Software Engineering and Programming Basics

Control Structures

Authors of slides:
Prof. Dr.-Ing. Janet Siegmund
Prof. Dr.-Ing. Norbert Siegmund
Prof. Christian Lengauer
Partly extracted from script of PD Dr. Christian Bachmaier

Catching Up II

- Methods implement behavior in Java
  - Methods have a **head** and a **body**
    - ReturnType Name ([Parameter, …]) { [Body] }
    - `void:` nothing is returned
    - `return` indicates, what a method returns

- There are special operations, that are defined in Java and which can be applied to variables and constants
  - Arithmetic, logical, relational, assignment
  - Have binding priorities

Catching Up III

- What categories of data types do exist?
  - Primitive
  - Complex
- What are complex data types?
  - User defined classes
  - Composed from other (primitive or complex) data types
- What are primitive data types?
  - Basic types that are provided by Java
  - Charactures, numbers, booleans

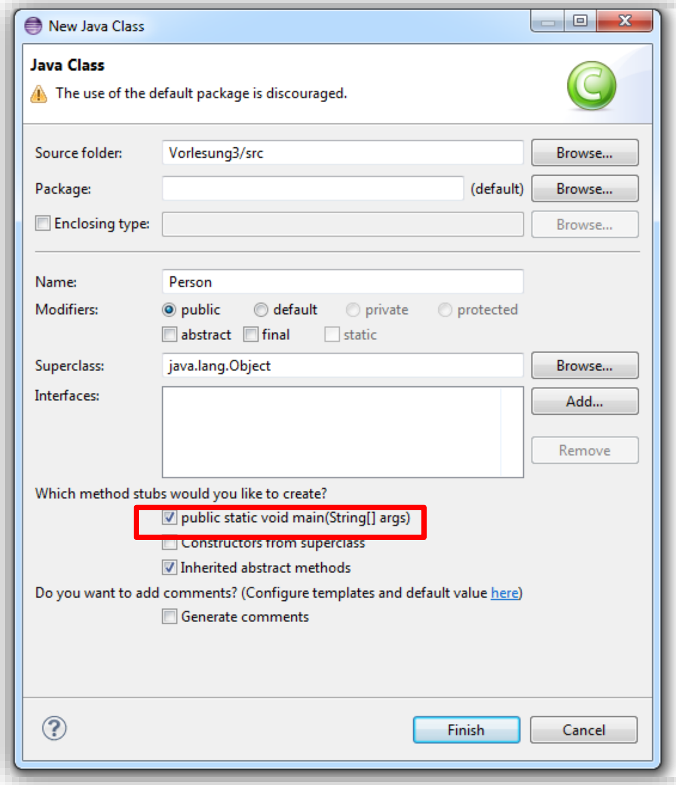
The main method

The „Start“-Method: main I

- Problem: how should Java know, at which point in a program it should start?
- Solution: special method that serves as entry point

```
public static void main(String[] args) {
// TODO Auto-generated method stub

}
```



New Java Class

Java Class

⚠ The use of the default package is discouraged.

Source folder: Vorlesung3/src

Package: (default)

☐ Enclosing type:

Name: Person

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

The „Start“-Method: main II

Visibility:
Can be called from everywhere (we‘ll get to that later)

Static:
The method can be called without creating an oject of this class (more details later)

```
public static void main(String[] args) {
// TODO Auto-generated method stub

}
```

Input paramters:
Array of String that are passed upon a program‘s start time

Return type:
Nothing is returned. Of course, when this metod ends, the program ends.

Identifier (the name of the method):
Java-specified name.

Eclipse Example

- Printing text:

```
System.out.println(…);
```

Hint:

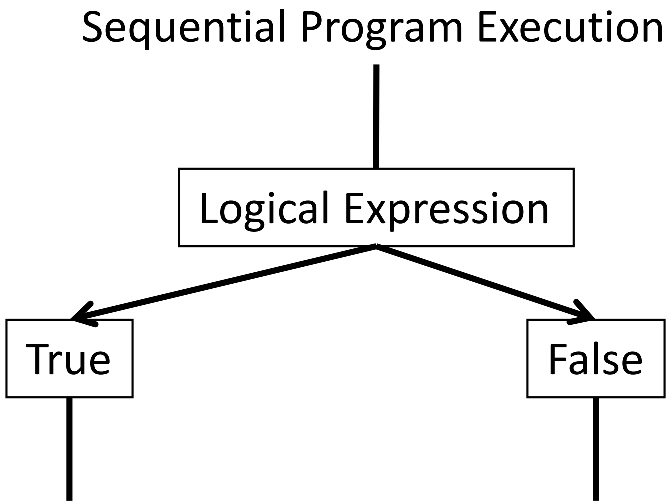
Type „syso“, then CTRL+SPACE to let Eclipse do the rest

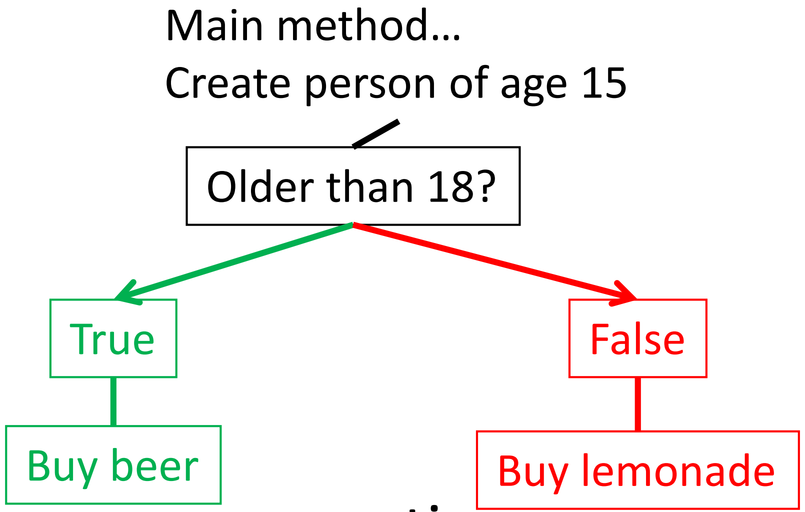
- Input of text:

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

Control Structures

Conditional Branching I





- Branches influence sequential program execution
- Keyword: **`if (logical expression)`**
  - Can be nested
  - **`else`** (i.e., false case) **is optional**
  - Then, a single statement or a bock of statements (written between „{“ and „}“; i.e., they define start and end of a block)

Conditional Branching II

```
public static void main(String[] args) {
    Person p = new Person(“Westley", “Crusher");
    p.age = 18 ;

    if (p.getAge() >= 18)
        System.out.println(“full age!");
    else if (p.getAge() >= 15){
        System.out.println(“not yet full age");
    }
    else {
        System.out.println(“Well, it takes some time.");
        p.isBirthday();
    }
```

Single statement, no curly braces necessary

Optional, but possible

Necessary for more than one statement

Jave needs to know what exactly should be executed in case true and in case false.
Curly braces denote which statements belong to which case.

Pecularity

```
int b = 1;
int c = -1;
if (b == 1)
    if (c > 0)
        System.out.println("c is greater than 0");
    else
        System.out.println("c is smaller than or equal to 0");
```

No braces necessary, because another conditional is following

Watch out for the „dangling else“

```
int i = 1;
if (i <= 0)
    if (i == 0)
        System.out.println("i is zero");
else
    System.out.println(i);
```

⚠ Always use braces!

else is bound to next if; nothing is returned

Selection: switch – case I

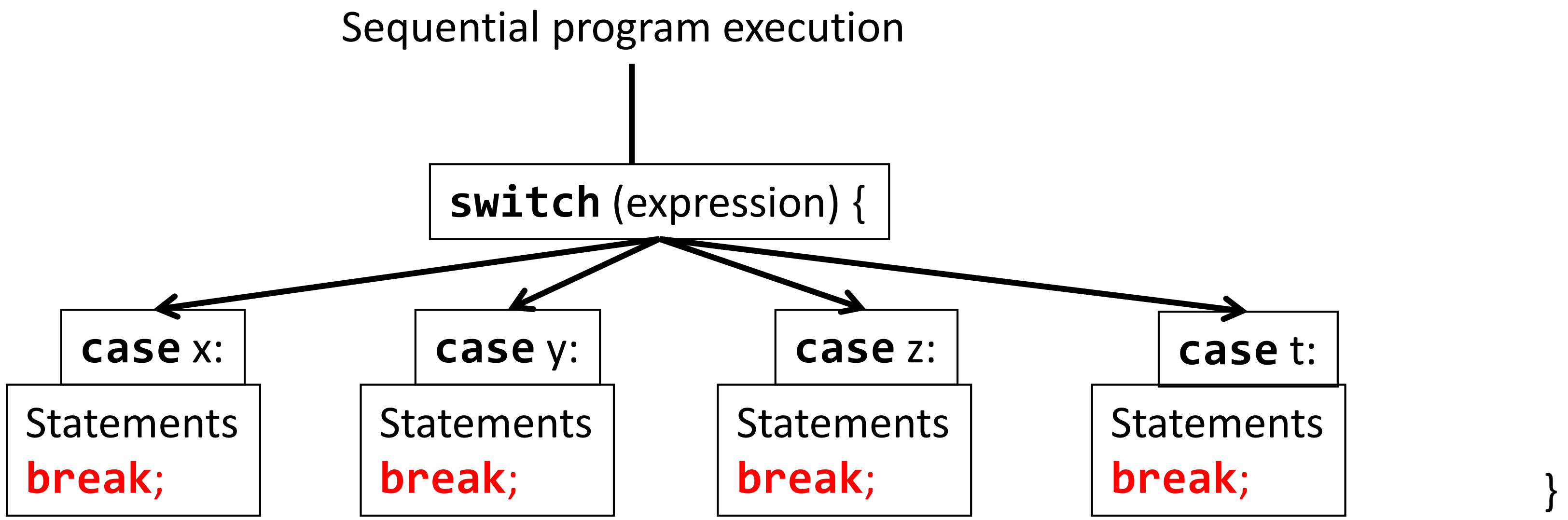
- What do to with lots and lots of if-else statement?

```java
if (a > 20) System.out.println(a);
else if (a > 19) System.out.println(a);
else if (a > 18) System.out.println(a);
else if (a > 17) System.out.println(a);
else if (a > 16) System.out.println(a);
else if (a > 15) System.out.println(a);
else if (a > 14) System.out.println(a);
else if (a > 13) System.out.println(a);
else if (a > 12) System.out.println(a);
else if (a > 11) System.out.println(a);
```

- Simpler alternative: switch – case!

Selection: switch – case II

- Enables multiple branching



- **case** describes constant
- Read: In case variable x == 10 evaluates to true, execute statements until break; for the case that x == 11 evaluates to true, execute statements until break;

Selection: switch – case II

Expressions can also contain computation

```
switch (3 * age) {
    case 9:
        System.out.println("You are 3 years old!");
        break;
    case 12:
        System.out.println("You are 4 years old!");
        break;
    case 15:
        System.out.println("You are 5 years old!"); ");
    case 18:
        System.out.println(" You are 5 or 6 years old!"); ");
        break;
    default:
        System.out.println(„I‘m not sure about your age.");
}
```

Normal case

Without **break**, the following statements will be executed until the next **break**

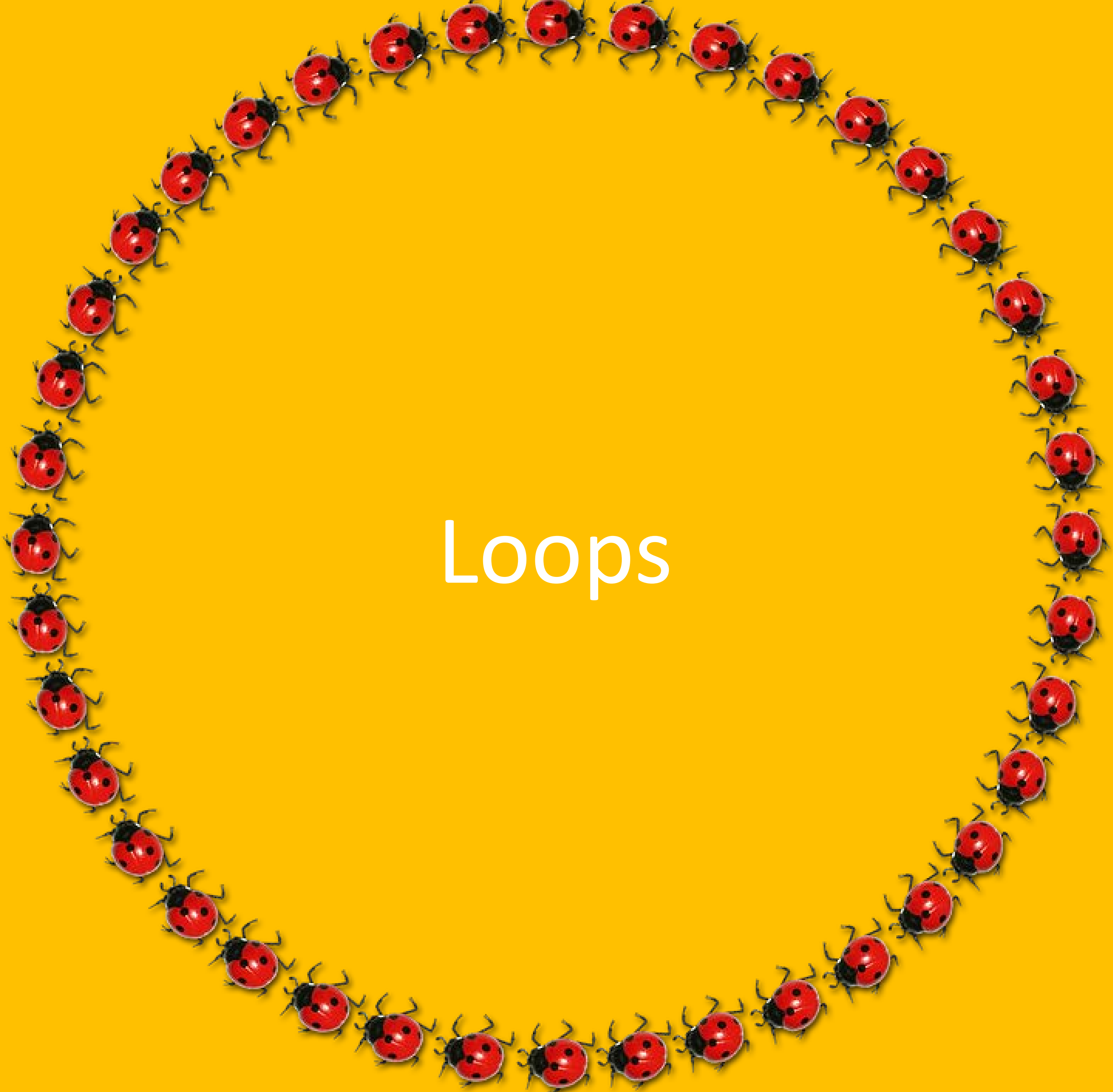
Default: Standard case. Here is everything that does not fulfull the conditions covered by all the **case**s. (e.g.,: 6, 3, 3000, -21).

Selection: switch – case IV

- Only certain expressions can be used with **switch**
- Depends on return type of expression
  - Some primitive data types (**char, byte, short, int, long**)
  - A few complex data types (specified by Java)
    - Character, Byte, Short, Integer, Long, String
  - Enumarations (more details later)
- Case expression needs to be constant

```
final int ONE = 1;
int three = 3;
```

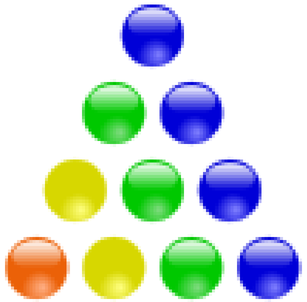
```
switch (a) {
  case ONE:
    System.out.println("One"); // Constant -> ok
    break;
  case ONE + 1:
    System.out.println("Two"); // Constant expression -> ok
    break;
  case three:
    System.out.println("Three"); // Error: No constant expression
    break;}
```



Loops

Loops I

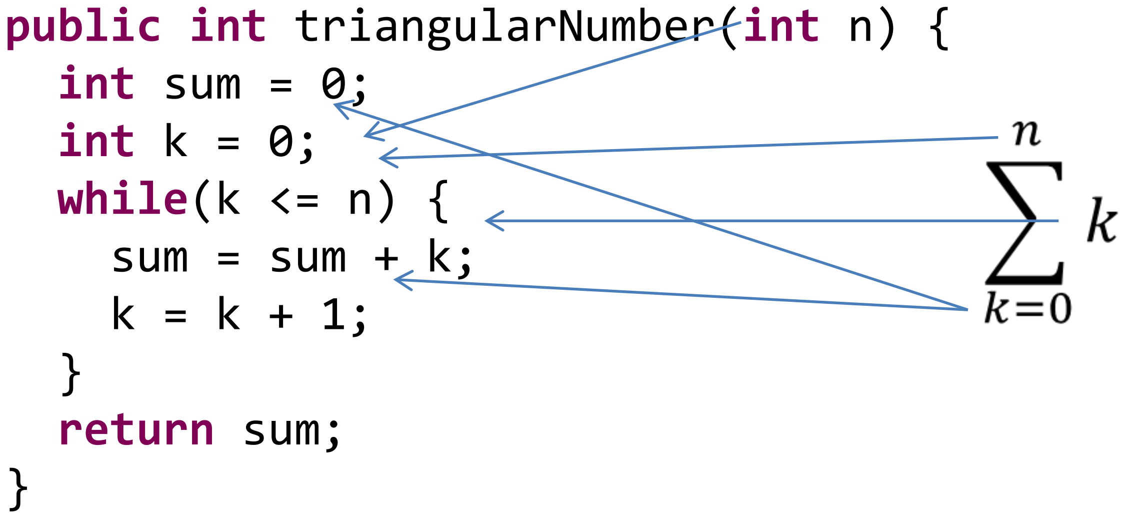
- Frequent problem: executing the same statements multiple times
  - Add a sequence of numbers
  - Play each song of music collection
- Different kinds of loops: while, do-while, for, foreach, etc.
- Example: Triangular number
  - Sum of numbers from 1 to n
  - $\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$



While – Loop

- Rejecting loop (first condition is evaluated, then loop body is executed if condition evaluates to true)

```
while ([logical expression])
{
  [statement]
}
```



```
public int triangularNumber(int n) {
  int sum = 0;
  int k = 0;
  while(k <= n) {
    sum = sum + k;
    k = k + 1;
  }
  return sum;
}
```

$$\sum_{k=0}^{n} k$$

Alternatively:

```
sum += k;
++k;
```

do-while-loop

- Non-rejecting loop (i.e., loop body will be executed at least once; post-test)

„;“ Don‘t forget!

```
do {
  [statement]
} while ([logical expression]);
```

```
public int triangularNumber2(int n) {
  int sum = 0;
  int k = 0;
  do {
    sum = sum + k;
    k = k + 1;
  } while(k <= n);
  return sum;
}
```

```
while(k <= n) {
    sum = sum + k;
    k = k + 1;
}
```

What Does the Following Code Print?

```
public void questionInLecture()
{
  int line = 1;
  while (line <= 5) {
    int star = 1;
    while (star <= 2 * line) {
      System.out.print("*");
      ++star;
    }
    System.out.println();
    ++line;
  }
}
```

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

```
System.out.print() – Print to console
System.out.println() – Print to console and make new line
```

Hints

- Termination: avoid infinite loops

```
while(true);
```

- Safe Stop condition:
  - Take care of overflow (e.g., values > 9)

```
public int unsafeAbort(int value)
{
  while(value != 9)
    ++value;
  return value;
}
```

better: →

```
public int safeAbort(int value)
{
  while(value < 9)
    {++value;}
  return value;
}
```

  - Consider possible inaccuracies in computation

```
public static void loop() {
  double d = 0.0;
  while (d != 1.0) {
    d += 0.1;}}
```

better: →

```
public void loop() {
  double d = 0.0;
  while (d >= 0.99 || d <= 1.01) {
    d += 0.1;}}
```

Counting Loop: for

- Counts ([Start] ; [END] ; [DELTA COUNTER]) { [Statements] }
  - `[Initialization]` is done once
  - Until `[logical expression]` evaluates to false, `[statements]` are executed
  - **After** each execution `[assignment]` is done

```
for ([Initialization]; [logical expression]; [assignment]) {
  [statements];
}
```

```
public int triangularNumber3(int n) {
  int sum = 0;
  for (int k = 0; k <= n; ++k) {
    sum = sum + k; //alternative: sum += k;

  }
  return sum;
}
```

```
while(k <= n) {
    sum = sum + k;
    k = k + 1;
}
```

$$\sum_{k=0}^{n} k$$

Hints

- A variable should be defined in head of loop and is then valid **only in the loop body** (not outside of loop)

```java
public int triangularNumber3(int n) {
  int sum = 0;
  for (int k = 0; k <= n; ++k) {
    sum = sum + k; //sum += k;
  }
  k++; //Error, k does not exist anymore!
  return sum;
}
```

- The variable initialized in the head can be used in the loop body (but you should not do this)

Relationship Between for and while

- Translate the for loop to a while loop

```
public int triangularNumber3(int n) {
  int sum = 0;
  for (int k = 0; k <= n; ++k) {
    sum = sum + k; //sum += k;
  }
  return sum;
}
```

```
public int triangularNumber(int n) {
  int sum = 0;
  int k = 0;
  while(k <= n) {
    sum = sum + k;
    k = k + 1;
  }
  return sum;
}
```

3 to 5 minutes

```
for ([Initialization]; [Logical expression]; [Assignment]) {
  [Statement];
}

[Initialization];
while ([logical expression]) {
  [Statement];
  [Assignment];
}
```

Don't forget to avoid inifite loops!

When `for`, when `while`?

- Rule of thumb: **`for`** only for clean counting loops
- Some informal critiera to use **`for`:**
  - Terminating condition is known before first iteration
  - Counting with a counting variable
  - All three expressions in loop head refer to the same variable
  - Assignments to counting variables do not appear in the loop body
  - Termination is easy to garanty, or even better…
  - … number of iterations (i.e., how often the loop is executed) is known before

„Go To“ Statements

- Premature termination with **break;**

```
boolean run = true;
while(run) {
  char option = readOption();
  if(option == 'q') {//quit
    break;
}}
```

- Execute next loop iteration with **continue;**

```
for (int i = 0; i <= 10; ++i) {
  if (i % 2 == 1) {
    continue;
  }
  System.out.println("Number: " + i);
}
```

- With nested loops, always the local loop is referred to

Scope I

- Until now: curly braces to indicate where statements belong to
  - **if** ([logical expression])

    { [Statements in case **true**] }

    else { [Statements in case **false**] }
  - Loops: **while** (…) { [Statements in loop] }
  - Method: **void** triangularNumber () { [Statements in Method] }
  - Klasse: **class** myClass { [Attributes, methods in class] }

- New: curly braces show how long variables are valid

Scope II

- Variables are valid only after their declaration in the block in which they are defined (and their sub blocks)

```
int i = 1;
if(i < 5){
  int j = 3;
  i++;
  System.out.println(i + j); // 5
}
System.out.println(i); // 2
System.out.println(j); // Error: j is unknown here, is declared in inner block!
```

- Variables with same name are not allowed in sub blocks (except class-/instance variables)

```
int i = 1;
if(i < 5){
    double i = 1;
    System.out.println(i); // unclear, which i is meant
```

Quizz!!!

- Each loop contains two errors. Try finding and fixing them!:

```
public void errorLoops(char k) {
  for(int i, i < 5; i++) {
    System.out.println(i);
  }

  boolean run = true;
  int x = 0;
  while(k){
    x++;
    if(x > 5)
      continue;
  }

  int counter;
  do {
    int counter = 10;
    counter = counter - 2;
  } while(counter > 0);
}
```

Semicolon instead of comma

i is not initialized. Fix: i = 0

No logical expression. Fix: while(run)

Infinite loop! Fix: break instead of continue

Scope-Error: counter declared twice

Infinite loop! counter is set to 10 again and again

Learning Goals

- Loops allow executing the same statements multiple times:
- `while` und `for`
  - Initialization of variables to evaluate terminating condition
  - Logical expression is checked at each iteration
  - Possibly change in counting variable

- Scope „{ … }“ defines where variables are valid

- Conditional branching with:
  `if` ([logical expression]) { [true] } + optional `else` { [false] }
- The issues with the case statements

Coming Up Next

- Object-oriented programming
- Behavior of objects (not classes!)
- How do I created objects of classes? -> With the constructor