

03. ЦИКЛИЧЕСКАЯ КОНСТРУКЦИЯ. Ю. АНТИПАТТЕРНЫ НАПИСАНИЯ КОДА

*курс лекций по информатике и программированию
для студентов первого курса ИТИС КФУ (java-поток)
2023/2024*

М.М. Абрамский

кандидат технических наук, доцент кафедры программной инженерии

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 (да, синтаксический сахар).

```
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.

! Подумайте о причинах появления этих операторов.

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

Здесь x виден.

Объявленное во внешнем блоке
доступно внутри вложенных блоков

Ошибка компиляции.

Здесь y не существует.

Объявленное во вложенных
блоках недоступно во внешних


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

Внешний блок

Область видимости (scope) переменных

Цикл **for**:

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



i существует только в цикле.

Значит, **for** и **while** – не совсем ЭКВИВАЛЕНТНЫ

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

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

в чем небольшая разница?

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 переключался 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



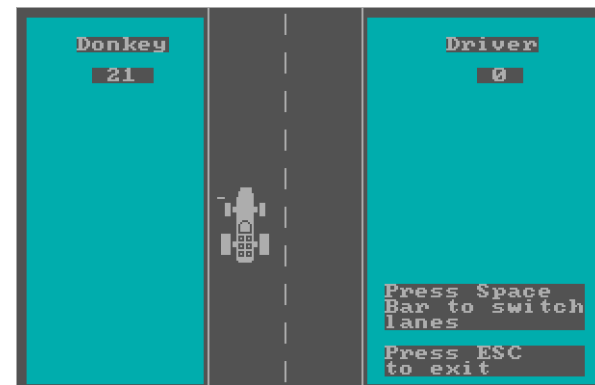
c xkcd.ru

- 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.

НЕМНОГО О ВВОДЕ И ВЫВОДЕ

Концепция черного ящика



Получить входные данные

Профессионально:

- От пользователя (человека, запустившего программу)
- От другой программы
- Из какого-то источника данных (файл, сеть, устройство)

Для удобства:

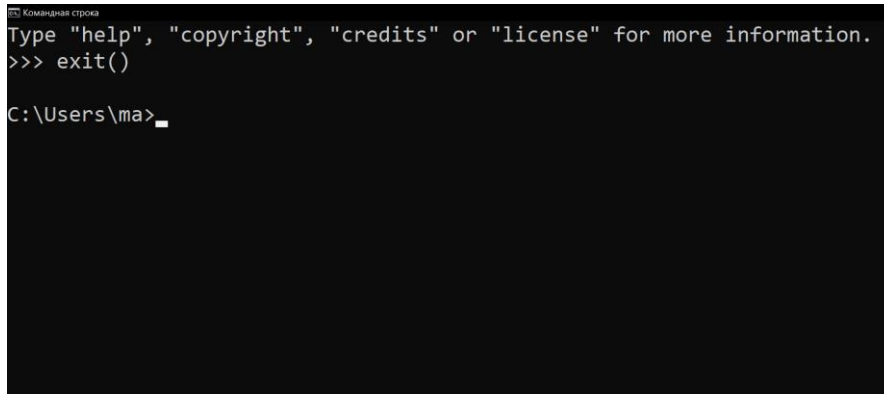
- Задать их в явном виде в самом начале программы
 - Т.е. выполнить что-то вроде “ $a = 2$ ”

А что происходит, когда программа запускается?

Где-то мы должны ввести данные, а затем посмотреть, что
выводится.

Консоль

- **Текстовый интерфейс** взаимодействия пользователя с программой.
- **Диалоговый** (ждет, что мы что-то введем, отвечает нам на наш ввод)
- Связан с **входными** данными (мы вводим данные в программу через консоль) и **выходными** данными (программа выводит в консоль данные для нас, «на экран»)



```
Командная строка
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

C:\Users\ma>
```

Объяснение термина «консоль»

IBM 701 Electronic Data Processing Machine

«первая коммерческая массовая большая компьютерная система для научных вычислений»

ЭВМ (железо) + вычислительная архитектура ЭВМ + ОС



«консоль оператора»

Как запрограммировать возможность ввода пользователем данных из консоли при работе программы

Ввод - Scanner

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

Объект (*), отвечающий за ввод данных
System.in – стандартный вход (тот самый «экран»)

Чтение данных:

```
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)

- Если при запуске после имени программы пишутся еще другие данные, то они приходят в нее как аргументы запуска.
- Java программа управляет этим через массив **args**
 - **String [] args** в **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);  
  
}
```

Отдать выходные данные

- **На экран пользователю** (человеку, запускавшему программу)
- В другую программу
- Куда-то еще (файл, сеть, устройство)

Вывод

```
System.out.print(x)
```

Если надо несколько значений:

```
System.out.print(x + ", " + y)
```

Если надо сделать перенос после вывода, то вместо print надо написать println

**ПОГОВОРИМ О ТРЕБОВАНИЯХ К АЛГОРИТМУ
(ПОМИМО ТЕХ, КОТОРЫЕ ПРЕДЪЯВЛЯЮТСЯ К
НЕМУ ПО УМОЛЧАНИЮ)**

Требования

- Эффективность
- Надежность написания, лаконичность, удобство поддержки кода
 - Code Conventions + Best Practices

Антипаттерны

- Дублирование кода – один из **антипаттернов** (как не надо делать).
- Есть еще паттерны – как стоит делать. Паттерны – формализация лучших практик.

Антипаттерн «Нагромождение переменных»

- Пример: подсчитать $(a - 1) * b + (b - 5) * d$

```
int x = a - 1;
```

```
int y = x * b;
```

```
int z = b - 5;
```

```
int u = z * d;
```

```
int v = y + u;
```

Все ли переменные нужны?

Антипаттерн «Дублирование кода»

```
if (x * x - 2 * x > 5 - x) {  
    System.out.println(x * x - 2 * x);  
}  
else {  
    System.out.println(5 - x);  
}
```

Некоторые выражения (одни и те же) придется считать несколько раз подряд.

Забегая вперед – это еще хуже с функциями: `f() * f() > 10 * f() && f() * f() - f() > 0`

Лучше:

```
int a = x * x - 2 * x;  
int b = 5 - x;  
int max = a > b ? a : b; // Math.max  
System.out.println(max);
```

Дублирование кода

(не влияющее на количество операций, но все равно неприятное – почему?)

- $n!! = 1 * 3 * 5 * \dots * n$ (если n – нечетное),
- и $2 * 4 * 6 * \dots * n$ (если четное).

```
if (n % 2 == 1) {  
    int p = 1;  
    for (int i = 1; i <= n; i+=2) {  
  
        p *= i;  
    }  
}  
else {  
    int p = 1;  
    for (int i = 2; i <= n; i+=2) {  
        p *= i;  
    }  
}
```

Типичное решение:

Причесываем

```
int p = 1;  
for (int i = 2 - n % 2; i <= n; i+=2) {  
    p *= i;  
}
```

Еще проще

Отнимая от n двойку, мы всегда получаем все четные или все нечетные числа.

```
int p = 1;  
while (n >= 1) {  
    p *= n;  
    n -= 2;  
}
```

Антипаттерн «хардкод»

Задача: вводится 10 чисел. Проверить, что сумма первой половины $>$ произведения 2й-половины, умноженного на удвоенное количество чисел.

Решение:

```
int s = 0;
for (int i = 0; i < 5; i++) {
    s += scanner.nextInt();
}
int p = 1;
for (int i = 5; i < 10; i++) {
    p *= scanner.nextInt();
}
System.out.println(s > p * 20);
```

Антипаттерн «хардкод»

Задача: вводится 10 чисел. Проверить, что сумма первой половины $>$ произведения 2й-половины, умноженного на удвоенное количество чисел.

Решение:

```
int s = 0;
for (int i = 0; i < 5; i++) {
    s += scanner.nextInt();
}
int p = 1;
for (int i = 5; i < 10; i++) {
    p *= scanner.nextInt();
}
System.out.println(s > p * 20);
```

Все хорошо, но вдруг нас попросят сделать не для 10, а для 12? 24? 1000? Другого четного числа?

Каждое изменение задачи потребует изменения кода, его перекомпиляции.

Как правильно?

Без хардкода

```
int n = 10;
int s = 0;
for (int i = 0; i < n / 2; i++) {
    s += scanner.nextInt();
}
int p = 1;
for (int i = n / 2; i < n; i++) {
    p *= scanner.nextInt();
}
System.out.println(s > p * n * 2);
```

