vogella SERVICES

Training

Books

Java concurrency (multi-threading) - Tutorial

Lars Vogel

Version 2.3

Copyright © 2008, 2009, 2010, 2011, 2012, 2013 Lars Vogel

02.08.2013

Revision History

Revision 0.1	28.12.2008	Lars Vogel	Created
Revision 0.2 - 2.3	12.01.2009 - 02.08.2013	Lars Vogel	bug fixes and enhancements

Java concurrency (multi-threading)

This article describes how to do concurrent programming with Java. It covers the concepts of parallel programming, immutability, threads, the executor framework (thread pools), futures, callables and the fork-join framework.

Northeastern NETWORKED FOR SUCCESS MS COMPUTER SCIENCE ONLINE + ON-CAMPUS TOP 50 NATIONAL RESEARCH UNIVERSITY LEARN MORE

Table of Contents

Ads

+concurrency +concurrent
Over 100 Million Visitors. Discover and Explore on Ask.com!
Ask.com/Answers

The Latest on Face To Face Dimensions Of
See what's trending on Yahoo & stay up to date with Face To Face Dimensions Of Flanged Valves Asme
search.yahoo.com

Find Java Concurrency (multi-
Search for Java Concurrency (multi-threading). Find Great Results on Yahoo.com!
search.yahoo.com

Find Java Concurrency (multi-
Search for Java Concurrency (multi-threading). Find Great Results on Yahoo.com!
search.yahoo.com

Looking for Java Concurrency (multi-
Find Java Concurrency (multi-threading) at Amazon
www.Amazon.com

Java Concurrency (multi-threading)
Top answers for Java Concurrency (multi-threading)
www.Answered-Questions.com

SAP (Mobile-Applications) && (Enterprise-Ready) && (SAP HANA Cloud Platform) == money++
Learn about OpenUI5, the open HTML5 toolkit from SAP
Learn more

Cubify by 3D Systems

Home of The Cube, CubePro & iSense. Bring Your 3D Designs to Life!

1. Concurrency

1.1. What is concurrency?

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improve the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

1.2. Process vs. threads

A *process* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A *thread* is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

2. Improvements and issues with concurrency

Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

2.1. Limits of concurrency gains

Concurrency promises to perform certain task faster as these tasks can be divided into subtasks and these subtasks can be executed in parallel. Of course the runtime is limited by parts of the task which can be performed in parallel.

The theoretical possible performance gain can be calculated by the following rule which is referred to as *Amdahl's Law.*

If F is the percentage of the program which can not run in parallel and N is the number of processes, then the maximum performance gain is 1/ (F+ ((1-F)/n)).

2.2. Concurrency issues

Threads have their own call stack, but can also access shared data. Therefore you have two basic problems, visibility and access problems.

A visibility problem occurs if thread A reads shared data which is later changed by thread B and thread A is unaware of this change.

An access problem can occur if several thread access and change the same shared data at the same time.

Visibility and access problem can lead to

- Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.
- Safety failure: The program creates incorrect data.

3. Concurrency in Java

3.1. Processes and Threads

A Java program runs in its own process and by default in one thread. Java supports threads as part of the Java language via the `Thread` code. The Java application can create new threads via this class.

Java 1.5 also provides improved support for concurrency with the in the `java.util.concurrent` package.

3.2. Locks and thread synchronization

Java provides *locks* to protect certain parts of the code to be executed by several threads at the same time. The simplest way of locking a certain method or Java class is to define the method or class with the `synchronized` keyword.

The *synchronized* keyword in Java ensures:

- that only a single thread can execute a block of code at the same time
- that each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock

Synchronization is necessary for mutually exclusive access to blocks of and for reliable communication between threads.

You can use the *synchronized* keyword for the definition of a method. This would ensure that only one thread can enter this method at the same time. Another threads which is calling this method would wait until the first threads leaves this method.

```
public synchronized void critial() {
  // some thread critical stuff
  // here
}
```

You can also use the `synchronized` keyword to protect blocks of code within a method. This block is guarded by a key, which can be either a string or an object. This key is called the *lock*. All code which is protected by the same lock can only be executed by one thread at the same time

For example the following datastructure will ensure that only one thread can access the inner block of the `add()` and `next()` methods.

```
package de.vogella.pagerank.crawler;

import java.util.ArrayList;
import java.util.List;

/**
 * Data structure for a web crawler. Keeps track of the visited sites and keeps
 * a list of sites which needs still to be crawled.
 *
 * @author Lars Vogel
 *
 */

public class CrawledSites {
  private List<String> crawledSites = new ArrayList<String>();
  private List<String> linkedSites = new ArrayList<String>();

  public void add(String site) {
    synchronized (this) {
      if (!crawledSites.contains(site)) {
        linkedSites.add(site);
      }
    }
  }


/**
   * Get next site to crawl. Can return null (if nothing to crawl)
   */

  public String next() {
    if (linkedSites.size() == 0) {
      return null;
    }
    synchronized (this) {
      // Need to check again if size has changed
      if (linkedSites.size() > 0) {
        String s = linkedSites.get(0);
        linkedSites.remove(0);
        crawledSites.add(s);
        return s;
      }
      return null;
    }
  }

}
```

3.3. Volatile

If a variable is declared with the *volatile* keyword then it is guaranteed that any thread that reads the field will see the most recently written value. The *volatile* keyword will not perform any mutual exclusive lock on the variable.

As of Java 5 write access to a *volatile* variable will also update non-volatile variables which were modified by the same thread. This can also be used to update values within a reference variable, e.g. for a *volatile* variable person. In this case you must use a temporary variable person and use the setter to initialize the variable and then assign the temporary variable to the final variable. This will then make the address changes of this variable and the values visible to other threads.

SAP SECURE_CLOUD && (build || extend || run) == SAP HANA Cloud Platform
Start building applications on an in-memory PaaS using cloud-based developer tools
Sign up for a free account ›

4. The Java memory model

4.1. Overview

The *Java memory model* describes the communication between the memory of the threads and the main memory of the application.

It defines the rules how changes in the memory done by threads are propagated to other threads. The *Java memory model* also defines the situations in which a thread re-fresh its own memory from the main memory.

It also describes which operations are atomic and the ordering of the operations.

4.2. Atomic operation

An atomic operation is an operation which is performed as a single unit of work without the possibility of interference from other operations.

The Java language specification guarantees that reading or writing a variable is an atomic operation(unless the variable is of type `long` or `double`). Operations variables of type `long` or `double` are only atomic if they declared with the `volatile` keyword. .

Assume `i` is defined as `int`. The `i++` (increment) operation it not an atomic operation in Java. This also applies for the other numeric types, e.g. long. etc).

The `i++` operation first reads the value which is currently stored in i (atomic operations) and then it adds one to it (atomic operation). But between the read and the write the value of i might have changed.

Since Java 1.5 the java language provides atomic variables, e.g. AtomicInteger or AtomicLong which provide methods like `getAndDecrement()`, `getAndIncrement()` and `getAndSet()` which are atomic.

4.3. Memory updates in synchronized code

The Java memory model guarantees that each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock.

5. Immutability and Defensive Copies

5.1. Immutability

The simplest way to avoid problems with concurrency is to share only immutable data between threads. Immutable data is data which cannot changed.

To make a class immutable make

- all its fields final
- the class declared as final
- the this reference is not allowed to escape during construction
- Any fields which refer to mutable data objects are
    - private
    - have no setter method
    - they are never directly returned of otherwise exposed to a caller
    - if they are changed internally in the class this change is not visible and has no effect outside of the class

An immutable class may have some mutable data which is uses to manages its state but from the outside this class nor any attribute of this class can get changed.

For all mutable fields, e.g. Arrays, that are passed from the outside to the class during the construction phase, the class needs to make a defensive-copy of the elements to make sure that no other object from the outside still can change the data

5.2. Defensive Copies

You must protect your classes from calling code. Assume that calling code will do its best to change your data in a way you didn't expect it. While this is especially true in case of immutable data it is also true for non-immutable data which you still not expect that this data is changed outside your class.

To protect your class against that you should copy data you receive and only return copies of data to calling code.

The following example creates a copy of a list (ArrayList) and returns only the copy of the list. This way the client of this class cannot remove elements from the list.

```
package de.vogella.performance.defensivecopy;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyDataStructure {
  List<String> list = new ArrayList<String>();

  public void add(String s) {
    list.add(s);
  }


/**
   * Makes a defensive copy of the List and return it
   * This way cannot modify the list itself
   *
   * @return List<String>
   */

  public List<String> getList() {
    return Collections.unmodifiableList(list);
  }
}
```

6. Threads in Java

The base means for concurrency are is the java.lang.Threads class. A Thread executes an object of type java.lang.Runnable.

Runnable is an interface with defines the run() method. This method is called by the Thread object and contains the work which should be done. Therefore the "Runnable" is the task to perform. The Thread is the worker who is doing this task.

The following demonstrates a task (Runnable) which counts the sum of a given range of numbers. Create a new Java project called de.vogella.concurrency.threads for the example code of this section.

```
package de.vogella.concurrency.threads;

/**
 * MyRunnable will count the sum of the number from 1 to the parameter
 * countUntil and then write the result to the console.
 * <p>
 * MyRunnable is the task which will be performed
 *
 * @author Lars Vogel
 *
 */

public class MyRunnable implements Runnable {
  private final long countUntil;

  MyRunnable(long countUntil) {
    this.countUntil = countUntil;
  }

  @Override
  public void run() {
    long sum = 0;
    for (long i = 1; i < countUntil; i++) {
      sum += i;
    }
    System.out.println(sum);
  }
}
```

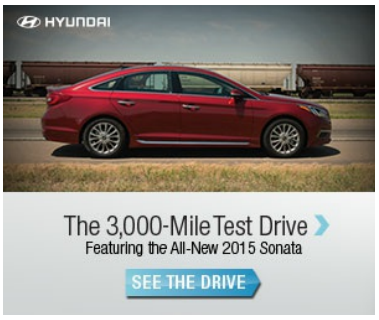
The following example demonstrate the usage of the Thread and the Runnable class.

```
package de.vogella.concurrency.threads;

import java.util.ArrayList;
import java.util.List;

public class Main {

  public static void main(String[] args) {
    // We will store the threads so that we can check if they are done
    List<Thread> threads = new ArrayList<Thread>();
    // We will create 500 threads
    for (int i = 0; i < 500; i++) {
      Runnable task = new MyRunnable(10000000L + i);
      Thread worker = new Thread(task);
      // We can set the name of the thread
      worker.setName(String.valueOf(i));
      // Start the thread, never call method run() direct
      worker.start();
      // Remember the thread for later usage
      threads.add(worker);
    }
    int running = 0;
    do {
      running = 0;
      for (Thread thread : threads) {
        if (thread.isAlive()) {
          running++;
        }
      }
      System.out.println("We have " + running + " running threads. ");
    } while (running > 0);

  }
}
```

Using the Thread class directly has the following disadvantages.

- Creating a new thread causes some performance overhead
- Too many threads can lead to reduced performance, as the CPU needs to switch between these threads.
- You cannot easily control the number of threads, therefore you may run into out of memory errors due to too many threads.

The java.util.concurrent package offers improved support for concurrency compared to the direct usage of Threads. This package is described in the next section.



7. Threads pools with the Executor Framework

Tip

You find this examples in the source section in Java project called *de.vogella.concurrency.threadpools*.

Thread pools manage a pool of worker threads. The thread pools contains a work queue which holds tasks waiting to get executed.

A thread pool can be described as a collection of Runnable objects (work queue) and a connections of running threads. These threads are constantly running and are checking the work query for new work. If there is new work to be done they execute this Runnable. The Thread class itself provides a method, e.g. execute(Runnable r) to add a new Runnable object to the work queue.

The Executor framework provides example implementation of the java.util.concurrent.Executor interface, e.g. Executors.newFixedThreadPool(int n) which will create n worker threads. The ExecutorService adds life cycle methods to the Executor, which allows to shutdown the Executor and to wait for termination.

Tip

If you want to use one thread pool with one thread which executes several runnables you can use the Executors.newSingleThreadExecutor() method.

Create again the Runnable.

```java
package de.vogella.concurrency.threadpools;

/**
 * MyRunnable will count the sum of the number from 1 to the parameter
 * countUntil and then write the result to the console.
 * <p>
 * MyRunnable is the task which will be performed
 *
 * @author Lars Vogel
 *
 */

public class MyRunnable implements Runnable {
  private final long countUntil;

  MyRunnable(long countUntil) {
    this.countUntil = countUntil;
  }

  @Override
  public void run() {
    long sum = 0;
    for (long i = 1; i < countUntil; i++) {
      sum += i;
    }
    System.out.println(sum);
  }
}
```

Now you run your runnables with the executor framework.

```java
package de.vogella.concurrency.threadpools;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    for (int i = 0; i < 500; i++) {
      Runnable worker = new MyRunnable(10000000L + i);
      executor.execute(worker);
    }
    // This will make the executor accept no new threads
    // and finish all existing threads in the queue
    executor.shutdown();
    // Wait until all threads are finish
    executor.awaitTermination();
    System.out.println("Finished all threads");
  }
}
```

In case the threads should return some value (result-bearing threads) then you can use the `java.util.concurrent.Callable` class.

8. Futures and Callables

The code examples for this section are created in a Java project called *de.vogella.concurrency.callables*.

The executor framework presented in the last chapter works with `Runnables`. Runnable do not return result.

In case you expect your threads to return a computed result you can use `java.util.concurrent.Callable`. The `Callable` object allows to return values after completion.

The `Callable` object uses generics to define the type of object which is returned.

If you submit a `Callable` object to an `Executor` the framework returns an object of type `java.util.concurrent.Future`. This `Future` object can be used to check the status of a `Callable` and to retrieve the result from the `Callable`.

On the `Executor` you can use the method submit to submit a `Callable` and to get a future. To retrieve the result of the future use the `get()` method.

```java
package de.vogella.concurrency.callables;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<Long> {
  @Override
  public Long call() throws Exception {
    long sum = 0;
    for (long i = 0; i <= 100; i++) {
      sum += i;
    }
    return sum;
  }

}
```

```
package de.vogella.concurrency.callables;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableFutures {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {

    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    List<Future<Long>> list = new ArrayList<Future<Long>>();
    for (int i = 0; i < 20000; i++) {
      Callable<Long> worker = new MyCallable();
      Future<Long> submit = executor.submit(worker);
      list.add(submit);
    }
    long sum = 0;
    System.out.println(list.size());
    // now retrieve the result
    for (Future<Long> future : list) {
      try {
        sum += future.get();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.println(sum);
    executor.shutdown();
  }
}
```

Ads

1. **+concurrency +concurrent** Over 100 Million Visitors. Discover and Explore on Ask.com! Ask.com/Answers
2. **New Policy in TX** If you drive in TX you better read this... Insurance.Comparisons.org
3. **Find Java Concurrency (multi-threading)** Search for Java Concurrency (multi- search.yahoo.com

9. Nonblocking algorithms

Java 5.0 provides supports for additional atomic operations. This allows to develop algorithm which are non-blocking algorithm, e.g. which do not require synchronization, but are based on low-level atomic hardware primitives such as compare-and-swap (CAS). A compare-and-swap operation check if the variable has a certain value and if it has this value it will perform this operation.

Non-blocking algorithm are usually much faster then blocking algorithms as the synchronization of threads appears on a much finer level (hardware).

For example this created a non-blocking counter which always increases. This example is contained in the project called *de.vogella.concurrency.nonblocking.counter*.

```
package de.vogella.concurrency.nonblocking.counter;

import java.util.concurrent.atomic.AtomicInteger;

public class Counter {
  private AtomicInteger value = new AtomicInteger();
  public int getValue(){
    return value.get();
  }
  public int increment(){
    return value.incrementAndGet();
  }

  // Alternative implementation as increment but just make the
  // implementation explicit
  public int incrementLongVersion(){
    int oldValue = value.get();
    while (!value.compareAndSet(oldValue, oldValue+1)){
       oldValue = value.get();
    }
    return oldValue+1;
  }

}
```

And a test.

```
package de.vogella.concurrency.nonblocking.counter;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Test {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {
    final Counter counter = new Counter();
    List<Future<Integer>> list = new ArrayList<Future<Integer>>();

    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    for (int i = 0; i < 500; i++) {
      Callable<Integer> worker = new  Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
          int number = counter.increment();
          System.out.println(number);
          return number ;
        }
      };
      Future<Integer> submit= executor.submit(worker);
      list.add(submit);

    }


    // This will make the executor accept no new threads
    // and finish all existing threads in the queue
    executor.shutdown();
    // Wait until all threads are finish
    while (!executor.isTerminated()) {
    }
    Set<Integer> set = new HashSet<Integer>();
    for (Future<Integer> future : list) {
      try {
        set.add(future.get());
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    if (list.size()!=set.size()){
      throw new RuntimeException("Double-entries!!!");
    }

  }


}
```

The interesting part is how the `incrementAndGet()` method is implemented. It uses a CAS operation.

```
public final int incrementAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }
```

The JDK itself makes more and more use of non-blocking algorithms to increase performance for every developer. Developing correct non-blocking algorithm is not a trivial task.

For more information on non-blocking algorithm, e.g. examples for a non-blocking Stack and non-block LinkedList, please see **http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html**

10. Fork-Join in Java 7

Java 7 introduce a new parallel mechanism for compute intensive tasks, the fork-join framework. The fork-join framework allows you to distribute a certain task on several workers and then wait for the result.

E For Java 6.0 you can download the package (jsr166y) from **Download site**

For testing create the Java project "de.vogella.performance.forkjoin". If you are not using Java 7 you also need to *jsr166y.jar* to the classpath.

Create first a `algorithm` package and then the following class.

```
package algorithm;

import java.util.Random;

/**
 *
 * This class defines a long list of integers which defines the problem we will
 * later try to solve
 *
 */

public class Problem {
  private final int[] list = new int[2000000];

  public Problem() {
    Random generator = new Random(19580427);
    for (int i = 0; i < list.length; i++) {
      list[i] = generator.nextInt(500000);
    }
  }

  public int[] getList() {
    return list;
  }

}
```

Define now the `Solver` class as shown in the following example coding.

Tip

The API defines other top classes, e.g. RecursiveAction, AsyncAction. Check the Javadoc for details.

```
package algorithm;

import java.util.Arrays;

import jsr166y.forkjoin.RecursiveAction;

public class Solver extends RecursiveAction {
  private int[] list;
  public long result;

  public Solver(int[] array) {
    this.list = array;
  }

  @Override
  protected void compute() {
    if (list.length == 1) {
      result = list[0];
    } else {
      int midpoint = list.length / 2;
      int[] l1 = Arrays.copyOfRange(list, 0, midpoint);
      int[] l2 = Arrays.copyOfRange(list, midpoint, list.length);
      Solver s1 = new Solver(l1);
      Solver s2 = new Solver(l2);
      forkJoin(s1, s2);
      result = s1.result + s2.result;
    }
  }
}
```

Now define a small test class for testing it efficiency.

```
package testing;

import jsr166y.forkjoin.ForkJoinExecutor;
import jsr166y.forkjoin.ForkJoinPool;
import algorithm.Problem;
import algorithm.Solver;

public class Test {

  public static void main(String[] args) {
    Problem test = new Problem();
    // check the number of available processors
    int nThreads = Runtime.getRuntime().availableProcessors();
    System.out.println(nThreads);
    Solver mfj = new Solver(test.getList());
    ForkJoinExecutor pool = new ForkJoinPool(nThreads);
    pool.invoke(mfj);
    long result = mfj.getResult();
    System.out.println("Done. Result: " + result);
    long sum = 0;
    // check if the result was ok
    for (int i = 0; i < test.getList().length; i++) {
      sum += test.getList()[i];
    }
    System.out.println("Done. Result: " + sum);
  }
}
```

11. Deadlock

A concurrent application has the risk of a *deadlock*. A set of processes are deadlocked if all processes are waiting for an event which another process in the same set has to cause.

For example if thread A waits for a lock on object Z which thread B holds and thread B wait for a look on object Y which is hold be process A then these two processes are looked and cannot continue in their processing.

Ads

1. **+concurrency +concurrent** Over 100 Million Visitors. Discover and Explore on Ask.com! Ask.com/Answers
2. **New Policy in TX** If you drive in TX you better read this... Insurance.Comparisons.org
3. **Find Java Concurrency (multi-threading)** Search for Java Concurrency (multi- search.yahoo.com

12. Support this website

This tutorial is Open Content under the **CC BY-NC-SA 3.0 DE** license. Source code in this tutorial is distributed under the **Eclipse Public License**. See the **vogella License** page for details on the terms of reuse.

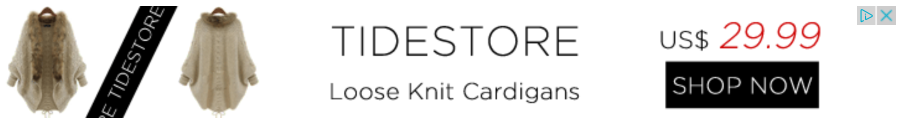
Writing and updating these tutorials is a lot of work. If this free community service was helpful, you can support the cause by giving a tip as well as reporting typos and factual errors.

12.1. Thank you

Please consider a **contribution** if this article helped you. It will help to maintain our content and our Open Source activities.

12.2. Questions and Discussion

If you find errors in this tutorial, please notify me (see the **top of the page**). Please note that due to the high volume of feedback I receive, I cannot answer questions to your implementation. Ensure you have read the **vogella FAQ** as I don't respond to questions already answered there.



13. Links and Literature

13.1. Source Code

Source Code of Examples

13.2. Concurrency Resources

http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html Introduction to Concurrency

http://www.briangoetz.com/pubs.html Article series from Brian Goetz including lots about concurrency

http://www.ibm.com/developerworks/library/j-jtp0730.html Thread pools and work queues by Brian Goetz

http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html Introduction to nonblocking algorithms by Brian Goetz

http://www.ibm.com/developerworks/java/library/j-jtp11137.html Java theory and practice: Stick a fork in it, Part 1 by Brian Goetz

http://www.ibm.com/developerworks/java/library/j-jtp03048.html Java theory and practice: Stick a fork in it, Part 2 by Brian Goetz

13.3. vogella Resources

vogella Training Android and Eclipse Training from the vogella team

Android Tutorial Introduction to Android Programming

GWT Tutorial Program in Java, compile to JavaScript and HTML

Eclipse RCP Tutorial Create native applications in Java

JUnit Tutorial Test your application

Git Tutorial Put all your files in a distributed version control system