320341 Programming in Java

JACOBS
UNIVERSITY

Fall Semester 2015

Lecture 2: Fundamental Structures

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Outline

- Program structure
- Data types
- Variables
- Constants
- Operators
- Flow Control

Objectives

The objective of this lecture is to:

- Introduce fundamental programming structures in Java

Java Basics

Java is case sensitive

By convention all Java classes are nouns that begin with a capital letter with the first letter of each word capitalized. This is called **CamelCase**.

Example: **`SampleClassName.`**

A Java class name is an identifier that:

- consists of *letters*, *digits*, *underscores (_), dollar signs ($)*
- *does not begin with a digit*
- *does not contain empty spaces*
- *has no limit on length*
- *Is not a reserved word*

Examples of valid identifiers: Welcome1, $value, _value, m_inputField1

Example

```
// Text-printing program

public class Welcome1  {
   // main method begins execution of Java application
   public static void main( String args[] )  {
      System.out.println( "Welcome to Java in Programming!" );

   } // end method main

} // end class Welcome1
```

"//" means *single-line* comment

Source file name is name of public class with extension .java

Every statement ends with semi-colon

Comments

Three ways of commenting

- // runs from // to end of line

```
// Text-printing program
public class Welcome1 { …
}
```

- /* ….. */ multiple line comments

```
/* Text-printing program
   Date: 06 September, 2012
*/
public class Welcome1 { …
}
```

- /** ….. */
- Generate documentation automatically

```
/** Text-printing program
  * Date: 06 September, 2012
  */
public class Welcome1 { …
}
```

Data Types

Java is a strongly typed language

- Each variable must have its type be declared before use
- Example:

```
double salary;
int vacationDays;
long population;
boolean done;
```

There are eight (8) primitive types in Java (*4 integer types*, *2 float types*, *1 char*, *1 boolean*)

In Java the sizes of all numeric types are platform-independent

- In C/C++ numeric type sizes are platform dependent

There are no `unsigned` types in Java

Data Types: boolean and char

JACOBS UNIVERSITY

boolean:

- Have only two possible values : `true` or `false`
- Used to evaluate logical conditions
- Size not precisely defined

char:

- *Single 16-bit Unicode character*
- Min value: `'\u0000'`; max value is `'\uffff'` (65, 535 inclusive)
- Use single quotes to represent character constants e.g., 'A'

Data Types: integer types

byte:

- *8-bit signed two's complement integer*
- Minimum value: `-128` and maximum value is `127` (inclusive)

short:

- *16-bit signed two's complement integer*
- Minimum value: `-32,768` and maximum value is `32,767` (inclusive)

Data Types: integer types

JACOBS UNIVERSITY

int:

- *32-bit signed two's complement integer*
- Min value: `-2,147,483,648` ; max value is `2,147,483,647` (inclusive)
- Usually the default type for storing integer numbers

long:

- *64-bit signed two's complement integer*
- Use if the range of values to be stored is wider than that provided by `int`
- Use suffix `L` to define `long` constants e.g., `100000L` will be stored as `long`

Data Types: floating-point types

JACOBS UNIVERSITY

float:

- *Single precision 32-bit IEEE 754 floating point*
- Use `float` (instead of `double`) to save memory in large arrays of floating-point numbers
- Has limited precision, thus may not be sufficient for some applications
- Use suffix `F` on constants to store them as float type (e.g., `3.14F`)

double:

- *Double precision 64-bit IEEE 754 floating point*
- Use `float` (instead of `double`) to save memory in large arrays of floating-point numbers
- Floating-point numbers without suffix `F` default to double type

Data Types: summary

JACOBS UNIVERSITY

- Primitive data types are held on the stack, thus efficiently processed

Primitive Type	Size	Minimum	Maximum	Wrapper
boolean	-	-	-	Boolean
char	16-bits	Unicode 0	Unicode 2^{16}-1	Character
byte	8-bits	-128	+127	Byte
short	16-bits	-2^{15}	$+2^{15}$-1	Short
int	32-bits	-2^{31}	$+2^{31}$-1	Integer
long	64-bits	-2^{63}	$+2^{63}$-1	Long
float	32-bits	IEEE754	IEEE754	Float
double	64-bits	IEEE754	IEEE754	Double

Data Types

Two classes for performing high-precision arithmetic

- import package `java.math`

- `BigInteger`: for arbitrary precision integer arithmetic

- `BigDecimal`: for arbitrary-precision floating-point arithmetic

Example

```
// c = a + b
BigInteger c = a.add(b);

// d = c*(b+2)
BigInteger d =
    c.multiply(b.add(BigInteger.valueOf(2)));
```

Java Basics: Variables

JACOBS UNIVERSITY

Instance variable

- None static class fields to store object's state
- Their values are unique to each class instance or object

Class variable

- Static class fields to store object's state – only one copy of variable exists for entire class
- Must be declare with the static key word

Local variable

- Used by a method to store the temporary state
- Declared inside a method and accessible only inside the method

Parameters

- Used to pass values in methods
- Treated as local variables to the method they pass values to

Java Basics: Variables

Variable names are case sensitive

Each variable must have a type in Java

- Declaration syntax
  - *type variableName;*

Example:

```
double salary;
int vacationDays;
long population;
boolean done;
```

Java Basics: Variable Names

Variable names are case sensitive

A variable name *must begin with a letter, a dollar sign ($)* or underscore (_) and *must be a sequence of letters or digits*

A letter in Java is: `'A'-'Z', 'a'-'z', '_'` or *any* `Unicode character` *that denotes a letter in a language*

Special symbols e.g., '+, -, %, ...', are not allowed

Variable name are *length unlimited*

Reserved words are not allowed as variable names

By convention variable names always start with a letter, not underscore or dollar sign

Java Basics: Choosing Variable Names

JACOBS UNIVERSITY

Choose full words instead of cryptic abbreviations

If one word is used, use all small letters, like:

```
double salary; // use full words all in small letters
```

If more than one word is used, use camelCase, like:

```
int vacationDays; // use camelCase if more than one word is used
```

When declaring constant values, capitalize every letter and separate words using underscore like:

```
static final int DAYS_OF_WEEK = 7; // declaring constant variable
```

Java Basics: Initializing Variables

JACOBS UNIVERSITY

Variables must be properly initialized

- The code below results in a compile-time error since *vacationDays* was not initialized:

```
int vacationDays; // assume this is a local variable
System.out.println("vacationDays");
```

- Initialize the variable by assigning value:

```
int vacationDays = 12;
System.out.println("vacationDays");
```

- Declaration can be put anywhere in your code

```
double salary = 65000.0;
System.out.println("vacationDays");
int vacationDays = 12; // ok to declare variable here
```

Constants

Use keyword `final` to denote a constant

```
public class Constants {
  public static void main(String[] args) {
     final double PI= 3.14; // use all capital for constants declaration
     float diameter = 5;
     System.out.println(„Circumference is „+ PI*diameter);
  }
}
```

- Constant variables' values are immutable
- It is customary to name constants in all upper case
- It is also customary to make constants classwide – class constants
  - Declare as `static final`

```
public class Constants  {
     static final double PI = 3.14;
.....}
```

Operators

JACOBS UNIVERSITY

Arithmetic Operators

Operator	Description	Example
+	Addition operator	10+2 =12
-	Subtraction operator	10-2 =8
%	Integer remainder (modulus) operator	15%2 = 1
*	Multiplication operator	10*2 =20
/	Integer division if both arguments are integers, float-point division otherwise	15/2 =7 15.0/2 =7.5

Short-cuts

Assignment Expression	Equivalent Representation
x = x+2	x += 2
x = x-2	x -= 2
x = x*2	x *= 2
x = x/2	x /= 2
x = x % 2	x %= 2

Operators

JACOBS UNIVERSITY

Increment/ Decrement Operators

Operator	Description	Example
++	increment by one unit	int n = 12; n++; // changes n to 13
--	decrement by one unit	int n = 12; n--; // changes n to 11

- Applied only to variables, not constants e.g., 4++ is illegal
- There is also the prefix form of operators
- Difference appears when used in expressions

```
int  m = 7;
int  n = 7;
int  a = 2 * ++m; // now a is 16, m is 8
int  b = 2 * n++; // now b is 14, n is 8
```

Operators

JACOBS UNIVERSITY

Relational Operators

Op	Description	Example
==	Equivalent (equal to)	3 == 7 is false
!=	Not Equivalent (not equal to)	3 != 7 is true
<	Less Than	3 < 7 is true
>	Greater Than	3 > 7 is false
<=	Less or Equal to	3 <= 7 is true
>=	Greater or Equal to	3 >= 7 is false

Operators

JACOBS UNIVERSITY

Logical Operators

Op	Description	Example
`&&`	Logical AND	`(3<7) && (7<10) is true`
`||`	Logical OR	`(3 != 7) || (3 >7) is true`
`!`	Logical NEGATION	`!(3 < 7) is false`

&& and || are evaluated in a short “circuit fashion”

- The second argument is not evaluated if the result is already determined by the first argument.

Operators

Ternary (`?:`) Operator

- The ternary operator has the following syntax:

$$condition\ ?\ exp_1\ :\ exp_2$$

- Evaluates to exp_1 if *condition* is true; evaluates to exp_2 otherwise
- Example:

```
x < y ? x : y; // gives the smaller value of x and y
```

Operators

JACOBS UNIVERSITY

Bitwise Operators

- Allow the manipulation of individual bits in integral primitive data types
- Allow to perform boolean algebra on corresponding bits

Operator	Description	Example
`&`	Bitwise AND	`1111 & 1101 is 1101`
`|`	Bitwise OR	`1111 | 1101 is 1111`
`^`	Bitwise EXCLUSIVE OR	`1111 ^ 1101 is 0010`
`~`	Bitwise NOT (ONE's COMPLEMENT)	`~1101 is 0010`
`<<`	LEFT SHIFT	`1111 << 1 is 11110`
`>>`	RIGHT SHIFT	`1111 >> 1 is 0111`
`>>>`	UNSIGNED RIGHT SHIFT	`1111 >>> 1 is 0111`

Mathematical Functions and Constants

Function	Meaning	Example
`Math.sqrt(x)`	Square root	`double y = Math.sqrt(4.0);` // y = 2.0
`Math.pow(x,a)`	Power	`double y = Math.pow(4.0, 2.0);` // $4.0^{2.0}$ = 16.0
`Math.sin(x)`	Sine	`double y = Math.sin(45.0);` // y = 0.8509035245341184
`Math.cos(x)`	Cosine	`double y = Math.cos(30.0);` // y = 0.1542514498875840 5
`Math.tan(x)`	Tangent	`double y = Math.tan(45.0);` // y = 1.5485777614681775
`Math.E`	Approx to E	2.718281828459045 // Constant value

Mathematical Functions and Constants

JACOBS UNIVERSITY

Function	Meaning?	Example
`Math.atan(x)`	Arc tangent	`double y = Math.atan(45.0);` // y = 1.5485777614681775
`Math.exp(x)`	Exponent	`double y = Math.exp(2.0);` // y = 7.38905609893065
`Math.log(x)`	Natural logarithm	`double y = Math.log(4.0);` // y = 1.3862943611198906
`Math.log10(x)`	Decimal logarithm	`double y = Math.log10(100.0);` // y = 2.0
`Math.PI`	Approx to PI	3.141592653589793 // Constant value

Mathematical Functions and Constants

Avoid the **Math** prefix by adding import line:

```
import static java.lang.Math.*;
```

Example:

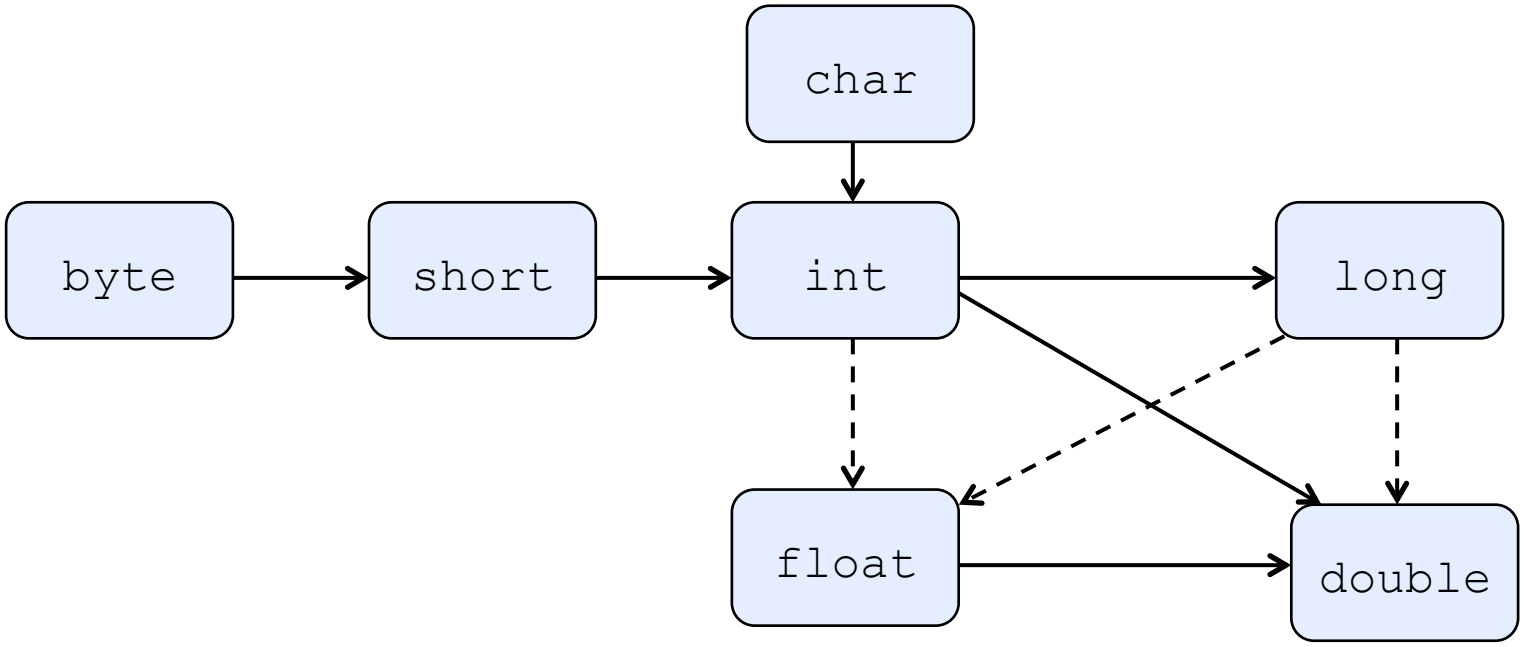
```
import static java.lang.Math.*;


System.out.println(„The square root of \u03C0 is  " +  sqrt(PI));
```

Conversions between Numeric Types

Solid line ⇨ no loss in precision

Dotted line ⇨ possible loss in precision



- Explain the loss in precision when converting from `long` to `double` yet both types use 64 bits?
- Explain also the loss in precision when converting from `int` to `float`

Conversions between Numeric Types

The following conversions take place when evaluating expressions:

- If either operand is `double`, the other is also converted to `double`
- Otherwise if either operand is `float`, the other is also converted to `float`
- Otherwise if either operand is `long`, the other is also converted to `long`
- Otherwise if both operands will be converted to `int`

Conversions between Numeric Types

JACOBS UNIVERSITY

Casting

- Forces one data type to another, but information may be lost
- Example

```
double x = 9.997;
int nx = (int) x; // nx is 9. 0.997 is lost
```

- nx has value 9, the fraction is discarded

Round

- Use `Math.round()` to round a floating point number to nearest integer
- Example

```
double x = 9.997;
int nx = (int) Math.round(x); // nx is 10.
```

Parenthesis and Operator Precedence

Operators	Associativity
[] . () method call	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instance of	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

Enumerated Types

JACOBS UNIVERSITY

Contain a finite number of named values

Example:

```
enum Size {SMALL, MEDIM, LARGE, EXTRA_LARGE};
```

Declare variables of this type and assign values

```
Size s = Size.MEDIUM;
```

A variable of type `Size` holds only one of the listed values or special value `null` indicating that the variable is not set

Input and Output

Use the **`Scanner`** (`java.util.`**`Scanner`**) class for reading data

- First, import `java.util.`**`Scanner`**` as follows`

```
import java.util.Scanner;
```

- Next, construct a `Scanner` attached the “standard input stream” as follows:

```
Scanner in = new Scanner(System.in);
```

The following methods from the `Scanner` class are used to read data:

Scanner Method	Description
`next()`	Reads single word delimited by white space
`nextLine()`	Reads a line of input
`next[PrimitiveType](), except char`	`nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), nextBoolean()`

Example

```
Scanner in = new Scanner(System.in);
System.out.println(„What is your name? “);
String name = in.nextLine(); // reads line of input
```

```
String firstname = in.next(); // reads a single word
System.out.println(„How old are you? “);
int age = in.nextInt(); // reads an integer value
```

```
String firstname = in.next(); // reads a single word
```

```
System.out.println(„How old are you? “);
int age = in.nextInt(); // reads an integer value
```

Example

JACOBS UNIVERSITY

```
import java.util.Scanner;
class DataInTest {
  public static void main (String [] args) {
      Scanner in = new Scanner(System.in); // Create a Scanner attached
      System.out.println ("Enter your first name: " );
      String firstName = in.nextLine(); // read entire line

      System.out.println ("Enter your last name: " );
      String lastName = in.nextLine(); // read entire line;

      System.out.println ("Enter your age: " );
      int age = in.nextInt(); // reads an int value

      System.out.println ("Hallo " +  firstName + " " + lastName + ".");
      System.out.println("You are "+ age + " years old now.");
  }
}
```

Formatting output

We can format our output using format specifiers & conversion characters

- Formatted `System.out.printf()`

```
double x = 10000.0/3.0;
System.out.print(x);  // prints 3333.3333333333335
System.out.printf("%8.2f",x); // prints 3333.33
```

prints x with field width of 8 and precision of 2 characters

- We can supply multiple parameters to `printf`

Formatting output

JACOBS UNIVERSITY

Commonly used conversion characters

Conversion character	Type	Example
`d`	Decimal integer	`120`
`x`	Hexadecimal integer	`78`
`o`	Octal integer	`170`
`f`	Fixed point floating point	`120.00`
`e`	Exponential floating point	`1.200000e+02`
`s`	String	`Hello World`
`c`	Character	`A`

Formatting output

JACOBS UNIVERSITY

Commonly used Flags

Flag	Purpose	Example
+	Prints positive/ negative numbers	+3333.33
0	Adds leading zero	003333.33
-	Left justified	|3333.33 |
,	Adds group separators	3,333.33
...	...	...

Formatting output

JACOBS UNIVERSITY

Common date conversions

Conversion	Type	Example
`c`	Complete date and time	`Mon Feb 09 18:30:45 PST 2004`
`F`	ISO 8601 date	`2004-02-09`
`D`	US formatted date (mm/dd/yy)	`02/09/2004`
`T`	24-hours time	`18:05:19`
`R`	`24`-hour time, no seconds	`18:05`
`Y`	Four digit year	`2004`
`Z`	Time zone	`PST`
…	…	…

Sequence Statements

Statements are executed in the order in which they are given

Block /Compound Statement

- Simple statements surrounded by a pair of braces
- Define the scope of variables
- Can be nested inside other blocks
- Example

Nested Block

```
public static void main(String [] args)  {
  int n;
  …
    {
        int k;
    } // k is only defined upto here
}
```

Selection Statements

if (condition) statement

- Executes one or more statements if a certain condition is true
- Example

```
if (sales >= target) {
        performance = "Satisfactory " ;
        bonus = 100;
                                                }
```

The statements inside the block are executed only if the condition evaluates to `true`; otherwise the next statement is executed

Selection Statements

`if (condition) statement`$_1$ `else statement`$_2$

- Executes statement$_1$ if *condition* is true, statement$_2$ otherwise

```
if (sales >= target)
{
      performance = "Satisfactory";
      bonus = 100 + 0.01 * (sales -
target);
}
else {
      performance = "Unsatisfactory";
      bonus =
0;
}
```

The statements in the first block (the `if` part) are executed if the condition is `true`; the second block (`else` part) is executed otherwise

Selection Statements

JACOBS UNIVERSITY

Multiple Branches

An `else` groups with the closest `if`

- Example

```
if (condition1)              // first if statement
    ……
if (condition2)              // second if statement

    ……
if (condition3)             // third if statement
    ……
else
     ……
```

- The `else` belongs to the third `if`

Selection Statements

Multiple branches are common

An `else` groups with the closest `if`

- Example

```
if (condition1)          // first if statement
    ……
else if (condition2)     // second if statement

    ……
else if (condition3)     // third if statement
    ……
else if (condition4)     // fourth if statement
    ……
else
    ……
```

- Only one branch can be followed

Selection Statements

JACOBS UNIVERSITY

Example

```
if (sales >= 2 * target) {
   performance = "Excellent";
   bonus = 1000;
} else if (sales >= 1.5 * target) {
   performance = "Fine";
   bonus = 500;
} else if (sales >= target) {
   performance = "Satisfactory";
   bonus = 100;
}
```

Multiple Selections

The `switch` statement is used for multiple selections

- Provides implementation for multi-way selection based on integral expression

```
switch (selector) {
    case selector-value1: statement;

    break;

    case selector-value2: statement;

    break;
    case selector-value3: statement;
    break;
    case selector-value4: statement;
    break;
    default: statement;

}
```

Multiple Selections

JACOBS UNIVERSITY

A case selector can be:

- An expression of type `char, byte, short, int` or their wrapper classes
- An enumerated constant
- A `String` literal (as of Java SE 7)

The selection is made according to the following rules:

- Results of the `selector` are compared to each `selector-value`
- If a match is found, the corresponding statement is executed
- If no match is found the `default` statement executes
- The `break` statement causes execution to jump to the end of the `switch` body

Example

Selects an option from four possible options

```
Scanner in = new Scanner(System.in);
System.out.println("Select an option (1, 2, 3 or 4)");
                                                    int
choice = in.nextInt();

switch (choice) {
    case 1:  menu 1 statement; break;
    case 2:  menu 2 statement; break;
    case 3:  menu 3 statement; break;
    case 4:  menu 4 statement; break;
    default: default statement;

}
```

Iteration (Loops)

The following looping control flow constructs will be presented:

- `while (condition) statement`

- `do statement while (condition)`

- `for (initialValues; condition; incrementValues) statement`

- `for (variable: collection) statement`

- `break statement`

- `continue statement`

Iteration – (`while`) Statement

Executes the statement /statement block as long as the condition is true

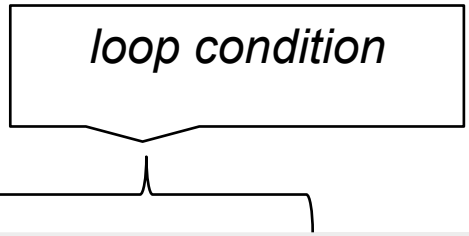
The general form is:

```
while (condition) {
    statement1;
                                        . . .
 statementn;
}
```

- The statements are repeated until the *condition* evaluates to `false`
- The `while` loop will never execute if the condition is false on the outset

Example

The loop is executed as long as `balance < goal` is `true`




```
while (balance < goal) {
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   years++;
}

System.out.println(years + " years.");
```

- The condition says “keep doing this loop until `balance < goal` is false

Iteration – (do- while) Statement

A `while` loop tests at the top; the code block may never be executed

The `do - while` guarantees that the loop is executed at least once

The general form is:

```
do  {
    statement1;
                                        . . .

 statementn;

}  while (condition);
```

- The loop executes the code block and only then tests the condition
- It then repeats the statement and retests the condition, and so on.

Example

Code segment to calculate retirement

```
do {
    balance += payment;
    double interest = balance * interestRate / 100;

    balance += interest;
    year++;
    // print current balance
                                        . . .
                                  // ask if ready to retire
and get input
                                . . .
                                     }  while
(input.equals(" N "));
```

- The `do – while` guarantees that the loop is executed at least once
- The loop is repeated as long as the user types “N” as input

Iteration (`for` loop)

JACOBS UNIVERSITY

An example of a determinate loop

The iteration is controlled by a *counter* which is updated in each iteration

1. The counter is initialized before each iteration
2. A condition is tested to determine if code should be executed
3. The counter is updated

Iteration (`for` loop)

The general form for the `for` loop is:

```
for (initialization; condition; step) {
    statement1;
                                        . . .
 statementn;
}
```

- Any of the statements (*initialization*, *condition*, and *step*) can be empty
- Note that any expression is allowed at the various slots of the `for` loop

Example

Simple `for` loop that prints numbers from 1 upto 10

```
for (int i = 1; i <= 10; i++)
    System.out.println(i);
```

Simple `for` loop that prints numbers from 10 downto 1

```
for (int i = 10; i > 0; i--)
                    System.out.println("counting
down . . . " +i); System.out.println("Blastoff!
");
```

Example

Defining multiple variables within a `for` loop

```
for (int i = 0, j = 1; i < 10 && j != 11; i++, j++)
          System.out.println("i = " +i+ ", j= " +j);
```

- Multiple variables defined within the `for` loop must be of same type

Testing Equality of Floats

```
for (double x = 0, x != 10; x +=0.1)
   System.out.println("x = " +x );
```

- Loop may never end due to round-off errors (no exact binary representation for 0.1)

Iteration (`for` loop)

JACOBS UNIVERSITY

Variable Scope

- Variables declared inside a for loop slot are visible until end of loop body

```
for (int i = 1; i <= 10; i++)
   System.out.println(i);
// i no longer defined here
```

- For example, *i* is not visible outside the loop

```
int i;
for (i = 1; i <= 10; i++)
   System.out.println(i);
// i still defined here
```

Iteration (`for each` loop)

Introduced in JDK 5.0

Allows to loop through elements of collections (arrays, sets, etc)

General Form

```
for (variable: collection) {
    statement1;
                                    . . .
 statementn;

}
```

- Sets the variable to each element of the collection, then executes code block
- Collection expression must be an array or object of a Collection class (e.g., Set)

Example

Defining multiple variables within a for loop

```
for (int element: a)
    System.out.println(element);
```

- Prints each element of the array *a* on a separate line
- The loop is read as “for each *element* in *a*”
- The loop traverses elements of the array, not the index

break

The `break` statement is used to break-out of a loop

- Example

```
while (years <= 100) {
    balance += payment;
    double interest = balance * interestRate/ 100;
    balance += interest;
    if (balance >= goal) break;
    years++;

}
```

- The loop is exited if the loop condition is true or if *balance >= goal* at the middle of the loop

Labeled `break`

- Allows to break from nested loops

```
label1:
outer-iteration
{
    inner-iteration {
                                        // . . .
                                        break
label1;
                                }
}
```

continue

Transfers program control to header of innermost enclosing loop

- Example

```
    Scanner in = new Scanner(System.in);

    while (sum < goal) {
        System.out.println(" Enter a number: ");

        n  = in.nextInt();
        if (n < 0) continue;
        sum += n;  // not executed if n <
0                                          }
```

- There is also a labeled `continue` statement much similar to labeled break

References

Cay S. Horstmann and Gary Cornell, Core Java(TM) 2. Vol. I. 9th Ed. Prentice Hall, 9th Edition. 2013. Chapters 3.