в Pascal*



Объявление переменной в цикле

```
// ...  
for (int i = 0; i < n; i++) {  
    int x = sc.nextInt();  
    double s = (x + 1) / (x - 100500);  
    System.out.println(s);  
}  
// ...
```

- С – ругается.
 - «Как можно несколько раз объявлять одну и ту же переменную?»
- Java – оптимизирует.
 - «А ты в курсе, что создание переменной x не зависит от цикла? Я поэтому это создание переменной вынесу перед циклом.»

for each

```
for (int i = 0; i < n; i++) {  
    s = s + a[i];  
}
```

А можно и так:

```
for (int x : a) {  
    s += x;  
}
```

Подробнее посмотрим, когда будут массивы.

break и continue

- **break** – обрывает выполнение текущей конструкции
 - чаще всего цикл
- **continue** – для цикла мгновенно начинает следующую итерацию.

Реальная причина `continue`

Подсчитать сумму положительных

Classical way

```
int s = 0;
for (int i = 0; i < n; i++) {

    if (a[i] >= 0)
        s += a[i];

}
```

И что?

Continue

```
int s = 0;
for (int i = 0; i < n; i++) {

    if (a[i] < 0)
        continue;

    s += a[i];

}
```


Трасса (trace)

- Последовательность команд, которые были вызваны при выполнении.
 - Всегда линейна (последовательна) – выписываются все вызванные команды подряд

Программа	Трасса
<pre>int i = 0; while (i < 5) { System.out.println(i); i++; }</pre>	<ul style="list-style-type: none">• $i = 0$• Проверка $i < 5$ – верно• Вывод 0• i становится равным 1• Проверка $i < 5$ – верно•

- Анализ трассы дает возможность проверять правильность программы
 - **Трассировка** (явный вывод результатов выполнения команд)
 - **Debugging** (отладка специальными средствами)

Как работали циклы и условия в **Assembler**

- Проверяли условие и перепрыгивали в нужное место в коде.
- Этот подход перекочевал в языки как оператор под названием ...

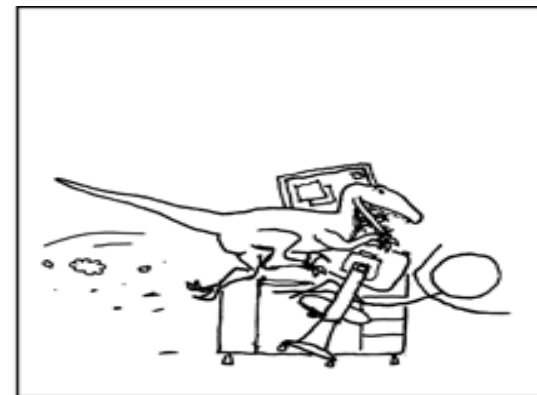
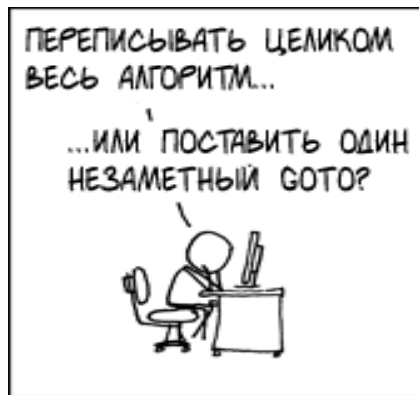
Go To (goto)

- Оператор **безусловного** перехода
- Каждая строка кода помечена меткой (label)
- Go To **перемещает управление** на нужную метку.
 - Начинает выполняться оператор, написанный в другом месте.

Пример: вывести числа от 1 до 4.

```
200: i := 1;  
300: write(i);  
400: i := i + 1;  
500: if (i < 5) then goto 300;
```

Критика



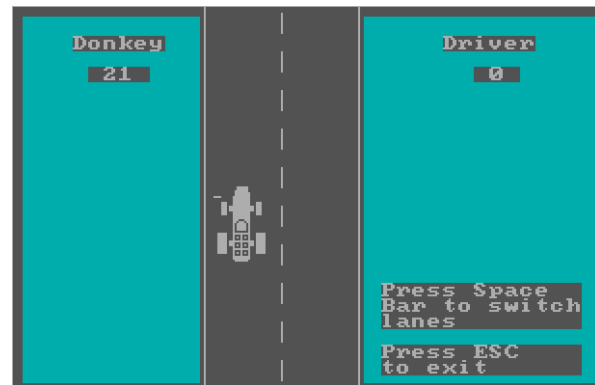
Критика

- 1968 Дейкстра
 - «Go To Statement Considered Harmful»
 - «Оператор Go To считается вредным»
- «Спагетти-код», нечитаемость, наследие низкоуровневых языков.

Donkey.bas

Bill Gates, Neil Konzen, 1981

```
1080 COLOR 15,0:LOCATE 17,4,0:PRINT "(C) Copyright IBM Corp 1981, 1982"
1090 COLOR 14,0:LOCATE 23,7,0:PRINT "Press space bar to continue"
1100 IF INKEY$<>" " THEN GOTO 1100
1110 CMD$ = INKEY$
1120 IF CMD$ = " " THEN GOTO 1110
1130 IF CMD$ = CHR$(27) THEN GOTO 1298
1140 IF CMD$ = " " THEN GOTO 1160
1150 GOTO 1110
1160 DEF SEG=0
1170 IF (PEEK(&H410) AND &H30)<>&H30 THEN DEF SEG:GOTO 1291
1180 WIDTH 80:CLS:LOCATE 3,1
```



Структурный подход

Теорема Бёма-Якопини (1965-1966 гг.) Любая программа, заданная в виде блок-схемы, может быть представлена с помощью последовательности, ветвления, цикла (; if-else while).

Т.е. любой goto можно выразить через циклический и условный оператор.

Область видимости (scope) переменных

Возникла из-за иерархических блоков и возможности объявлять переменную в любом месте программы.

Переменная существует только в блоке, в котором она объявлена, а также в блоках, содержащихся в нем. После окончания работы блока переменная освобождает имя и память

Пример:

Вложенный блок

```
{  
    int x = 0;  
    if (x > 0) {  
        int y = -x;  
    }  
    System.out.println(y);  
}
```

Здесь **x** виден.

Объявленное во внешнем блоке
доступно внутри вложенных блоков

Ошибка компиляции.

Здесь **y** не существует.


Объявленное во вложенных
блоках недоступно во внешних

Внешний блок

Область видимости (scope) переменных

Цикл **for**:

```
for (int i = 0; i < n; i++) {  
    System.out.println(i);  
}
```



i существует только в цикле.

Ввод - Scanner

Самой первой строчкой .java-файла:

```
import java.util.*;
```

Затем в коде:

```
Scanner sc = new Scanner(System.in);
```

Затем вызываем нужный метод

```
int x = sc.nextInt(); целое число
```

```
double y = sc.nextDouble(); вещественное число
```

```
String s = sc.nextLine(); строка (до переноса)
```

```
String s2 = sc.next(); строка (слово, до пробела)
```

Ввод – args

Аргументы запуска

- Помните `hl.exe -console`?
- Если после имени программы пишутся еще другие данные, то они приходят в нее как аргументы запуска
- Java программа управляет этим через массив args
– `String [] args` у `main`
- Можно передавать данные через этот массив, не забывая их конвертировать в нужный тип

args

```
public static void main(String[] args) {  
  
    int x = Integer.parseInt(args[0]);  
    int y = Integer.parseInt(args[1]);  
  
    int z = x + y;  
  
    System.out.println(z);  
  
}
```