The Basic Parts of Java

- Data Types
  - Primitive
  - Composite
    - array (will also be covered in the lecture on Collections)
- Lexical Rules
- Expressions and operators
- Methods
  - Parameter list
  - Argument parsing
- Control structures
  - An overview, very much C like

http://www.tiobe.com/tpci.htm

Position	(Position)	Programming Language	Ratings	(Rati
1	Java	22.442%	+6.55%	A
2	C	19.160%	+2.04%	A
3	C++	11.168%	-3.75%	A
4	Perl	9.274%	+0.31%	A
5	PHP	8.895%	+0.66%	A
6	(Visual) Basic	6.509%	-5.14%	A
7	C#	3.290%	+1.66%	A
8	Python	3.032%	-2.57%	A
9	JavaScript	1.768%	+0.26%	A
10	Delphi/Kylix	1.670%	-4.20%	A
11	SAS	1.299%	+0.65%	A
12	PL/SQL	0.959%	-0.39%	A
13	COBOL	0.855%	+0.36%	A
14	Lisp/Scheme/Dylan	0.718%	+0.40%	A

Primitive Data Types

- **`boolean`** {true, false}
- **`byte`** 8-bit
- **`short`** 16-bit
- **`int`** 32-bit
- **`long`** 64-bit
- **`float`** 32-bit
- **`double`** 64-bit
- **`char`** 16-bit uni-code

Natural numbers

Floating point numbers

- Also called built-in types
- All numbers are signed
- Have fixed size on all hardware platforms

Declarations, Example

```
// create some integers
int x, y, z;
x = 1234; y = 3;
z = Integer.MAX_VALUE;
// or similar for doubles
double v = 3.14e-23,
       w = 5.5,
       vv = Double.NEGATIVE_INFINITY;
// create some chars
char c1 = 'a';
Character c2;

// use a wrapper class
c2 = new Character ('b'); // read only

// A well-known constant
final static double PI = 3.14;
```

Declarations

- A declaration is the introduction of a new name in a program.
- All variables must be declared in advance.

- General forms

  type variableName1, *variableName2, variableName3;*

  type variableName1 = value1,
  variableName2 = value2,
  variableName3 = value3;

- Variables of primitive type are initialized with default values.
  - 0 (binary) what ever that means for the specific data type
- Constants are declared as **`final static`** variables

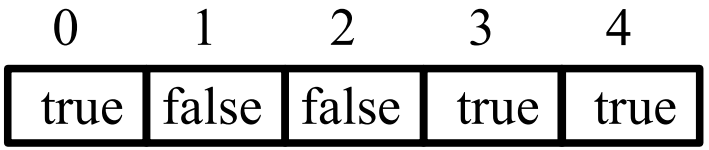
Array: A Composite Data Type

- An array is an indexed sequence of values of the *same type*.
- Arrays are defined as classes in Java.

- Example:

```
boolean[] boolTable = new boolean[MAXSIZE]
```

  - Elements are all of type **`boolean`**
  - The index type is always an integer
  - Index limits from 0 to **`MAXSIZE-1`**



- Bound-check at run-time.
- Arrays are first class objects (not pointers like in C)
- There are no struct/record type in Java.
  - Are replaced by classes
- Type safe enumeration added in Java 1.5

Lexical Rules

- A name in Java consists of [0-9][a-z][A-Z][_$]
  - cannot start with a number
  - national language letters can be used, e.g., æ, ø , and å.
  - no maximum length **`thisIsAVeryLongVariableName`**
  - Tip: avoid the use of $ in names (used for inner classes)

- All resevered word in Java are lower case, e.g., **`if.`**
- Case matters **`myVariable, myvariable`**

Java's Naming Convention

- Words run together, no underscore
- Intermediate words capitalized.
  - Okay: **`noOfDays, capacity, noInSequence`**
  - Not okay **`no_of_days, noofdays`**
- Name of classes: first letter upper case
  - Okay: **`Person, Pet, Car, SiteMap`**
  - Not okay: **`vehicle, site_map, siteMap`**
- Name of method or variable: first letter lower case
- Name of constants: all upper case, separated by underscore, e.g.,

  `java.math.E` and **`java.math.PI`**

Part of JavaSoft programming standard

Java's Naming convention
(http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html#367)

Commands in Java

- Very similar to the C programming language
- Assignment
  - **`variable = <expression>`**

- Method call
  - call by value
  - call by reference (almost)

- Control Structures
  - sequential (follow the flow)
  - branching (selective)
  - looping (iterative)

Block Statement

- Several statements can be grouped together into a *block statement.*
- A block is delimited by braces **`{ <statement list> }`**
- Variables can be declared in a block.
- A block statement can be used wherever a statement can be used.

```
// statements
if (weight < 20000)
  doStuffMethod();
else
  doOtherMethod();
```

```
// blocks
if (weight < 20000)
{
  doStuffMethod();
}
else
{
  doOtherMethod();
}
```

Expresions and Operators

- An *expression* is a program fragment that evaluates to a single value.
  - `double d = v + 9 * getSalary() % Math.PI;`
  - `e = e + 1;` (here `e` is used both as an *lvalue* and a *rvalue*)

- Arithmetic operators
  - Additive +, -, ++, -- `i = i + 1, i++, --i`
  - Multicative *, /, % (mod operator) `9%2 = 1, 7%4 = 3`

- Relational operators
  - Equality == (two '=' symbols) `i == j vs i = j`
  - Inequality != `i != j`
  - Greater-than >, >= `i > j, i >= j`
  - Less-than <, <= `i < j, i <= j`

Expresions and Operators, cont.

- Logical operators
  - and **&&** `bool1 && bool2`
  - or **||** `bool1 || bool2 || bool3`
  - not **!** `!(bool1)`
  - All are *short-circuit*

- Bitwise operators
  - and **&** `255 & 5 = 5` `15 & 128 = 0`
  - or **|** `255 | 5 = 255` `8 | 2 = 10`
  - xor **^** `3 ^ 8 = 11` `16 ^ 31 = 15`
  - shift left << `16 << 2 = 64` `7 << 3 = 56`
  - shift right >> `16 >> 2 = 4` `7 >> 2 = 1`

Expresions and Operators, cont.

- Assignment operators
  - can be combined with other binary operators
  - +=, -=, *=, /=, %=, >>=, <<=, &=, ^=
- Conditional Operator
  - Ternary operator
  - ?:
  - `int max = n > m ? n : m;`
- Precendence rules similar to C
  - `5 * 3 + 1 = ?`
- Associtivity rules similar to C
  - `5 - 3 - 1 = ?`

Methods in Java

- All procedures and functions in Java are methods on classes.
- The difference between a procedure and a function is the return type
  - `void myProcedure()`
  - `int myFunction()`
  - `MyClass myOtherFunction()`
- Methods cannot be nested!
- Returning
  - Implicit: When the last command is executed (for procedures).
  - Explicit: By using the **`return`** command.
    - Good design: only to have one **`return`** command each method

Methods in Java, cont.

- General format

```
ReturnType methodName (/* <argument list> */){
        // <method body>
        return stuff;
}
```

- Calling methods

```
double y = getAverageSalary();    // returns double
double z = getAverageSalary;      // an error!
boolean b = exists(/*args*/);     // returns boolean
exists(/*args*/);                 // ignore return val.
Person p = getPerson(/*args*/);   // returns Person
```

- Methods can be overloaded
  - **`double set(int i)`**
    `double set(int i, int j)`
  - Cannot overload on return value!

Class **IPAddress** Example

```
public class IPAddress{
   public static final String DOT = ".";
   private int[] n;           // example 127.0.0.1
   private String logical;    // example localhost
   /* Constructor */
   public IPAddress(){n = new int[4]; logical = null;}
   /* Sets the logical name */
   public void setName(String name){logical = name;}
   /* Gets the logical name */
   public String getName(){ return logical; }
   /* Sets numerical name */
   public void setNum(int one, int two, int three, int four){
       n[0] = one; n[1] = two; n[2] = three; n[3] = four;}
   /* Sets numerical name */
   public void setNum(int[] num){
      for (int i = 0; i < 4; i++){n[i] = num[i];} }
   /* Gets the numerical name as a string */
   public String getNum(){
      return "" + n[0] + DOT + n[1] + DOT + n[2] + DOT + n[3];    }
```

Class **IPAddress** Example, cont.

```
public class IPAddress{
 // <snip>
 public static void main(String[] args){
    // create a new IPAddress
    IPAddress luke = new IPAddress();
     luke.setName("luke.cs.aau.dk");
     System.out.println(luke.getName());
     luke.setNum(130, 225, 194, 177);
     String no = luke.getNum();
     System.out.println(no);
    // create another IPAddress
    IPAddress localHost = new IPAddress();
     localHost.setName("localhost");
     int[] lNum = {127, 0, 0, 1}; // array initialization
     localHost.setNum(lNum);
     System.out.print(localHost.getName());
     System.out.print(" ");
     System.out.println(localHost.getNum());
 }
```

Class `IPAddress`, Ugly Stuff

```
public class IPAddress{
    /* <snip> as previous class*/

    /* What is ugly here? */
    public void setName1(String name){
      logical = name;
      return;
    }
    /* What is ugly here? */
    public void setName2(String name, int i){
      logical = name;
    }

    /* What is ugly here? */
    public int setName3(String name){
      logical = name;
      return 1;
    }
```

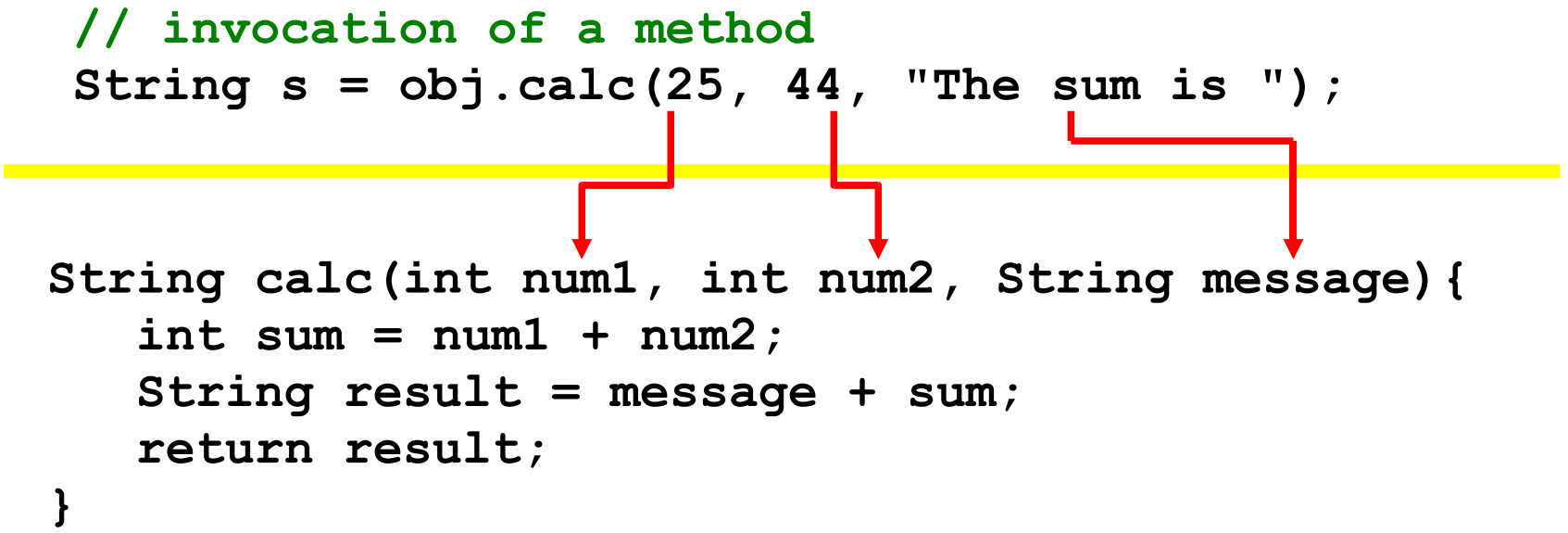
Parameter Mechanism

- All parameters in Java are pass-by-value.
  - The value of the actual parameter is copied to the formal parameter.
- A variable number of arguments is supported in Java 1.5
  - **`printf`** functionality
  - Not supported in Java 1.4
    - **`public static void main (String[] args)`**

- Passing Objects
  - Objects are accessed via a reference.
  - References are pass-by-value.
    - The reference is copied
    - The object itself is not copied
  - Via a formal parameter it is possible to modify the object directly.
  - The reference to the object cannot be modified.

Actual and Formal Parameters

- Each time a method is called, the actual parameters in the invocation are copied into the formal parameters.



Class **IPAddress** Example, cont.

```
public class IPAddress{
   /* Call by value */
   public int callByValue(int i){ i += 100; return i; }
   /* Call by value */
   public String callByValue(String s){s = "modified string"; return s;
     }
   /* Call by reference like method */
   public int callByRefLike(int[] a){
       int sum = 0;
       for(int j = 0; j < a.length; j++){ sum += a[j]; a[j] = 255;}
       return sum;
   }
   // in main method
   IPAddress random = new IPAddress()
   int dummy = 2;
   random.callByValue(dummy); // dummy unchanged
   String str = "not using new";
   random.callByValue(str); // str unchanged, also if called with new
   int[] ranIPNum = new int[4];
   random.callByRefLike(ranIPNum); // ranIPNUM to 255.255.255.255
```

The `static` Keyword

- For data elements
  - Are shared between all the instances of a class
  - `public static int i;`
  - `public static ArrayList = new ArrayList();`
  - `public static final char DOT = '.';`

- For methods
  - Can be accessed without using an object
  - `public static void main(String args[]){}`
  - `public static int getCount(){}`

Class **IPAddress** Example, cont.

```
public static void main (String[] args){
   private static int count = 0;
   public static final String DOT = ".";
   <snip>
   /* Constructor */
   public IPAddress(){
     n = new int[4];  logical = null;
     count++;}
   /* Get the number of IPAddress objects created */
   public static int getCount() { return count;}
   <snip>
   /* Handy helper method */
   public static void show(IPAddress ip){
      System.out.print(ip.getName()); System.out.print(" ");
      System.out.println(ip.getNum());
   }
}
```

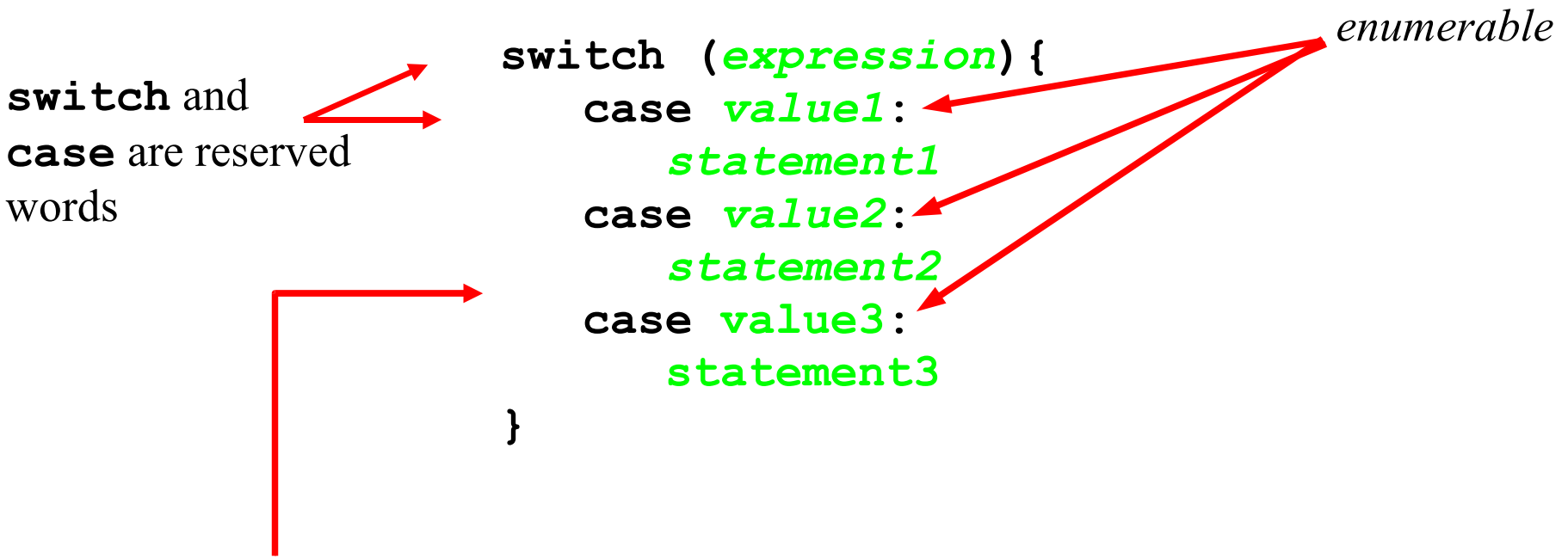
Control Structures

- Basic parts of the programs
- Borrowed from C programming language

- Branching
  - **`if-else`**
  - **`switch`** (case statement)
  - **`break`**, **`continue`**, **`goto`**
- Looping
  - **`while-do`**
  - **`do-while`**
  - **`for`**
  - **„`for-each`“**

The `switch` Statement

- The general syntax of a switch statement is



- enumerables can appear in any order
- enumerables do not need to be consecutive
- several case constant may select the same substatement
- enumerables must be distinct
- enumerable cannot use `case 1..9:`

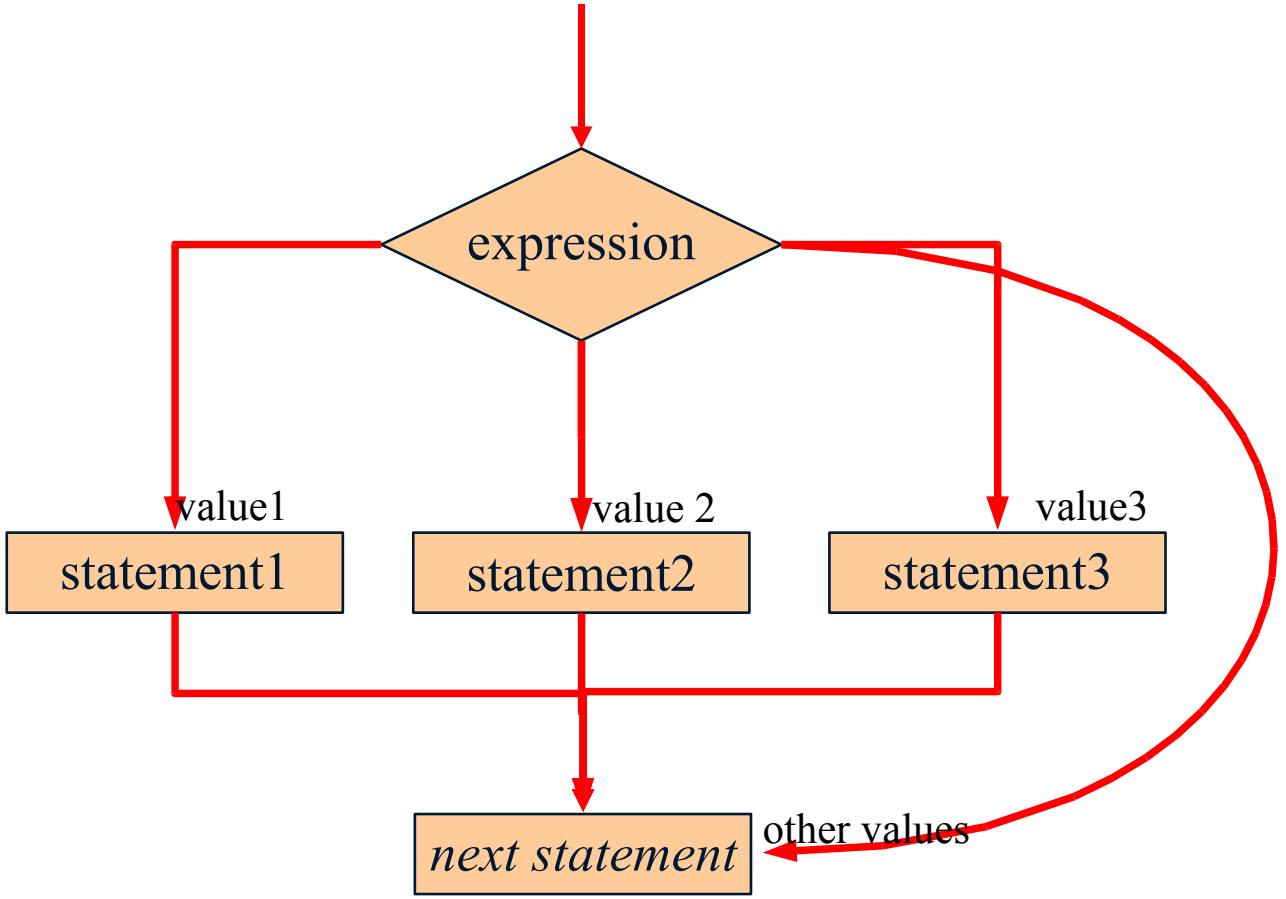
The `switch` Statement, cont.

- Often a break statement is used as the last statement in each case's statement list
- A break statement causes control to transfer to the end of the switch statement
- If a break statement is not used, the flow of control will continue into the next case

`break` exits the innermost enclosing loop or **`switch`**

```
switch (expression){
    case value1:
        statement1
        break;
    case value2:
        statement2
        break;
    case value3 :
        statement3
        break;
}
```

Logic of an **switch** Statement




```
switch (expression){
    case value1:
        statement
        break;
    case value2:
        statement2
        break;
    case value3:
        statement3
        break;
}
// next statement
```

The `switch` Statement, cont.

- A switch statement can have an optional default case.
- The default case has no associated value and simply uses the reserved word **`default`**.
- If the default case is present, control will transfer to it if no other case value matches.
- The default case can be positioned anywhere in the switch
  - it is usually placed at the end.

- Control falls through to the statement after the switch if there is no other value matches and there is no default case.

The **`switch`** Statement, cont.

- The expression of a switch statement must result in an integral data type
  - Works: **`byte`**, **`short`**, **`int`**, **`long`**, and **`char`**
  - No not work **`boolean`**, **`float`**, **`double`**, and **`String`**
- Note that the implicit boolean condition in a switch statement is equality (==), i.e., match the expression with a value.
- You cannot perform relational checks with a switch statement.,

```
switch (i < 7)
{
    case true :
        statement1
        break;
    case "Hello" :
        statement2
        break;
}
```

illegal, relational checking

not integral types

The **`switch`** Statement, Example

```
// function that gets a salary
int salary = getSalary();

switch(salary/20000) {
     case 0:         //[0,20.000)
          System.out.println("poor");
          break;
     case 1:         //[20.000,40.000)
          System.out.println("not so poor");
          break;
     case 2:         //[40.000,60.000)
          System.out.println("rich");
          break;
     case 3:         //[60.000,80.000)
          System.out.println("really rich");
          break;
     default:        //[80.000, max)
          System.out.println("Hi, Bill Gates");
}
```

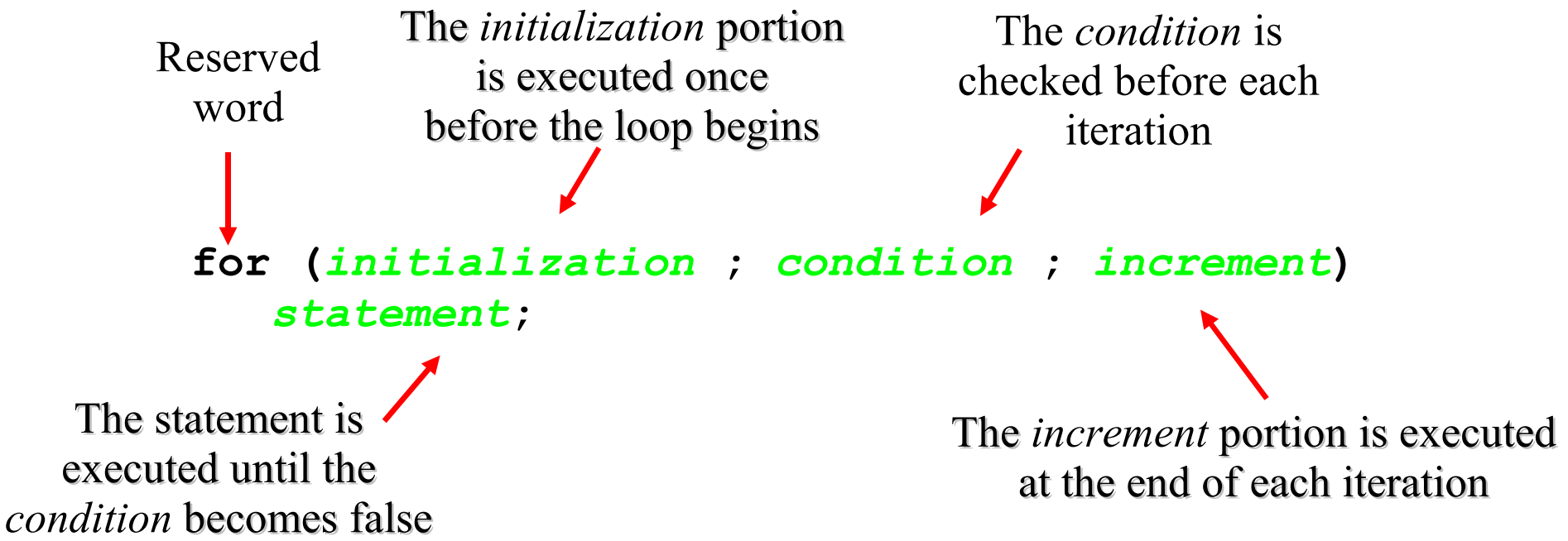
The **switch** Statement, Example

```
// What is wrong here?
switch(choice) {
     case 0..10:
          // small no stuff
          break;
     case 11..20:
          // big no stuff
          break;
     default:
          // default stuff
          break;

}
```

```
// What is ugly here?
switch(choice) {
     case 0:
          // do stuff one
          break;
     case 1:
          // do stuff two
          break;
     case 0:
          // do stuff three
          break;

}
```

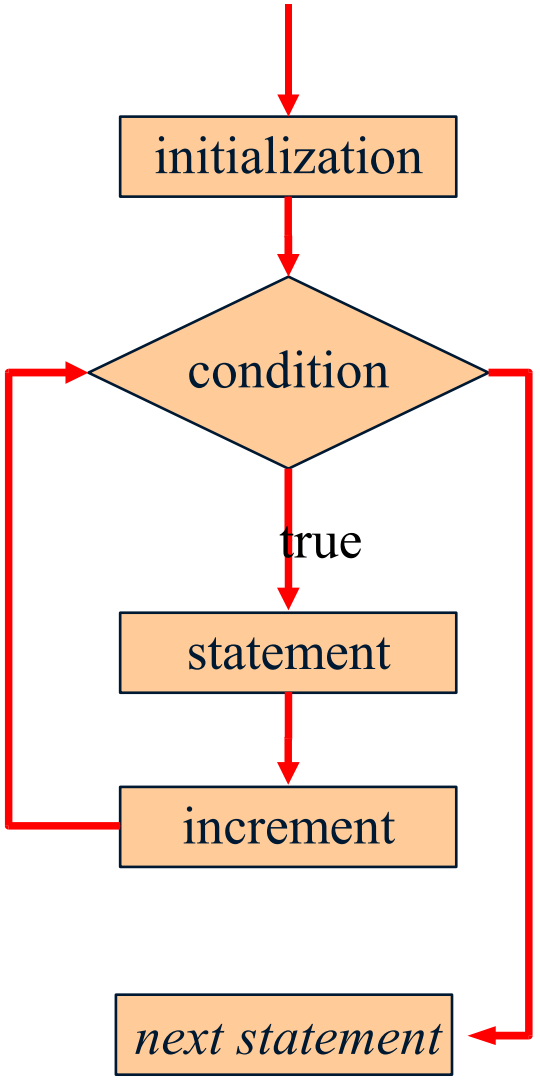
The `for` Statement

- The *for* statement has the following syntax




```
// equvivalent while statement
initialization
while (condition){
   statement;
   increment;
}
```

Logic of the `for` Statement



false

```
// Count from 1 to 10
int n = 10;
for (int i = 1; i <= n; i++)
  System.out.println(i);
// next statement

// what is wrong here?
for (int i=0; i < 10; i++){
    System.out.println(i);
    i--;
}

// what is ugly here?
for (int i = 0; i < 10;){
  i++;
  // do stuff
}

// what is ugly here?
int i;
for (i = 0; i < 10; i++){
  // do stuff
}
```

The `for` Statement, cont

- Like a *while* loop, the condition of a *for* statement is tested prior to executing the loop body.
- Therefore, the body of a for loop will execute zero or more times.
- It is well-suited for executing a specific number of times that can be determined in advance.
- Each expression in the header of a for loop is optional
  - Both semi-colons are always required in the for loop header.

```
// an infintive loop
for (;;){
  // do stuff
}
```

The `for` (each) Statement

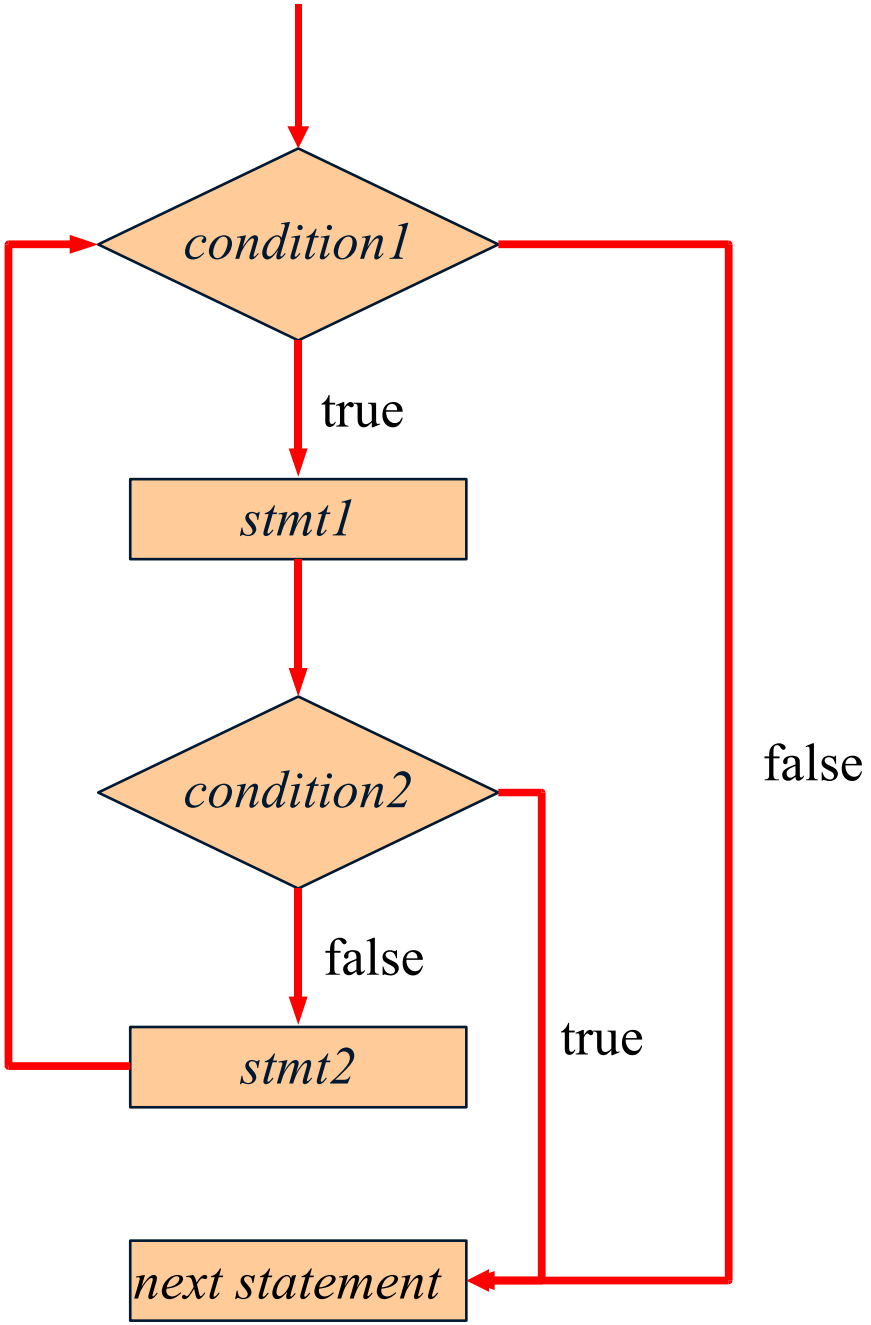
- „For each“ added in Java 1.5
- Will be introduced in more details in the lecture on collections.

```
public class ForEach {
     public static void main(String[] args) {
          // iterate over the String array
          for (String s : args) {
               System.out.println(s);
          }
          // iterator over an integer array
          int[] a = { 1, 2, 3, 4, 5 };
          for (int i : a) {
               System.out.println(i);
          }
     }
}
```

Branching

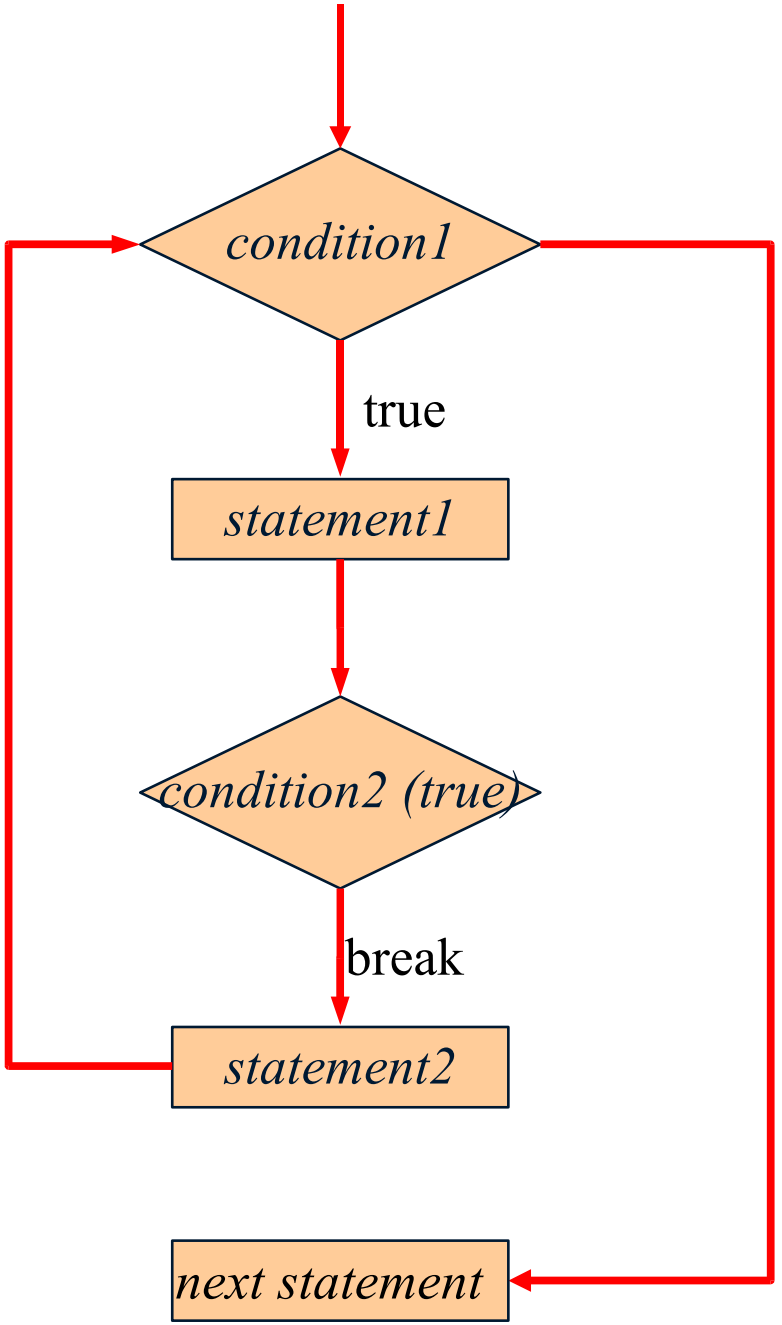
- **`break`**
  - Can be used in any control structure
  - Exits from the innermost enclosing loop
  - **`break <label>`**
- **`continue`**
  - Cycles a loop, e.g., jump to the condition checking
- **`return`**
  - Only from methods
  - Jumps out of the current method and returns to where the method was called from
  - **`return <expression>`**
- **`goto`**
  - Reserved word, not implemented
  - **`goto`** is extremely powerful, but even more dangerous

Logic of the **`break`** Statement



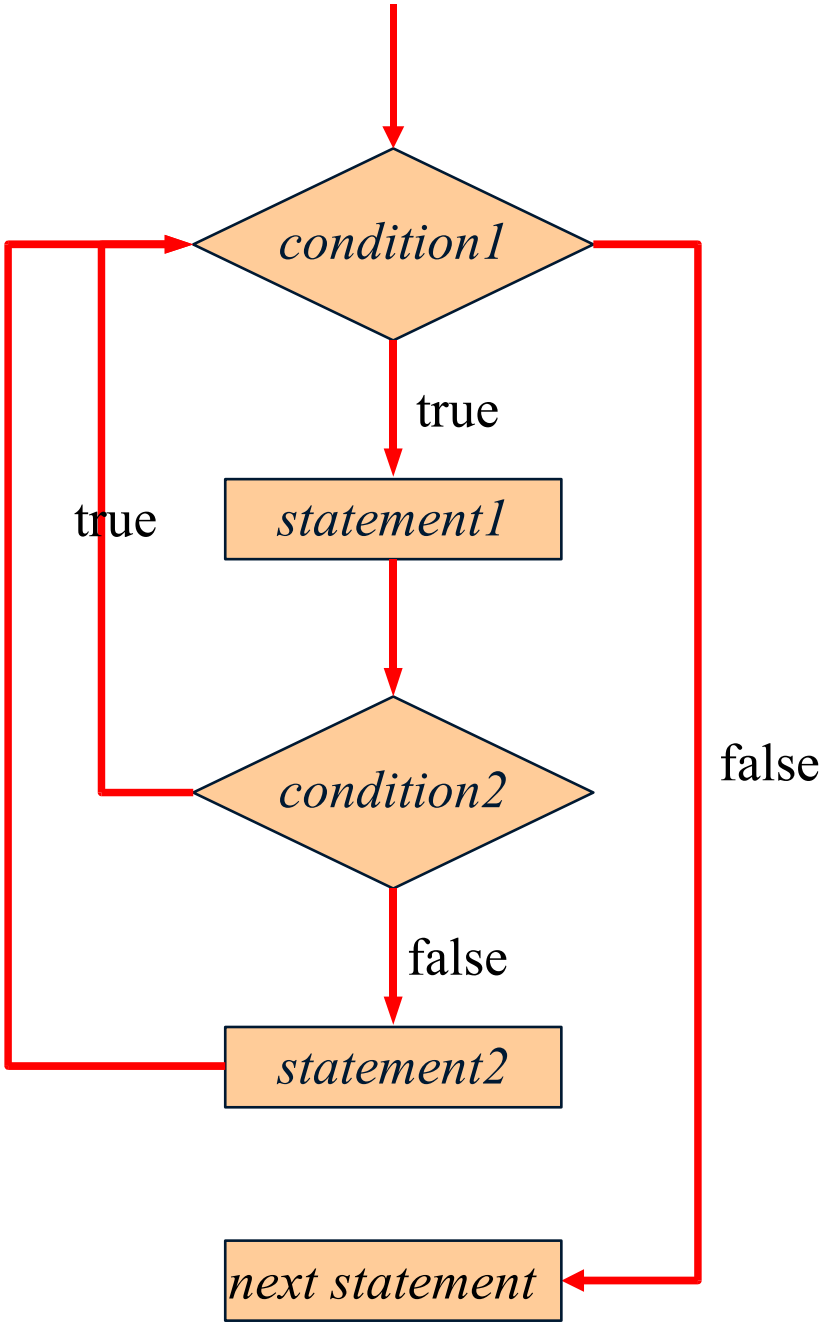

```
while (condition1) {
      stmt1;
      if (condition2)
        break;
      stmt2;
}
// next statement
```

Logic of the **`break`** Statement, cont




```
while (condition1) {
    statement1;
    while (true){ // condition2
        break;
    }
    statement2;
}
// next statement
```

Logic of the `continue` Statement




```
while (condition1) {
      statement1;
      if (condition2)
        continue;
      statement2;
}
// next statement
```

```
// what is wrong here?
while (condition){
      // many more statements
      continue;
}
```

`continue` Example

```
public void skipPrinting(int x, int y){
    for(int num = 1; num <= 100; num++){
          if((num % x) == 0){
                continue;
          }
          if((num % y) == 0){
                continue;
          }
          // This num is not divisible by x or y
          System.out.println(num);
    }
}
```

`break` and `continue` Example

```
for (int i = 3; i <= max; i++) {
    // skip even numbers
    if (i % 2 == 0)
        continue;
    // check uneven numbers
    boolean isPrime = true;
    for (int j = 2; j < i - 1; j++) {
        // is i diviseable with any number in [2..i-i]
        // then it is not a prime number so we break
        // of efficiency reasons
        if (i % j == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime)
        System.out.println(i + " is a prime number");
}
```

Summary

- Set of built-in data types
- Array are supported
  - No support for C-like structs (Pascal records)
- Methods
  - Procedure
  - Functions
- Argument passing
  - Always pass-by-value in Java
  - Actual and formal parameters.
- Control structures
  - Prefer **`for`** statement over **`while`** statement
  - Use **`break`**, **`continue`**, and **`return`** as little as possible

The `if` Statement

- The *if statement* has the following syntax:

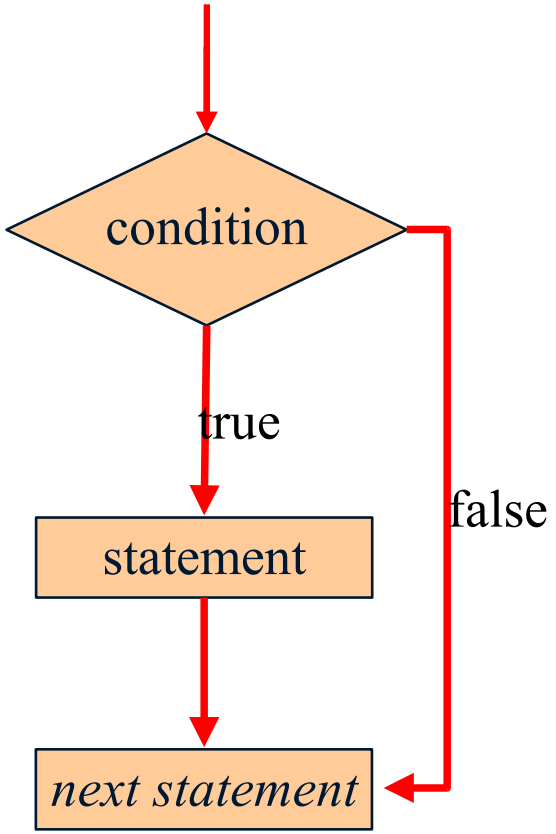
`if` is a reserved word

The condition must be a *boolean expression, i.e.* it must evaluate to true or false.

```
if (condition){
    statement;
}
```

If the condition is true the statement is executed.
If the condition is false, the statement is skipped.

Logic of an `if` Statement




```
// example 1
if (weight < 20000)
  doStuffMethod();
// same thing
if (weight < 20000){
  doStuffMethod();
}
// example 2
if (weight < 20000)
  doStuffMethod();
  doMoreStuff();

// NOT the same thing
if (weight < 20000){
  doStuffMethod();
  doMoreStuff();
}
// What is ugly here?
boolean b = getChoice();
if (b == true){
  // do stuff
}
```

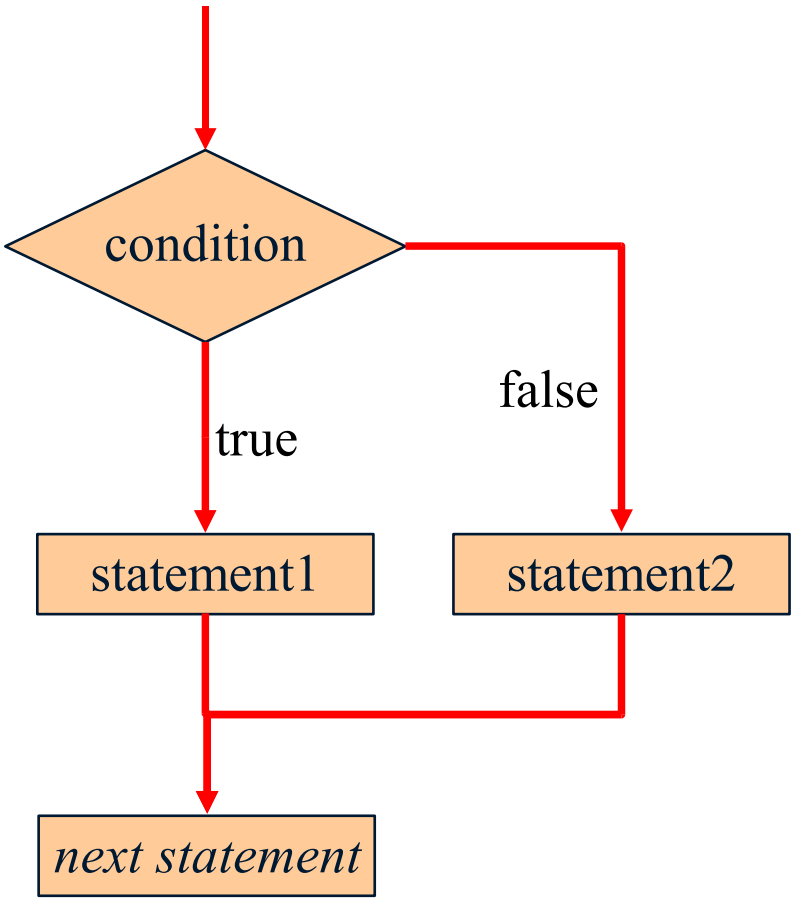
The `if-else` Statement

- The *if-else statement* has the following syntax:

```
if (condition){
     statement1;
}
else{
     statement2;
}
```

- If the condition is true *statement1* is executed. If the condition is false *statement2* is executed
- One or the other will be executed, but not both
- An **`else`** clause is matched to the last unmatched **`if`** (no matter what is implied by the indentation)

Logic of an `if-else` Statement

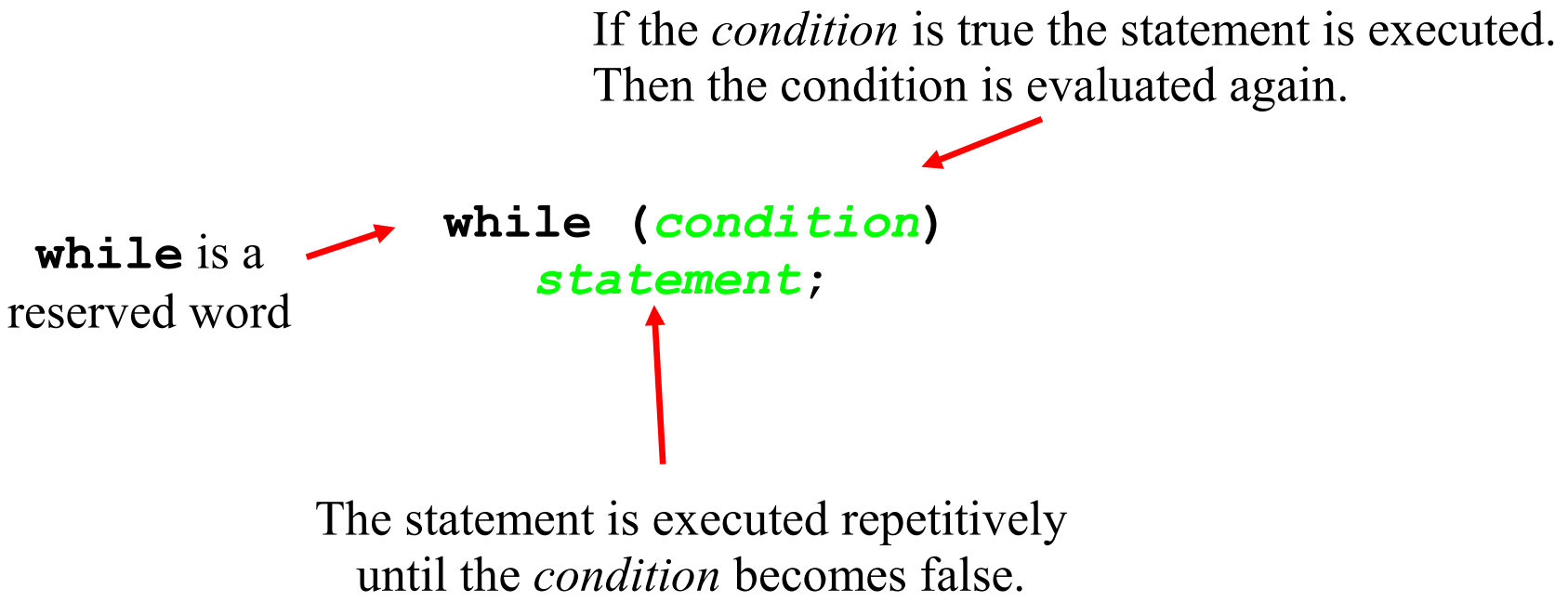



```
if (income < 20000)
  System.out.println ("poor");
else if (income < 40000)
  System.out.println ("not so poor")
else if (income < 60000)
  System.out.println ("rich")
else
  System.out.println ("really rich")
```

```
// what is wrong here?
char c = getChoice();
if (c == 'A')
  doStuffA();
if (c == 'B')
  doStuffB();
if (c == 'C')
  doStuffC();
```

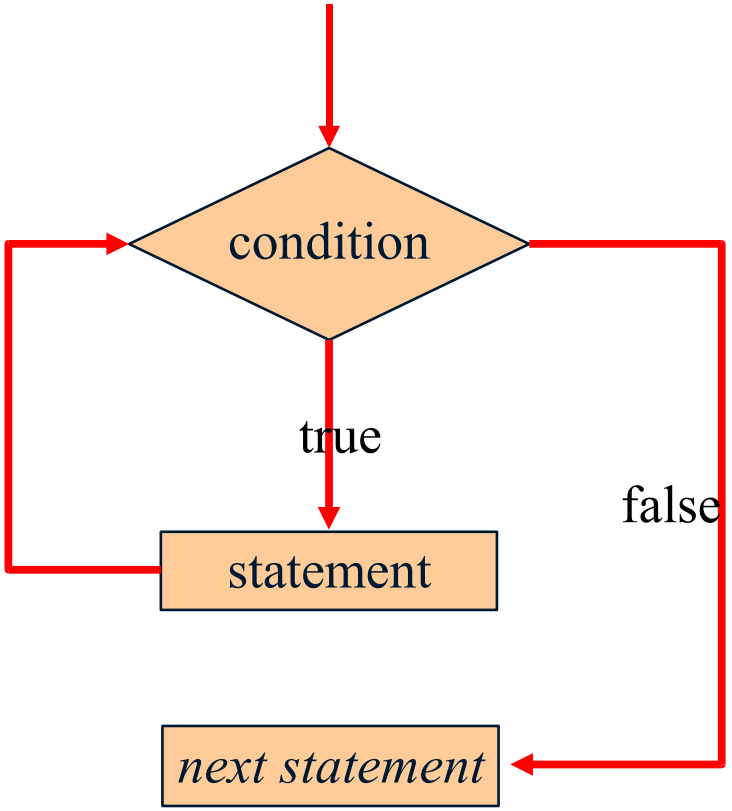
The `while` Statement

- The *while* statement has the following syntax



- Note, if the condition of a while statement is false initially, then *statement* is never executed
  - Tthe body of a while loop is executed zero or more times

Logic of the **`while`** Statement




```
// Count from 1 to 10
int n = 10;
int i = 1;
while (i <= n) {
     System.out.println(i);
     i = i + 1;
}
// next statement
```

```
// what is wrong here?
int i = 0;
while(i < 10){
    System.out.println(i);
    // do stuff
}
```

The `while` Statement, cont.

- The body of a while loop must eventually make the *condition* false.
- If not, it is an *infinite loop*, which will execute until the user interrupts the program.
  - This is a common type of logical error.
  - You should always double check to ensure that your loops will terminate normally.
- The while statement can be nested
  - That is, the body of a *while* could contain another loop
  - Each time through the outer *while*, the inner *while* will go through its entire set of iterations

The **do** Statement

- The *do statement* has the following syntax

do and **while** are reserved words

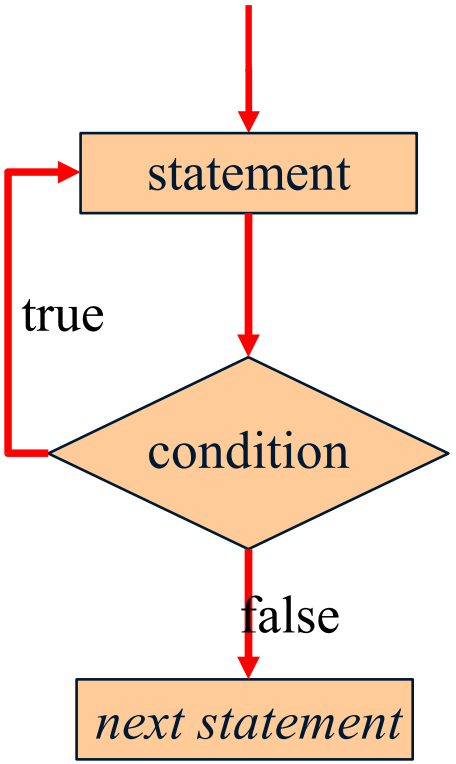
```
do {
   statement;
}
while (condition)
```

The *statement* is executed once initially, then the *condition* is evaluated.

The *statement* is executed until the *condition* becomes false.

- A *do* loop is similar to a *while* loop, except that the condition is evaluated after the body of the loop is executed.
  - Therefore the body of a do loop will execute at least one time.

Logic of the **do** Statement




```
// Count from 1 to 10
int n = 10;
int i = 1;
do {
     System.out.println(i)
      i = i + 1;
} while (i <= 10);
// next statement
```