

# **02. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ (JAVA И СИНТАКСИЧЕСКИЙ САХАР)**

*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

# Что такое «программировать»

1. Понять поставленную задачу.
2. Определить входные данные (вход, input) и то, что является ответом (выходными данными, output).
3. Составить алгоритм, преобразующий input в output.
4. Реализовать алгоритм в виде текста программы на языке программирования.

# Входные данные (input)

- «То, что необходимо для решения задачи / работы алгоритма»,
- Входные данные передаются в программу *некоторым образом*.

ЗАДАЧА	ВХОДНЫЕ ДАННЫЕ (INPUT)
Сложить 2 числа	Два числа
Отсортировать файл	Файл
Найти друзей пользователя x	Сведения о пользователях, сведения о факте дружбы, имя пользователя x
Найти периметр треугольника	Стороны треугольника

# Выходные данные (output)

- «ответ на задачу»,
- выходные данные передаются программой куда-то «наружу».

ЗАДАЧА	ВЫХОДНЫЕ ДАННЫЕ (OUTPUT)
Сложить 2 числа	Сумма
Отсортировать файл	Новый файл
Найти друзей пользователя x	Список друзей
Найти периметр треугольника	Число - периметр

# Для работы с данными

- Программа: «мне нужно данные где-то хранить»
- Фон Нейман: «в оперативной памяти (там же, где и хранишься ты сама)»
- Мы: «Все верно, но надо, чтобы было удобнее работать с этими данными при программировании»

# Переменная

## Ячейка в памяти

- Имеющая имя
- Имеющая размер в байтах
- Имеющая тип содержимого
  - Целое число, символ, дробь и др.

*«Переменная» - так как данные могут изменяться в процессе работы.*

# Имя переменной

## «Идентификатор»

[A-Za-z\_][A-Za-z0-9\_]\*

*Первый символ* – англ.буква (заглавная/строчная) или \_

*Следующие символы* – англ.буквы (заглавные/строчные), цифры или \_

Имя ОК	Имя не ОК
x	1x
abc20	A-B
_friends_of_User	;%\$x
sum	x y

# Тип переменной

- Определяет все возможные значения, которые могут храниться в переменной
- Определяет размер данных, которые хранятся в переменной
- Определяет, как интерпретировать хранимые биты



## Аналогия – кодовый замок

- **Размер** – 3 десятичные цифры
- **Значения** – наборы из 3х цифр от 000 до 999



# Какие могут быть типы данных?

- Число (целое, вещественное)
- Символ
- Булев тип (boolean) – истина/ложь
- Наборы однотипных данных
  - Набор чисел
  - Набор символов (строка)
- Наборы разнотипных данных
  - Записи / объекты
- Набор битов / байтов (файлы / потоки данных)
- и др.

*примитивные  
(скалярные)  
типы*

*ссылочные  
(non-primitive)  
типы*

# Примитивные (скалярные) типы в Java

- Числа
  - Целые
    - **byte** – 1 байт
    - **short** – 2 байта
    - **int** – 4 байта
    - **long** – 8 байт
  - Вещественные
    - **float** – 4 байта
    - **double** – 8 байтов
- Символы
  - **char** - 2 байта
    - Часто считается тоже целочисленным
- Логический тип
  - **boolean**


# Скалярные типы

- Переменная хранит одно значение
- Как же значение в переменную положить?

# Дальше будет код

Поэтому надо разобраться с тем, куда его писать в случае Java.

```
public class Task {  
    public static void main(String [] args) {  
        ...  
    }  
}
```



«пока сюда»

# Объявление переменных в коде

Тип имя

Пример:

```
int x
```

# Инициализация

```
int x;
```

```
x = 1
```

или

```
int x = 1;
```

**Неинициализированных переменных быть не должно!**

- %Username%, сколько будет  $x + 5$ ?
- А что такое  $x$ ?

# Как понимать присваивание

**x = 1**



в переменную (ячейку, контейнер) под названием **x**  
помещается значение 1

В «старых» языках:

**x := 1** или даже **x ← 1**

# А что такое $x = y$ ?

$x, y$  – переменные

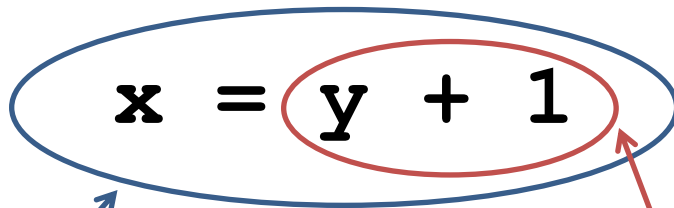
но трактовать их надо по разному!

$$x = y$$

«в переменную (ячейку)  $x$  нужно добавить значение, хранящееся в ячейке  $y$ »



# Statement and Expression



**Statement**  
(инструкция,  
команда,  
*оператор*)

**Expression**  
(выражение)  
*всегда чему-  
то равно*

# Когда определяется тип переменной

- При объявлении (`int x`);
- Это позволяет проверять корректность типов на этапе *компиляции*;
- Языки с таким подходом называют языками со **статической типизацией**;
- Но раз есть статические, то есть и динамические?

# Пример - Python

Динамическая («утиная») типизация

`x = 5` *# я - целое число*

`x = "s"` *# нет, я передумал, я - строка*

`x = 0.5` *# нет, я все же число, но вещественное*

**ВЕРНЕМСЯ К КОНКРЕТНЫМ ТИПАМ,  
РАЗБЕРЕМ ИХ УСТРОЙСТВО НА ПРИМЕРЕ  
ЦЕЛЫХ ЧИСЕЛ**

# Целые типы данных

*(на примере byte)*

- Какие числа поместятся в 1 байт?
- 1 байт – 8 бит, каждый бит – 0 или 1.
  - сколько возможных различных вариантов значений byte?

# 256

- Комбинаторика:  $2^8 = 256$
- Или в лоб:

$$00000000 = 0$$

$$00000001 = 1$$

$$00000010 = 2$$

...

$$11111110 = 254$$

$$11111111 = 255$$

Но как быть с отрицательными числами?

# Реализация знака -

- Представьте, что был бы еще 9й бит

$$\begin{array}{r}
 1 \ 0000 \ 0000 \ - \\
 \phantom{1 \ 0000 \ 0000} 1 \\
 1111 \ 1111 \\
 \hline
 -1
 \end{array}$$

$$\begin{array}{r}
 1111 \ 1111 \ - \ 1 \ = \ 1111 \ 1110 \\
 \hline
 -2
 \end{array}$$

и т.д.

# Числа со знаком

данные в byte, short, int, long – не всегда соответствуют своему двоичному коду

00000000 = 0

00000001 = 1

00000010 = 2

...

01111111 = 127

10000000 = -128 (хотя это 128 в двоичной)

...

11111110 = -2

11111111 = -1 (хотя это 255 в двоичной)

! Подумайте об алгоритме, как по целому числу узнать обратное ему число



# Число со знаком

- Первый бит – знак (0 – плюс, 1 – минус).
- Получается на само число остается на 1 бит меньше.
- byte – целые числа из  $[-128; 127]$ 
  - $128 = 2^7$ , хотя в байте 8 бит
  - По аналогии с int, short, long

# boolean

true, false

значения выражений сравнения имеют тип boolean.

Логические операции:

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

Разница:

$\&$ ,  $|$  – считают всё (оба аргумента).

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

- Ленивые (lazy) выражения.

# Ленивые операции – хорошо, но 100% всегда

Пример

Цель: выполнить операции `oper1` и `oper2`, и узнать, обе ли они успешны.

```
boolean result = oper1() & oper2()
```

В случае ленивого оператора `&&` и `false` по 1й операции `oper2` выполнена не будет.

# Сравнение значений

$<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $==$  (равно),  $!=$  (не равно)

На всякий случай:

- |                           |       |
|---------------------------|-------|
| • Чему равно $2 > 3$ ?    | False |
| • Чему равно $3 < 5$ ?    | True  |
| • Чему равно $3 \leq 3$ ? | True  |
| • Чему равно $3 < 3$ ?    | False |

# Вещественные типы

Канонический вид десятичной дроби:  $2.83 \cdot 10^{-9}$

Соответственно, надо хранить **мантиссу** и показатель.

Типы с *плавающей* точкой:

- *float* – 4 байта;
- *double* – 8 байтов.

Значение вещественного типа в коде также можно записать в каноническом виде: **2.83e-9** означает  $2.83 \cdot 10^{-9}$

# Загадка

Если подсчитать значение выражения  $2.0 - 1.1$  и вывести его на экран, то мы увидим:

0,8999999999999999

Почему?

# Конечные и бесконечные дроби

$$\frac{1}{3} = 0, (3)_{10} = 0,1_3$$

0.9 невозможно разложить на конечную сумму степеней двойки (т.е. двоичную дробь):

$$0.9 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

# Поэтому

**НИКОГДА** не делайте сравнение 2х вещественных значений с помощью:

- $==$
- $!=$
- $<=$
- $>=$

*! Подумайте, как сравнивать вещественные числа друг с другом правильно*



# Арифметика (со спецификой Java)

$+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$  (остаток от деления)

- любая целочисленная константа – `int`
- любая вещественная константа – `double`
- результат арифметического выражения с целыми числами – всегда `int` (если нет `long`). Даже такой:

```
byte b1 = 50;
```

```
byte b2 = 51;
```

```
byte b3 = b1 + b2; // не работает!
```

# Приведение типов

(тип)

```
byte b3 = (byte) (b1 + b2);
```

**Пример :**

```
byte b1 = 100;
```

```
byte b2 = 100;
```

```
byte b3 = (byte) (b1 + b2);
```

если результат не помещается в нужный тип, у двоичного представления убирают слева все «лишние» биты – например, для byte все, кроме последних 8ми

# Еще операторы (Java)

## ***a*++** (инкремент)

- Увеличение на единицу
- Разница в “ $x = ++a$ ”, “ $x = a++$ ”
  - » Постфиксный и префиксный инкременты

$a = 2; b = 3;$

$c = a+++++b$

Аналогично ***a*--** (декремент)

# **++, -- - пример синтаксического сахара**

Можно обойтись без этого оператора, но он в некотором смысле «удобнее»

`a = x;`

`x = x + 1;`



`a = x++;`

- Код пишется быстрее, при этом его размер уменьшается,
- НО! Практически всегда синтаксический сахар снижает читаемость кода.

# «Еще сахару»

$+=$ ,  $-=$ ,  $*=$ ,  $/=$  и т.д.

$x += y$

Для Java: Это не “ $x = x + y$ ”, а “ $x = (\text{тип } x)(x + y);$ ”

# Приоритет операций

- Приведение типов
- Скобки
- $*$ ,  $/$
- $+$ ,  $-$

# 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.**

# Приоритет операции приведения типа

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

*Ожидание: 0.5*

*Реальность: 0.5*

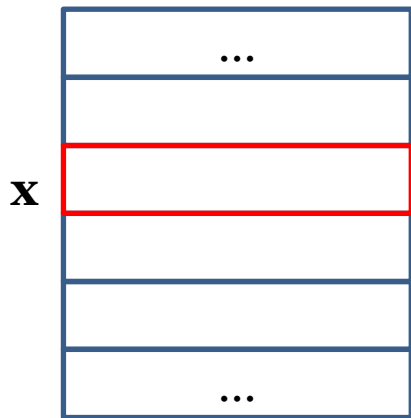


# Хранение значений переменных в ОП

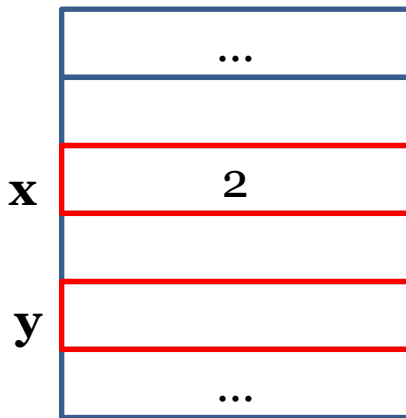
- **Стек (Stack)** – место хранения переменных примитивного типа
  - Небольшой размер (Stack Overflow)
  - Быстрые операции добавления и удаления
- **Куча (Heap)** – место хранения данных ссылочного типа
  - Динамически выделяемая память (большого размера)

# Память и примитивные переменные

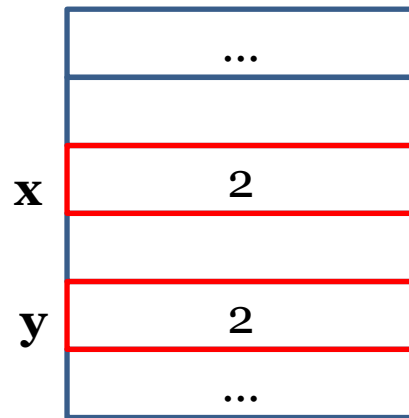
**int x;**



**x = 2;**  
**int y;**



**y = x;**



*Значение скопировалось*

Память под переменную примитивного типа выделяется в момент объявления  
 Примитивная переменная «хранит» в себе значение (можно это себе так представлять)  
 Пишу переменную, подразумеваю значение

# Ссылочные типы в Java (все остальные)

- Переменная хранит не содержимое, а ссылку на область памяти, где содержимое лежит.
  - К данным нужно обращаться по другому
    - `user.username`
- Объявление переменной и выделение памяти
  - 2 разных действия
    - `StringBuilder s = new StringBuilder("ITIS");`
      - » `StringBuilder` – изменяемая строка в Java

# Память и ссылочные переменные

**Object x;**

**null**

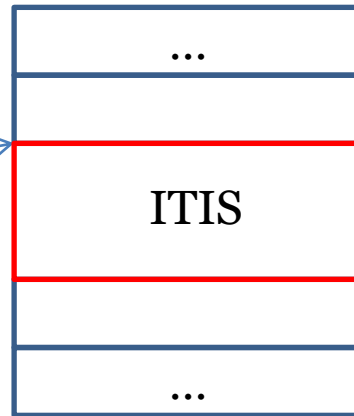


**x**



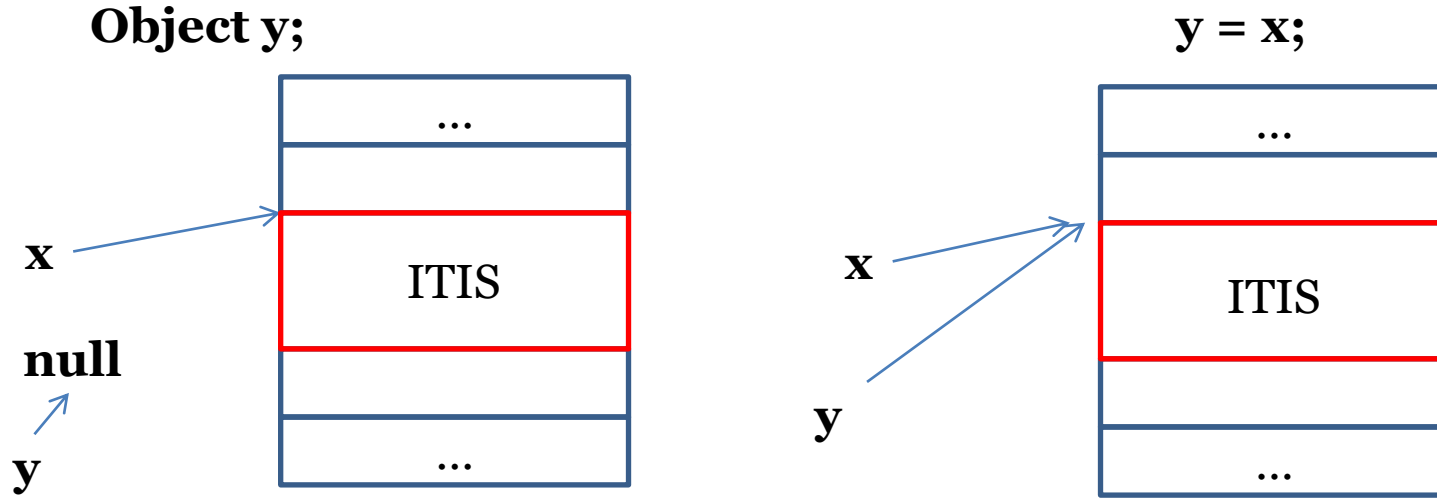
**x = new Object("ITIS");**

**x**



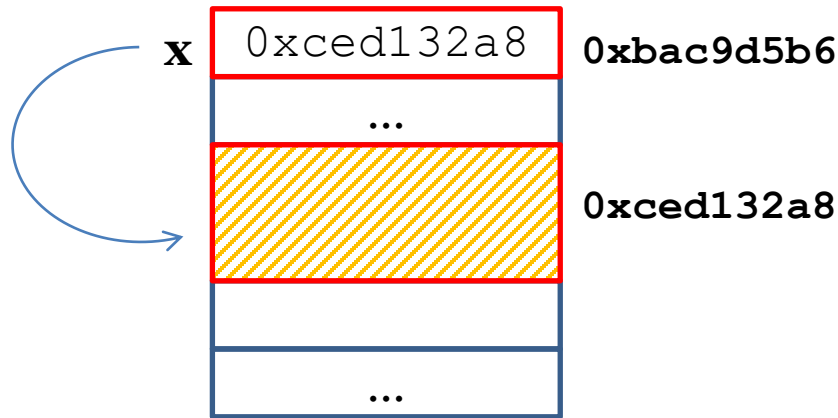
Ссылочная переменная «хранит» в себе ссылку на то место в памяти, где данные лежат. Т.е. сама по себе хранит просто число (номер ячейки, адрес).  
null – пустое значение такой ссылки.

# Память и ссылочные переменные



При присваивании не происходит копирования данных. Копируется только ссылка. (логично – у вас у всех есть ссылка на универ, в котором вы учитесь, но универ при этом не дублируется)

# Память – подход С.



***x (ссылочная переменная)*** хранит в себе явно адрес (сам хранит число как примитивная переменная) и в то же время через этот адрес указывает на данные.

*В си вообще говоря все не так просто, тут просто продемонстрирована модель.*

В Java прямого доступа к адресам нет, эту работу JVM прячет от греха подальше.

Но модель **такая же** (для понимания)

# 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*

# Что за структуры

- Последовательное соединение инструкций
- Ветвление
- Цикл



# Последовательное соединение операторов (Java)

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

Команда1; Команда2; ... КомандаN;

– **последовательность команд**

{ Команда1; Команда2; ... КомандаN; }

– **блок**

# Условный оператор (Java)

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**

```
}
```

# Заголовок условия

- 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!

# ; в заголовке нет!

Иначе получается очень интересный оператор.

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

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

# Dangling else

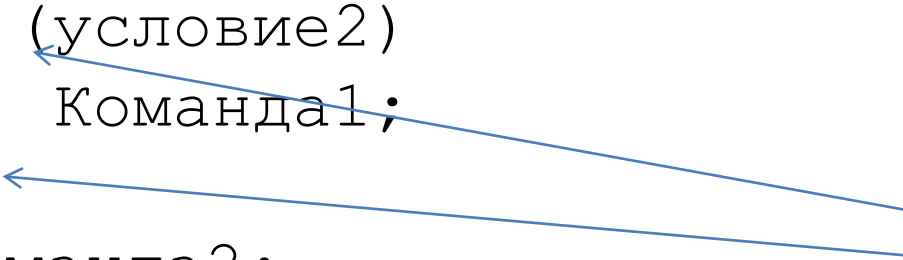
- Если нет операторных скобок, то else приклеивается к ближайшему if
- Может возникнуть вот такая ситуация:

```
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.*

*! Подумайте над примером*

# Тернарный оператор

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

```
double y = P ? A : 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»*

