Teach Yourself Java in 21 Minutes

Department of Computer Science, Lund Institute of Technology

Author: **Patrik Persson** Contact: klas@cs.lth.se

This is a brief tutorial in Java for you who already know another object-oriented language, such as Simula or C++. The tutorial is organized as a number of examples showing the details of Java. The intention is to give you enough information about the Java language to be able to follow the course in real-time programming.

There are books claiming to teach you Java in 21 days, but since you already know object-orientation your learning time will probably be closer to 21 minutes – hence the title.

This document may be freely copied and distributed for non-commercial use. In case of changes or enhancements, *title*, *department*, *author*, and *contact* must be preserved, and changes that are redistributed or made public must be mailed to the contact above.

Table of contents

What is Java, and why?

The Java programming language was developed at Sun Microsystems and originally became popular as a language for Internet applications (applets). Such applets are embedded within WWW pages and executed in the user's browser. A special format called *byte code* is used instead of ordinary machine code, and by using a special Java interpreter program that code can be executed on any computer. Such an interpreter is called a Java Virtual Machine (JVM) and is available for most modern computer systems.

(There is nothing about the Java language itself that enforces the byte code technique – there are actually some compilers who generate real machine code, known as native code, directly.)

The Java language is not limited to Internet applications. It is a complete general object-oriented language and can be used to develop many kinds of applications. Although the syntax of Java is very similar to that of C++, many complicated and error-prone features of C++ have been removed. The result can be described as a Simula with C++ syntax.

Sun Microsystems (who created the Java language) provide free tools for developing Java software. The Java home page `<http://java.sun.com>` has links to Java compilers for most computer systems (such as Unix and Microsoft Windows), as well as a lot of documentation. It is possible to download a Java compiler and use it for free.

Compiling and running Java programs

In Java, every source file usually contains exactly one class. The file must have the same name as the class; a class named `TurtleMaze` would be stored in the source file `TurtleMaze.java`. This source file can then be compiled using the `javac` compiler:

```
% javac TurtleMaze.java
```

The output of the compiler is a file with the same name as the source file, but with the extension `.class` instead of `.java` (i.e., `TurtleMaze.class` in the above example). That class file contains the byte code mentioned earlier, so it cannot be executed right away. Instead it is executed using the JVM (byte code interpreter) as follows:

```
% java TurtleMaze
```

This command loads the `TurtleMaze` class and executes its `main` method (that is, starts the program). If the `TurtleMaze` class in turn uses other classes, these are loaded automatically when needed.

Since every class should be in its own file, several files can need to be recompiled at the same time. The `javac` compiler has a special option `-depend` to compile all files that depend on a particular file. The command

```
% javac -depend TurtleMaze.java
```

will compile not only `TurtleMaze.java`, but also all changed files it depends upon.

Finding out more about Java

Many details of the Java language have been left out in this tutorial. If you want to know more about the Java programming language, refer to one of the following sources:

- Per Holm: *Objektorienterad programmering och Java*. Studentlitteratur, 1998.
- Mary Campione and Kathy Walrath: *The Java Tutorial* (second edition). Addison-Wesley, 1998. Also available on WWW: `<http://java.sun.com/docs/books/tutorial/index.html>`.
- Ken Arnold and James Gosling: *The Java Programming Language* (second edition). Addison-Wesley, 1998.
- Sun Microsystems: Java Technology Home Page: `<http://java.sun.com>`. Includes detailed documentation about the Java class libraries.

If you have a question about Java which this short tutorial does not answer, feel free to ask any of the teachers in your course.

1 Simple declarations and expressions

This section shows how to declare and use variables of the simple types, such as integers or booleans. Declarations and uses of object references are shown in Section 3 on page 6.

Note that declarations and statements can be mixed freely (in contrast to Simula and Pascal).

1.1 Simple declarations

Java supports the usual set of simple types, such as integer, boolean, and real variables. Here are a few of the most common ones:

```
int m, n;       // Two integer variables
double x, y;    // Two real coordinates
boolean b;      // Either ‘true’ or ‘false’
char ch;        // A character, such as ‘P’ or ‘@’
```

1.2 Numeric expressions and assignments

Numeric expressions are written in much the same way as in other languages.

```
n = 3 * (5 + 2);
x = y / 3.141592653;
n = m % 8;      // Modulo, i.e. n is now (m mod 8)
b = true;
ch = ‘x‘;
```

Note: the assignment is written using ‘=’ as opposed to ‘:=’ in many other languages. Another symbol, ‘==’, is used to compare two values to each other (see Section 2.1 on page 5). If you try to compare two values using ‘=’ you will get an error.

It is possible to assign a variable an initial value directly when declaring it. Example:

```
double f = 0.57;
boolean flag = true;
```

Unlike Simula, the initial value of a local variable is undefined (unless, of course, an initial value is explicitly given as just shown).

Pitfall: differences between integer and real division

The Java division operator (‘`/`’) can actually mean two different things: real division for real numbers, and integer division for integers. Usually this is not a problem, but it can occasionally lead to some surprising results:

```
double f;
f = 1 / 3;              // f is now 0.0
f = 1.0 / 3.0;         // f is now 0.33333333...
```

In the first case an integer division is performed, giving an integer result (0). To get the result 0.33333, the 1 and 3 are expressed as real values (1.0 and 3.0), which means the division becomes a real division.

1.3 Type conversion (casting)

In some languages it is possible to assign, for instance, a real value to an integer variable. The value is then automatically converted (in this case, rounded) to the right type.

Java does not perform all such conversions automatically. Instead the programmer must indicate where the conversions must be made by writing the desired type in parentheses before the expression. In Java, such a conversion is called a *cast*. Example:

```
double radians;
int degrees;
...
degrees = radians * 180 / 3.141592653;          // Error
degrees = (int) (radians * 180 / 3.141592653);  // OK
```

It is, however, possible to assign an integer value to a real variable without casting. In general, no cast is necessary as long as the conversion can be made without any loss of information.

2 Statements

Java statements are written in much the same way as in other languages. Just like in Simula or Pascal, statements can be grouped together in blocks using ‘`{`‘ and ‘`}`’ (corresponding to `begin` and `end` in these languages).

2.1 If statements and boolean expressions

A simple if statement is written as follows:

```
if (n == 3)
    x = 3.2;
```

Note:

- There is no `then` keyword
- The condition must be of boolean type and written within parentheses
- Comparison is made using '`==`'

There are of course a number of other comparison operators, such as '`<`', '`>`', '`<=`', '`>=`', and so on. The only one that looks different from Simula and Pascal is the 'not equals' operator '`!=`', which is used in the example below.

```
if (x != 0)
    y = 3.0 / x;      // Executed when x is non-zero
else
    y = 1;            // Executed when x is zero
```

Pitfall: semicolons and 'else' statements

Note that, unlike Simula and Pascal, there should be **a semicolon before the `else` keyword** in the example above.

However, when one uses braces ('`{`' and '`}`') to form a block of statements, **the right brace should NOT be followed by a semicolon**. (In fact, a right brace is never followed by a semicolon in Java.)

```
if (x != 0) {
    y = 3.0 / x;
    x = x + 1;
} else                // <--- Note: no semicolon
    y = 1;
```

It is common practice to always include the braces, even if they only contain a single statement. This avoids forgetting them whenever another statement is added.

More about boolean expressions

For boolean expressions, one needs to use logical operators corresponding to 'and', 'or', and 'not'. In Java, they are written as follows:

and	`&&`
or	`||`
not	`!`

For example:

```
int x, y;
boolean b;
...
if ((x <= 9 || y > 3) && !b) {
    b = true;
}
```

2.2 While and for statements

```
// Calculate exp(1). End when the term is less than 0.00001
double sum = 0.0;
double term = 1.0;
int k = 1;
while (term >= 0.00001) {
    sum = sum + term;
    term = term / k;
    k++;                 // Shortcut for 'k = k + 1'
}
```

As the example shows, there is nothing special about Java's while statement. The for statement is quite general and can be used in some very advanced ways. However, the most common use is to repeat some statement a known number of times:

```
// Calculate 1 + (1/2) + (1/3) + ... + (1/100)
int i;
double sum = 0.0;
for (i = 1; i <= 100; i++) {
    sum = sum + 1.0 / i;
}
```

As indicated in these examples, the statement `i++` is a shortcut for `i = i + 1`. Actually, there are at least four ways to increment an integer variable[1]:

```
i = i + 1;
i++;
++i;
i += 1;
```

As long as these statements are not used as parts of a larger expression, they mean exactly the same thing. There corresponding operators for decrementing variables are `--` and `-=`.

3 Classes and objects

As already mentioned, one file normally contains one class.

1. Supporting several ways to write essentially the same thing has historical reasons – it is a heritage from the C programming language.

3.1 Classes

A class declaration typically contains a set of attributes (sometimes called *instance variables*) and functions (called *methods* in Java). So far a Java class declaration is very similar to one in Simula. Attributes are declared almost as usual:

```
class Turtle {
    private boolean penDown;
    protected int x, y;

    // Declare some more stuff
}
```

The `private` and `protected` keywords require some explanation. The `private` declaration means that those attributes cannot be accessed outside of the class. In general, attributes should be kept private to prevent other classes from accessing them directly.

There are two other related keywords: `public` and `protected`. The `public` keyword is used to declare that something can be accessed from other classes. The `protected` keyword specifies that something can be accessed from within the class and all its subclasses, but not from the outside.

3.2 Methods

In Java, functions and procedures are called *methods*. Methods are declared as follows:

```
class Turtle {
    // Attribute declarations, as above

    public void jumpTo(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public int getX() {
        return x;
    }
}
```

This example contains two methods. The first is called `jumpTo` and has two integer parameters, `newX` and `newY`.

The second method is called `getX`, has no parameters, and returns an integer. Note that the empty pair of parentheses must be present.

Both method declarations begin with the keyword `public`, to make sure they can be accessed from other classes. (It is however possible to declare methods `private` or `protected`, which can be useful for internal methods which should not be used from other classes.)

Before the method's name, a type is written to indicate the method's return type. The `jumpTo` method does not return a value (i.e., it is a procedure, not a function). For this reason, it is declared as `void` (meaning 'nothing'). The `getX` method returns an integer, so it is declared as `int`.

3.3 Using objects

The `new` operator is used to create objects in much the same way as in other languages. If we assume the `Turtle` class requires two integer parameters (say, X and Y coordinates) a `Turtle` object can be created as follows:

```
Turtle t;
t = new Turtle(100, 100);
```

The first line is a declaration of a reference variable to a `Turtle` object, just like a `ref(Turtle)` declaration in Simula. The second line creates a new `Turtle` object and sets the `t` variable to refer to it.

There's nothing strange about calling methods in objects, as the following examples show.

```
int a = t.getX();
t.jumpTo(300, 200);
```

Java has garbage collection, so there is no need to destroy objects manually.

3.4 Parameters to classes: constructors

In the example above, the `Turtle` class was assumed to take two parameters. This must of course be specified in the class in some way, and in Java this is done in a special method called the *constructor*. The constructor is automatically called when an object is created, and the parameters of the constructor match those given when an object is created.

The constructor is written just like any ordinary method but with *the same name as the class*, and *no return type* (not even `void`).

The `Turtle` constructor could, for instance, look like this:

```
public Turtle(int initX, int initY) {
    x = initX;
    y = initY;
    penDown = false;
}
```

Unlike Simula class parameters, the constructor's parameters are not attributes. Instead they are used to give initial values to the attributes.

3.5 The `main` method

In Java, statements can only be written within methods in classes. This means that there must be some method which is called by the system when the program starts executing. This method is called `main` and must be declared in the class which is started from the command line (for example, in the `TurtleMaze` class if one runs `java TurtleMaze`).

A `main` method usually creates a few objects and does some small work to get things going. For `Turtle` a simple `main` method may look as follows:

```
public static void main(String[] args) {
    Turtle t = new Turtle(100, 200);
    t.right(90);
    while (t.getX() < 150) {
        t.forward(2);
    }
}
```

There are two new things about `main`, which can both safely be ignored for now. The first is the `static` keyword. It means that when the `main` method is called, it is not associated with an object, but with the class. (This implies that the method cannot access any attributes.)

The other new thing is the parameter named `args`. If the Java interpreter is given any more information than the class name, this data is passed on to the `main` method in this parameter.

3.6 Inheritance

To declare a subclass of another class, use the `extends` keyword in the class declaration:

```
class NinjaTurtle extends Turtle {
    // Declarations for Ninja turtles
}
```

So far, this works in exactly the same way as subclasses in Simula. If the superclass' constructor has any parameters, it must be called first using the keyword `super`. The constructor for `NinjaTurtle` might look like this:

```
public NinjaTurtle(int initX, int initY, String name) {
    super(initX, initY);      // Call superclass' constructor
    // ... do some more initialization stuff...
}
```

Virtual methods

In Java, all methods are virtual, so there is no need for any special syntax for virtual methods. A method in a class automatically overrides any method with the same name and parameters in any superclass.

It is possible to declare *abstract* methods. Such methods are really just declarations without any associated implementation, meaning that the method must be implemented in some subclass. Consider, for example, a class for graphic figures: `Figure`. That class is then specialized into `Circle`, `Square` and so on. All figures can be drawn, but the implementation is left to subclasses. The draw method in `Figure` could be declared as follows:

```
public abstract void draw();
```

A class with one or more abstract methods is itself called abstract, and must be declared as such by writing `abstract class` instead of `class`. It is not possible to create objects from abstract classes.

3.7 Interfaces and listeners

An *interface* can be used to specify that a class has to provide a certain set of methods. This can be useful in a number of situations and is perhaps best shown with an example as follows.

Programs with graphical user interfaces often need to be informed whenever the mouse is clicked. Usually the program has some method which should be automatically called by the system whenever the user clicks the mouse.

Java provides a very flexible way of specifying an object and a method to call in such situations. Suppose the window system declares an *interface*, written as follows:

```
interface MouseListener {
    void processMouseClick(int x, int y);
}
```

This declaration essentially says that if an object should be used to handle mouse clicks, its class should contain a `processMouseClick` method with two integer parameters.

A class can then be declared to *implement* that interface:

```
class SomeClass extends SomeOtherClass implements MouseListener {
    // ...declarations...

    public void processMouseClick(int x, int y) {
        // Do something sensible here
    }
}
```

Finally, the window system should have some method to register `MouseListener` objects to inform whenever a mouse is clicked. Such a method might look like as follows:

```
class WindowSystem {
    public void addMouseListener(MouseListener m) {
        // Insert m into some clever data structure
    }
    // ... and loads of more stuff...
}
```

Note that the type of the parameter `m` is not a class, but an interface. In other words, it does not matter of which class the listener is, as long as that class implements the `MouseListener` interface. This turns out to be a quite flexible technique in practice, since the different components need very little information about each other.

4 Exceptions

Many things can go wrong during the execution of a program. These *run-time errors* can be divided into two broad categories:

- Faults introduced by the programmer, such as division by zero or calling a method with a null reference.
- Things out of the program's control, such as a user entering a garbage on the keyboard when the program expects a positive integer.

The latter category is the one that programmers usually take care of. The traditional way of handling these errors is to put the code in question in some method which returns a value to indicate whether whings went well or not. A method to read a positive integer from the keyboard could, for instance, look like this:

```
public int getNatural() { ... }
```

Suppose the special value -1 is used to indicate that an invalid number was entered by the user. The code that calls this method would then have to check the return value with an if statement. If that code is part of some method, that method may in turn have to return some value to indicate that things went wrong. This kind of programming can easily turn into a lot of if statements and special return values, and very few statements that actually do something.

4.1 Catching exceptions

Using Java exceptions, the method above could be declared as follows:

```
public int getNatural() throws IOException { ... }
```

This method is said to *throw* an *exception* (more specifically, an `IOException`) when something else than a natural number is entered on the keyboard. The code that calls the method could look like this:

```
int m, n;
try {
    n = getNatural();
    m = n * 2; // If an exception is thrown, this is not executed
}
catch (IOException e) {
    // The user entered something wrong. Use 1 as default.
    n = 1;
    m = 2;
}
```

The statement(s) within the `try` clause are executed as usual, but whenever an exception occurs, the `try` clause is interrupted and the statements within the corresponding `catch` clause are executed. The execution then continues after the `try`/`catch` clauses.

The `try` clause can contain many statements that throw exceptions, and there can be several different `catch` clauses. This means that the error handling is separated from the code that actually does the work. It often also helps in reducing the complexity of the error handling code.

If a method does not handle an exception (for instance, if it uses the `getNatural()` method without any `try`/`catch` clauses), the exception must be passed on to the calling method. This is done using a `throws` declaration as indicated above:

```
public void doStuff() throws IOException {
    int n = getNatural();   // May throw an exception
    // Clever calculations using n...
}
```

The method that calls `doStuff()` must in turn either catch the exception or pass it on. If the exception is not caught (even the `main` method passes it on), the execution is aborted with an error message.

4.2 Throwing exceptions

It is of course also possible to throw exceptions yourself when things go wrong. The `getNatural()` method could look as follows (in Java-pseudo-code):

```
public int getNatural() throws IOException {
    char ch;
    while (more input) {
        ch = (read character);
        if (ch < '0' || ch > '9') {
            throw new IOException("bad natural number");
        }
        ...
    }
    ...
}
```

Note the `new` keyword in the `throw` statement above. This reveals that the exception is actually an object which can contain information. In particular, it contains a string describing the error, as indicated above.

4.3 Declaring new exceptions

Although there are a number of pre-defined exception classes in Java, you may occasionally need to declare your own exceptions. This can be done by creating a subclass of the existing `Exception` class. Suppose we want to throw an exception when some external equipment is overheating. Such an exception could hold information about the current temperature, as follows:

```
class OverheatedException extends Exception {
    public OverheatedException(String s, double temp) {
        super(s);
        myTemperature = temp;
    }

    public double getTemperature() {
        return myTemperature;
    }

    private double myTemperature;
}
```

4.4 Unchecked exceptions

Some exceptions do not have to be caught: the so-called *unchecked* exceptions which are thrown by the run-time system in some cases. For instance, when an integer division by zero occurs, an `ArithmeticException` is thrown by the system. This exception is normally not caught anywhere, so when it is thrown the execution of the program is aborted with an error message (just like in most other languages).

It is possible to catch these unchecked exceptions, which can occasionally be useful.

5 Miscellaneous

This section contains a few details about Java that might be useful to know when writing Java programs.

5.1 Comments

One kind of comments has already been shown: the line comment, which starts with '`//`' and extends to the end of a line. Multi-line comments are written using '`/*`' and '`*/`' as follows:

```
/* This is a comment
   which continues on to a second line */
```

(A special case of such multi-line comments are the *documentation comments*. They are written immediately before classes and methods and begin with '`/**`' (two asterisks) and end, as usual, with '`*/`' (one asterisk). Such comments are used by a special tool, `javadoc`, to automatically generate low-level documentation of the program.)

5.2 Using packages

Some of the library classes, as well as some of the help classes for the laboratories, are provided in *packages*. To use the classes from a package, they must be *imported*.

For instance, many of the classes used for graphical user interfaces in Java (AWT, or Abstract Window Toolkit) belong to the `java.awt` package. To use these classes, the following must be written first in the source file:

```
import java.awt.*;
```

5.3 Arrays

A Java array is similar to an object in some ways. It is for example accessed using reference variables. Array references can be declared as follows:

```
int[] someInts;       // An integer array
Turtle[] turtleFarm;  // An array of references to Turtles
```

Since these variables are only references to arrays, the array sizes are not given in the declarations, but when the arrays are actually created.

```
someInts = new int[30];
turtleFarm = new Turtle[100];
```

The array elements can then be used as any simple scalar variables. (Note that indices always start at 0 and end at the size **minus one**, so the elements of the `someInts` array have the indices 0 to 29.)

```
int i;
for (i = 0; i < someInts.length; i = i + 1) {
    someInts[i] = i * i;
}
```

The expression `someInts.length` means in the length of the vector, 30 in this case.

5.4 Writing to the terminal

To write something to the terminal, call one of the methods `print` and `println` in the object `System.out`. They both write the argument (a `String`) to the terminal. The latter method, `println`, also ends the line. Example:

```
System.out.print(“Jag vill bo “);
System.out.println(“i en svamp“);
System.out.println(“Annars får jag kramp“);
```

The resulting output is:

```
Jag vill bo i en svamp
Annars får jag kramp
```

Variable values can be printed like this:

```
int a;
a = 6 * 7;
System.out.println(“6 * 7 = “ + a);
```

6 A complete Java program

The following example program displays a window with graphical figures (squares and circles). The program is not intended to be useful in any way except as an example of a complete Java program.

Figure.java

```
import java.awt.*;

/**
 * Simple abstract class for graphic figures that can be drawn in windows.
 */
abstract class Figure {

    /**
     * Constructor: takes two parameters, the X and Y coordinates.
     */
    public Figure(int inX, int inY) {
        x = inX;
        y = inY;
    }

    /**
     * Abstract method for drawing this thing.
     *
     * The g parameter is a 'pen' that can be used to draw things
     * in the window.
     */
    public abstract void draw(Graphics g);

    /**
     * Move the figure to (newX, newY).
     */
    public void move(int newX, int newY) {
        x = newX;
        y = newY;
    }

    protected int x, y;         // X and Y coordinates
}
```

Square.java

```
import java.awt.*;

/**
 * A square that can be drawn in a window. The coordinates represent the
 * upper left corner of the square.
 */
class Square extends Figure {

    /**
     * Constructor: first two parameters are the coordinates, the third is
     * the side.
     */
    public Square(int inX, int inY, int inSide) {
        super(inX, inY);
        side = inSide;
    }

    /**
     * Drawing method for squares.
     */
    public void draw(Graphics g) {
        g.drawRect(x, y, side, side);
    }

    private int side;       // Square side
}
```

Circle.java

```
import java.awt.*;

/**
 * Circle class. The coordinates represent the circle's center.
 */
class Circle extends Figure {

    /**
     * Constructor: the first two parameters are the coordinates,
     * the third is the diameter.
     */
    public Circle(int inX, int inY, int inDiam) {
        super(inX, inY);
        d = inDiam;
```

```
    }

    /**
     * Drawing method for circles.
     */
    public void draw(Graphics g) {
        g.drawOval(x, y, d, d);
    }

    private int d;            // Circle diameter
}
```

FigureWindow.java

```
import java.awt.*;

/**
 * A simple window to display graphic figures in. The window is a subclass
 * of the Java 'Frame' class, which describes graphic windows. The window
 * keeps its figures in an array.
 *
 * The Java window system (AWT) automatically calls the paint method in
 * the Frame class whenever the window's contents need to be redrawn. A
 * new implementation of paint is provided in FigureWindow to handle the
 * drawing.
 */
class FigureWindow extends Frame {

    /**
     * Constructor: the parameter indicates the maximal number of figures.
     */
    public FigureWindow(int max) {
        super("Fabulous Figures");     // Window title
        figures = new Figure[max];
        nbrOfFigures = 0;
    }

    /**
     * Add the figure f to the window. If the maximal number of figures has
     * been reached, nothing happens.
     */
    public void addFigure(Figure f) {
        if (nbrOfFigures < figures.length) {
            figures[nbrOfFigures] = f;
            nbrOfFigures++;
        }
    }
```

```
    /**
     * This method is called automatically by the system. Draws the
     * raphic figures associated with the window.
     *
     * The g parameter is a drawing 'pen' provided by the system.
     */
    public void paint(Graphics g) {
        int i;
        for(i = 0; i < nbrOfFigures; i++) {
            figures[i].draw(g);
        }
    }

    // Array of graphic figures
    private Figure[] figures;

    // Current number of figures
    private int nbrOfFigures;

    /**
     * Main method: creates a FigureWindow and a few figures inside it.
     */
    public static void main(String[] args) {
        FigureWindow w = new FigureWindow(10);
        w.setSize(400, 300);
        w.addFigure(new Square(50, 50, 200));
        w.addFigure(new Circle(200, 100, 150));
        w.addFigure(new Circle(300, 200, 200));
        w.show();
    }
}
```