

# UDP Programming

RES, Lecture 4

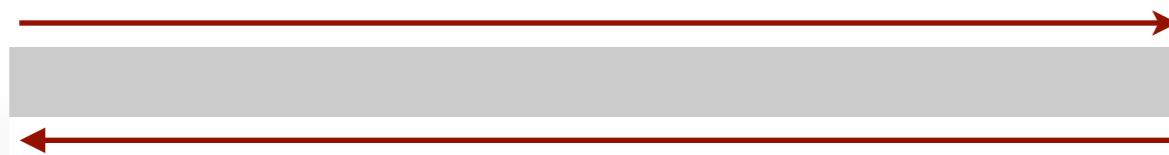
---

Olivier Liechti

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

	<b>Monday</b>	<b>Tuesday</b>	<b>Friday</b>
<b>UDP</b> Docker Music (challenge)	10.4	11.4	14.4
<b>Docker</b> Docker Music (challenge)	24.4	25.4	28.4
	<b>1.5 TE</b>	<b>2.5 challenge</b>	<b>5.5 challenge</b>
<b>HTTP</b> Code HTTP client	8.5	9.5	12.5
<b>HTTP</b> HTTP infrastructure	15.5	<b>16.5 challenge</b>	<b>19.5 challenge</b>
<b>No lecture</b> HTTP infrastructure	22.5	23.5	26.5
<b>HTTP</b> HTTP infrastructure	29.5	30.5	2.6
	5.6	6.6	9.6
Présentations HTTP infra	<b>12.6 challenge</b>	13.6	16.6



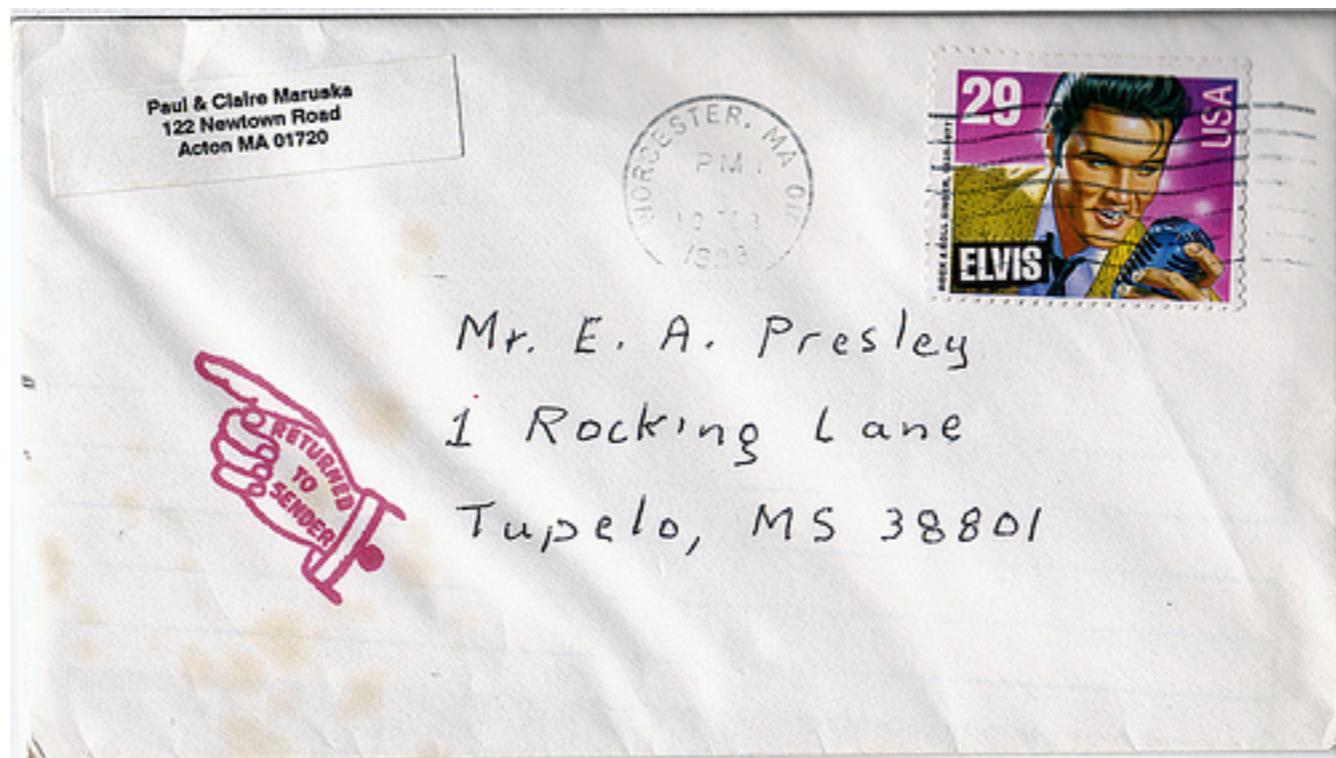
TCP



UDP



# The UDP Protocol



# The UDP Protocol

- When using UDP, **you do not work with the abstraction of an IO stream.**
- Rather, **you work with the abstraction of individual datagrams**, which you can send and receive. Every datagram is independent from the others.
- You **do not have any guarantee**: datagrams can get lost, datagrams can arrive out-of-order, datagrams can be duplicated.
- What do you put “**inside**” the datagram (i.e. what is the payload)?
  - A notification.
  - A request, a query, a command. A reply, a response, a result.
  - A portion of a data stream (managed by the application).
- What do you put “**outside**” of the datagram (i.e. what is the header)?
  - A destination address (IP address in the IP packet header + UDP port)
  - A source address (IP + port)

# Unicast, Broadcast, Multicast

---

## Destination Address

192.168.10.2



255.255.255.255



239.255.22.5

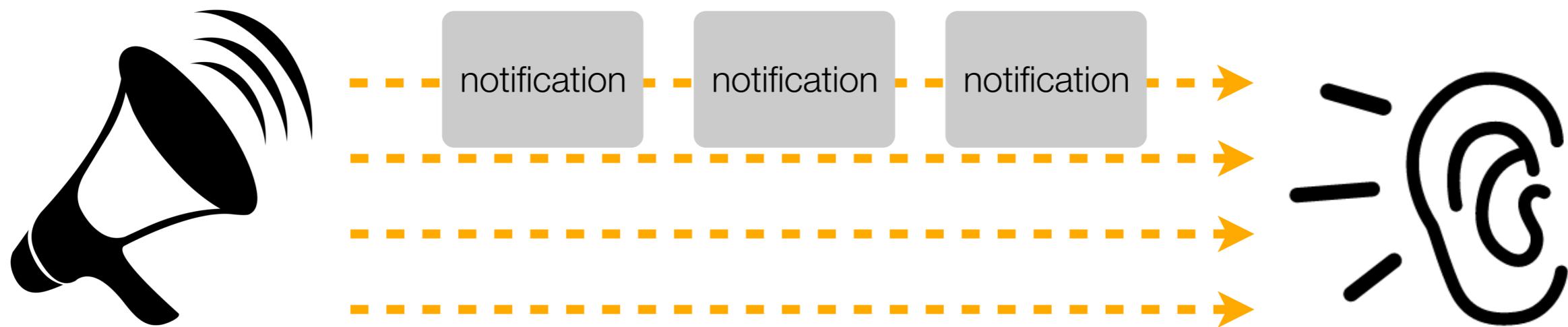


# Messaging Patterns



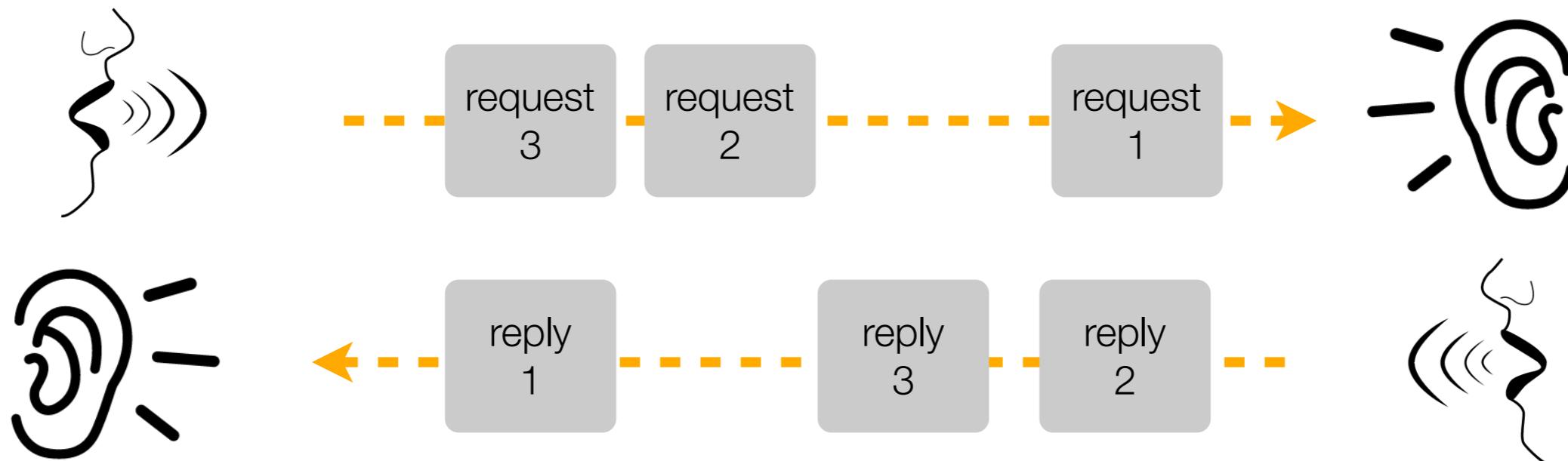
# Pattern: Fire-and-Forget

- The sender generates messages, sends them to a single receiver or to a group of receivers.
- The sender may do that on a **periodic basis**.
- The sender does not expect any answer. **He is telling, not asking.**



# Pattern: Request-Reply

- This pattern is used to implement the **typical client-server model**. Both the client and the server **produce and consume** datagrams.
- **The client produces requests** (aka queries, commands). The client also listens for incoming replies (aka responses, results). The server listens for requests. **The server also sends replies back to the client**.
- Can the client send a new request, even if he has not received the response to the previous request yet? If yes, and because UDP datagrams can be delivered out of order, how can the client **associate a reply with the corresponding request**?



# Service Discovery Protocols

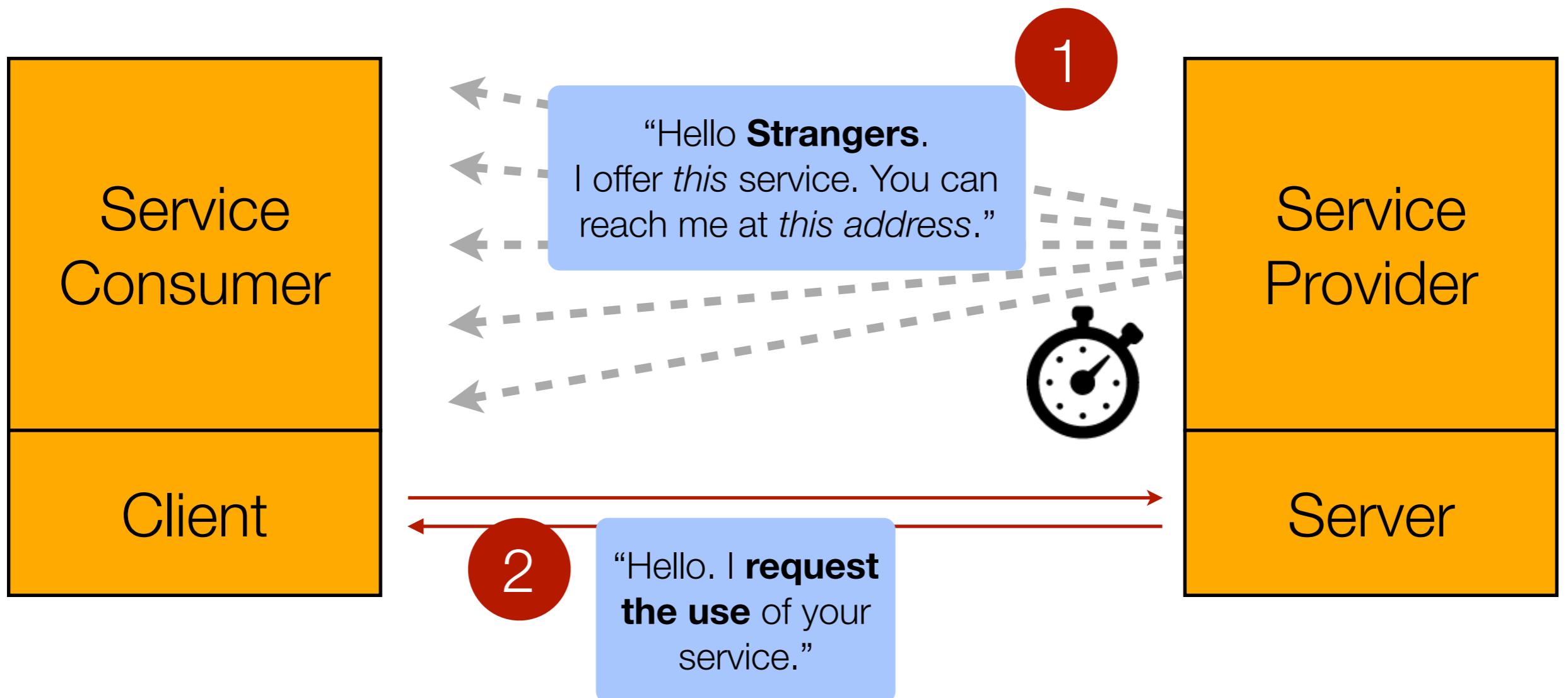


# Service Discovery Protocols

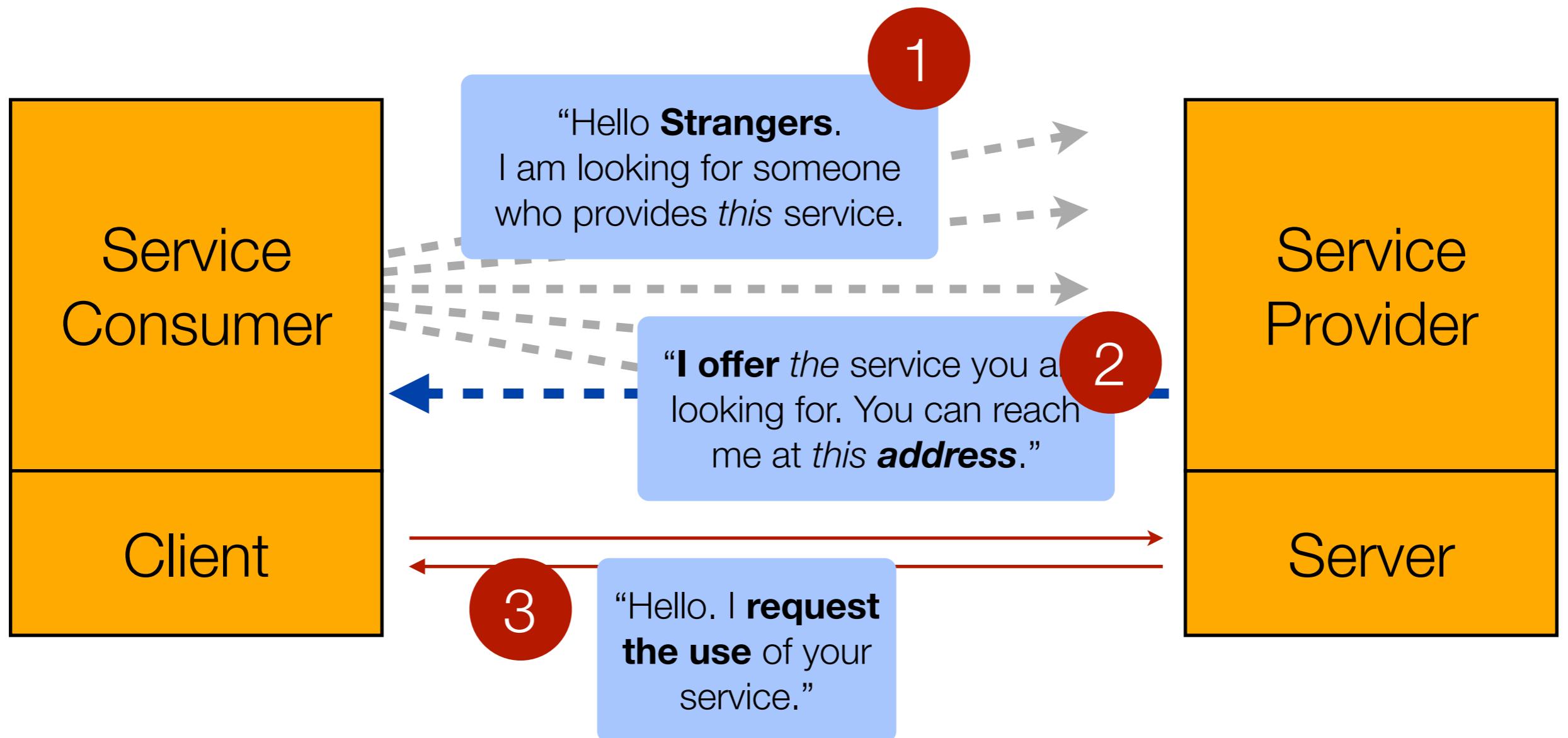
- With **unicast** message transmission, the sender must know who the receiver is and must know how to reach it (i.e. know its address).
- With **broadcast** and **multicast**, this is not required. The sender knows that nodes that are either *nearby*, or that have expressed *their interest*, will receive a copy of the message.
- This property can be used to **discover the availability of services or resources** in a *dynamic environment*.



# Model 1: Advertise Service Periodically



# Model 2: Query Service Availability



# Reliability



# Reliability

- UDP is **not a reliable transport protocol**:
  - Datagrams can arrive in a different order from which they were sent.
  - Datagrams can get lost.
- For some application-level protocols, this is not an issue because they are **tolerant to data loss**. For example, think of a **media streaming** protocol.
- But **if no data loss can be tolerated at the application level** (which is typically the case for file transfer protocols), does it mean that it is impossible to use UDP?

# Who Is Responsible for Reliability?

- It is actually possible to implement reliable application-level protocols on top of UDP, but...
  - It is the **responsibility of the application-level protocol specification** to include appropriate mechanisms to overcome the limitations of UDP.
  - It is the **responsibility of the application developer** to implement these mechanisms.
- In other words, it is up to the application developer to “**do the kind of stuff that TCP usually does**”.
- Do you remember about **acknowledgments, timers, retransmissions, stop-and-wait, sliding windows**, etc?

# A First Example: TFTP

- **Trivial File Transfer Protocol (TFTP)**
  - <http://tools.ietf.org/html/rfc1350>
  - “TFTP is a very simple protocol used to transfer files. It is from this that its name comes, Trivial File Transfer Protocol or TFTP. **Each nonterminal packet is acknowledged separately.** This document describes the protocol and its types of packets. The document also explains the reasons behind some of the design decisions.”
  - “Any transfer begins with a request to read or write a file, which also serves to request a connection. If the server grants the request, the connection is opened and the file is sent in fixed length blocks of 512 bytes. Each data packet contains one block of data, and must be acknowledged by an acknowledgment packet **before** the next packet can be sent.”

# A Second Example: CoAP

- **Constrained Application Protocol (CoAP)**

- <http://tools.ietf.org/html/draft-ietf-core-coap-18>
- “The Constrained Application Protocol (CoAP) is a **specialized web transfer protocol** for use with constrained nodes and constrained (e.g., low-power, lossy) networks. [...] The protocol is designed for **machine-to-machine (M2M) applications** such as smart energy and building automation.”
- “CoAP provides a **request/response interaction model** between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URLs and Internet media types.”
- “**2.1. Messaging Model.** The CoAP messaging model is based on the exchange of messages over UDP between endpoints. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. This message format is shared by requests and responses. The CoAP message format is specified in Section 3. **Each message contains a Message ID used to detect duplicates and for optional reliability.** (The Message ID is compact; its 16-bit size enables up to about 250 messages per second from one endpoint to another with default protocol parameters.)
- “**4.2. Messages Transmitted Reliably.** The reliable transmission of a message is initiated by marking the message as Confirmable in the CoAP header. [...] A recipient MUST **acknowledge** a Confirmable message with an Acknowledgement message [...].”

# Using the Socket API for a UDP Sender

1. Create a datagram socket (without giving any address / port)
2. Create a datagram and put some data (bytes) in it
3. Send the datagram via the socket, specifying the destination address and port

**If a reply is expected:**

4. Accept incoming datagrams on the socket



If we do not specify the port when creating the socket, the operating system will **automatically assign a free one** for us.

This port will be in the “source port” field of the UDP header. The receiver of our datagram will extract this value and will use it to send us the reply (it is a part of the **return address**).

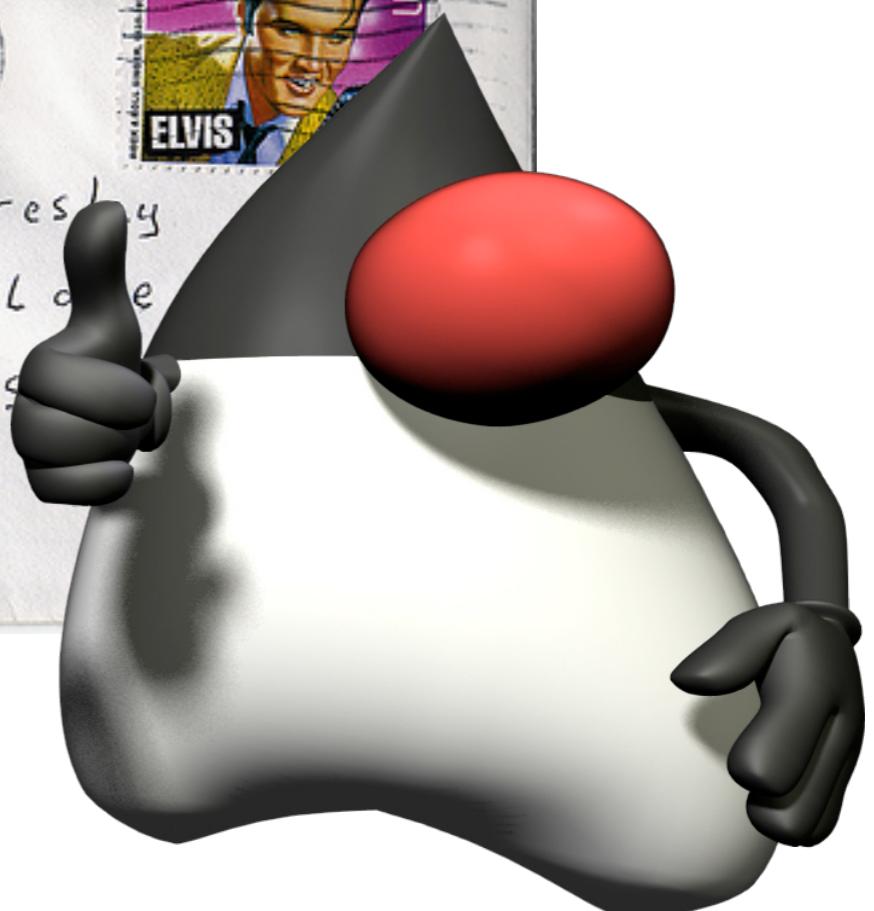
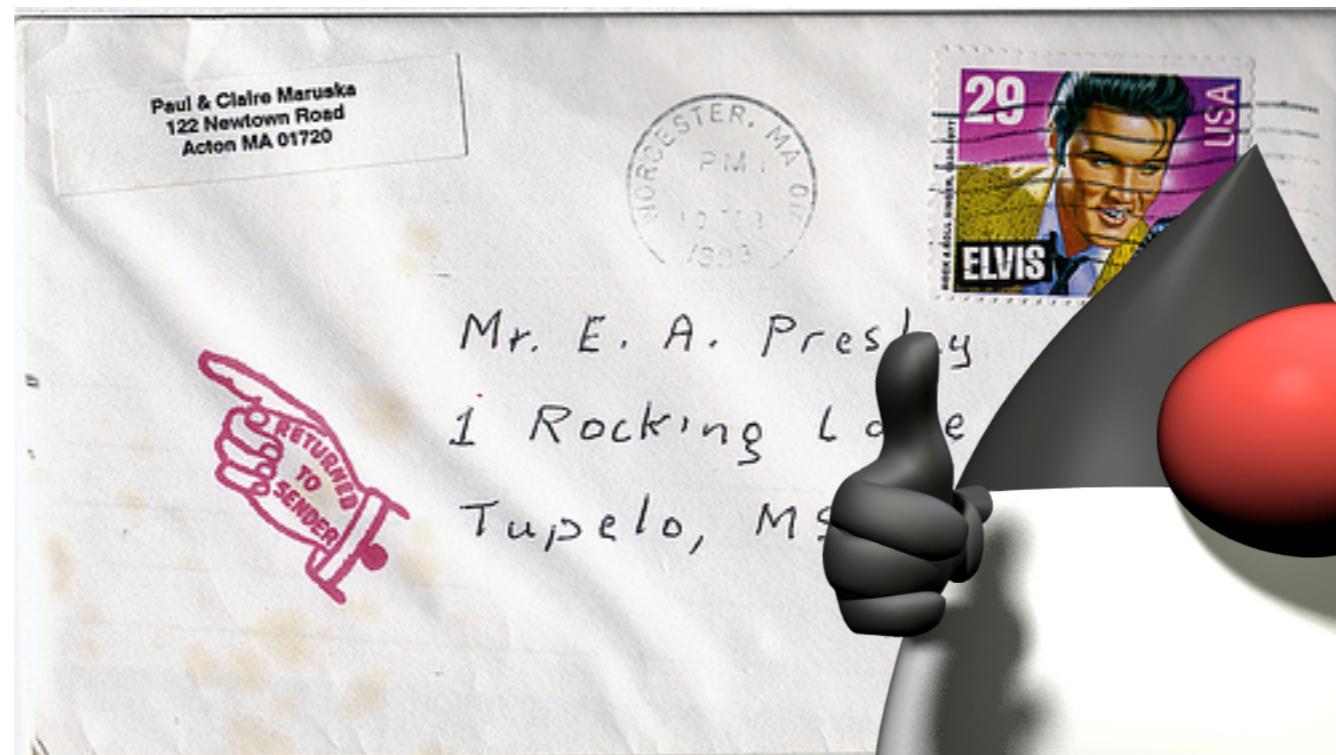
# Using the Socket API for a UDP Receiver

Application-level protocol specifications often define **a standard UDP port**, where the clients can send their requests. When this is the case, we have to specify the port number when creating the socket.

1. Create a datagram socket, specifying a particular port
2. Loop
  - 2.1. Accept an incoming datagram via the socket
  - 2.2. Process the datagram
  - 2.3. If a reply is expected by the sender
    - 2.3.1. Extract the return address from the request datagram
    - 2.3.2. Prepare a reply datagram
    - 2.3.3. Send the reply datagram via the socket

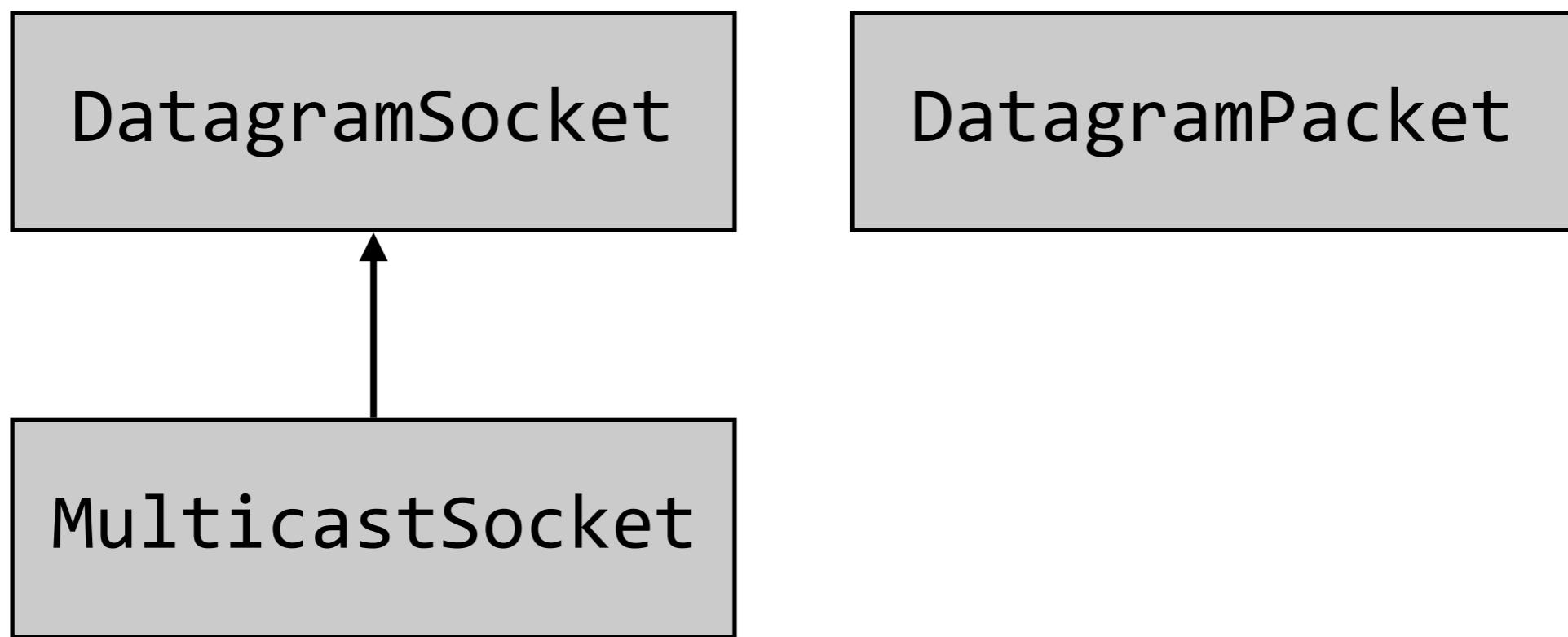
It is at this stage that we extract the source port and source IP address from the incoming datagram. We have found the **return address**, so we know where to send the response.

# Using UDP in Java



# Using UDP in Java

`java.net`



```
// Sending a message to a multicast group

socket = new DatagramSocket();
byte[] payload = "Java is cool, everybody should know!!".getBytes();

DatagramPacket datagram = new DatagramPacket(payload, payload.length,
InetAddress.getByName("239.255.3.5"), 4411);

socket.send(datagram);
```

## Publisher (multicast) Subscriber (multicast)

```
// Listening for broadcasted messages on the local network

MulticastSocket socket = new MulticastSocket(port);
InetAddress multicastGroup = InetAddress.getByName("239.255.3.5");
socket.joinGroup(multicastGroup);

while (true) {
    byte[] buffer = new byte[2048];
    DatagramPacket datagram = new DatagramPacket(buffer, buffer.length);
    try {
        socket.receive(datagram);
        String msg = new String(datagram.getData(), datagram.getOffset(), datagram.getLength());
        LOG.log(Level.INFO, "Received a datagram with this message: " + msg);
    } catch (IOException ex) {
        LOG.log(Level.SEVERE, ex.getMessage(), ex);
    }
}
socket.leaveGroup(multicastGroup);
```

```
// Broadcasting a message to all nodes on the local network

socket = new DatagramSocket();
socket.setBroadcast(true);

byte[] payload = "Java is cool, everybody should know!!".getBytes();

DatagramPacket datagram = new DatagramPacket(payload, payload.length,
InetAddress.getByName("255.255.255.255"), 4411);

socket.send(datagram);
```

## Publisher

## Subscriber

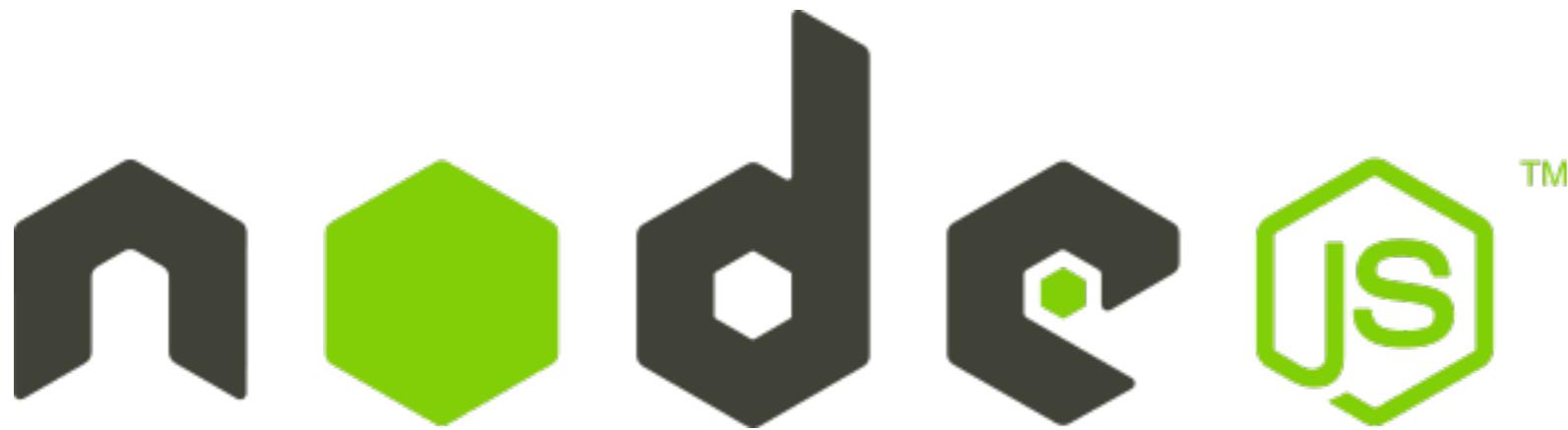
```
// Listening for broadcasted messages on the local network

DatagramSocket socket = new DatagramSocket(port);

while (true) {
    byte[] buffer = new byte[2048];
    DatagramPacket datagram = new DatagramPacket(buffer, buffer.length);
    try {
        socket.receive(datagram);
        String msg = new String(datagram.getData(), datagram.getOffset(), datagram.getLength());
        LOG.log(Level.INFO, "Received a datagram with this message: " + msg);
    } catch (IOException ex) {
        LOG.log(Level.SEVERE, ex.getMessage(), ex);
    }
}
```

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud



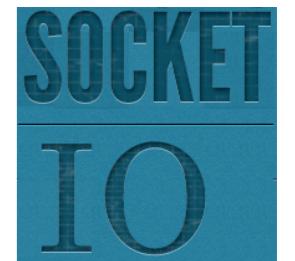
# Node.js 101

# JavaScript - End to End



Client

express



Server



heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

# Douglas Crockford: JavaScript: The Good Parts

[https://www.youtube.com/watch?v=\\_DKkVvOt6dk](https://www.youtube.com/watch?v=_DKkVvOt6dk)

# JavaScript is built on some very good ideas and a few very bad ones.

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

**JavaScript is an important language** because it is the language of the web browser. Its association with the browser makes it one of the most popular programming languages in the world. **At the same time, it is one of the most despised programming languages in the world.** [...]

Most people in that situation **don't even bother to learn JavaScript first**, and then they are surprised when JavaScript turns out to have significant differences from the some other language they would rather be using, and that those differences matter.

The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming. It is a language with enormous expressive power. It is even better when you know what you're doing. **Programming is difficult business. It should never be undertaken in ignorance.**

*Unearthing the Excellence in JavaScript*



Douglas Crockford



“Node.js is a **platform** built on Chrome's JavaScript runtime for easily building **fast, scalable network applications.** **not only!**

Node.js uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient, perfect for **data-intensive real-time applications that run across distributed devices.**”

# Node.js v0.10.32 Manual & Documentation

- [Assertion Testing](#)
- [Buffer](#)
- [C/C++ Addons](#)
- [Child Processes](#)
- [Cluster](#)
- [Console](#)
- [Crypto](#)
- [Debugger](#)
- [DNS](#)
- [Domain](#)
- [Events](#)
- [File System](#)

- [Globals](#)
- [HTTP](#)
- [HTTPS](#)
- [Modules](#)
- [Net](#)
- [OS](#)
- [Path](#)
- [Process](#)
- [Punycode](#)
- [Query Strings](#)
- [Readline](#)
- [REPL](#)
- [Stream](#)
- [String Decoder](#)
- [Timers](#)
- [TLS/SSL](#)
- [TTY](#)
- [UDP/Datagram](#)
- [URL](#)
- [Utilities](#)
- [VM](#)
- [ZLIB](#)

```
console.log("hello");
```

index.js

```
$ node index.js
hello
```



## Let's look at a **first example (NOT** good)

```
/*global require */ ←  
  
var fs = require("fs"); ←  
  
/**  
 * Simple function to test the synchronous readFileSync function provided by  
 * Node.js  
 * @param {string} filename - the name of the file we want to read in the test  
 */  
  
function testSyncRead(filename) {  
    console.log("I am about to read a file: " + filename);  
    var data = fs.readFileSync(filename); ←  
    console.log("I have read " + data.length + " bytes (synchronously).");  
    console.log("I am done.");  
}  
  
// We get the file name from the argument passed on the command line  
var filename = process.argv[2]; ←  
  
console.log("\nTesting the synchronous call");  
testSyncRead(filename);
```

*This is for the Brackets editor*

We use a standard **Node module** for accessing the file system

**fs.readFileSync** is synchronous: it blocks the main thread until the data is available.

**process** is a global object provided by Node.js



```
$ node sample2.js medium.txt
```

Testing the synchronous call  
I am about to read a file: medium.txt  
I have read 1024 bytes (synchronously).  
I am done.

*Synchronous functions are easier to use, but they have **severe** performance implications!!*



## Let's look at a **second example (BETTER)**

```
/*global require */  
  
var fs = require("fs");  
  
/**  
 * Simple function to test the asynchronous readFile function provided by Node.js  
 * @param {string} filename - the name of the file we want to read in the test  
 */  
function testAsyncRead(filename) {  
  console.log("I am about to read a file: " + filename);  
  
  fs.readFile(filename, function (err, data) {  
    console.log("Nodes just told me that I have read the file.");  
  });  
  
  console.log("I am done. Am I really????");  
}  
// We get the file name from the argument passed on the command line  
var filename = process.argv[2];  
  
console.log("\nTesting the asynchronous call");  
testAsyncRead(filename);
```

**fs.readFile** is asynchronous: it does not block the main thread until the data is available.

We must provide a **callback function**, which Node.js will invoke when the data is available.

**Problems** can happen when an (asynchronous) function is called.



```
$ node sample2.js medium.txt  
  
Testing the asynchronous call  
I am about to read a file: medium.txt  
I am done. Am I?  
Nodes just told me that I have read the file.
```

Node.js developers **have to** learn the asynchronous programming style.



## Let's look at a **third example**

```
/*global require */  
  
var http = require("http"); ←  
  
/**  
 * This function starts a http daemon on port 9000. It also  
 * registers a callback handler, to handle incoming HTTP  
 * requests (a simple message is sent back to clients).  
 */  
function runHttpServer() {  
    var daemon = http.createServer(); ←  
  
    daemon.on("request", function (req, res) { ←  
        console.log("A request has arrived: URL=" + req.url);  
        res.writeHead(200, {  
            'Content-Type': 'text/plain' ←  
        });  
        res.end('Hello World\n'); ←  
    });  
  
    console.log("Starting http daemon...");  
    daemon.listen(9000); ←  
}  
  
runHttpServer();
```

We use a standard **Node module** that takes care of the HTTP protocol.

Node can provide us with a **ready-to-use** server.

We can attach **event handlers** to the server. Node will notify us asynchronously, and give us access to the request and response.

We can **send back** data to the client.

We have wired everything, let's **welcome** clients!

HTTP Node.js v0.10.32 Manual & Documentation

nodejs.org/api/http.html#http\_event\_request

NPM REGISTRY

DOCS

BLOG

COMMUNITY

LOGOS

JOB

@nodejs

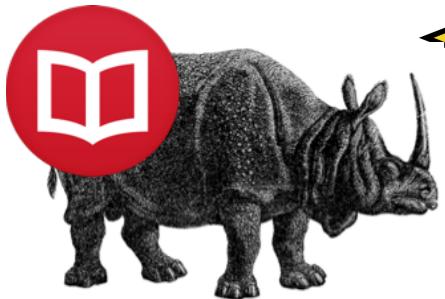
# Node.js v0.10.32 Manual & Documentation

[Index](#) | [View on single page](#) | [View as JSON](#)

## Table of Contents

- [HTTP](#)
  - [http.STATUS\\_CODES](#)
  - [http.createServer\(\[requestListener\]\)](#)
  - [http.createClient\(\[port\], \[host\]\)](#)
  - Class: [http.Server](#)
    - [Event: 'request'](#)
    - [Event: 'connection'](#)
    - [Event: 'close'](#)
    - [Event: 'checkContinue'](#)
    - [Event: 'connect'](#)
    - [Event: 'upgrade'](#)
    - [Event: 'clientError'](#)
    - [server.listen\(port, \[hostname\], \[backlog\], \[callback\]\)](#)
    - [server.listen\(path, \[callback\]\)](#)
    - [server.listen\(handle, \[callback\]\)](#)
    - [server.close\(\[callback\]\)](#)
    - [server.maxHeadersCount](#)
    - [server.setTimeout\(msecs, callback\)](#)
    - [server.timeout](#)
  - Class: [http.ServerResponse](#)
    - [Event: 'close'](#)
    - [Event: 'finish'](#)
    - [response.writeContinue\(\)](#)

These are the events that are **emitted** by the class. You can write callbacks and **react** to these events.



# How does Node.js use an **event loop** to offer an asynchronous programming model?

```
on('request', function(req, res) { // my code});
```

```
on('data', function(data) { // my code});
```

**Callback functions** that **you** have written and registered

'request' **event**

'request' **event**

'data' **event**

'request' **event**

⋮

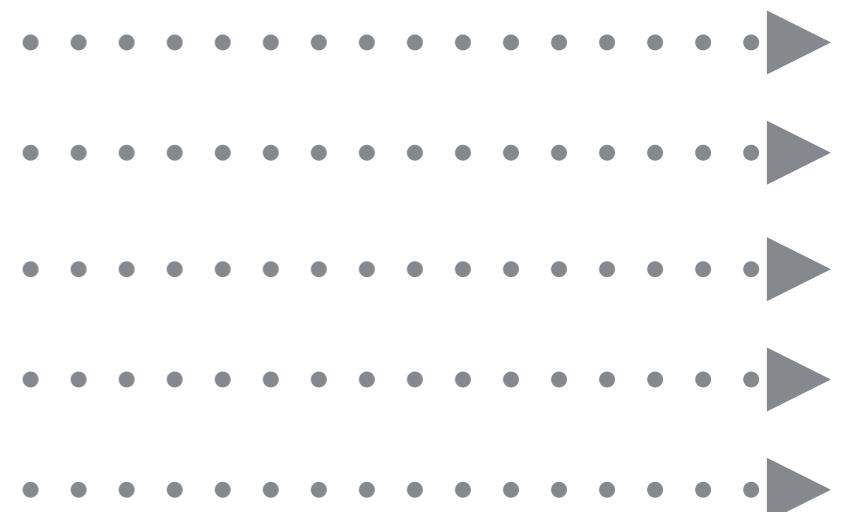
⋮

Queue of events that have been **emitted**

## Event Loop

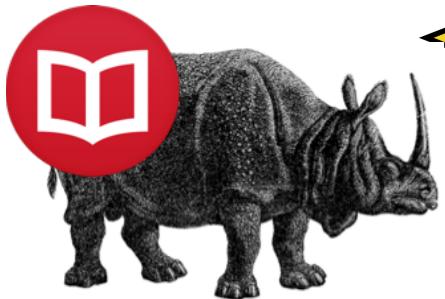
get the next event in the queue; invoke the registered callbacks in sequence; delegate I/O operations to the Node platform

All the code that **you** write runs on a **single thread**



The **long-running tasks** (I/Os) are executed by Node in parallel; Node emits events to report progress (which triggers your callbacks).

Another pattern is to provide a callback to node when invoking an asynchronous function.



## What is **npm**?

*"**npm** is the **package manager** for the Node JavaScript platform. It puts **modules** in place so that node can find them, and manages **dependency** conflicts intelligently."*

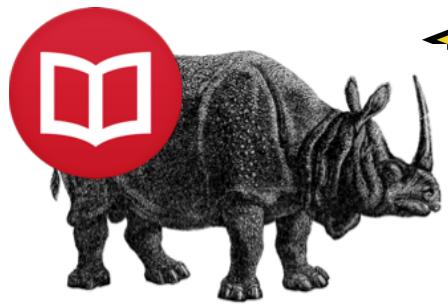
*It is extremely configurable to support a wide variety of use cases. Most commonly, it is used to **publish**, **discover**, **install**, and **develop** node programs."*

<https://www.npmjs.org/doc/cli/npm.html>



You **have to** read this:

<https://www.npmjs.org/doc/misc/npm-faq.html>



npm is a set of command line tools that work together with the **node registry**

The screenshot shows a web browser window for the npm website (<https://www.npmjs.com>). A large yellow promotional overlay is displayed at the top, featuring a cartoon penguin holding a sign that says "private npm is here." Below the overlay, the main website content is visible, including the navigation bar with links like "nutty penguin music", "npm private modules", "npm for the Enterprise", "documentation", "blog", "npm weekly", "jobs", and "support". The main heading "npm is the package manager for" is centered above four statistics: "187,190 total packages", "94,194,137 downloads in the last day", "568,188,116 downloads in the last week", and "2,515,481,679 downloads in the last month". At the bottom of the main content area, the text "packages people 'npm install' a lot" is written in red.

<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

The screenshot shows a web browser window with the URL <https://www.npmjs.com/package/chance>. A large, semi-transparent promotional overlay covers the top half of the page. It features a cartoon character of a brown dog wearing a white hoodie with 'npm' on it, holding a white mug with 'npm' on it. The text 'private npm is here.' is displayed in bold, followed by 'publish unlimited private modules for just \$7/month'. Below the overlay are two buttons: 'sign up' (in red) and 'no thanks' (in grey). The rest of the page below the overlay is the standard npm package page for 'chance'. The package page includes the npm logo, the package name 'chance' with a 'public' badge, a description 'Chance - Utility library', and a thumbnail image of a six-sided die. The search bar at the top has 'expr' typed into it. To the right of the search bar are links for 'sign up or log in' and a user profile icon. The main content area shows the package details for 'express', including its description as a minimalist web framework, and other packages like 'express-session' and 'express-handlebars'. On the right side of the package page, there are sections for 'npm install chance', a recent commit from 'victorquinn' (published 3 weeks ago), the latest version '0.7.7', a link to the GitHub repository 'github.com/victorquinn/chancejs', the 'MIT license', and a 'Collaborators' section.

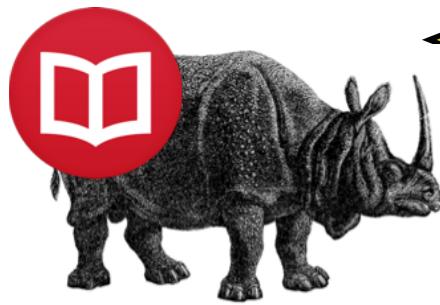
<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

The screenshot shows the npmjs.com package page for the 'express' module. The page has a header with the npm logo and a search bar. The main title is 'express' with a 'public' badge. Below it is a brief description: 'Fast, unopinionated, minimalist web framework'. A large 'express' logo is centered on the page. On the right side, there's a sidebar with various details: a star icon, a download icon labeled 'npm install express', a profile picture of 'dougwilson' with the note 'published 2 months ago', the latest version '4.13.3' (which is the 272nd release), the GitHub repository link 'github.com/strongloop/express', and the 'MIT license'. Below these are sections for 'Collaborators' (with icons for several people) and 'Stats' (showing 135,585 downloads in the last day, 818,847 in the last week, and 3,533,048 in the last month), as well as '67 open issues' and '28 open pull requests' on GitHub. At the bottom left, there's an 'Installation' section with the command '\$ npm install express' and a 'Features' section with a single bullet point: '• Robust routing'.

<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

```
Last login: Wed Sep 23 19:59:09 on console
$ npm help

Usage: npm <command>

where <command> is one of:
  access, add-user, adduser, apihelp, author, bin, bugs, c,
  cache, completion, config, ddp, dedupe, deprecate, dist-tag,
  dist-tags, docs, edit, explore, faq, find, find-dupes, get,
  help, help-search, home, i, info, init, install, issues, la,
  link, list, ll, ln, login, ls, outdated, owner, pack,
  prefix, prune, publish, r, rb, rebuild, remove, repo,
  restart, rm, root, run-script, s, se, search, set, show,
  shrinkwrap, star, stars, start, stop, t, tag, test, tst, un,
  uninstall, unlink, unpublish, unstar, up, update, v,
  verison, version, view, whoami

  npm <cmd> -h      quick help on <cmd>
  npm -l            display full usage info
  npm faq          commonly asked questions
  npm help <term>   search for help on <term>
  npm help npm     involved overview

Specify configs in the ini-formatted file:
  /Users/admin/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

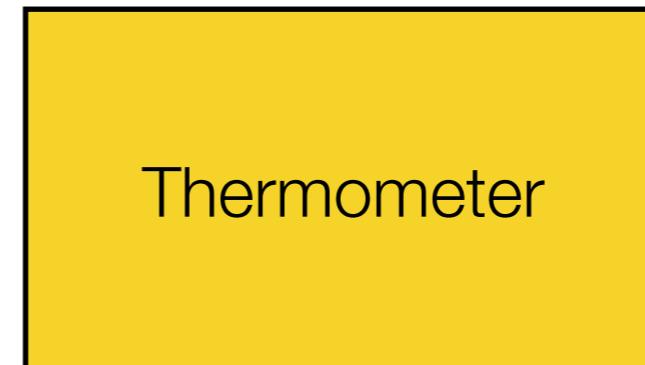
npm@2.5.1 /usr/local/lib/node_modules/npm
$
```

init  
install  
publish  
update

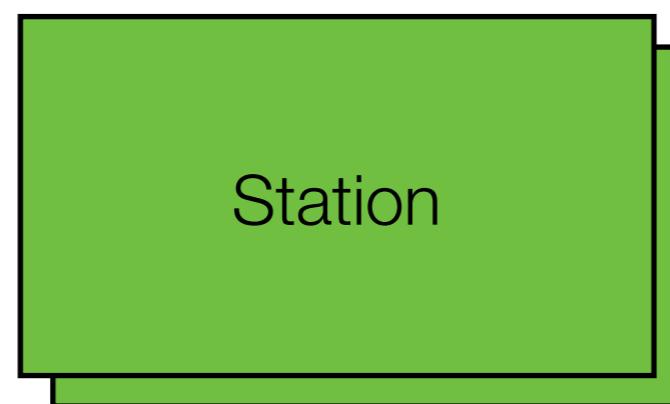
# Example: 09-thermometers



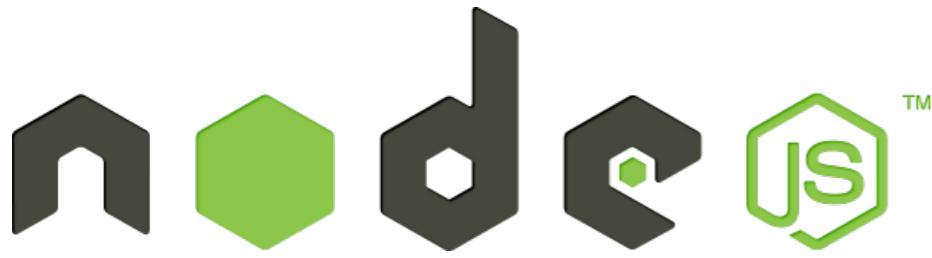
node thermometer.js office 20 3



**multicast**  
(239.255.22.5:9907)



node station.js



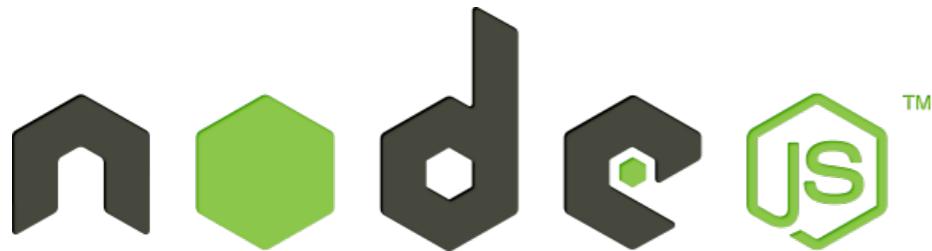
# thermometer.js (fragment)

```
// We use a standard Node.js module to work with UDP
var dgram = require('dgram');

// Let's create a datagram socket. We will use it to send our UDP datagrams
var s = dgram.createSocket('udp4');

// Create a measure object and serialize it to JSON
var measure = new Object();
measure.timestamp = Date.now();
measure.location = that.location;
measure.temperature = that.temperature;
var payload = JSON.stringify(measure);

// Send the payload via UDP (multicast)
message = new Buffer(payload);
s.send(message, 0, message.length, protocol.PROTOCOL_PORT, protocol.PROTOCOL_MULTICAST_ADDRESS,
function(err, bytes) {
  console.log("Sending payload: " + payload + " via port " + s.address().port);
});
```



## station.js (fragment)

```
// We use a standard Node.js module to work with UDP
var dgram = require('dgram');

// Let's create a datagram socket. We will use it to listen for datagrams published in the
// multicast group by thermometers and containing measures
var s = dgram.createSocket('udp4');
s.bind(protocol.PROTOCOL_PORT, function() {
  console.log("Joining multicast group");
  s.addMembership(protocol.PROTOCOL_MULTICAST_ADDRESS);
});

// This call back is invoked when a new datagram has arrived.
s.on('message', function(msg, source) {
  console.log("Data has arrived: " + msg + ". Source IP: " + source.address + ". Source
port: " + source.port);
});
```

# JavaScript 101

# JavaScript 101

- Types
- Scopes
- Objects
- Prototypal inheritance
- Functions
- Constructors
- Arrays



*Things are changing... we do not cover ES 6 in these slides*

# Rule #1

## JavaScript defines 6 types

```
var aNumber = 3.12;
var aBoolean = true;
var aString = "HEIG-VD";
var anObject = {
    aProperty: null
};

// t is true for all of these:
var t;
t = typeof aNumber === "number";
t = typeof aBoolean === "boolean";
t = typeof aString === "string";
t = typeof anObject === "object";
t = typeof anObject.aProperty ===
    "object";
t = typeof anObject.foobar ===
    "undefined";
```

- The 6 types are:
  - number
  - boolean
  - string
  - object
  - undefined
  - null
- null is a type, but `typeof null === object`.
- JavaScript is a dynamic language: when you declare a variable, you don't specify a type (and the type can change over time).

## Rule #2

There are 2 scopes for variables:

the (evil) global scope and the function scope

```
var aVariableInGlobalScope;

function myFunction() {
    var aVariableInFunctionScope;
    anotherVariableInGlobalScope;
}

function myFunction2() {
    for (i=0; i<10; i++) {
        //i is in global scope!
    }
    for (var j=0; j<10; j++) {
        //j is in function scope!
    }
}
```

- A variable declared within a function is **not accessible** outside this function.
- Unless using **strict mode**, it is not mandatory to declare variables (beware of typos...)
- Two scripts loaded from the same HTML page share the same global scope (beware of **conflicts...**).
- There is **no block scope**.

## Rule #3

# Objects are dynamic bags of properties

```
// let's create an object
var person = {
  firstName: 'olivier',
  lastName: 'liechti'
};

// we can dynamically add properties
person.gender = 'male';
person['zip'] = 1446;

// and delete them
delete person.zip;

for (var key in person) {
  console.log(key + " : " +
  person[key]);
};
```

- There are different ways to **access properties** of an object.
- JavaScript is **dynamic**: it is possible to **add** and **remove** properties to an object at any time.
- Every object has a different list of properties (**no class**).

## Rule #4

# The language has no support for classes

## There are 3 ways to create objects

```
// create an object with a literal
var person = {
  firstName: 'olivier',
  lastName: 'liechti'
};

// create an object with a prototype
var child = Object.create(person);

// create an object with a constructor
var child = new Person('olivier',
  'liechti');
```

- **class** is a reserved word in JavaScript, but it is not used in the current version of the language (reserved for the future).
- A **constructor** is function like any other (uppercase is a coding convention).
- It is the use of the **new** keyword that triggers the object creation process.

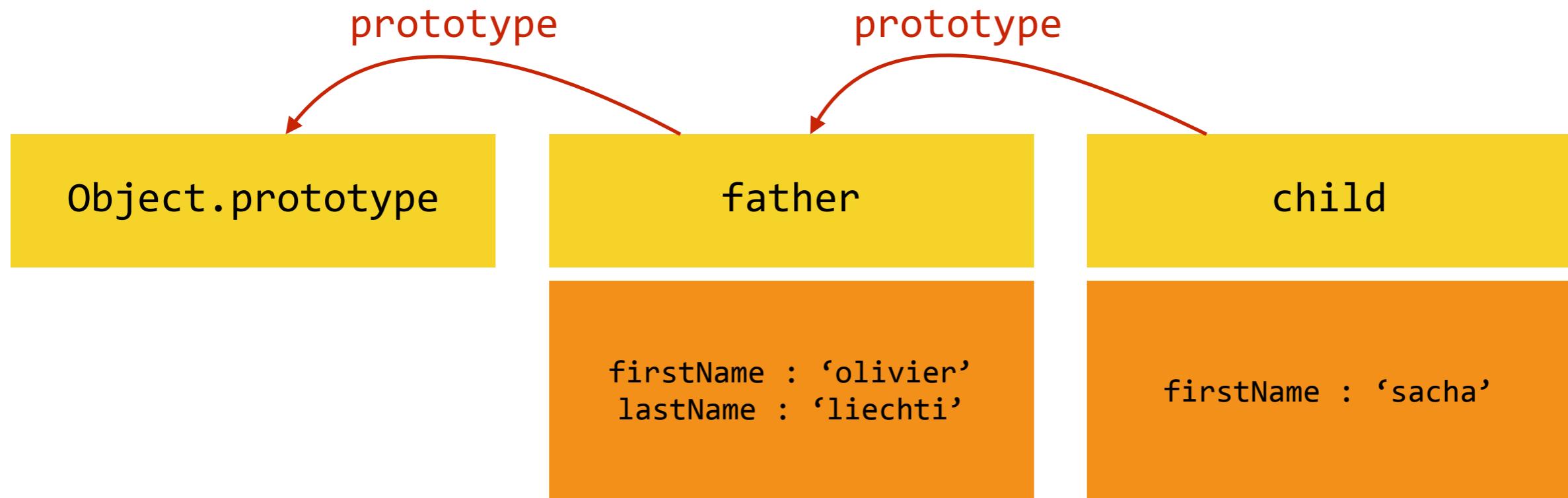
# Rule #5

## Every object inherits from a prototype object

```
var person = {  
    firstName: "olivier",  
    lastName: "liechti"  
};  
// person's prototype is Object.prototype  
  
var father = {};  
var child = Object.create(father);  
// child's prototype is father  
  
function Person(fn, ln) {  
    this.firstName = fn;  
    this.lastName = ln;  
}  
var john = new Person("John", "Doe");  
// john's prototype is Person.prototype
```

## Rule #5

# Every object inherits from a prototype object



```
console.log(child.lastName);
// prints 'liechti' on the
console
```

- Every object inherits from a prototype object. **It inherits and can override its properties**, including its methods.
- Objects created with object literals inherit from **Object.prototype**.
- When you access the property of an object, JavaScript **looks up the prototype chain** until it finds an ancestor that has a value for this property.

# Rule #6

## With patterns, it is possible to implement class-like data structures

```
function Person(fn, ln) {  
    var privateVar;  
    this.firstName = fn;  
    this.lastName = ln;  
    this.badGreet = function() {  
        console.log("Hi " + this.firstName);  
    };  
};  
  
Person.prototype.greet = function() {  
    console.log("Hey " + this.firstName);  
};  
  
var p1 = new Person("olivier", "liechti");  
  
p1.badGreet();  
p1.greet();
```

- **badGreet** is a property that will be replicated for every object created with the Person constructor:
  - poor memory management
  - not possible to alter behavior of all instances at once
- **greet** is a property that will be shared by all instances (because it will be looked up along the object inheritance chain).
- **privateVar** is not accessible outside of the constructor.
- **firstName** is publicly accessible (no encapsulation).

# Rule #7

## Arrays are objects

```
var fruits = ["apple", "pear"];

fruits.push("banana");
console.log(Object.getPrototypeOf(fruits));

for (var i=0; i<fruits.length; i++) {
  console.log("fruits[" + i + "] = " + fruits[i]);
}

var transformedFruits = fruits.map( function(fruit) {
  return fruit.toUpperCase();
});
transformedFruits.forEach( function(fruit) {
  console.log(fruit);
});

var count = fruits.reduce( function(val, fruit) {
  console.log("reducer invoked with " + val);
  return val+1;
}, 0);
console.log("There are " + count + " fruits in the array");
```

# Rule #8

## Functions are objects

```
function aFunction() {  
}  
  
var f = function() {  
}  
  
var g = function g() {  
    g(); // recursive call  
}  
  
var h = function(functionParam) {  
    functionParam();  
}  
  
h(f);  
h(g);
```

## #1 Functions are objects

```
var aFunction = function() {  
    console.log("I am doing my job");  
};  
aFunction.aProperty = "aValue";  
aFunction();
```

## #2 Passing functions as arguments

```
var aFunction = function() {...};  
var anotherFunction = function( f1 ) {... f1(); ...};  
anotherFunction( aFunction );
```

## #3 Return functions as values

```
var aFunction = function() {  
    return function() {...}  
};  
aFunction()();
```

## Rule #9

**Whenever a function is defined, an object is created.**

Its prototype is Function.prototype.

Its prototype property is the object that will be the prototype of instances created with the function used as a constructor with the **new** keyword.

```
function Robot() {  
}  
var r2d2 = new Robot();  
  
var t1 = Object.getPrototypeOf(Robot) === Function.prototype;  
var t2 = Robot.__proto__ === Function.prototype; //  
deprecated  
  
var t3 = Object.getPrototypeOf(r2d2) === Robot.prototype;  
var t4 = Object.getPrototypeOf(Robot.prototype) ===  
Object.prototype;
```

## Rule #10

# "new" makes constructors behave in a special way, but constructors are normal functions

```
function Animal(name) {  
    this.name = name;  
}  
var cat = new Animal("Félix");
```

1. A **new object** is created.
2. Its **prototype** is set to `Animal.prototype`
3. The constructor function is called; **this** is bound to the newly created object.
4. In general, the constructor does not return any object. In this case, the result of the `new` expression is the newly created object.

# Rule #11

## Functions can be nested

```
function f1(p1) {  
    console.log("f1 can see " + p1);  
    function f2(p2) {  
        console.log("f2 can see " + p2 + " " + p1);  
        function f3(p3) {  
            console.log("f3 can see " + p3 + " " + p2 + " " + p1);  
        }  
        f3(3);  
    }  
    f2(2);  
}  
f1(1);
```

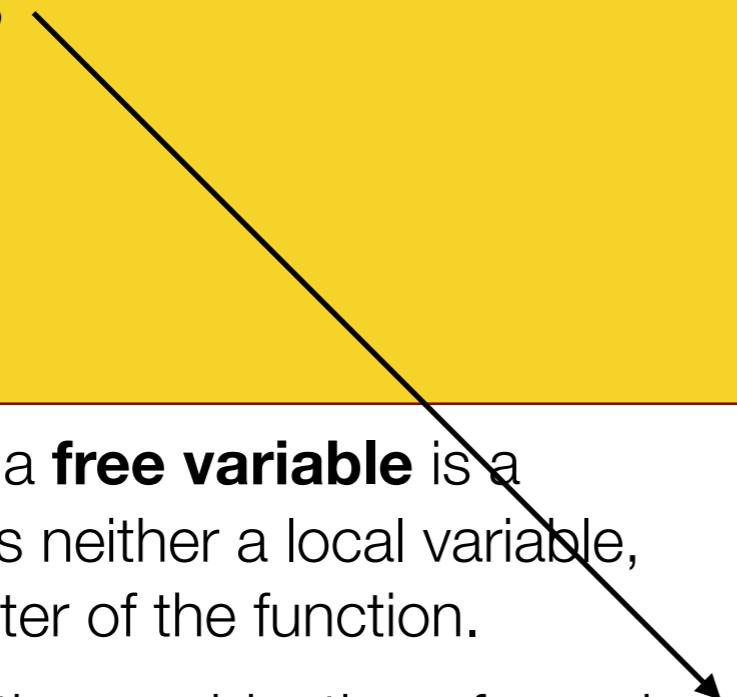
- An **object** is created for every function.
- Each function has access to variables defined in the **parent** functions (an in the **global scope**).

# Rule #12

## A closure is formed when a nested function accesses a *free variable*

```
function f1(p1) {  
    console.log("f1 can see " + p1);  
    function f2(p2) {  
        console.log("f2 can see " + p2 + " " + p1);  
        function f3(p3) {  
            console.log("f3 can see " + p3 + " " + p2 + " " + p1);  
        }  
        f3(3);  
    }  
    f2(2);  
}  
f1(1);
```

- In a function, a **free variable** is a variable that is neither a local variable, nor a parameter of the function.
- A **closure** is the combination of a code block (the function code) and saved parent environments.



```
▼ function f3(p3) { console.log("f3 can see " + p3 + " " + p2 + " " + p1); }  
  arguments: null  
  caller: null  
  length: 1  
  name: "f3"  
  ▶ prototype: f3  
  ▶ __proto__: function Empty() {}  
  ▶ <function scope>  
    ▶ Closure  
      p2: 2  
    ▶ Closure  
      p1: 1  
  ▶ Global: Window
```

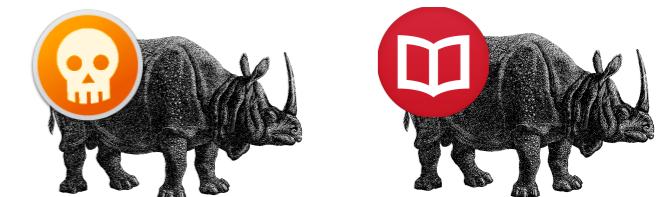
# Rule #13

## When 2 functions are created in the same scope, closures are formed on the same environment.

```
function parent() {  
  
    var sharedVariable;  
  
    function child1() {  
        sharedVariable = 1;  
    }  
  
    function child2() {  
        sharedVariable = 2;  
    }  
  
    function getValue() {  
        return sharedVariable;  
    }  
  
    return {  
        toggleToOne: child1,  
        toggleToTwo: child2,  
        getValue: getValue  
    }  
}  
  
var myToggle1 = parent();  
var myToggle2 = parent();  
myToggle1.toggleToTwo();  
myToggle2.toggleToOne();  
console.log(myToggle1.getValue());  
console.log(myToggle2.getValue());
```

- This pattern can be used to implement **private variables** in Javascript.
- But remember that in this case, the code of the functions is **cloned** in all instances (unlike when you use `Class.prototype.method`).
- Be aware of a **well known issue** when closures are formed within a loop.

```
function test() {  
    var functions = [];  
  
    for (var i=0; i<10; i++) {  
        functions.push(function() {  
            console.log(i);  
        });  
    };  
  
    return functions;  
}  
  
var fs = test();  
fs.forEach( function(f) { f(); } );
```



- **What is happening?**
- **Why?**
- **How do you fix this?**

# Rule #14

## Patterns are applied to create modules

```
var myModule = (function() {  
  
    var aPrivateVar;  
    var privateFunction_1 = function() {}  
    var privateFunction_2 = function() {}  
  
    return {  
        publicFunction: privateFunction_1  
    }  
  
})();  
  
myModule.publicFunction();
```

<http://codepen.io/wasadigi/full/GxsfC/>

- The function is **immediately invoked**.

- When `privateFunction_1` accesses `aPrivateVar`, a **closure** is formed.
- is **available even after** the immediately invoked function has returned.
- `privateFunction_1` and `privateFunction_2` share the same parent scope.

```
Hello world
> dir(myModule)
▼ Object ⓘ
  ▼ publicFunction: function () {
    arguments: null
    caller: null
    length: 0
    name: ""
    ► prototype: Object
    ► __proto__: function Empty() {}
    ▼ <function scope>
      ▼ Closure
        aPrivateVar: "world"
        ► Global: Window
        ► __proto__: Object
      ◀ undefined
    >
```

# Rule #15

## The value of this depends on the way a function has been called

```
function Robot(name) {  
  this.name = name;  
}
```

```
Robot.prototype.greet = function greet() {  
  debugger;  
  console.log("hello " + this.name);  
  if (this.name === undefined) {  
    return; // oops...  
  }  
  greet(); // no dot notation!  
}
```

```
var r2d2 = new Robot("r2d2");  
r2d2.greet();
```

- When a function is called on an object (i.e. **as a method**), then this refers to the object.
- When a function is called **as a function** (no dot notation), then this refers to the global object).
- There are methods defined on **Function.prototype** to control the value of this: **apply** and **call**.