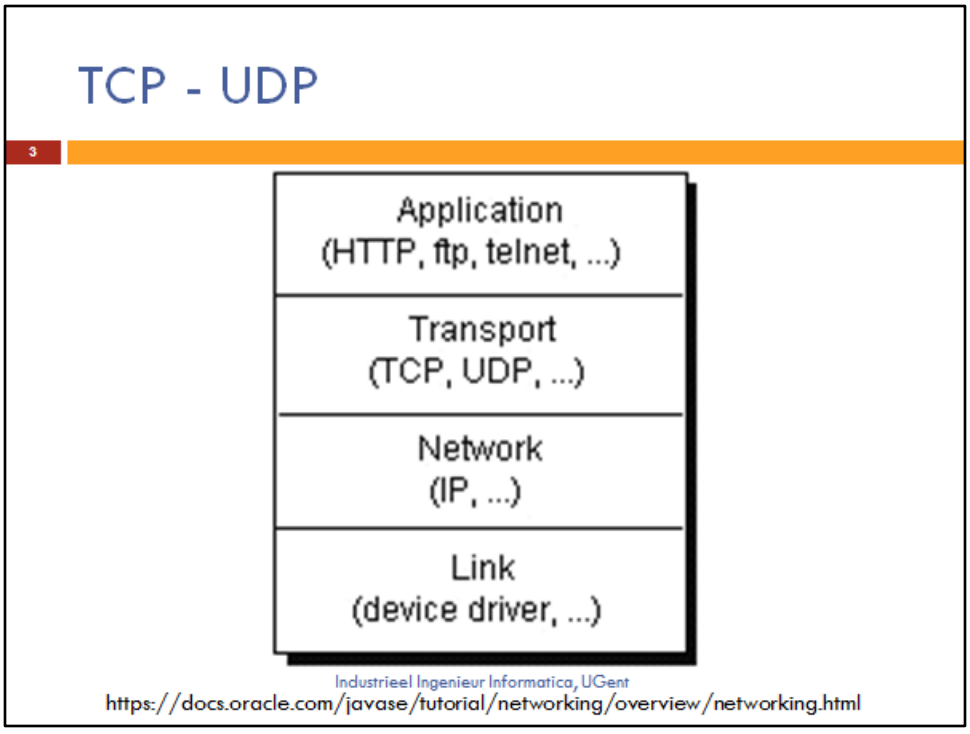
NETWERKPROGRAMMATIE JAVA

Veerle Ongenae

Overzicht

- Herhaling



Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

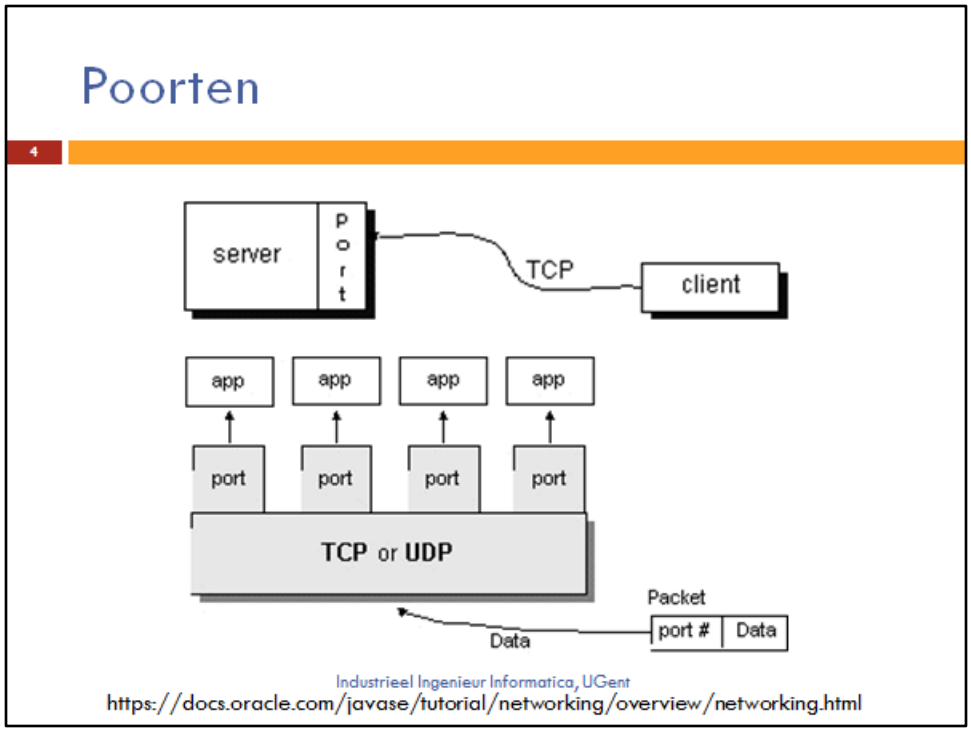
When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the java.net package. These classes provide system-independent network communication. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

TCP (*Transmission Control Protocol*) is a connection-based protocol that provides a reliable flow of data between two computers.
TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

UDP (*User Datagram Protocol*) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival.

UDP is not connection-based like TCP.
The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.
Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.
Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network. In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.
The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data from one application to another. Sending datagrams is much like sending a letter through the mail service: The order of delivery is not important and is not guaranteed, and each message is independent of any others.



Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? Through the use of ports.

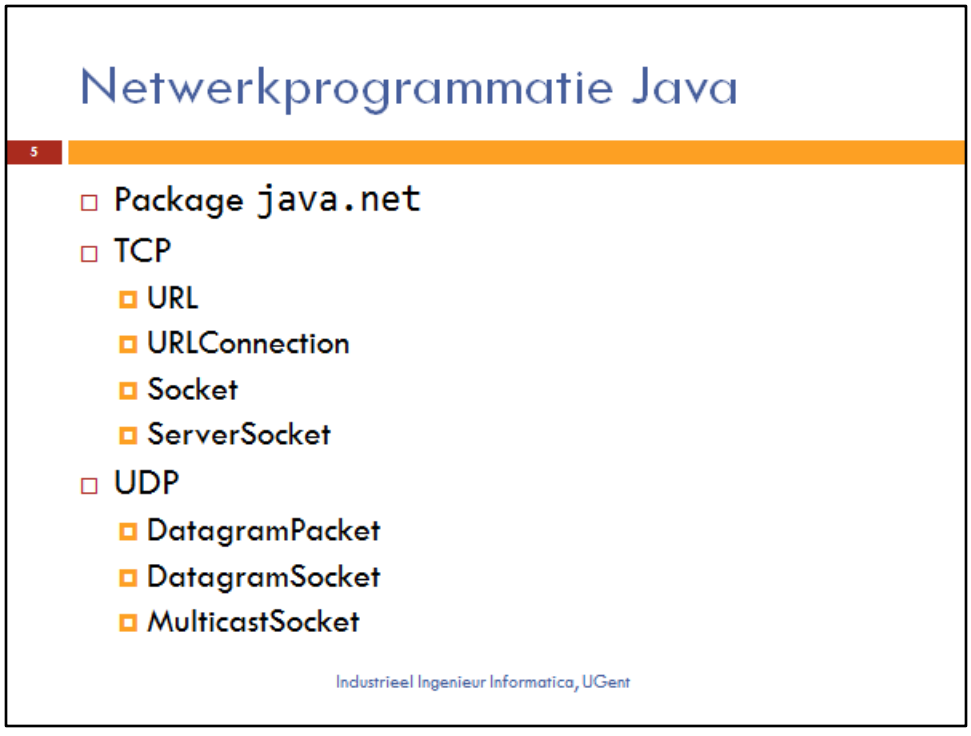
Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port.

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.

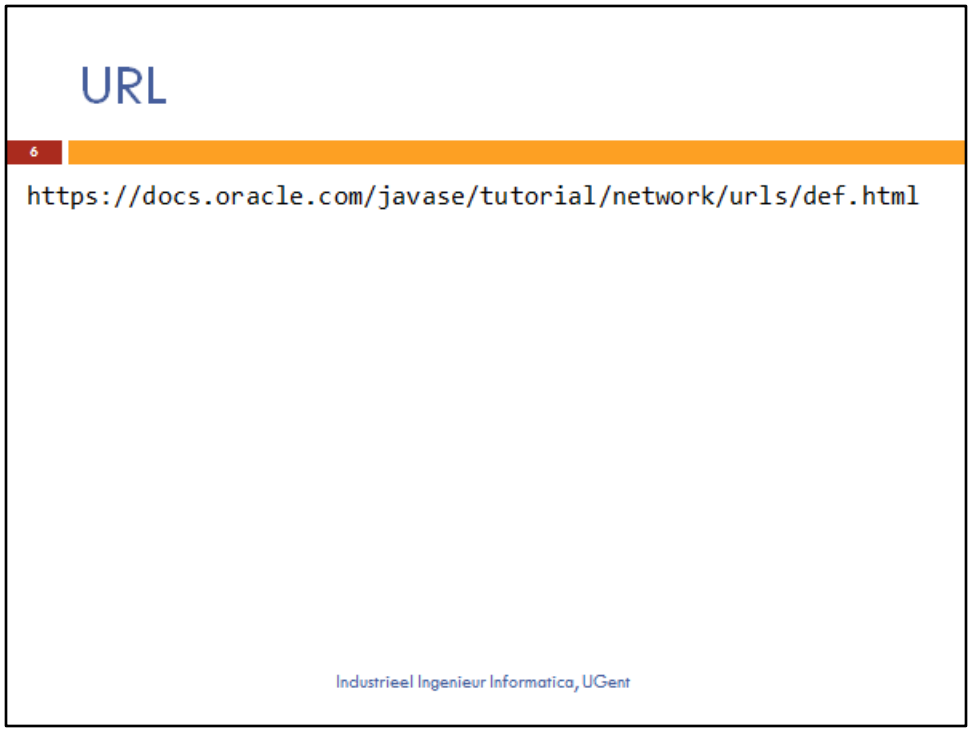
Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved

for use by well-known services such as HTTP and FTP and other system services. These ports are called *well-known ports*. Your applications should not attempt to bind to them.



Networking Classes in the JDK

Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet. The URL, URLConnection, Socket, and ServerSocket classes all use TCP to communicate over the network. The DatagramPacket, DatagramSocket, and MulticastSocket classes are for use with UDP.



URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.
You provide URLs to your favorite Web browser so that it can locate files on the Internet in the same way that you provide addresses on letters so that the post office can locate your correspondents.

A URL has two main components:
Protocol identifier: For the URL http://example.com, the protocol identifier is http.
Resource name: For the URL http://example.com, the resource name is example.com.

The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

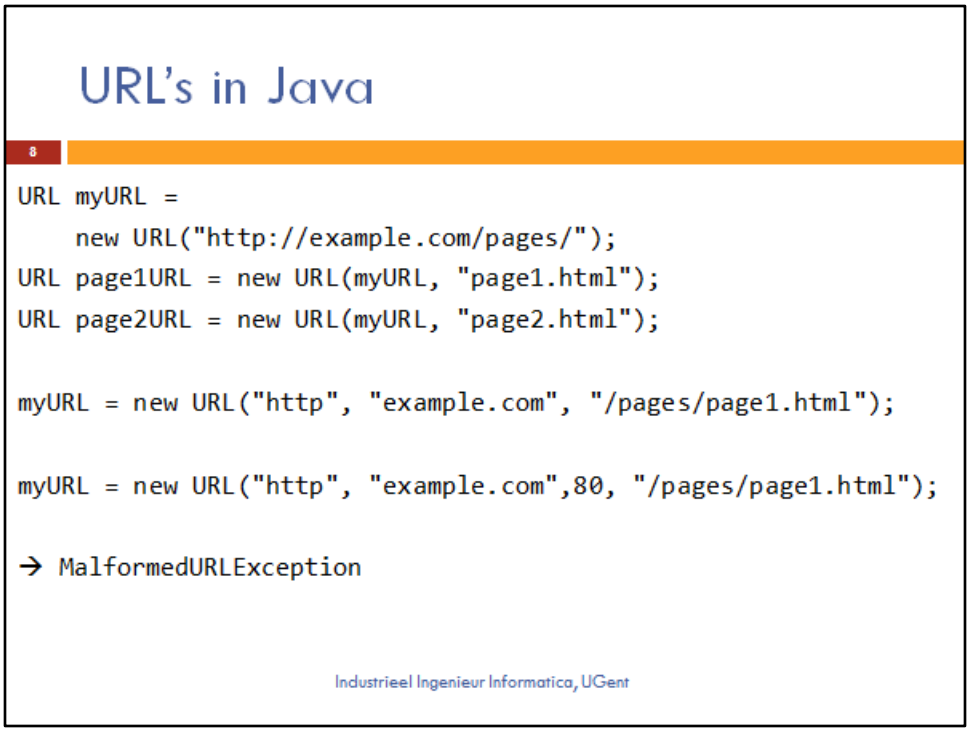
The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:
Host Name The name of the machine on which the resource lives.
Filename The pathname to the file on the machine.

Port Number The port number to which to connect (typically optional).
Reference A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).
For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. Java programs can use a class called URL in the java.net package to represent a URL address.

Overzicht

- Herhaling
- URL's in Java



The easiest way to create a URL object is from a String that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

URL myURL = new URL("http://example.com/");

The URL object created above represents an absolute URL. An absolute URL contains all of the information necessary to reach the resource in question. You can also create URL objects from a relative URL address.

Creating a URL Relative to Another
A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL.

In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:
http://example.com/pages/page1.html
http://example.com/pages/page2.html

You can create URL objects for these pages relative to their common base URL: http://example.com/pages/ like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:
URL(URL baseURL, String relativeURL)

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. If baseURL is null, then this constructor treats relativeURL like an absolute URL specification. Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.

This constructor is also useful for creating URL objects for named anchors (also called references) within a file. For example, suppose the page1.html file has a named anchor called BOTTOM at the bottom of the file. You can use the relative URL constructor to create a URL object for it like this:

```
URL page1BottomURL = new URL(page1URL,"#BOTTOM");
```

Other URL Constructors

The URL class provides two additional constructors for creating a URL object. These constructors are useful when you are working with URLs, such as HTTP URLs, that have host name, filename, port number, and reference components in the resource name portion of the URL. These two constructors are useful when you do not have a String containing the complete URL specification, but you do know various components of the URL.

For example, suppose you design a network browsing panel similar to a file browsing panel that allows users to choose the protocol, host name, port number, and filename. You can construct a URL from the panel's components. The first constructor creates a URL object from a protocol, host name, and filename. The following code snippet creates a URL to the page1.html file at the example.com site:

```
new URL("http", "example.com", "/pages/page1.html");
```

This is equivalent to

```
new URL("http://example.com/pages/page1.html");
```

The first argument is the protocol, the second is the host name, and the last is the pathname of the file. Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.
The final URL constructor adds the port number to the list of arguments used in the previous constructor:

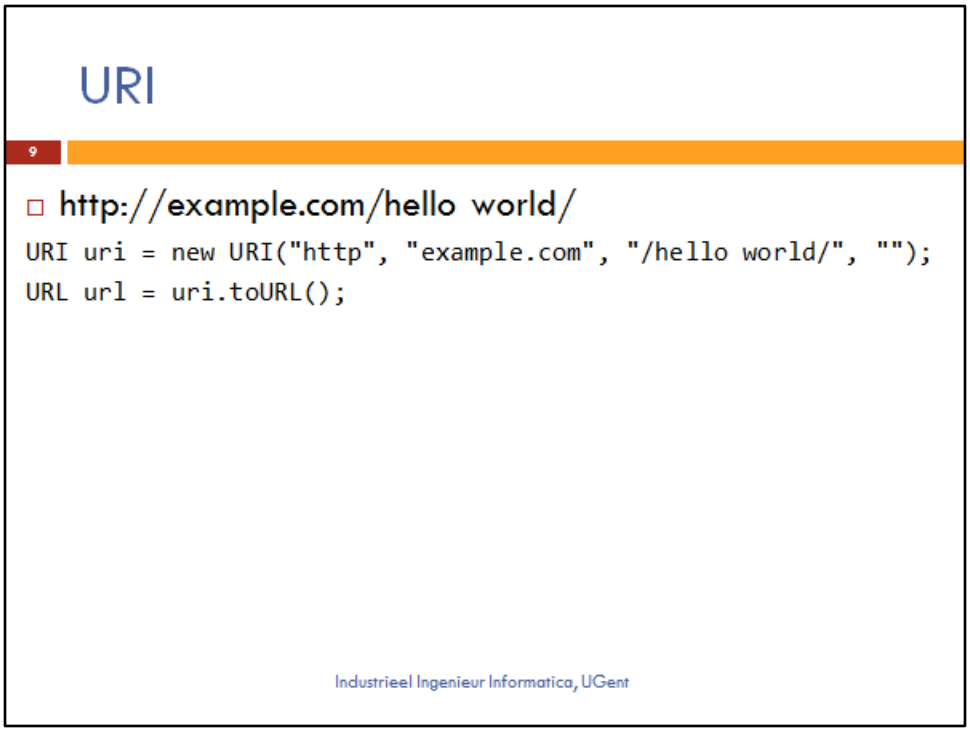
```
URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");
```

This creates a URL object for the following URL:
http://example.com:80/pages/page1.html
If you construct a URL object using one of these constructors, you can get a String containing the complete URL address by using the URL object's toString method or the equivalent toExternalForm method.

URLs are "**write-once**" objects. Once you've created a URL object, you cannot change any of its attributes (protocol, host name, filename, or port number).

Each of the four URL constructors throws a **MalformedURLException** if the arguments to the constructor refer to a null or unknown protocol. Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this:

```
try {
   URL myURL = new URL(...);
}
catch (MalformedURLException e) {
   // exception handler code here
   // ...
}
```



URL addresses with Special characters

Some URL addresses contain special characters, for example the space character. Like this:

http://example.com/hello world/

To make these characters legal they need to be encoded before passing them to the URL constructor.

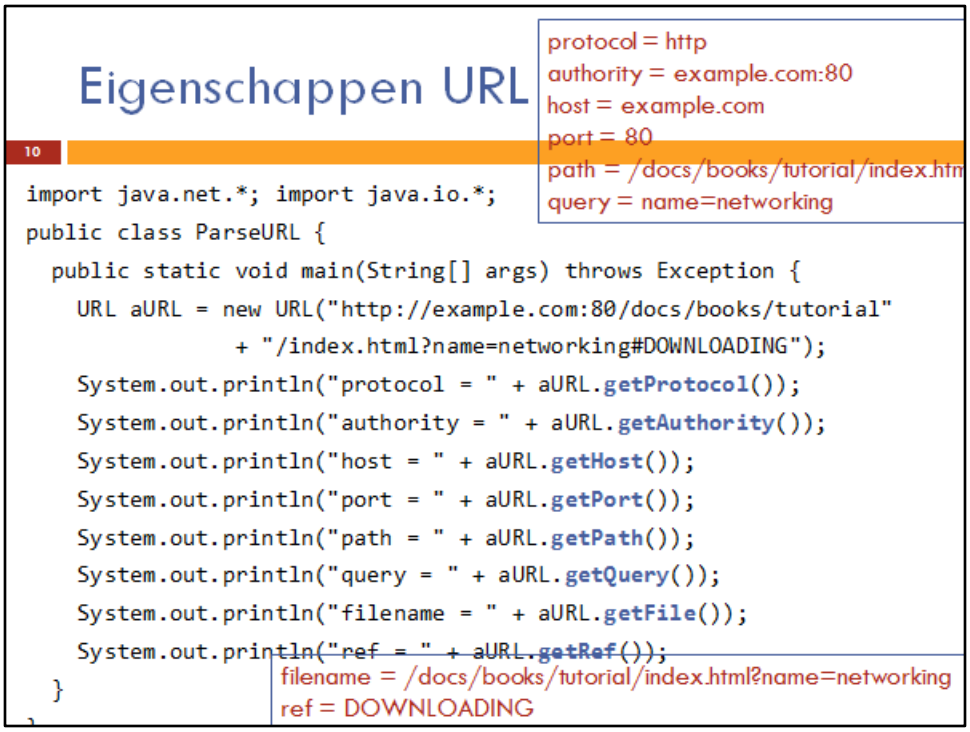
URL url = new URL("http://example.com/hello%20world");

Encoding the special character(s) in this example is easy as there is only one character that needs encoding, but for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the java.net.URI class to automatically take care of the encoding for you.

URI uri = new URI("http", "example.com", "/hello world/", "");

And then convert the URI to a URL.

URL url = uri.toURL();



The URL class provides several methods that let you query URL objects. You can get the protocol, authority, host name, port number, path, query, filename, and reference from a URL using these accessor methods:
getProtocol Returns the protocol identifier component of the URL.
getAuthority Returns the authority component of the URL.
getHost Returns the host name component of the URL.
getPort Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
getPath Returns the path component of this URL.
getQuery Returns the query component of this URL.
getFile Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.
getRef Returns the reference component of the URL.

Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric. You can use these get*XXX* methods to get information about the URL regardless of the constructor that you used to create the URL object.
The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need.

Lezen van een URL

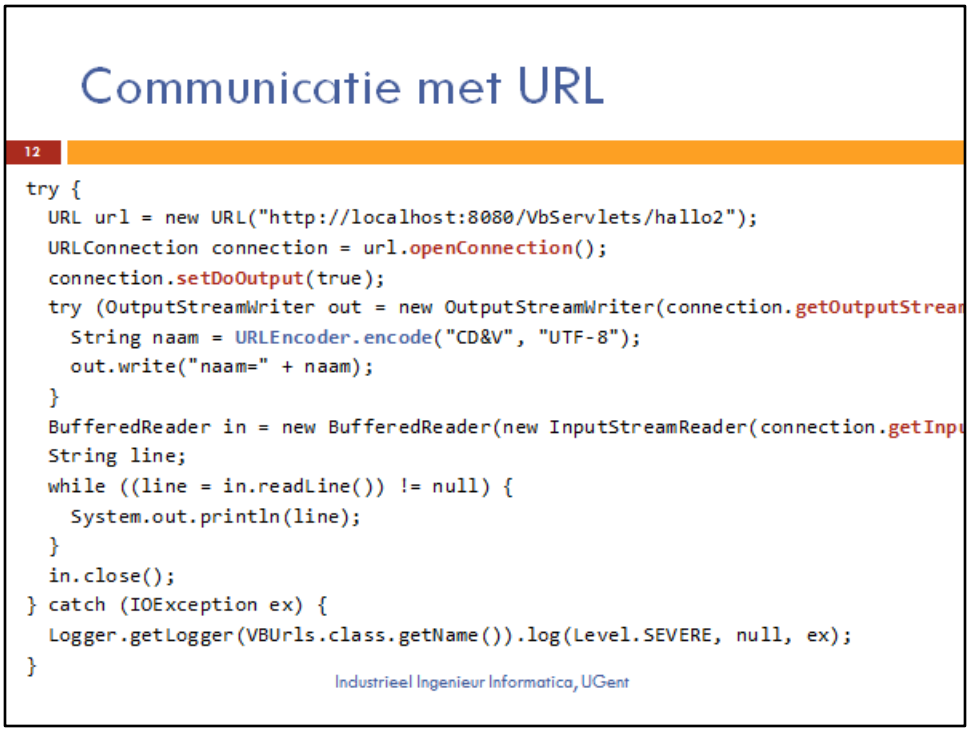
11

```
try {
  URL ugent = new URL("http://www.ugent.be/");
  try (BufferedReader in = new BufferedReader(
    new InputStreamReader(ugent.openStream()))) {
      String inputLine;
      while ((inputLine = in.readLine()) != null) {
        System.out.println(inputLine);
      }
  } catch (IOException ex) {
    Logger.getLogger(
      VBUrls.class.getName()).log(Level.SEVERE, null, ex);
  }
} catch (MalformedURLException ex) {
  Logger.getLogger(
    VBUrls.class.getName()).log(Level.SEVERE, null, ex);
}
```

Industrieel Ingenieur Informatica, UGent

After you've successfully created a URL, you can call the URL's openStream() method to get a stream from which you can read the contents of the URL. The openStream() method returns a java.io.InputStream object, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses openStream() to get an input stream on the URL http://www.ugent.be/. It then opens a BufferedReader on the input stream and reads from the BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream.



The URLConnection class contains many methods that let you communicate with the URL over the network. URLConnection is an HTTP-centric class; that is, many of its methods are useful only when you are working with HTTP URLs. However, most URL protocols allow you to read from and write to the connection.

Writing to a URLConnection

Many HTML pages contain *forms* — text fields and other GUI objects that let you enter data to send to the server. After you type in the required information and initiate the query by clicking a button, your Web browser writes the data to the URL over the network. At the other end the server receives the data, processes it, and then sends you a response, usually in the form of a new HTML page.

Many of these HTML forms use the HTTP POST METHOD to send data to the server. Thus writing to a URL is often called *posting to a URL*. The server recognizes the POST request and reads the data sent from the client.

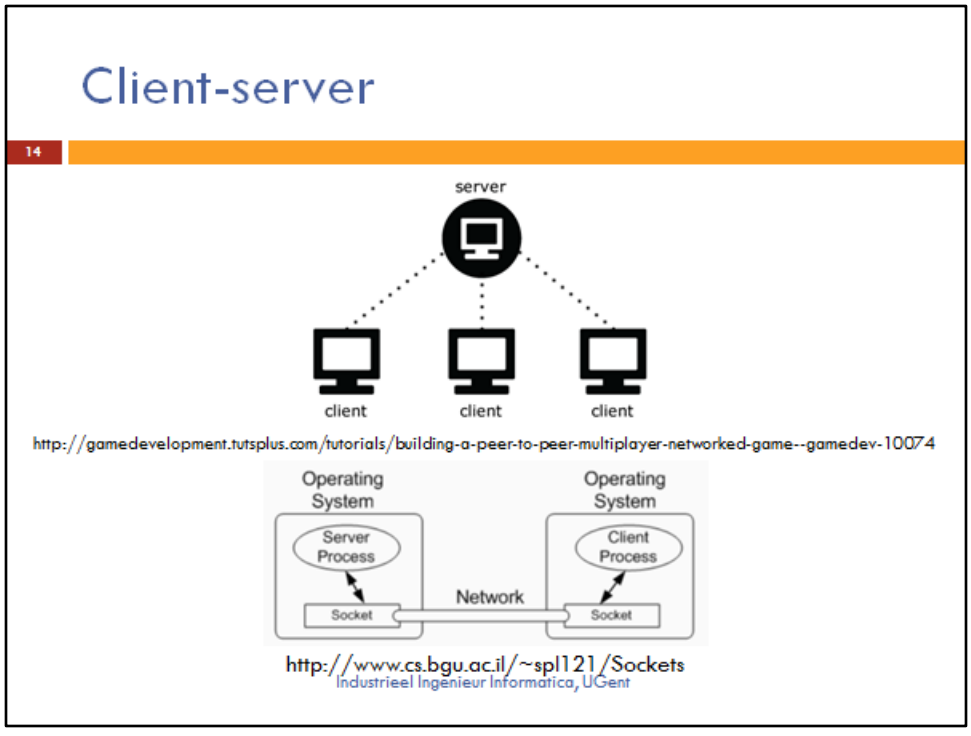
For a Java program to interact with a server-side process it simply must be able to write to a URL, thus providing data to the server. It can do this by following these steps:

Create a URL.
Retrieve the URLConnection object.
Set output capability on the URLConnection.
Open a connection to the resource.

Get an output stream from the connection.
Write to the output stream.
Close the output stream.

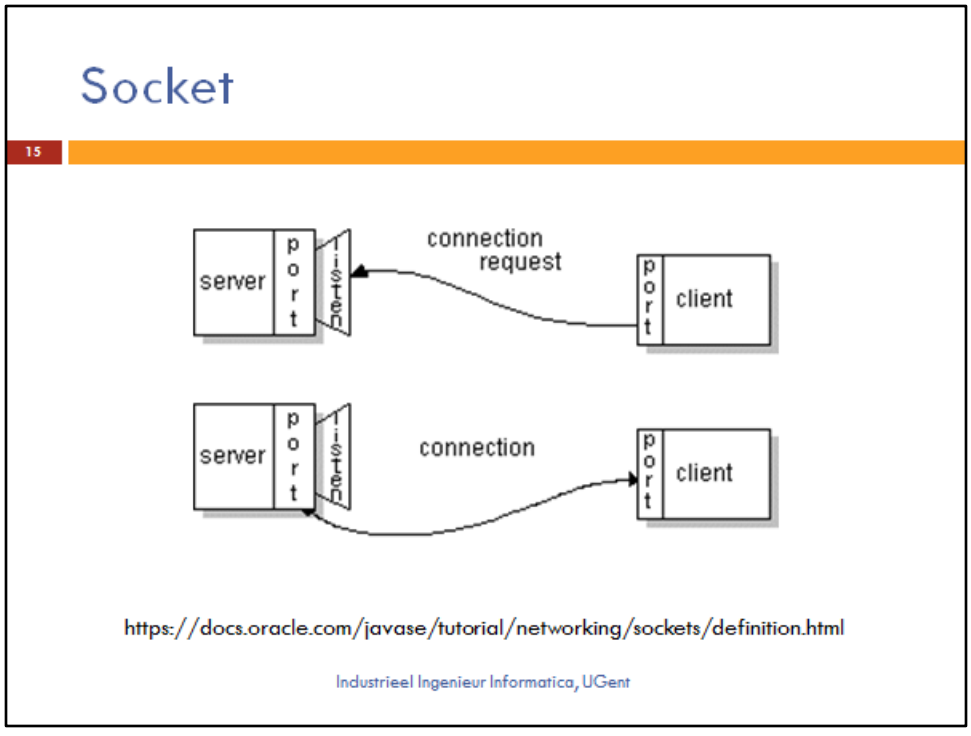
Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie



URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application. In **client-server applications**, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it. TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

A **socket** is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.



Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.
On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

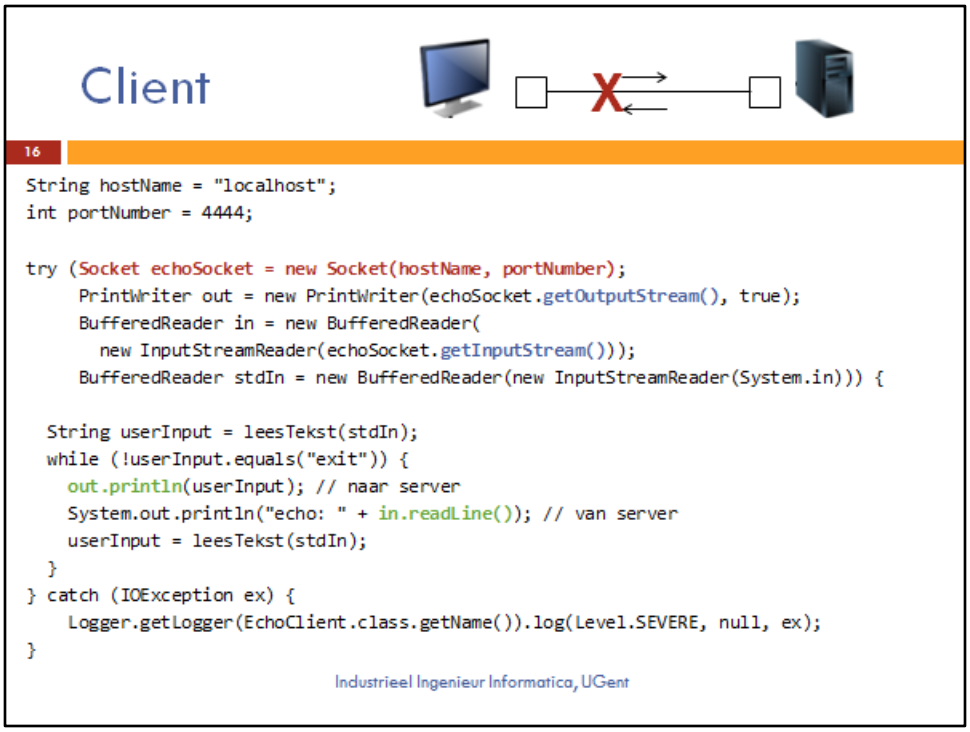
If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
The client and server can now communicate by writing to or reading from their sockets.

A *socket* is one **endpoint** of a **two-way communication** link between two programs

running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.
An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The java.net package in the Java platform provides a class, **Socket**, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion. Additionally, java.net includes the **ServerSocket** class, which implements a socket that servers can use to listen for and accept connections to clients.



Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, EchoClient, that connects to an echo server. The echo server receives data from its client and echoes it back.
The EchoClient example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.
Note that the EchoClient example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

The statements in the try-with-resources statement in the EchoClient example are critical. These lines establish the socket connection between the client and the server and open a PrintWriter and a BufferedReader on the socket.
The first statement in the try-with resources statement creates a new Socket object and names it echoSocket. The Socket constructor used here requires the name of the computer and the port number to which you want to connect. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect.

The second statement in the try-with resources statement gets the socket's output stream and opens a PrintWriter on it. Similarly, the third statement gets the socket's input stream and opens a BufferedReader on it. The example uses readers and writers so that it can write Unicode characters over the socket.
To send data through the socket to the server, the EchoClient example needs to write to the PrintWriter. To get the server's response, EchoClient reads from the BufferedReader object stdIn, which is created in the fourth statement in the try-with resources statement.

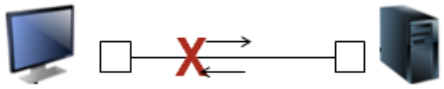
The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the PrintWriter connected to the socket.
The second statement in the while loop reads a line of information from the BufferedReader connected to the socket. The readLine method waits until the server echoes the information back to EchoClient. When readline returns, EchoClient prints the information to the standard output.

The while loop continues until the user types “exit”. That is, the EchoClient example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until the user types “exit”. The while loop then terminates, and the Java runtime automatically closes the readers and writers connected to the socket and to the standard input stream, and it closes the socket connection to the server. The Java runtime closes these resources automatically because they were created in the try-with-resources statement. The Java runtime closes these resources in reverse order that they were created. (This is good because streams connected to a socket should be closed before the socket itself is closed.)

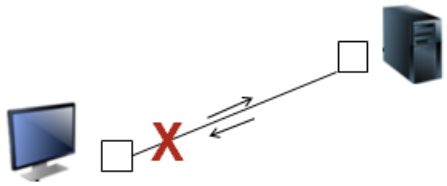
This client program is straightforward and simple because the echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:
Open a socket.
Open an input stream and output stream to the socket.
Read from and write to the stream according to the server's protocol.
Close the streams.
Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

Server

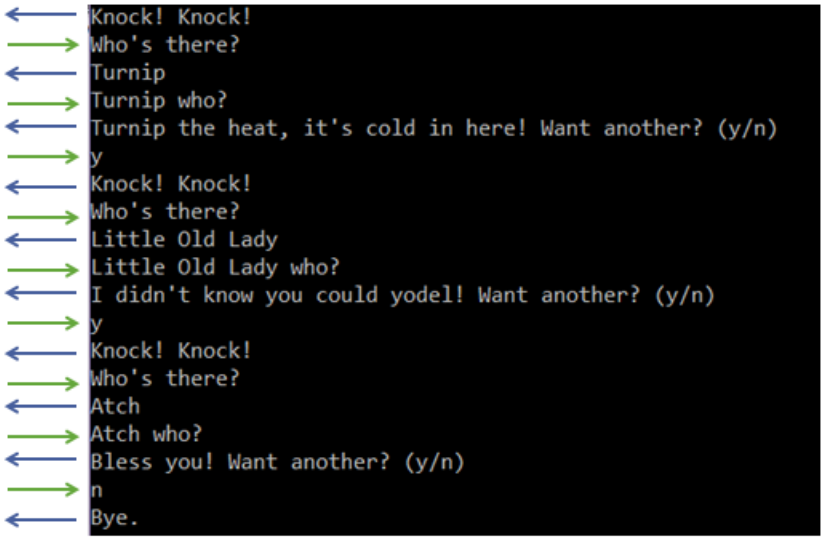


Server



KnockKnockServer

19



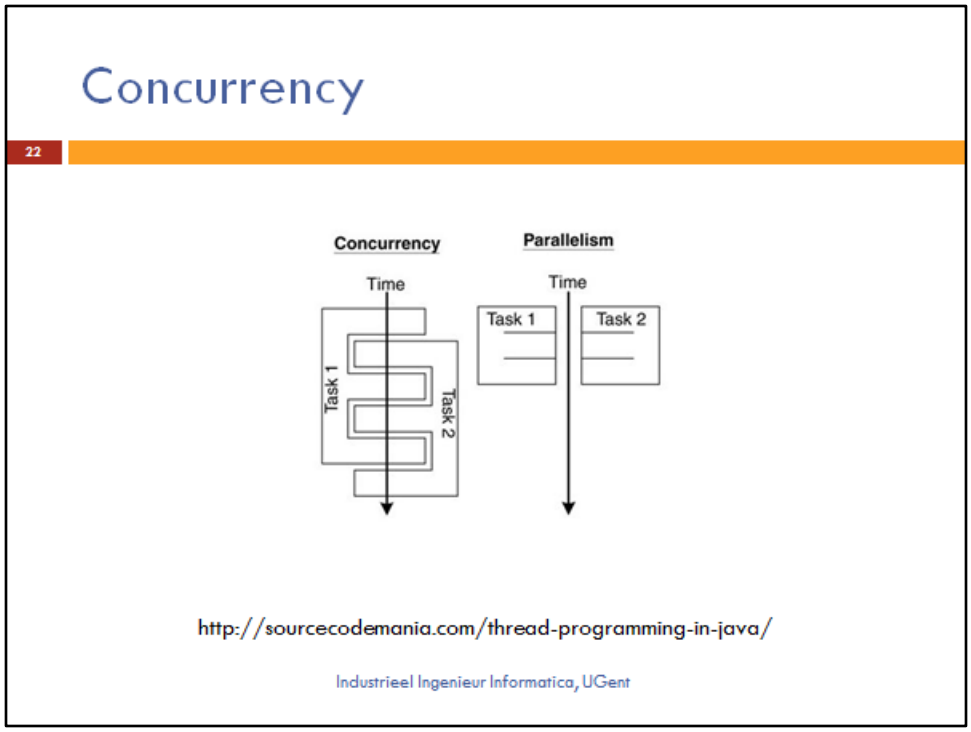
KnockKnockServer

20

```
try {
  ServerSocket serverSocket = new ServerSocket(4444);
  while (true) {
    try (Socket clientSocket = serverSocket.accept())
      try (PrintWriter out
           = new PrintWriter(clientSocket.getOutputStream(), true);
         BufferedReader in = new BufferedReader(
          new InputStreamReader(clientSocket.getInputStream()))){
       String inputLine, outputLine;
       while ((inputLine = in.readLine()) != null) {
         outputLine = kkp.processInput(inputLine);
         out.println(outputLine);
         if (outputLine.equals("Bye.")) {break;}
       }
    } …
```

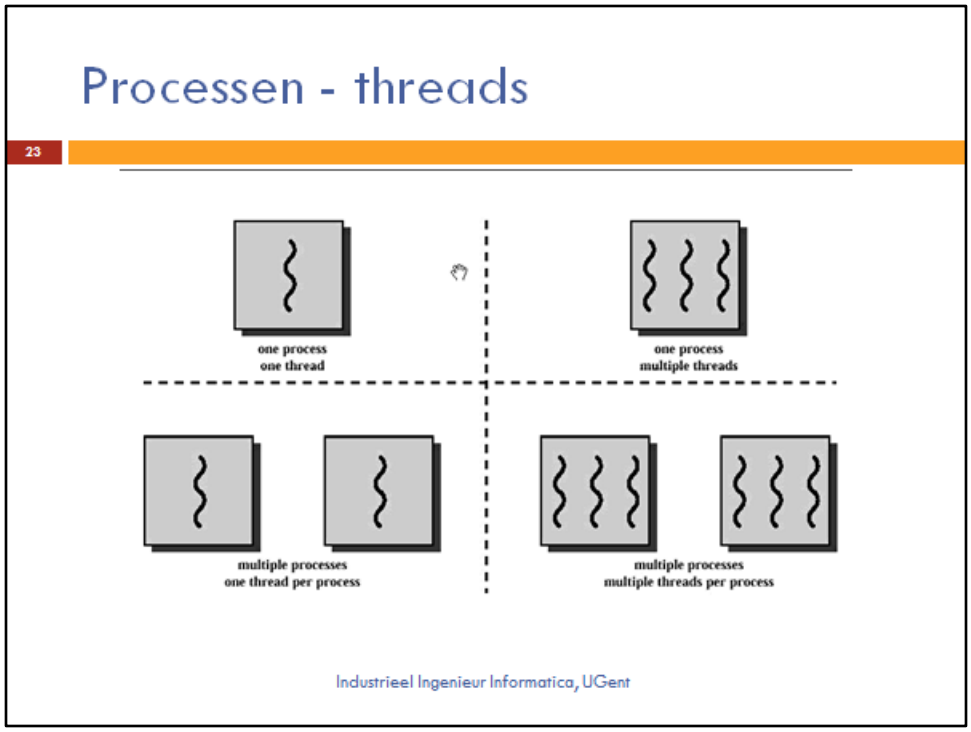
Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency



Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as *concurrent* software.

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the java.util.concurrent packages.



In concurrent programming, there are two basic units of execution: processes and threads. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating

processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

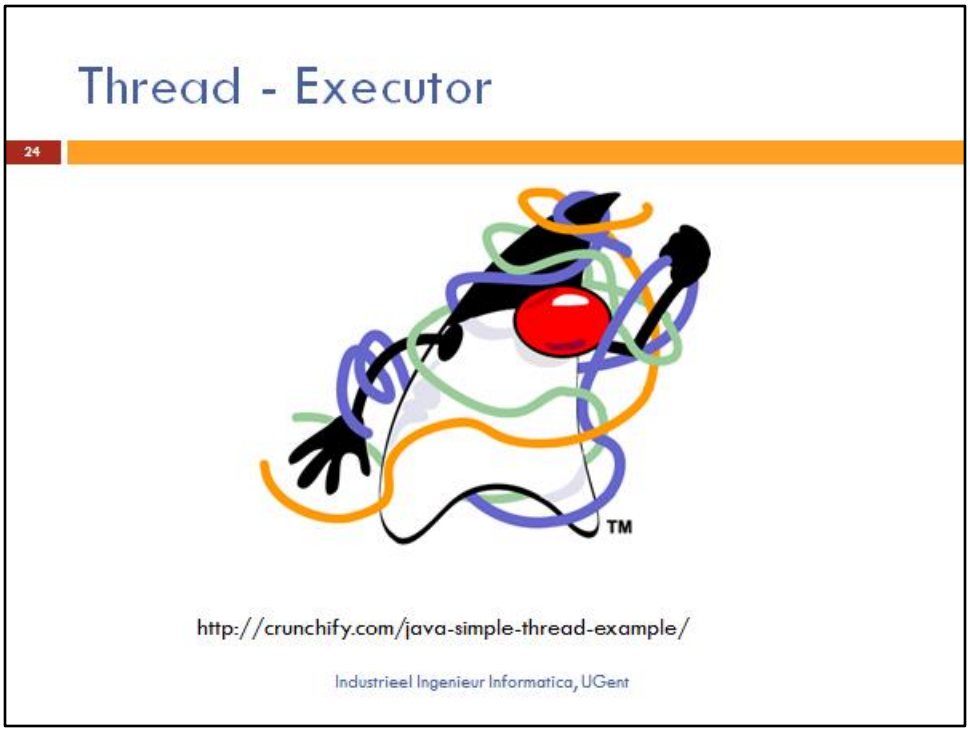
Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object.

Threads

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads.



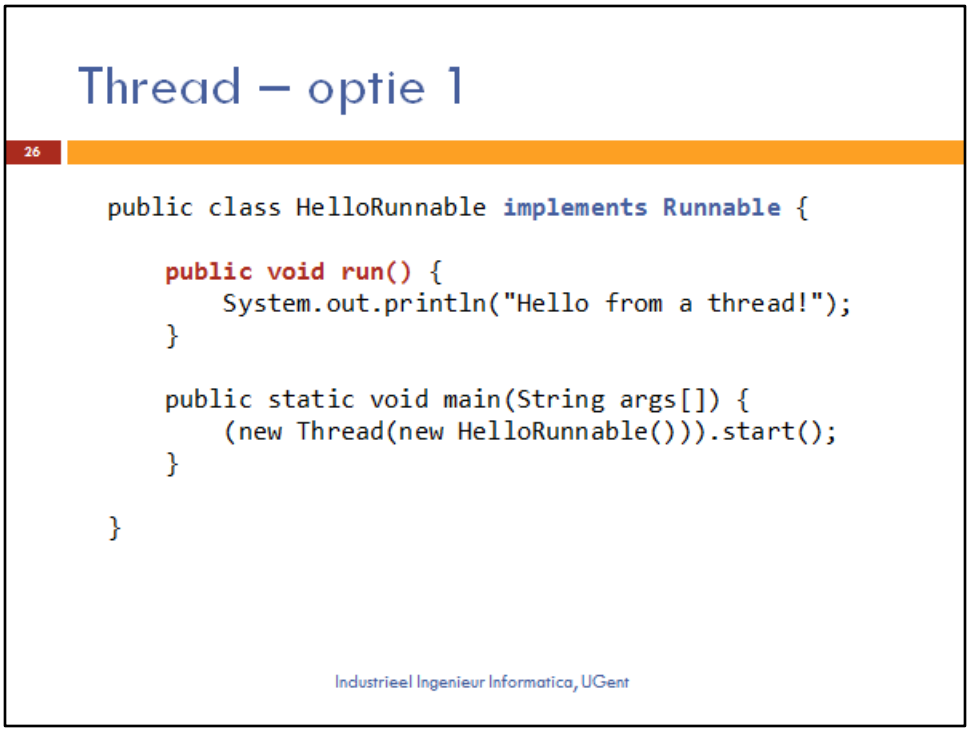
Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.

To abstract thread management from the rest of your application, pass the application's tasks to an executor.

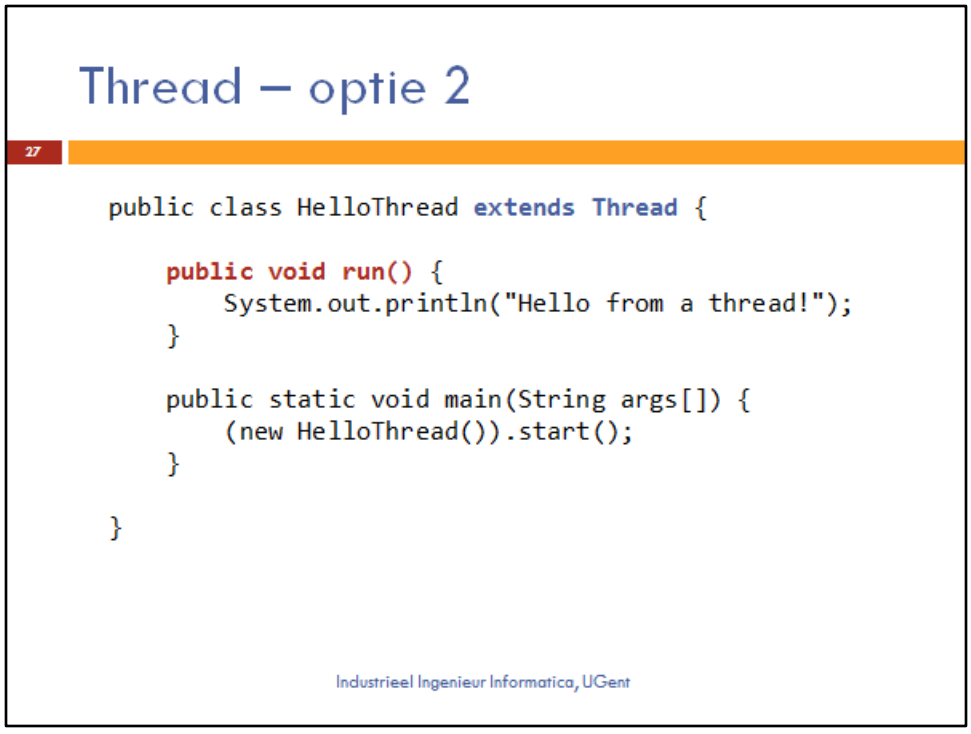
Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads



An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

Provide a Runnable object. The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor.



Subclass Thread. The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run.

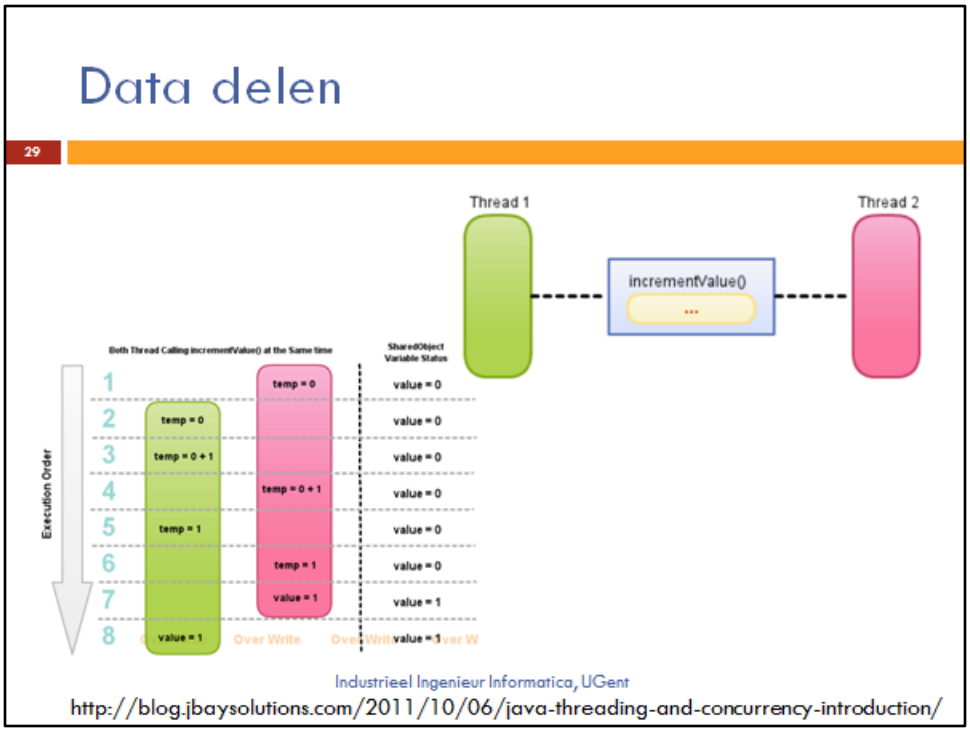
Notice that both examples invoke Thread.start in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.

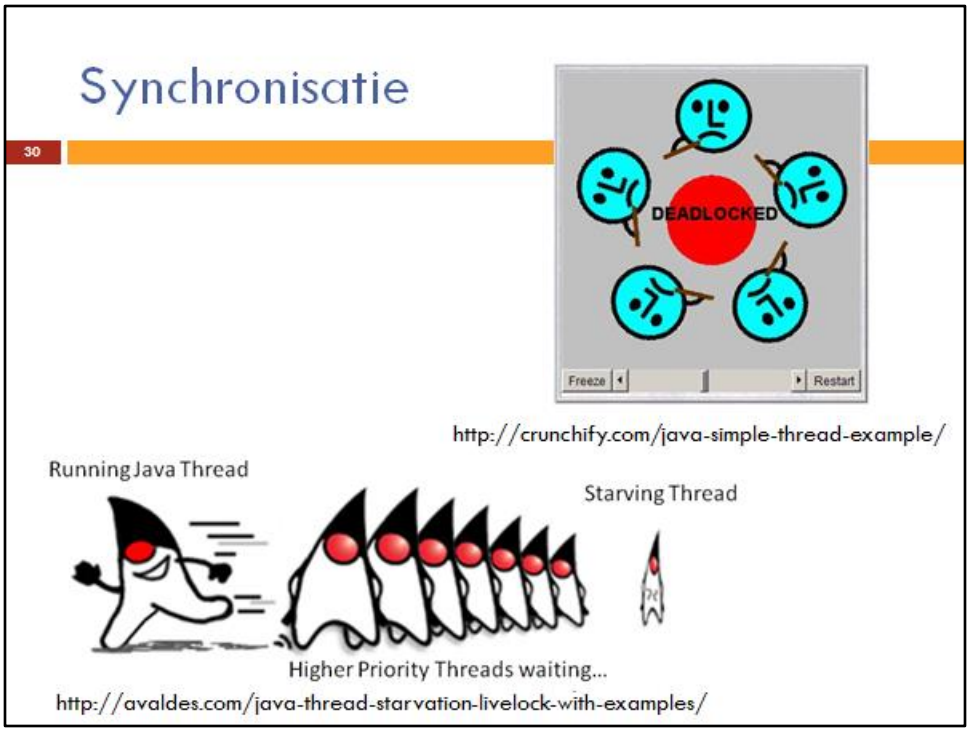
The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and Thread object: sleep, interrupt, join, …

Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie



Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.



However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Deadlock, starvation and livelock are forms of thread contention.

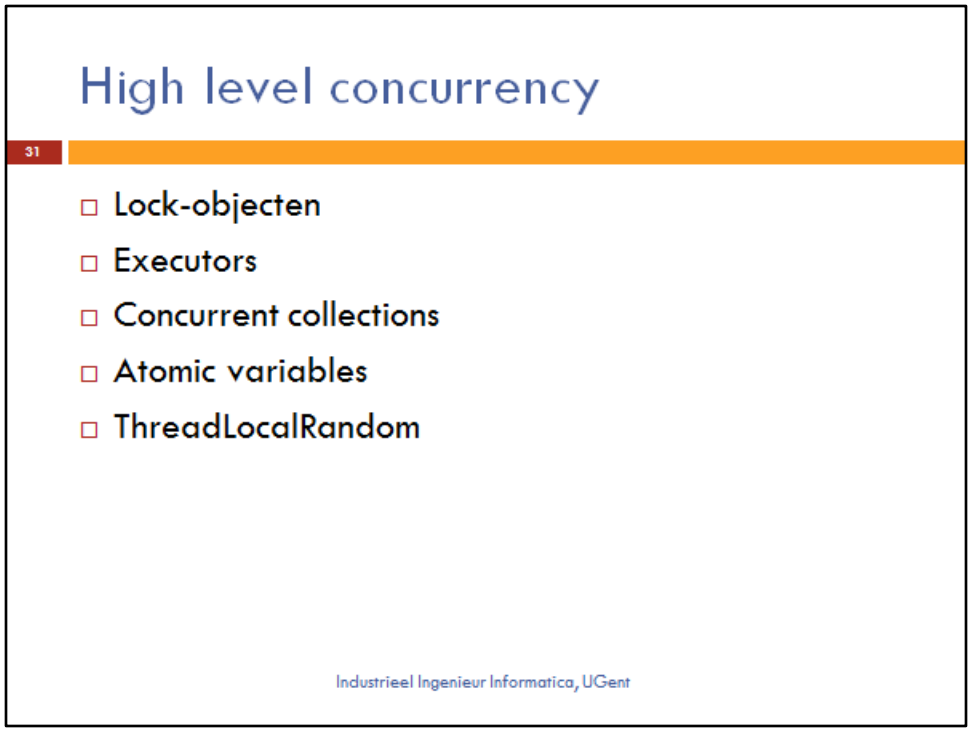
Thread Interference: errors are introduced when multiple threads access shared data.

Memory Consistency Errors: errors that result from inconsistent views of shared memory.

Synchronized Methods: a simple idiom that can effectively prevent thread interference and memory consistency errors.

Implicit Locks and Synchronization: a more general synchronization idiom, synchronization is based on implicit locks.

Atomic Access: operations that can't be interfered with by other threads, volatile instance variables.



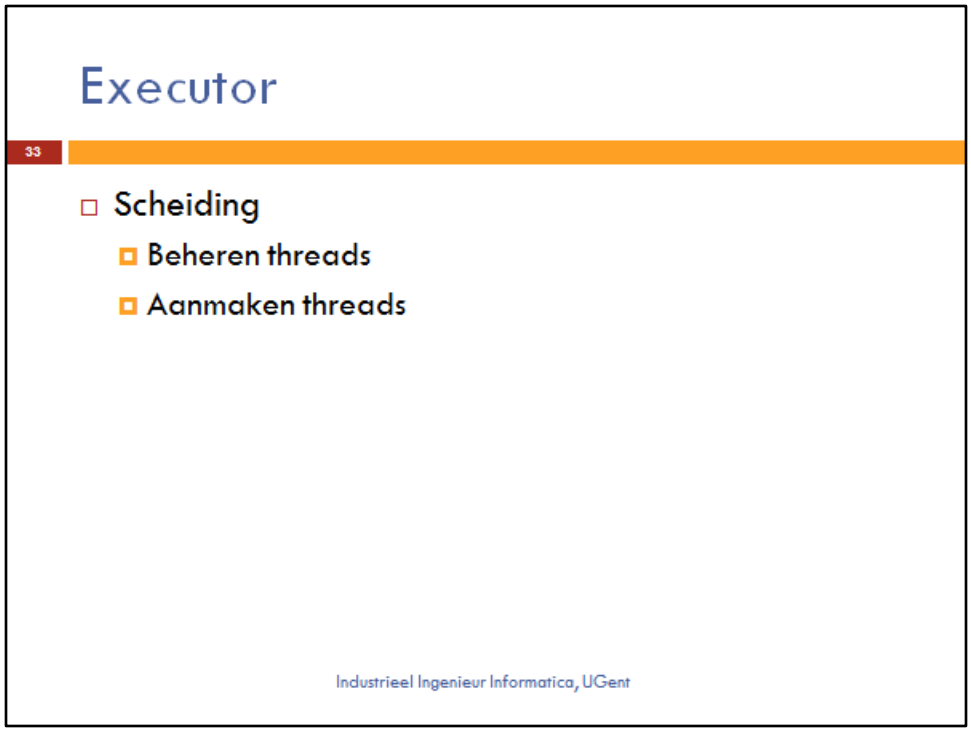
So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new java.util.concurrent packages. There are also new concurrent data structures in the Java Collections Framework.

Lock objects support locking idioms that simplify many concurrent applications.
Executors define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
ThreadLocalRandom (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

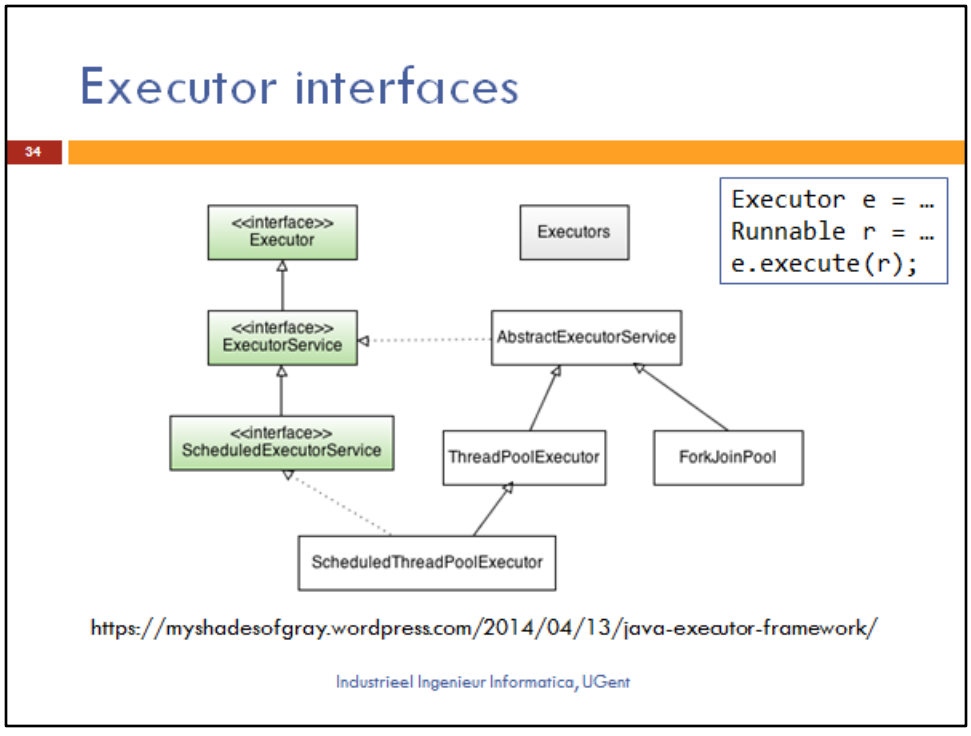
Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie
  - Executor



In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

Executor Interfaces define the three executor object types.
Thread Pools are the most common kind of executor implementation.
Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.



The java.util.concurrent package defines three executor interfaces:

Executor, a simple interface that supports launching new tasks.
ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

The Executor Interface
The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();

with

e.execute(r);

However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

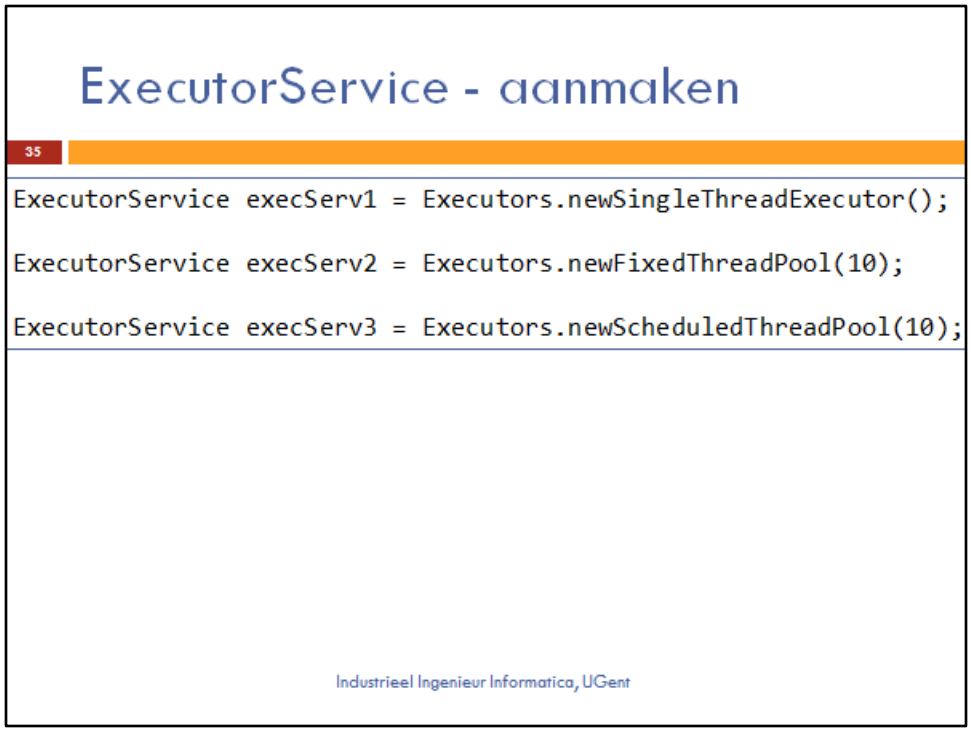
The ExecutorService Interface

The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

The ScheduledExecutorService Interface

The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.



How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too.

ExecutorService - gebruik

36

```
executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
```

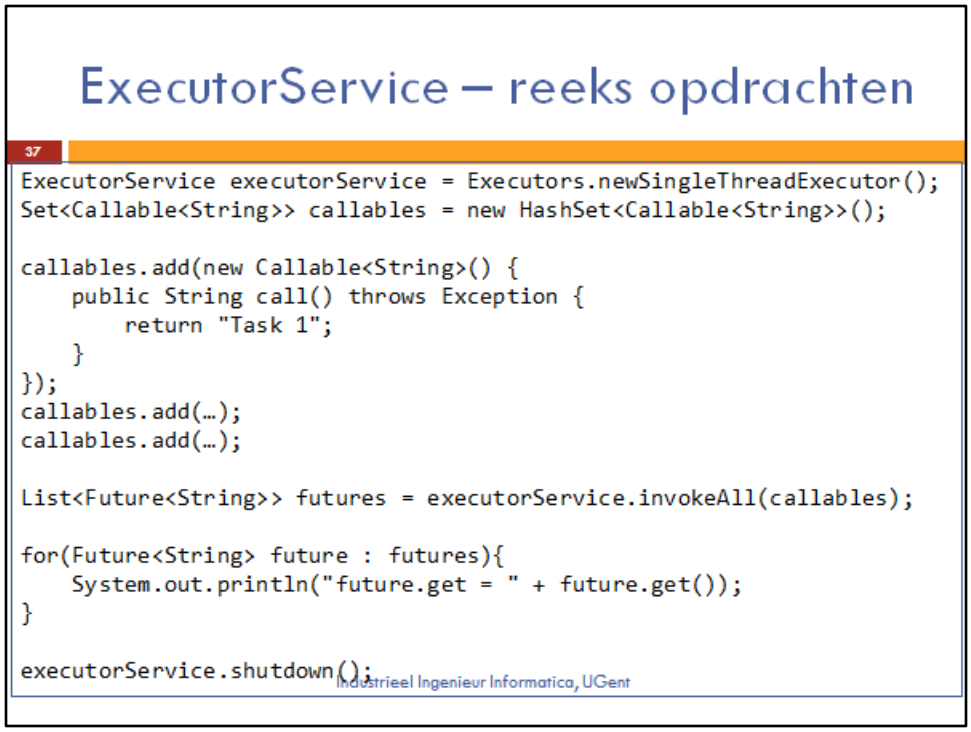
```
executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
```

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.
The Callable's result can be obtained via the Future object returned by the submit(Callable) method.



The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.
Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.
For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.
To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.
If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

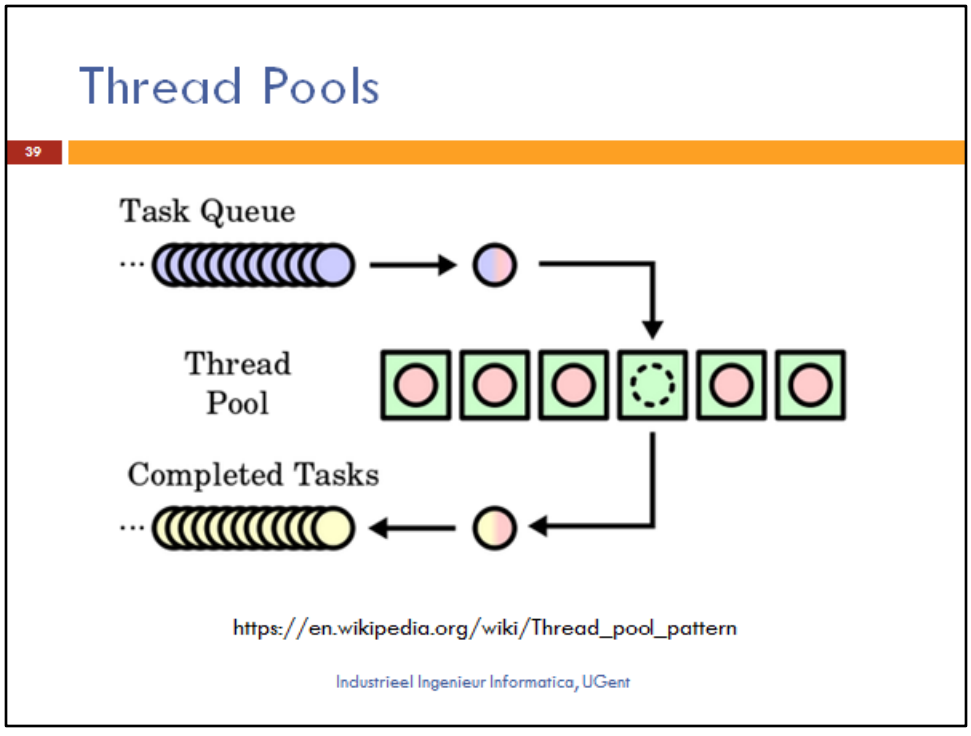
ExecutorService – reeks opdrachten

38

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(…);
callables.add(…);

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();
```

Industrieel Ingenieur Informatica, UGent

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.
If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.



Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be

servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

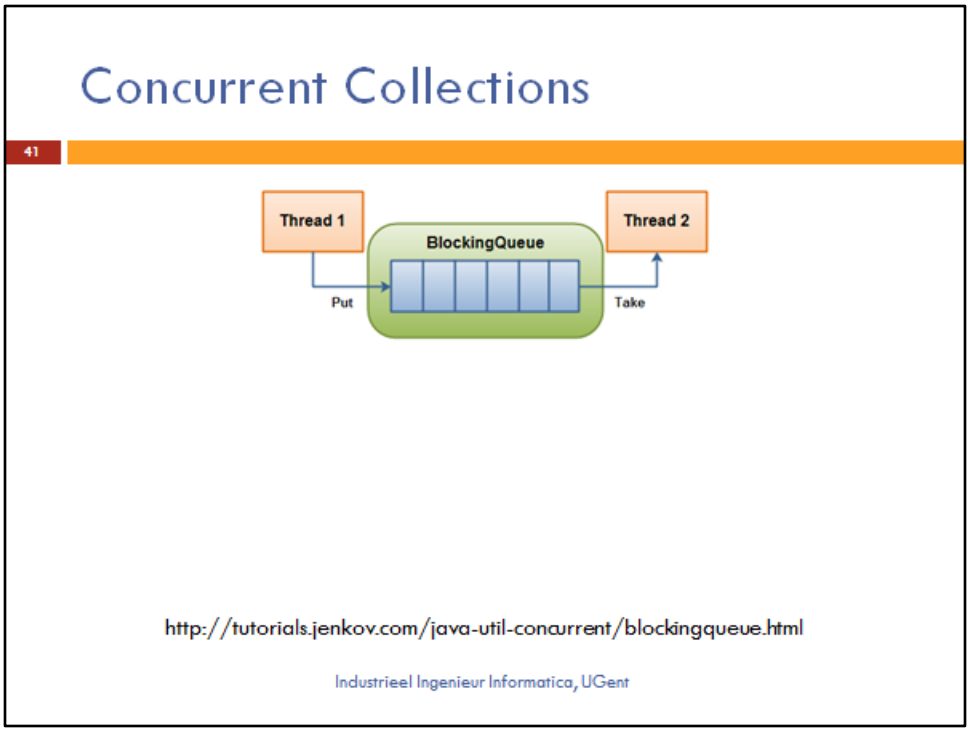
A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors This class also provides the following factory methods:

The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
The newSingleThreadExecutor method creates an executor that executes a single task at a time.
Several factory methods are ScheduledExecutorService versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor will give you additional options.

Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie
  - Executor
  - Concurrent Collections



The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.

ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.

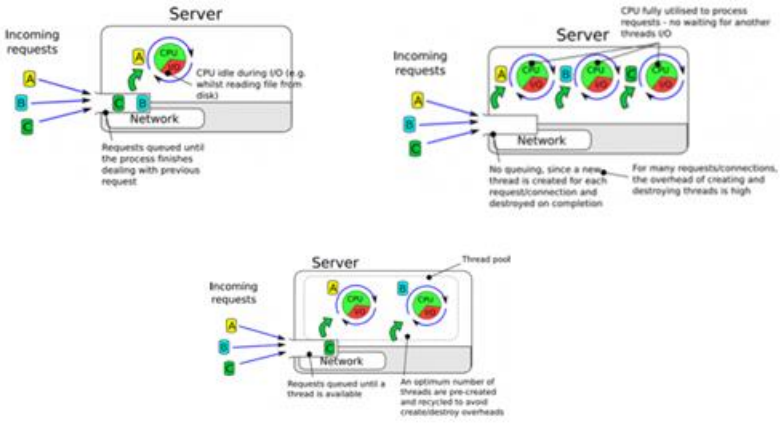
ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

Overzicht

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie
  - Executor
  - Concurrent Collections
- Multithreaded Server

Singlethreaded Server



https://myshadesofgray.wordpress.com/2014/04/13/java-executor-framework/

KnockKnockServer

44

```
int portNumber = 9999;
boolean listening = true;
try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
  while (listening) {
    new KnockKnockThread(serverSocket.accept()).start();
  }
} catch (IOException e) {
  Logger.getLogger(
    KnockKnockMultiServer.class.getName()).log(Level.SEVERE,
       null, e);
  System.err.println("Could not listen on port " + portNumber);
  throw new RuntimeException(e);
}
```

Industrieel Ingenieur Informatica, UGent

This example describes a simple multithreaded server implemented in Java. The code is based on the singlethreaded KnockKnockServer. The main difference is the server loop. Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that will process the request.

Note: This code uses a "thread per connection" design which most of us originally thought less efficient than a thread pooled server.

Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that processes the request. That way the thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call. That way the risk is minimized for clients being denied access to the server because the listening thread is not inside the accept() call.

Multithreaded Server Advantages
The advantages of a multithreaded server compared to a singlethreaded server are summed up below:

- Less time is spent outside the accept() call.
- Long running client requests do not block the whole server

As mentioned earlier the more time the thread calling serverSocket.accept() spends inside this method call, the more responsive the server will be. Only when the listening thread is inside the accept() call can clients connect to the server. Otherwise the clients just get an error or have to wait.

In a singlethreaded server long running requests may make the server unresponsive for a long period. This is not true for a multithreaded server, unless the long-running request takes up all CPU time time and/or network bandwidth.

KnockKnockServer - Thread

45

```
public class KnockKnockThread extends Thread {
  private Socket socket = null;
  public KnockKnockThread(Socket socket) {
    super("KKMultiServerThread");
    this.socket = socket;
  }
  @Override
  public void run() {
    try (
      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));)
      {…}
      socket.close();
```

KnockKnockServer - Executor

46

```
int portNumber = 9999;
boolean listening = true;
try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
  ExecutorService execServ = Executors.newFixedThreadPool(10);
  while (listening) {
    try {
      Socket clientSocket = serverSocket.accept();
      execServ.submit(new KnockKnockRunnable(clientSocket));
    } catch (IOException ex) {… }
  }
} catch (IOException e) {
  Logger.getLogger(
    KnockKnockPool.class.getName()).log(Level.SEVERE, null, e);
  System.err.println("Could not listen on port " + portNumber);
  throw new RuntimeException(e);
}
```

Industrieel Ingenieur Informatica, UGent

This example describes a simple thread pooled server implemented in Java. The code is based on the multithreaded KnockKnock server. The main difference is the server loop. Rather than starting a new thread per incoming connection, the connection is wrapped in a Runnable and handed off to a thread pool with a fixed number of threads. The Runnable's are kept in a queue in the thread pool. When a thread in the thread pool is idle it will take a Runnable from the queue and execute it.

Rather than starting a new thread per incoming connection, the KnockKnockRunnable is passed to the thread pool for execution when a thread in the pool becomes idle.

Thread Pooled Server Advantages

The advantages of a thread pooled server compared to a multithreaded server is that you can control the maximum number of threads running at the same time. This has certain advantages.

First of all if the requests require a lot of CPU time, RAM or network bandwidth, this may slow down the server if many requests are processed at the same time. For instance, if memory consumption causes the server to swap memory in and out of disk, this will result in a serious performance penalty. By controlling the maximum number of threads you can minimize the risk of resource depletion, both due to limiting the memory taken by the processing of the requests, but also due to the limitation and reuse of the threads. Each thread take up a certain amount of memory too, just to represent the thread itself.

Additionally, executing many requests concurrently will slow down all requests processed. For instance, if you process 1.000 requests concurrently and each request takes 1 second, then all requests will take 1.000 seconds to complete. If you instead queue the requests up and process them say 10 at a time, the first 10 requests will complete after 10 seconds, the next 10 will complete after 20 seconds etc. Only the last 10 requests will complete after 1.000 seconds. This gives a better service to the clients.