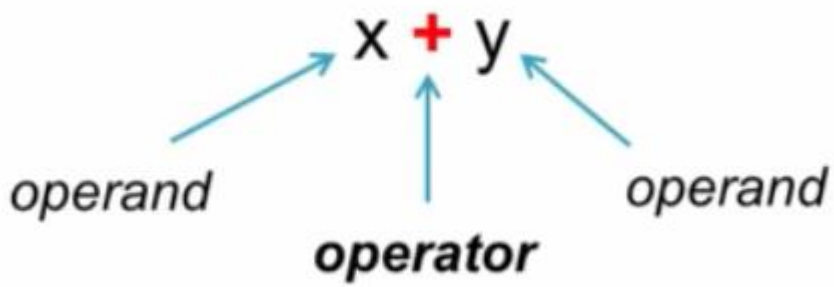
Основи на Java

TABLE 2.1 Numeric Data Types

Name	*Range*	*Storage Size*	
`byte`	-2^7 to 2^7-1 (-128 to 127)	8-bit signed	byte type
`short`	-2^{15} to $2^{15}-1$ (-32768 to 32767)	16-bit signed	short type
`int`	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	32-bit signed	int type
`long`	-2^{63} to $2^{63}-1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed	long type
`float`	Negative range: $-3.4028235E+38$ to $-1.4E-45$ Positive range: $1.4E-45$ to $3.4028235E+38$	32-bit IEEE 754	float type
`double`	Negative range: $-1.7976931348623157E+308$ to $-4.9E-324$ Positive range: $4.9E-324$ to $1.7976931348623157E+308$	64-bit IEEE 754	double type

Съдържание:

- Оператори
- Аритметични оператори
- Логически оператори
- Control-flow statement
- Примери



Operator Types

- Assignment
- Arithmetic
- Comparison
- Logical
- Bitwise
- Bit shift
- instanceof

Unary, Binary & Ternary Operators

- Unary
  - ***operator*** operand, e.g., **-**x
  - operand ***operator***, e.g., x**++**
- Binary
  - operand **operator** operand, e.g., x **+** 3
- Ternary (**?:**)
  - operand **operator** operand **operator** operand, e.g., (x > 3) **?** x **:** 0

Arithmetic Operators

- Addition (**+**) ~ *int i = 5 + 2;* → ~ Unary plus: int i = +5;
  ~ string concatenation

other uses

System.out.println("User name: " **+** name);

- Subtraction (**-**) → ~ Unary minus: int i = -x;
- Multiplication (*)
- Division (*/*)
- Modulus (**%**) ~ int i = 5 % 2; number is *odd* or *even*?

Shorthand Operators

- Pre & post increment/decrement
  - Applies to **addition** & **subtraction**
  - ++ or --
  - Increment/decrement by **1**, e.g. x**++** x = x + 1;
- Compound Arithmetic Assignment Operators
  - Applies to **all** arithmetic operations
  - +=, -=, *=, /=, %=
  - x **+=** 5; x = x + 5;

Post & Pre

```
int x = 5;
```

Post: x++; // 6

```
int y = x++; // y = 5, x = 6
```

```
int y = x;
x = x + 1;
int y = x--;   // y = 5, x = 4
```

Pre: ++x; // 6

```
int y = ++x; // y = 6, x = 6
```

```
x = x + 1;
int y = x;
int y = --x;   // y = 4, x = 4
```

Защо следния код ще даде грешка?!

```
    int index = 0;
    int[] array = new int[3];
    array[++index] = 10;

    array[++index] = 20;

    array[++index] = 30;

    System.out.println(index);
}
```

```java
static void compoundArithmeticAssignment() {
    int x = 100;

    System.out.println("x += 5: " + (x += 5));
    System.out.println("x -= 5: " + (x -= 5));
    System.out.println("x *= 5: " + (x *= 5));
    System.out.println("x /= 5: " + (x /= 5));
    System.out.println("x %= 5: " + (x %= 5));

    // Invalid
    //System.out.println("x =+ 5: " + (x =+ 5)); // Unary plus ~ x = +5
    //System.out.println("x =- 5: " + (x =- 5)); // Unary minus ~ x = -5
    /*System.out.println("x =* 5: " + (x =* 5));
    System.out.println("x =/ 5: " + (x =/ 5));*/
}
```

Operator Precedence

5 + 9 - 3 + 2 * 5

Rule 1: Multiplicative operators (*, /, %) have higher precedence over additive operators (+,-)

5 + 9 - 3 + **(**2 * 5**)**

Rule 2: Operators in *same group* are evaluated *left* to *right*

((5 + 9) - 3) + (2 * 5)

Use *parenthesis* to change evaluation order

((5 + 9) – (3 + 2) * 5)

Operand Promotion

Operands smaller than **int** are *promoted to int*

127 (byte) + 1 (byte) → 127 (int) + 1 (int) → 128 (int)

'a' + 'b' → 195

Same-Type Operations

If **both** operands are *int*, *long*, *float* or *double*, then operations are carried in that type and evaluated to a value of that type

5 + 6 → 11

1 / 2 → **0**, *not 0.5*

Mixed-Type Operations

If operands belong to different types, then smaller type is promoted to larger type

Order of promotion: int → long → float → double

1/2.0 or 1.0/2 → 1.0/2.0 → 0.5

char + float → int + float → float + float → float

9 / 5 * 20.1 → (9 / 5) * 20.1 → 1 * 20.1 → 1.0 * 20.1 → 20.1

Type of final result will be of *largest* data type

```
  static void charTypePromotion() {
    System.out.println("\nInside charTypePromotion ...");
    char char1 = 50; // Will be assigned corresponding UTF16 value 2
    System.out.println("char1: " + char1);
    System.out.println("(73 - char1): " + (73 - char1)); // char1 gets promoted to int, i.e.
    decimal equivalent 50 in UTF16 is used
    System.out.println("(char1 - '3'): " + (char1 - '3')); // char1 & '3' are promoted to
    ints
    System.out.println("('a' + 'b'): " + ('a' +'b')); // 'a' & 'b' are promoted to ints and
    the respective equivalents 97 & 98 are added
  }

  public static void main(String[] args) {
    // Language Basics 1
    //print();
    //primitives();
    //typeCasting();
    //arrays();
    //threeDimensionalArrays();
    /*varargsOverload(true, 1, 2, 3);
    varargsOverload(true, 1, 2, 3, 4, 5, 6, 7, 8);
    varargsOverload(true);*/
    charTypePromotion();
  }
}
```

~ operator precedence rule

~ operand promotion rule

~ same-type operations rule

~ mixed-type operations rule

Операции за сравнение

```java
// Comparison or Relational operators
static void comparisonOperators() {
    int age = 25;
    if (age > 21) {
        System.out.println("Graduate student");
    }
}
```

```java
// Comparison or Relational operators
static void comparisonOperators() {
    int age = 25;
    /*if (age > 21) {
        System.out.println("Graduate student");
    }*/
    System.out.println("age > 21: " + (age > 21));
    System.out.println("age >= 21: " + (age >= 21));
    System.out.println("age < 21: " + (age < 21));
    System.out.println("age <= 21: " + (age <= 21));
    System.out.println("age == 21: " + (age == 21)); // equal to (equality operator)
    System.out.println("age != 21: " + (age != 21)); // not equal to (equality
    operator)
}
```

```java
// Comparison or Relational operators
static void comparisonOperators() {
    int age = 20;
    /*if (age > 21) {
        System.out.println("Graduate student");
    }*/
    System.out.println("age > 21: " + (age > 21));
    System.out.println("age >= 21: " + (age >= 21));
    System.out.println("age < 21: " + (age < 21));
    System.out.println("age <= 21: " + (age <= 21));
    System.out.println("age == 21: " + (age == 21)); // equal to (equality operator)
    System.out.println("age != 21: " + (age != 21)); // not equal to (equality
    operator)

    boolean isInternational = true;
    //System.out.println("isInternational <= true: " + (isInternational <= true));
    System.out.println("isInternational == true: " + (isInternational == true));
    System.out.println("isInternational != true: " + (isInternational != true));
}
```

ЛОГИЧЕСКИ оператори

&&
AND

=

||
OR

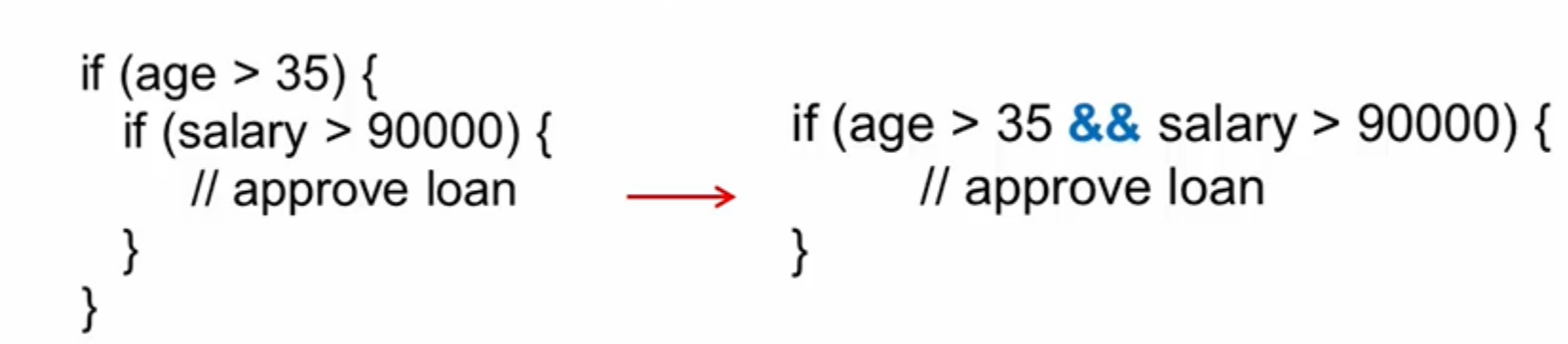
=

10% off

!
NOT *good credit history*

=

ЛОГИЧЕСКИ оператори -1



ЛОГИЧЕСКИ оператори -2

age = 37 and salary = 80000

(age > 35 **&&** salary > 90000) **false**

(age > 35 **||** salary > 90000) **true**

!(age > 35) **false**

ЛОГИЧЕСКИ оператори -3

Truth Table

x	true	true	false	false
y	true	false	true	false
x && y	true	false	false	false
x || y	true	true	true	false
!x	false	false	true	true

Short Circuit Operators ~ &&, ||

&&

left operand is *false*, return *false*

Conditional-And

||

left operand is *true*, return *true*

Conditional-Or

Short Circuit Operators

&& prevents **NullPointerException**

```
if (s.age > 21) {
        …
}

if (s != null && s.age > 21) {
            …
}
```

Ще даде грешка компилатора, защо?

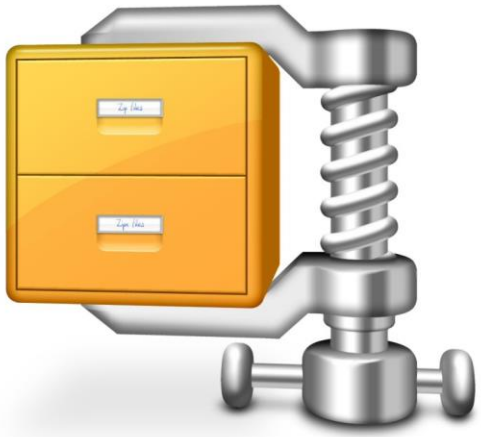
```
static void logicalOperators() {
    System.out.println("\nInside logicalOperators ...");
    int age = 37;
    int salary = 95000;
    boolean hasBadCredit = false;

    // 1. Core (AND, OR, NOT & Operator Chaining)

    if (age > 35 && salary > 90000) {
        System.out.println("Loan approved!");
    } else {
        System.out.println("Loan not approved!");
    }
```

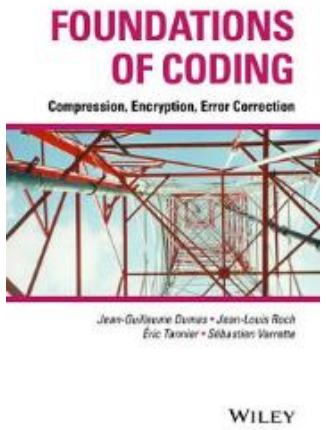
```
static void logicalOperators() {
    System.out.println("\nInside logicalOperators ...");
    int age = 37;
    int salary = 95000;
    boolean hasBadCredit = false;

    // 1. Core (AND, OR, NOT & Operator Chaining)

    if (!age <= 35 && salary > 90000 && !hasBadCredit) {
        System.out.println("Loan approved!");
    } else {
        System.out.println("Loan not approved!");
    }
```

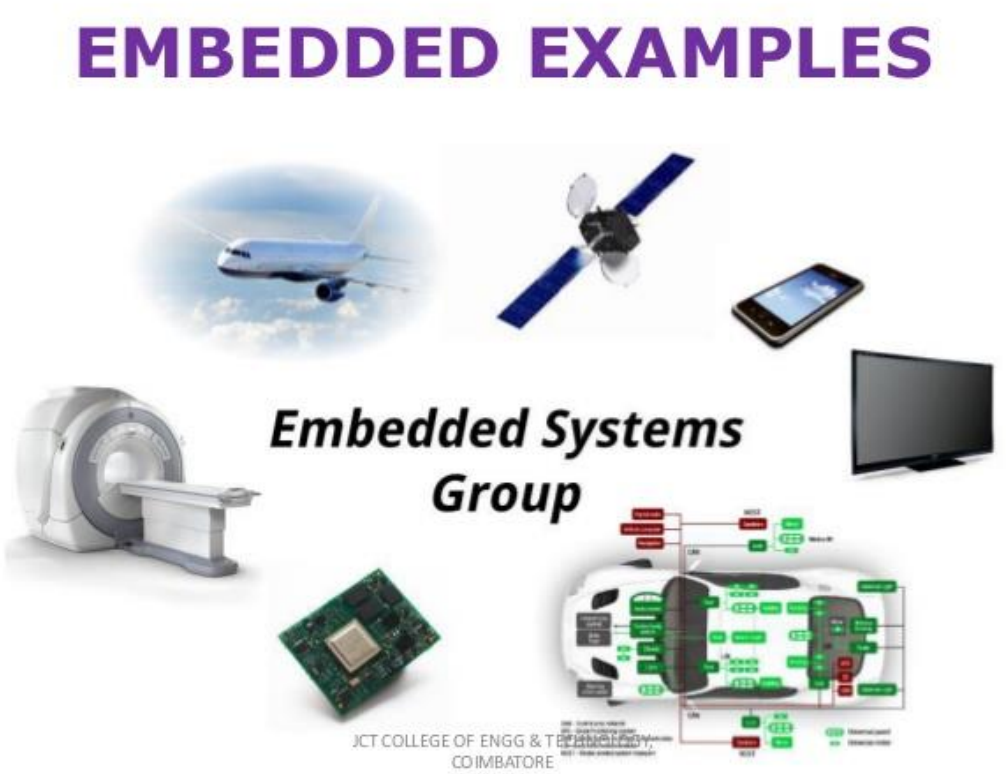
Bitwise Operators

- Operate on individual **bits** of operands
- Operands:
  - **Integer primitives** ~ *operand promotion* rule applies
  - **Boolean** (*rare*)

Приложения, които използват побитови операции







Побитови операции

- & → Bitwise AND
- | → Bitwise OR
- ^ → Bitwise XOR (Exclusive OR)
- ~ → Bitwise NOT

Bitwise AND (&)

- Returns 1 if ***both*** input bits are 1
- Let x = 1, y = 3

  (x & y) → 1

true = 1 & false = 0

```
00000000 00000000 00000000 00000001 (x)
00000000 00000000 00000000 00000011 (y)
00000000 00000000 00000000 00000001
```

Bitwise OR (|)

- Returns 1 if ***either*** of input bits is 1
- Let x = 1, y = 3

  (x | y) → 3

  00000000 00000000 00000000 00000**01** (x)

  00000000 00000000 00000000 00000**11** (y)

  00000000 00000000 00000000 00000**11**

Bitwise XOR (^)

- Returns 1 *ONLY* if one of the input bits is 1, *but not both*
- Let x = 1, y = 3

  (x ^ y) → 2

  00000000 00000000 00000000 0000000**0**1 (x)

  00000000 00000000 00000000 0000000**1**1 (y)

  00000000 00000000 00000000 0000000**1**0

Bitwise NOT (~)

- Inverts bits
- Let x = 1

  ~x → -2

  ~ (00000000 00000000 00000000 00000001) =

  11111111 11111111 11111111 11111110

Non Short Circuit Operators

- **&** and |
- Always checks *both* operands

Compound Bitwise Assignment

- operand1 **=** operand1 **&** operand2

  operand1 **&=** operand2

  e.g., boolean b = true

  b **&=** false; // Assigns false

Bit Shift Operators

- **Shifts bits**
- Operands ~ **integer primitives**
- << → Left-shift
- >>> → Unsigned right-shift
- >> → Signed right-shift

Left-shift Operator (<<)

- Left shifts left operand by # bits specified on right
- Example

  6 → 00000000 00000000 00000000 00000110

  6 **<<** 1 → 00000000 00000000 00000000 00001100 → 12
- Inserts *zeroes* at lower-order bits
- Same as *multiplication by powers of 2*

  6 << 1 → 6 * 2^1 → 12

  6 << 3 → 6 * 2^3 → 48

Unsigned Right-shift Operator (>>>)

- Right shifts left operand by # bits specified on right
- Inserts *zeroes* at higher-order bits
- Example

12 → 00000000 00000000 00000000 00001100

12 >>> 1 → 00000000 00000000 00000000 00000110 → 6

- Same as *division by powers of 2*

12 >>> 1 → 12 / 2^1 → 6

Signed Right-shift Operator (>>)

- Same as >>>, but padded with MSB
- Sign is preserved
- Example

-2,147,483,552 → 10000000 00000000 00000000 01100000

-2,147,483,552 >> 4 → 11111000 00000000 00000000 00000110

(-134,217,722)

ПРИЛОЖЕНИЯ:

- *Compiler optimizations*: Replace multiplication & division
- Hash tables, e.g., Java HashMap's hash function
- Embedded programming
- Games programming
- Systems with no floating-point support

Compound Bit Shift Assignment

- operand1 = operand1 << operand2

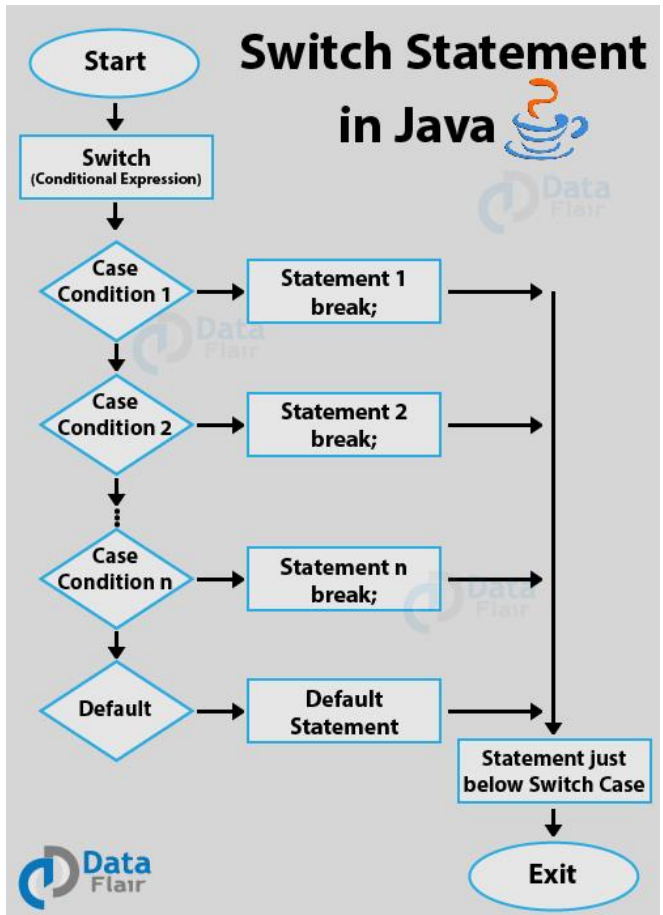
  operand1 **<<=** operand2
- operand1 = operand1 >>> operand2

  operand1 **>>>=** operand2
- operand1 = operand1 >> operand2

  operand1 **>>=** operand2

If-statements

```
static boolean ifStatement() {
    boolean approved = false;

    int age = 57;
    int salary = 75000;
    boolean hasBadCredit = false;

    if (age >= 25 && age <= 35 && salary >= 50000) {
        approved = true;
        System.out.println("age >= 25 && age <= 35 && salary >= 50000");
    } else if (age > 35 && age <= 45 && salary >= 70000) {
        approved = true;
        System.out.println("age > 35 && age <= 45 && salary >= 70000");
    } else if (age > 45 && age <= 55 && salary >= 90000) {
        approved = true;
        System.out.println("age > 45 && age <= 55 && salary >= 90000");
    } else {
        if (age > 55 && !hasBadCredit) {
            approved = true;
            System.out.println("age > 55 && !hasBadCredit");
        }
        System.out.println("else block");
    }


    System.out.println("outside if");
    return approved;
}
```


Start
Switch Statement in Java
Switch
(Conditional Expression)
Case Condition 1
Statement 1
break;
Case Condition 2
Statement 2
break;
Case Condition n
Statement n
break;
Default
Default Statement
Statement just below Switch Case
Exit
Data Flair

switch-statements

- Алтернативен начин на If-statement
- Пример:

```
int month = 3;
if (month == 1) {
   System.out.println(“January”);
} else if (month == 2) {
   System.out.println(“February”);
} else if (month == 3) {
   System.out.println(“March”);
} else {
   System.out.println(“April”);
}
```

```
int month = 3;
switch (month) {
   case 1: System.out.println(“January”);
           break;
   case 2: System.out.println(“February”);
           break;
   case 3: System.out.println(“March”);
           break;

}
```

- Още се нарича case-statement

switch Example

```
int month = 3;
switch (month) {
   case 1: System.out.println(“January”);
           break;
   case 2: System.out.println(“February”);
           break;
   case 3: System.out.println(“March”);
           break;
   default: System.out.println(“April”);
}
```

Типове, използвани в конструкцията case

- **Integer**, e.g., 7, x, x + y

  byte
  short
  char
  int

  cannot be long

- **String** (since Java 7)
- **enum**

Ограничения в конструкцията CASE

- Стойността трябва да е в обхвата на типа данни от условието;
- Константно условие : стойност, която е известна по време на компилирането
- Стойността трябва да е уникална
- Не може да е NULL

Стойността трябва да е в обхвата на типа данни от условието

```
byte month = 3;
switch (month) {
   case 1: System.out.println("January");
           break;
   case 128: System.out.println("February");
             break;
   default: System.out.println("April");
}
```

Константно условие : стойност, която е известна по време на компилирането

```
byte month2 = 2;
byte month = 3;
switch (month) {
   case 1: System.out.println(“January”);
            break;
   case month2: System.out.println(“February”);
            break;
   default: System.out.println(“April”);
}
```

```
static void switchExample() {
    System.out.println("\nInside switchExample ...");
    final byte month2 = 2;
    byte month = 3;
    switch (month) {
        case 1: System.out.println("January");
                break;
        case month2: System.out.println("February");
                break;
        case 3: System.out.println("March");
                break;
        default: System.out.println("April");
    }
}
```

Стойността трябва да е уникална

```
static void switchExample() {
    System.out.println("\nInside switchExample ...");
    //final byte month2 = 2;
    byte month = 3;
    switch (month) {
        case 1: System.out.println("January");
                break;
        case 1: System.out.println("February");
                break;
        case 3: System.out.println("March");
                break;
        default: System.out.println("April");
    }
}
```

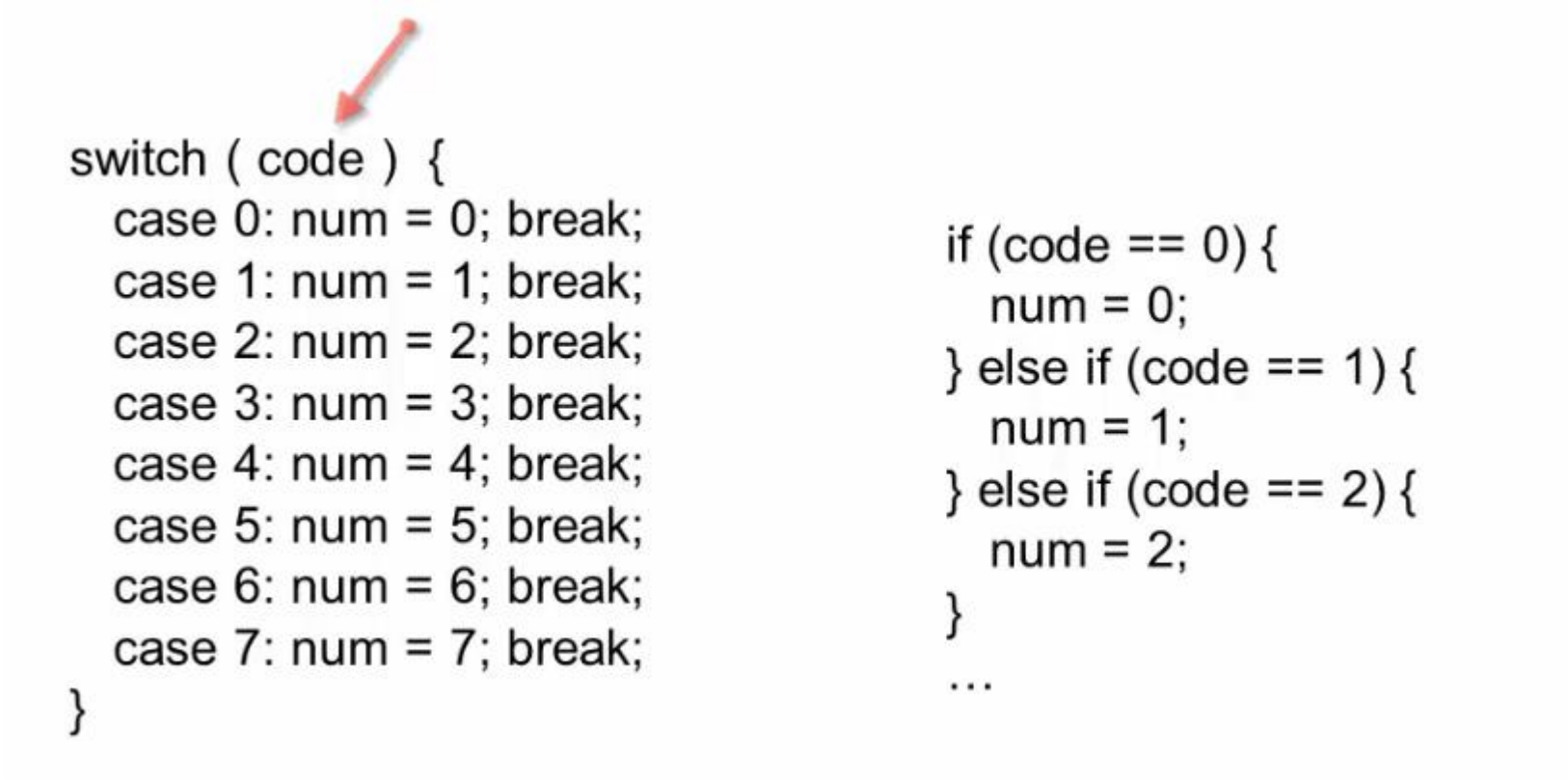
Стойността не може да е *null*

```
static void switchExample() {
    System.out.println("\nInside switchExample ...");
    //final byte month2 = 2;
    byte month = 3;
    switch (month) {
        case 1: System.out.println("January");
                break;
        case null: System.out.println("February");
                break;
        case 3: System.out.println("March");
                break;
        default: System.out.println("April");
    }
}
```

Кога конструкцията switch , не е подходяща да се използва:

- Когато имаме повече от едно условие;
- Когато искаме да проверим за повече условия;
- Условието в switch не е от типа integer, string или enum;
- Или в случаите, когато не може да се приложи ограничението;

Кога е удачно да се използва switch конструкцията:



```
switch ( code ) {
   case 0: num = 0; break;
   case 1: num = 1; break;
   case 2: num = 2; break;
   case 3: num = 3; break;
   case 4: num = 4; break;
   case 5: num = 5; break;
   case 6: num = 6; break;
   case 7: num = 7; break;
}
```

```
if (code == 0) {
   num = 0;
} else if (code == 1) {
   num = 1;
} else if (code == 2) {
   num = 2;
}
...
```

Кога е удачно да се използва switch конструкцията:

- Кода изглежда по-четлив;
- ЦЕЛТА: Когато имаме състояние, които с една стойност можем да ги опишем с една променлива;
- СКОРОСТ:
  - изпълнява се по-бързо и използва константни променливи;
  - Когато If условията са N на брой, ще имаме сложност на алгоритъма O(N). НО при конструкцията switch O(1);
  - При брой if условия > 100, се достигне границата за micro-optimization, тогава и switch кнструкцията няма да е ефективна; освен това съвременните компилатори сами избират как да реализира if или switch-конструкцията;

Ternary Operator

- Shorthand for *if-else with single statements*
- result = (boolean-expression) **?** *true-expr* **:** *false-expr*;

```
if (boolean-expression) {
   result = true-expr;
} else {
   result = false-expr;
}
```

- Пример:

```
int min = (x < y) ? x : y;
```

Кога се използва:

- По-четливо е@
- интелигентен начин на констуиране на стингов низ

```
greeting = "Hello " + (user.isMale() ? "Mr. " : "Ms. ") + user.name();
```

- Вместо:

```
String greeting = “Hello ”;
if (user.isMale()) {
   greeting += “Mr. “;
} else {
   greeting += “Ms. “;
}

greeting += user.name();
```

Кога се използва TERNARY

Logging

```
System.out.println( “John is " + (s.isMale() ? “male" : “female"));
```

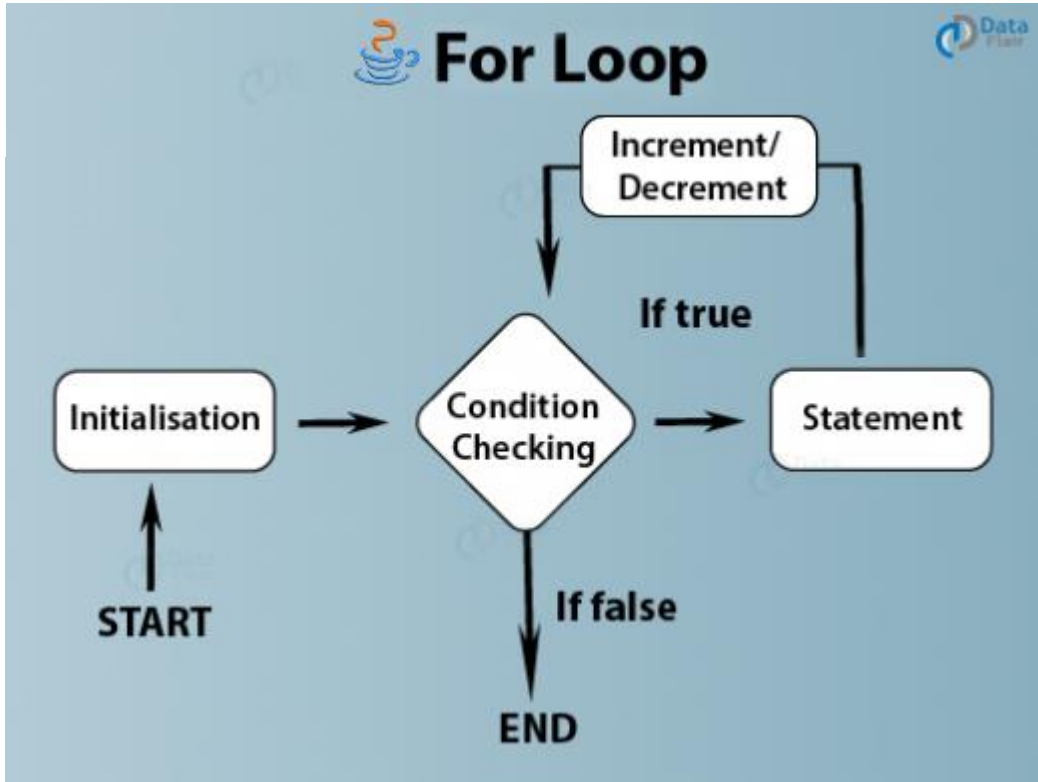
for Statement

Iteration statement

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i = 0; i < iArray.length; i++) {
    System.out.println(iArray[i]);
}
```

Iteration 1: i = 0 0 < 10 print 0 i = 1

Iteration 2: 1 < 10 print 1 i = 2



for Syntax


```
for (initialization; condition-expression; expression-list) {
    …
}
```

Инициализацията не е задължителна:

```
int iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int i = 0;
for ( ; i < iArray.length; i++) {
   System.out.println(iArray[i]);
}
```

Initialization ~ Declaration Statement

```
for (int i = 0; ; )
for (int i = 0, j = 1; ; )
for (int i = 0, int j = 1; ; ) // invalid
for (int i = 1, double d = 1.0; ; ) // invalid
```

Условието в цикъла:

- Трябва да е резултата Boolean
- Когато е true една итерация изпълнява инструкциите отт цикъла,

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

Списък в израза

```
for (int i = 0; i < iArray.length; System.out.println(iArray[i]), i++);
for (int i = 0; i < iArray.length; System.out.println(iArray[i++]));
```

Различен пример за цикъла FOR:

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int i = 0, j = 0;
for (i = 1, j = 1; i < iArray.length && j < iArray.length; i++, j++) {
    System.out.println(iArray[i] + " " + iArray[j]);
}
```

Разпечатване на четните стойности в масива:

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i = 0; i < iArray.length; i += 2) {
  System.out.println(iArray[i]);
}
```

Разпечатване на масиваа в рекурсен вид:

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i = iArray.length-1; i >= 0; i--) {
  System.out.println(iArray[i]);
}
```

Още примери ...

- Реализация на reverse метода – елемента ор позиция [0] да отиде на позиция [9], от [1] – [8] и т.н.

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i = 0, j = iArray.length-1; i < j; i++, j--) {
  int temp = iArray[i];
  iArray[i] = iArray[j];
  iArray[j] = temp;
}

for (int i = 0; i < iArray.length; i++) {
  System.out.print(iArray[i] + " ");
}
```

ОптимизаЦИЯ НА FOR

Първи вариант :

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i = 0, j = iArray.length-1, middle = iArray.length/2; i < middle; i++, j--) {
  int temp = iArray[i];
  iArray[i] = iArray[j];
  iArray[j] = temp;
}

for (int i = 0; i < iArray.length; i++) {
  System.out.print(iArray[i] + " ");
}
```

Втори вариант :

```
int[] iArray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i = 0, j = iArray.length-1, middle = iArray.length >>> 1; i < middle; i++, j--
  int temp = iArray[i];
  iArray[i] = iArray[j];
  iArray[j] = temp;
}

for (int i = 0; i < iArray.length; i++) {
  System.out.print(iArray[i] + " ");
}
```

Пример за намиране на най-голямото число в двумерен масив:

```
System.out.println("\nDisplaying Student Grades ... ");
int[][] studentGrades = {{77, 52, 69, 83, 45, 90}, {22, 71, 67, 69, 40}, {53, 87, 91
for (int i = 0; i < studentGrades.length; i++) {
    int max = 0;
    System.out.println("\nDisplaying grades of section " + i);

    for (int j = 0; j < studentGrades[i].length; j++) {
        if(studentGrades[i][j] > max ){
            max = studentGrades[i][j];
        }
        System.out.print(studentGrades[i][j] + " ");
    }
    System.out.print("\nmax: " + max);
}
```