Software Development in Java

Software Development in Java

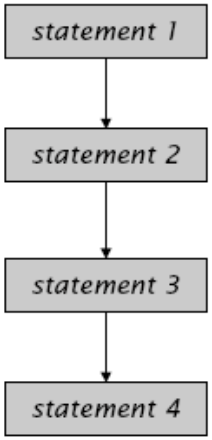
Week3

Conditionals & Loops

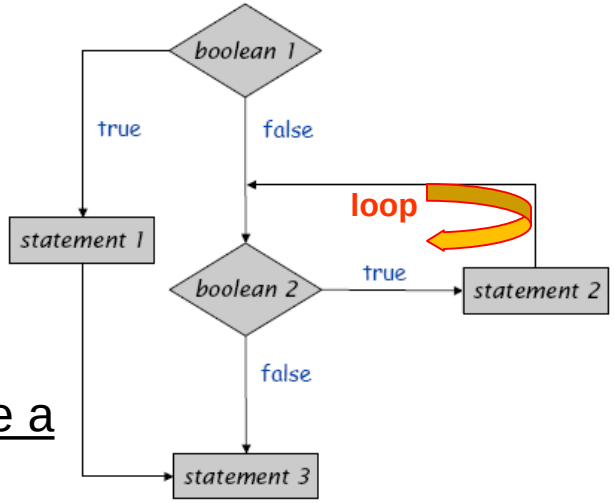
Control Flow

- **Control flow** is the ***execution* order** of instructions in a program

- Three control flow elements:
  1. **Sequential**
     - Statements are executed in the order they are written
     - All the codes we have seen so far are sequential



  2. **Branching (Decision making) -- Conditionals**
     - Provides computer programs with ability to make decisions and to carry out different actions according to different conditions



  3. **Repetition (Iteration) – Loops**
     - When the given condition satisfied, execute a block of statements repeatedly

Conditionals

if statement

if-else statement

Nested branches

switch statement

The *if* Structure

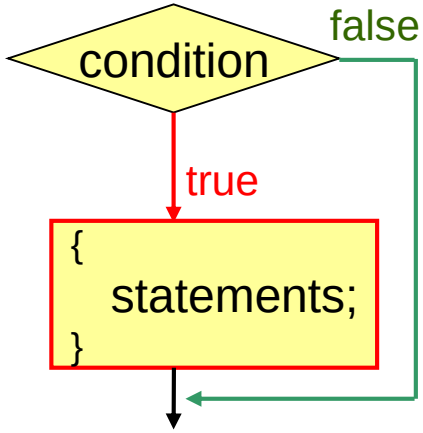
- The ***if*** structure allows to run statement(s) depending on a ***condition***

- The ***if*** structure has the syntax as:

condition is a boolean expression

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

if the condition is ***true,*** execute the statement(s)
if the condition is ***false***, the statement(s) will ***not*** be executed



- **if and only if** the *condition* is ***true,*** the `{statement(s)}` block is to be executed

The *if* Statement—Example1

```
public class IFStatement {
/*    demonstrate how to use if structure
 */
        public static void main(String[] args) {
        //reads a number from the command-line input
        //print the output comments accordingly

                final int ZERO = 0;
                int input=Integer.parseInt(args[0]);
                System.out.print("the input " + input + " is a ");

                if (input < ZERO){
                        System.out.print("negative ");
                }
                System.out.println("number");
}
```

true

false

```
java IFStatement -52
the input -52 is a negative number
```

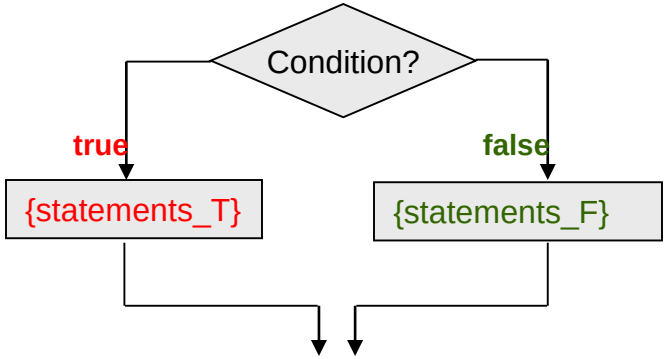
```
java IFStatement 52
the input 52 is a number
```

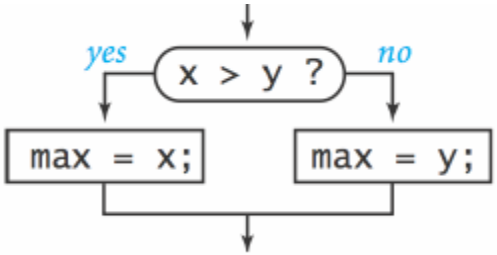
The *if/else* Structure

- The syntax of ***if/else*** structure :

```
if (condition){
    statements_T
}
else{
    statements_F;
}
```

- If the *condition* is ***true***, execute {statement_T};
- Otherwise, (indicating the *condition* is ***false***) then execute {statement_F}






```
if (x > y) max = x;
else       max = y;
```

Example—Quadratic.java

- Print the real roots of $ax^2+bx+c=0$
- Condition for real roots: $b^2-4ac>=0$ *(non-negative)*
- Solution: ***if-else*** is used for testing the condition, and then print output accordingly

```
public class Quadratic { //calculate real roots for any user-defined (user inputs for a, b and c ) quadratic function
        public static void main(String[] args) {
                double a=Double.parseDouble(args[0]);
                double b=Double.parseDouble(args[1]);
                double c=Double.parseDouble(args[2]);
                double discriminant=b*b-4.0*a*c;
                if (discriminant < 0.0) //no real roots
                {
                        System.out.println("No real roots");
                }
                else //calculate the roots
                {
                        System.out.println((-b+Math.sqrt(discriminant))/2.0*a);
                        System.out.println((-b-Math.sqrt(discriminant))/2.0*a);
                }
        }
}
```

Comparing Floating-Point Numbers

- Consider this code:

```
double r = Math.sqrt(2); // calculate √2 → r
double d = r * r -2;  // calculate (√2)² - 2 → d
if (d == 0)
    System.out.println("sqrt(2)squared minus 2 is 0");
else
    System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```

- It prints:

  sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16

Avoid using (Don't use!) "==" to compare floating-point numbers; instead, testing whether they are *close enough*:

```
//|x - y| ≤ ε

final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)

// x is approximately equal to y
//ε is a small number such as 1.0e-14
```

Comparing Strings

We assume that *input is a* pre-defined string; eg.: **String input=args[0];**

- **Don't use == for strings!**

  if (input == "Y") //**NOT appropriate!!!**

- Use ***equals()*** **method** from **String** class to test whether two strings contain exactly same characters in the same order

  √ if (input.***equals***("Y")) //return Boolean result

  √ if (input.***equals***("Mickey Mouse"))

- Case insensitive test ("Y" or "y")

  if (input.**equalsIgnoreCase**("Y"))

http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

Example—String comparisons

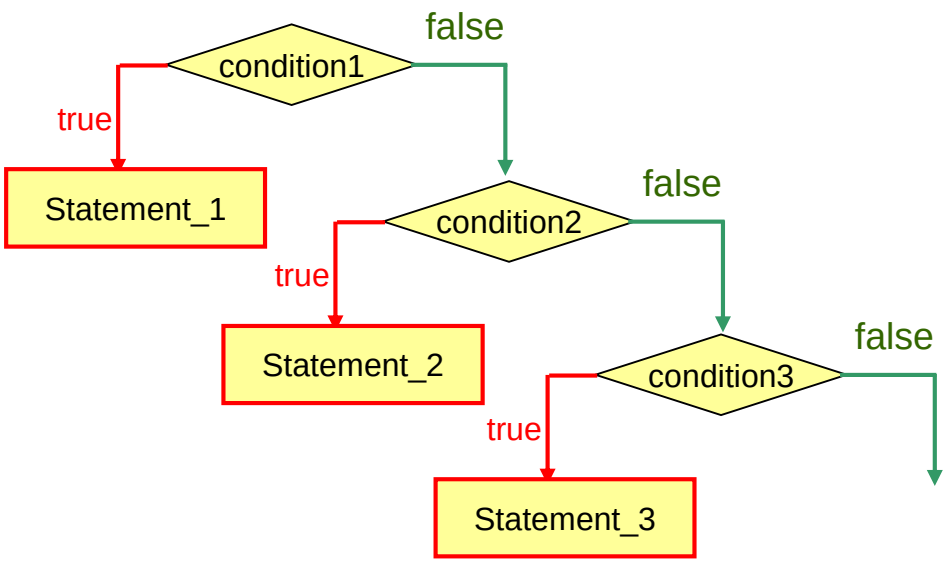
```
public class StrComp {
    public static void main(String[] args) {
        /* read two strings from command-line argument
        * and test whether they are equal
        */
        String s1=args[0];
        String s2=args[1];
        System.out.print(s1+" & "+s2 +" are ");
        if(s1.equals(s2)) System.out.println("equal strings");
        else System.out.println("not equal");
    }
}
```

Input: James james

Output: James & james are not equal

Multiple Alternatives: Sequence of Comparisons

```
if (condition_1) {
    statement_1;
}
else if (condition_2) {
    statement_2;
}
else if    . . .
else {
    statement_n;
}
```



Example—String comparison2

```
public class StrComp {
      public static void main(String[] args) {
      /* compare two string arguments
       * practice String comparison & multiple alternatives
       */
          String s1=args[0];
          String s2=args[1];
          int comp=s1.compareTo(s2);
          if(comp==0) System.out.println("equal strings");
          else if (comp<0) System.out.println(s1+" is before " +s2);
          else System.out.println(s1+" is after " +s2);        }
}
```

The ***String*** class has method ***compareTo()*** to test whether one string comes before the other

Suppose *s1* and *s2* are two defined strings

s1.***compareTo***(s2)

-- returns **a negative value** if *s1* is **before** *s2* in the dictionary order

-- returns **0** if *s* and *t* are **equal**

-- returns **a positive value** if *s1* is **after** *s2* in the dictionary order

Input: wang Wang

Output: wang is after Wang

Input: car cargo
Output: car is before cargo

Multiple Alternatives

```
public class Earthquake
{//measure how severe an earthquake is and print the corresponding warning
    public static void main(String [] args)
    {
        double richter = Double.parseDouble(args[0]);
        String r;
        if (richter >= 8.0)          r = "Most structures fall";
        else if (richter >= 7.0)   r = "Many buildings destroyed";
        else if (richter >= 6.0)   r = "Many buildings considerably damaged, some collapse";
        else if (richter >= 4.5)   r = "Damage to poorly constructed buildings";
        else if (richter >= 3.5)   r = "Felt by many people, no destruction";
        else if (richter >= 0.0)   r = "Generally not felt by people";
        else                        r = "Negative numbers are not valid";
        System.out.println(r);

    }
}
```

The *switch* Statement: Another Way for Multibranch

- ***switch*** statement is to compare **a single value** against several **constant alternatives**.

```
switch (single_Control_Variable)
{
   case Constant_No0:
         statements …
         break;
   case Constant_No1:
         statements …
         break;
   case ……
   default:
         statements…
         break;
}
```

If Control_Variable matches Constant_No1, control jumps to here and the statement block will be executed until *break*.

If optional *default* case presents, control jumps to here when NO other case value matches.

Type of *Control_Variable* can be:

- **Integer (byte, short, int, long); or**
- **character (char)**
- **String**

Not floating value, Not Boolean value

- When *break* statement is encountered, control leaves the *switch block*
- If a *break* statement is not used, the flow of control will continue to next case

Do not omit ***break***!

The ***switch*** Statement: Example

- limit the range of input from 0~9

```
int digit=Integer.parseInt(args[0]);

if (digit == 1) System.out.print("one");
else if (digit == 2) System.out.print("two");
else if (digit == 3) System.out.print("three") ;
else if (digit == 4) System.out.print("four");
else if (digit == 5) System.out.print("five") ;
else if (digit == 6) System.out.print("six") ;
else if (digit == 7) System.out.print("seven") ;
else if (digit == 8) System.out.print("eight") ;
else if (digit == 9) System.out.print("nine") ;
else System.out.print("error") ;
```

```
int digit=Integer.parseInt(args[0]);

switch (digit)
{
    case 1: System.out.print("one"); break;
    case 2: System.out.print("two"); break;
    case 3: System.out.print("three"); break;
    case 4: System.out.print("four"); break;
    case 5: System.out.print("five"); break;
    case 6: System.out.print("six"); break;
    case 7: System.out.print("seven"); break;
    case 8: System.out.print("eight"); break;
    case 9: System.out.print("nine"); break;
    default: System.out.print("error"); break;
}
```

Loops

while
do-while
for

Nested loops

Repetition/Iteration

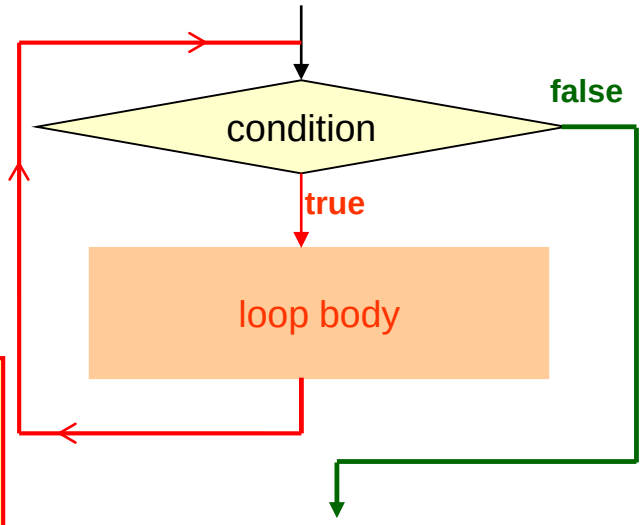
- **Loop** is a programming structure that repeats an action for a certain number of times according to a Boolean condition
- **Body of loop** is the part to be repeated
- **Iteration** is every repetition of the loop body

- Three types of loops in Java:
  - ***while***
  - ***do-while***
  - ***for***

while Loop

- As long as the condition is true, a ***while*** structure executes a block of code (loop body) repeatedly.

```
while (condition)  {
        //loop body to be repeated
}
```



The condition is evaluated first and if it is true, the loop body will be executed

This procedure will be repeated until the condition becomes false

while Loop

- Most commonly, the loop body is a block statement (set of statements delimited by { })

```
while (i <= number) {
    System.out.println(i);
    i++;
}
```

Revising the condition!!!

How many times this loop body will be executed if we set:

1. i=0 and number=5?
2. i=1 and number=5?
3. i=1 and number=0?

```
int i=0; number=5;
while (i <= number) {
    System.out.println(i);
    i++;
}
System.out.println("no looping");
```

<u>If the condition of a *while* loop is *false* initially, the loop body will not be executed</u>.

while Loop -- Tracing

When you trace a loop, you keep track of the current line of code and the current values of the variables. Whenever a variable's value changes, you cross out the old value and write in the new value.

```
int i = 1;
int sum =0;
while (i <= 5) {
    sum=sum+i;
    i++;
}
```

//sum=1+2+…+5

sum	i	i <= 5
0	1	true
1	2	true
3	3	true
6	4	true
10	5	true
15	6	false

Common Error: Infinite Loops

- The body of a loop **must eventually make the loop condition false**, otherwise, the loop will keep running (infinite loop) and needs user's intervention to terminate it.

- Example 1:

this is an infinite loop!

```
int years = 0;
while (years < 10)  {
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

Change the control condition in the loop body!

years++;

- Example 2:

```
int years = 10;
while (years > 0)  {
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

years--;

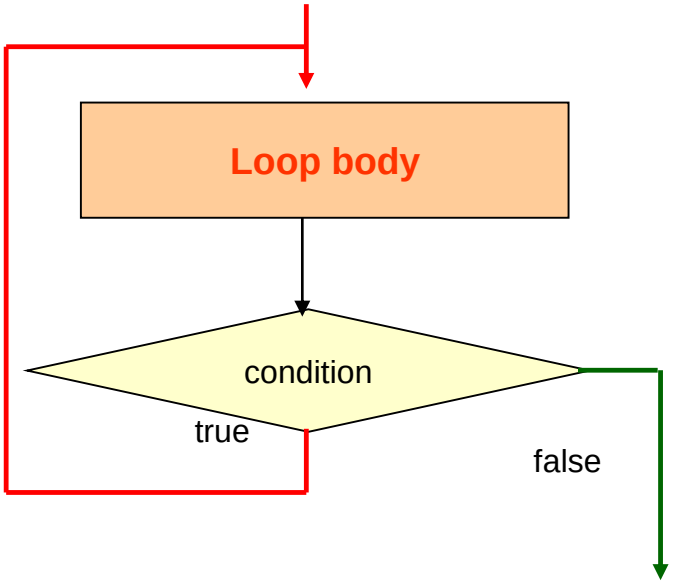
// Oops, the condition always be true
// the loop will not stop……

! Loops run forever –must kill program (CTRL-C for Windows OS)

do-while Loop

- Executes **<u>loop body at least once</u>**:

```
do {
    loop body
} while (condition);
```



Do/Execute the loop body once initially
and then evaluate the condition; while condition is true, the loop body will be repeated

do-while Loop

- Example:

do loop

```
do
{
   System.out.println(i);
   i=i-1;
} while (i > 0);
```

while loop

```
while (i> 0)
{
   System.out.println(i);
   i=i-1;
}
```

do-while loop executes <u>loop body at least once</u>

What is the output when initial value of i is: 5, 0, -5?

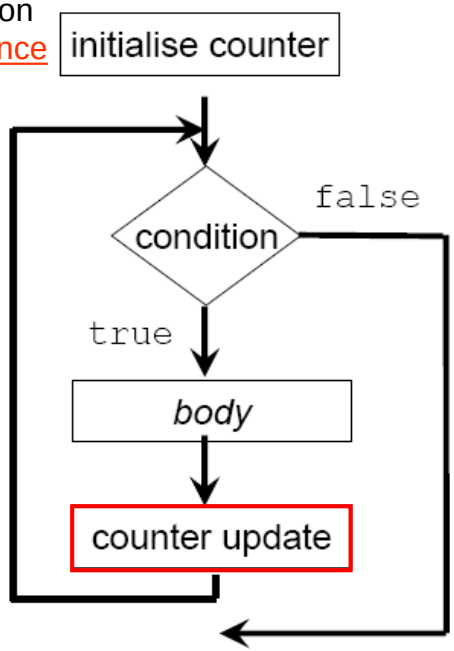
for Loop

```
for (initialization; condition; update) {
    loop body
}
```

Step 0: The **initialization** is executed **only once** before the loop begins
Step 1: If the condition is true, the **loop body** is executed
Step 2: The **update** is executed at the end of each iteration
Step 1 & 2 will be repeated until the condition becomes false

initialization executed only once

Similar to *while* loop, the condition is tested before the execution of the loop body



Equivalent

```
initialization;
while (condition)
{
    loop body;
    update;
}
```

for Loop

for (initialization; condition; update) {
    statement
}

To execute an initialization once, then **while the condition is true** keep executing the loop body and updating an expression.

When the number of iterations is known in advance, the *for* loop structure can be used

```
int sum = 0;
for (int i = 1; i <= 10; i++) //sum=1+2+3+…+10
    sum = sum + i;
System.out.println(sum);
```

Loop body

↔

```
int i = 1;
int sum =0;
while (i <= 10) {
    sum=sum+i;
    i++;
}
```

Scope

- The **scope of a variable** is the section of the program where the variable is defined.
- Generally, the scope of a variable is comprised of the statements that follow the declaration in the same block as its declaration.
- **A variable is visible or accessible only within its scope**
- For instance, in a typical *for* loop, the incrementing variable is not available for use beyond the loop structure

```
for (int i=1; i<=4; i++ )
    System.out.println(i);

System.out.println(i);
```

Loop body & the scope of incrementing variable **i**

//out of scope---error!

A semicolon that shouldn't be there:

```
sum = 0;
for (int i = 1; i <= 10; i++);

    sum = sum + i;

System.out.println(sum);
```

Loop body become empty statement and will do nothing

i is out of the scope and cannot be accessed

Nesting

- With nesting, you can compose loops and conditionals to build programs to solve complex problems.
  - Nest conditionals within conditionals √
  - Nest loops within loops
  - Nest conditionals within loops

Nesting

- Create triangle pattern with nested loops

```
*
* *
* * *
* * * *
```

1. <u>Loop through rows</u>

```
for (int i = 1; i <= n; i++)
{
    // make each specific row
}
```

Put loops together → Nested loops

2. <u>Make a specific row via another loop</u>

```
for (int j = 1; j <= i; j++)
    System.out.print("* ");
System.out.println();
```

```
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= i; j++)
        System.out.print("* ");
    System.out.println();
}
```

Change to a new line

Control Flow Summary

- Control flow.
  - Sequence of statements that are actually executed in a program.
  - Conditionals and loops: enables us to choreograph the control flow.

Control Flow	Description	Examples
Sequential programs	All statements are executed in the order given.	

Conditionals	Certain statements are executed depending on certain condition.	`if`
		`if-else`
		`switch`

Loops	Certain statements are executed repeatedly as long as certain conditions are true.	`while`
		`for`
		`do-while`