COMP284 Scripting Languages

Lecture 1: Overview of COMP284
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

How many programming languages should you learn?

1. Academic / Educational viewpoint:
   Learn programming language concepts and
   use programme languages to gain practical experience with them
   – imperative / object-oriented — C, Java
   – functional — Maude, OCaml, Haskell
   – logic/constraint — Prolog, DLV
   – concurrent
   then all (other) programming languages can be learned easily
2. An employer's viewpoint:
   Learn exactly those programming languages that the specific employer needs
3. Compromise: Spend most time on 1 but leave some time for 2 to allow more than one language from a class/paradigm to be learned
4. Problem: Which additional language do you cover?
   ⇝ Look what is used/demanded by employers

Programming languages: Job ads

Software Developer
(Digital Repository)
University of Liverpool - University Library
£31,020 - £35,939 pa

To work as part of a small team based in the University Library, working closely with the University's Computing Services Department on the institutional digital repository, recommending and developing technical solutions, tools and functionality to integrate the repository with other internal systems and to enable research outputs to be shared externally. You will be an experienced Software Developer with knowledge of LAMP technologies such as XML, XSLT, Perl and Javascript. You will hold a degree in Computer Science or a related discipline and/or have proven industrial experience of software development. The post is full time, 35 hours per week.

Job Ref: A-576989

Programming languages: Job ads

Senior Software Development Manager
IMDb Video and Recommendations (Seattle, WA)

IMDb (a wholly-owned subsidiary of Amazon) is recruiting for a Senior Software Development Manager to lead our "What to Watch" team. You'll be charged with transforming IMDb from a reference site to a place where hundreds of millions of people find and discover what to watch across a variety of video providers, and seamlessly connect them with watching the movies and TV shows best suited for them, wherever and whenever they may be.

Basic qualifications:

- Bachelor's degree in Computer Science, Computer Engineering or related technical discipline
- 10+ years of experience as a software developer
- 5+ years experience managing people
- Software development experience in OOP, Java, Perl, HTML, CSS, JavaScript, Linux/UNIX, AJAX, MySQL

Programming languages: Job ads

Full-time Remote Worker
AOL Tech (Engadget, TUAW, Joystiq, Massively)

AOL Tech is looking for a great front-end developer who can help us take Engadget and our other blogs to new levels.
The ideal candidate is highly proficient in JavaScript/jQuery, comfortable with PHP / mySQL and experienced in web design, optimization and related technologies for desktop and mobile. A solid understanding of mobile-first design is a must.

Requirements:

- High proficiency in JavaScript/jQuery
- Familiar with spriting, lazy loading, and other general performance-optimized techniques
- Mac access for compatibility with current tools
- HTML5/CSS3
- Git, SSH

Websites and Programming Languages

Website	Client-Side	Server-Side	Database
Google	JavaScript	C, C++, Go, Java, Python, ~~PHP~~	BigTable, MariaDB
Facebook	JavaScript	Hack, PHP, Python, C++, Java, ...	MariaDB, MySQL, HBase Cassandra
YouTube	~~Flash~~, JavaScript	C, C++, Python, Java, Go	BigTable, MariaDB
Yahoo	JavaScript	PHP	MySQL, PostgreSQL
Amazon	JavaScript	Java, C++, Perl	Oracle Database
Wikipedia	JavaScript	PHP, Hack	MySQL, MariaDB
Twitter	JavaScript	C++, Java, Scala	MySQL
Bing	JavaScript	ASP.NET	MS SQL Server

Wikipedia Contributors: Programming languages used in most popular websites. Wikipedia, The Free Encyclopedia, 20 October 2017, at 11:28. `http://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites` [accessed 23 October 2017]

Scripting languages

Script

A user-readable and user-modifiable program that performs simple operations and controls the operation of other programs

Scripting language

A programming language for writing scripts

Classical example: Shell scripts


```
#!/bin/sh
for file in *; do
    wc -l "$file"
done
```

Print the number of lines and name for each file in the current directory

Scripting languages: Properties

- Program code is present at run time and starting point of execution
  - compilation by programmer/user is not needed
  - compilation to bytecode or other low-level representations may be performed 'behind the scenes' as an optimisation
- Presence of a suitable runtime environment is required for the execution of scripts
  - includes an interpreter, or just-in-time compiler, or bytecode compiler plus virtual machine
  - typically also includes a large collection of libraries
- Executation of scripts is typically slower then the execution of code that has been fully pre-compiled to machine code

```
#!/bin/sh
for file in *; do
    wc -l "$file"
done
```

Scripting languages: Properties

- Rich and easy to use interface to the underlying operating system, in order to run other programs and communicate with them
  - rich input/output capabilities, including pipes, network sockets, file I/O, and filesystem operations
- Easy integration within larger systems
  - often used to glue other systems together
  - can be embedded into other applications


```
#!/bin/sh
for file in *; do
    wc -l "$file"
done
```

Scripting languages: Properties

- Variables, functions, and methods
  typically do not require type declarations
  (automatic conversion between types, e.g. strings and numbers)
- Some built-in data structures
  (more than in C, fewer than in Java)
- Ability to generate, load, and interpret source code at run time
  through an eval function

JavaScript

```
var x   = 2;
var y   = 6;
var str = "if (x > 0) { z = y / x } else { z = -1 }";
console.log('z is ', eval(str));  // Output: z is 3
x       = 0;
console.log('z is ', eval(str));  // Output: z is -1
```

Scripting languages: Properties

- The evolution of a scripting language typically starts with a limited set of language constructs for a specific purpose

  Example: PHP started as set of simple 'functions' for tracking visits to a web page

- The language then accumulates more and more language constructs as it is used for a wider range of purposes

- These additional language constructs may or may not fit well together with the original core and/or may duplicate existing language constructs
- During this evolution of the language, backward compatibility may or may not be preserved

$\leadsto$ Language design of scripting languages is often sub-optimal

Aims

1. To provide students with an understanding of the nature and role of scripting languages

2. To introduce students to some popular scripting languages and their applications

3. To enable students to write simple scripts using these languages for a variety of applications

Learning Outcomes

At the end of the module students should be able to

1. compare and contrast languages such as JavaScript, Perl and PHP with other programming languages

2. document and comment applications witten using a scripting language

3. rapidly develop simple applications, both computer and web-based, using an appropriate scripting language

Delivery of the module (1)

1. Lectures
   - Structure:
     16 to 18 lectures
   - Schedule:
     1 or 2 lectures per week spread over 9 weeks

     See your personal timetable and e-mail announcements for details
   - Lecture notes and screencasts are available at
     `cgi.csc.liv.ac.uk/~ullrich/COMP284/notes`
   - Revise the lectures before the corresponding practical
   - Additional self study using the recommended textbooks
     and the on-line material is essential

Delivery of the module (1)

2 Practicals

- Structure:
  - 7 practicals with worksheets (3 Perl, 2 PHP, 2 JavaScript)
    - ⇝ gain understanding via practice
    - ⇝ get answers to questions about the lecture material
  - Up to 3 additional practicals for questions about the assignments
- Schedule:
  1 practical per week for about 10 weeks

  Practicals start in week 2
- Practicals assume familiarity with Linux and departmental Linux systems
  ⇝ To recap, use the worksheets available at
  `cgi.csc.liv.ac.uk.uk/~ullrich/COMP284/notes`
- Practicals assume familiarity with the related lecture material

How to learn a new programming language

- Once you know how to program in one programming language, additional programming languages are best learned by a process of enquiry and practice guided by existing experience

- Typically, the questions that guide you are
  - What kind of . . . are there?
    Example: What kind of control structures are there?

  - What is the syntax for . . . ?
    Example: What is the syntax for conditional statements?
  - What happens if . . . ?
    Example: What happens if 1 is divided by 0?

  - How do I . . . ?
    Example: How do I catch an exception?
- Talk to other people who are currently trying to learn the same language or have already learned it
  ⇝ Ask what has surprised them most

How to learn a new programming language

- Once you know how to program in one programming language, additional programming languages are best learned by a process of enquiry and practice
- The best kind of learning is learning by doing
  - ⇝ The questions posed on the previous slide are often best explored by experimenting with small sample programs (‘toy’ programs)
- Work on substantive programs
  - ⇝ You need to convince employers that you have worked on programs more substantive than ‘toy’ programs
  - ⇝ The assignments are ‘pretend’ substantive programs but in reality are too small
- Employers value experience, in particular, the experience that you get from overcoming challenges
  - ⇝ Assignments that are not challenging are of limited value

Delivery of the module (3)

3. Office hours

   Monday, 16:00 Ashton, Room 1.03

   but always arrange a meeting by e-mail first
   (`U.Hustadt@liverpool.ac.uk`)

4. Announcements will be send by e-mail
   - You should check you university e-mail account at least every other day
   - Always use your university e-mail account
     if you want to contact me or any other member of staff

Recommended texts

- Core reading
  - R. Nixon:
    Learning PHP, MySQL, & JavaScript.
    O'Reilly, 2009.
    Learning PHP. . . , 4th edition.
    O'Reilly, 2014.
    Harold Cohen Library: 518.561.N73 or e-book
  - R. L. Schwartz, brian d foy, T. Phoenix:
    Learning Perl.
    O'Reilly, 2011.
    Learning Perl, 7th edition.
    O'Reilly, 2016.
    Harold Cohen Library: 518.579.86.S39 or e-book
- Further reading
  - M. David:
    HTML5: designing rich Internet applications.
    Focal Press, 2010.
    Harold Cohen Library: 518.532.D24 or e-book
  - N. C. Zakas:
    Professional JavaScript for Web Developers.
    Wiley, 2009.
    Harold Cohen Library: 518.59.Z21 or e-book

Assessment

- This is a coursework-based module (no exam)
- Three assessment tasks need to be completed throughout the semester:
  - Perl Deadline: Friday, 2 March, 17:00
  - PHP Deadline: Monday, 9 April, 12:00
  - JavaScript Deadline: Friday, 27 April, 17:00
- Effort required: about 10 hours each
- Available at: `http://cgi.csc.liv.ac.uk/~ullrich/COMP284/`

Attendance and Performance

	Students	Average Lecture Attendance	Average Practical Attendance	Average Module Mark
2011-12	33	76.0%	70.0%	63.1
2012-13	58	82.0%	69.0%	64.5
2013-14	107	80.0%	60.0%	59.1
2014-15	119	71.8%	65.2%	54.5
2015-16	76	67.4%	46.8%	57.9
2016-17	114	43.8%	38.3%	53.0

- From 2014-15, screencasts of the lectures were available to students
- From 2015-16, the requirement to write a report on each program was dropped
- Hypothesis 1:
  Lecture Attendance $>$ 75% and Practical Attendance $>$ 65% $\Leftrightarrow$ Module Mark $>$ 62
- Hypothesis 2:
  Screencasts Available $\Leftrightarrow$ Module Mark $<$ 59

Academic Integrity

- Plagiarism occurs when a student misrepresents, as his/her own work, the work, written or otherwise, of any other person (including another student) or of any institution
- Collusion occurs where there is unauthorised co-operation between a student and another person in the preparation and production of work which is presented as the student's own
- Fabrication of data occurs when a student enhances, exaggerates, or fabricates data in order to conceal a lack of legitimate data

If you are found to have plagiarised work, colluded with others, or fabricated data, then you may fail COMP284

Serious 'offenders' may be excluded from the University

Do not try to take a 'shortcut'
You must do the work yourself!

Academic Integrity: Lab rules

- Do not ask another student to see any part of their code for a COMP284 assignment
  - ⇝ contravention of this leads to collusion
- Do not show or make available any part of your code relating for a COMP284 assignment to any other student
  - ⇝ contravention of this leads to collusion

- Do not share (links to) on-line material that might help with a COMP284 assignment
  - ⇝ contravention of this leads to collusion

- Lock your Lab PC when you leave it alone
- Where you use any material/code found on-line for a COMP284 assignment, you must add comments to your code indicating its origin by a proper academic reference
  - ⇝ contravention of this is plagiarism
  - ⇝ acknowledged code re-use may still result in a lower mark

COMP284 Scripting Languages

Lecture 2: Perl (Part 1)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Perl

- Originally developed by Larry Wall in 1987
  Perl 6 was released in December 2015
- Borrows features from
  - C
    imperative language with variables, expressions, assignment statements, blocks of statements, control structures, and procedures / functions
  - Lisp
    lists, list operations, functions as first-class citizens
  - AWK (pattern scanning and processing language)
    hashes / associative arrays, regular expressions
  - sed (stream editor for filtering and transforming text)
    regular expressions and substitution `s///`
  - Shell
    use of sigils to indicate type (`$` – scalar, `@` – array, `%` – hash, `&` – procedure)
  - Object-oriented programming languages
    classes/packages, inheritance, methods

Perl: Uses and applications

- Main application areas of Perl
  - text processing
    ⇝ easier and more powerful than sed or awk
  - system administration
    ⇝ easier and more powerful than shell scripts
- Other application areas
  - web programming
  - code generation
  - bioinformatics
  - linguistics
  - testing and quality assurance

Perl: Applications

- Applications written in Perl
  - Movable Type – web publishing platform
    `http://www.movabletype.org/`
  - Request Tracker – issue tracking system
    `http://bestpractical.com/rt/`
  - Slash – database-driven web application server
    `http://sourceforge.net/projects/slashcode/`

Perl: Applications

- Organisations using Perl
  - Amazon – online retailer
    http://www.amazon.co.uk
  - BBC – TV/Radio/Online entertainment and journalism
    http://www.bbc.co.uk
  - Booking.com – hotel bookings
    http://www.booking.com
  - craigslist – classified ads
    http://www.craigslist.org
  - IMDb – movie database
    http://www.imdb.com
  - Monsanto – agriculture/biotech
    http://www.monsanto.co.uk/
  - Slashdot – technology related news
    http://slashdot.org

Java versus Perl: Java

```java
/* Author: Clare Dixon
 * The HelloWorld class implements an application
 * that prints out "Hello World".
 */
public class HelloWorld {
  // ---------------METHODS-------------
  /* Main Method */
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```


Edit-compile-run cycle:

1. Edit and save as `HelloWorld.java`
2. Compile using `javac HelloWorld.java`
3. Run using `java HelloWorld`

Java versus Perl: Perl

```perl
#!/usr/bin/perl
# Author: Ullrich Hustadt
# The HelloWorld script implements an application
# that prints out "Hello World".

print "Hello␣World\n";
```


Edit-run cycle:

1. Edit and save as `HelloWorld`
2. Run using `perl HelloWorld`

---

1. Edit and save as `HelloWorld`
2. Make it executable `chmod u+x HelloWorld`
   This only needs to be done once!
3. Run using `./HelloWorld`

Perl

- Perl borrows features from a wide range of programming languages including imperative, object-oriented and functional languages

- Advantage: Programmers have a choice of programming styles
- Disadvantage: Programmers have a choice of programming styles

- Perl makes it easy to write completely incomprehensible code
  ⇝ Documenting and commenting Perl code is very important

Perl

- Perl makes it easy to write completely incomprehensible code
  ⇝ Documenting and commenting Perl code is very important


```
#!/usr/bin/perl
# Authors: Schwartz et al. / Ullrich Hustadt
# Text manipulation using regular expressions
#
# Retrieve the Perl documentation of function 'atan2'
@lines = `perldoc -u -f atan2`;

# Go through the lines of the documentation, turn all text
# between angled brackets to uppercase and remove the
# character in front of the opening angled bracket, then
# print the result
foreach (@lines) {
    s/\w<([^\>]+)>/\U$1/g;
    print;
}
```

In the example, there are more lines of comments than there are lines of code

Perl for Java programmers

- In the following we will consider various constructs of the Perl programming language
  - numbers, strings
  - variables, constants
  - assignments
  - control structures
- These will often be explained with reference to Java ('like Java', 'unlike Java')
- Note that Perl predates Java
  ⇝ common constructs are almost always inherited by both languages from the programming language C

Perl scripts

- A Perl script consists of one or more statements and comments
  ⇝ there is no need for a main function (or classes)
- Statements end in a semi-colon
- Whitespace before and in between statements is irrelevant
  (This does not mean its irrelevant to someone reading your code)
- Comments start with a hash symbol # and run to the end of the line
- Comments should precede the code they are referring to

Perl scripts

- Perl statements include
  - Assignments
  - Control structures

  Every statement returns a value
- Perl data types include
  - Scalars
  - Arrays / Lists
  - Hashes / Associative arrays
- Perl expressions are constructed from values and variables using operators and subroutines
  - Perl expressions can have side-effects
    (evaluation of an expression can change the program state)

  Every expression can be turned into a statement by adding a semi-colon

Scalar data

- A scalar is the simplest type of data in Perl
- A scalar is either
  - an integer number
    `0   2012   -40   1_263_978`
  - a floating-point number
    `1.25   256.0   -12e19   2.4e-10`
  - a string
    `'hello world'   "hello world\n"`

- Note:
  - There is no 'integer type', 'string type' etc
  - There are no boolean constants (true / false)

Integers and Floating-point numbers

- Perl provides a wide range of pre-defined mathematical functions

`abs(`*number*`)`	absolute value
`log(`*number*`)`	natural logarithm
`random(`*number*`)`	random number between 0 and *number*
`sqrt(`*number*`)`	square root

- Additional functions are available via the `POSIX` module

`ceil(`*number*`)`	round fractions up
`floor(`*number*`)`	round fractions down

  Note: There is no pre-defined `round` function

```
use POSIX;
print ceil(4.3);  // prints '5'
print floor(4.3); // prints '4'
```

- Remember: Floating-point arithmetic has its peculiarities

  David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic. Computing Surveys 23(1):5–48.
  `http://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf`

Mathematical functions and Error handling

- Perl, PHP and JavaScript differ in the way they deal with applications of mathematical functions that do not produce a number

  In Perl we have

  - `log(0)` produces an error message: `Can't take log of 0`
  - `sqrt(-1)` produces an error message: `Can't take sqrt of -1`
  - `1/0` produces an error message: `Illegal division by zero`
  - `0/0` produces an error message: `Illegal division by zero`

  and execution of a script terminates when an error occurs

- A possible way to perform error handling in Perl is as follows:

```
eval {   ...run the code here...            # try
         1;
} or do { ...handle the error here using $@... # catch
};
```

  The special variable `$@` contains the Perl syntax or routine error message from the last `eval`, `do`-FILE, or `require` command

Strings

Perl distinguishes between

- single-quoted strings and
- double-quoted strings

single-quoted strings ('taken literally')		double-quoted strings ('interpreted'/'evaluated')	
'hello'	⇝ hello	"hello"	⇝ hello
'don\'t'	⇝ don't	"don't"	⇝ don't
'"hello"'	⇝ "hello"	"\"hello\""	⇝ "hello"
'backslash\\'	⇝ backslash\	"backslash\\"	⇝ backslash\
'glass\\table'	⇝ glass\table	"glass\\table"	⇝ glass\table
'glass\table'	⇝ glass\table	"glass\table"	⇝ glass able

In Java, single quotes are used for single characters and double quotes for strings

Double-quoted string backslash escapes

- In a single-quoted string `\t` is simply a string consisting of `\` and `t`
- In a double-quoted string `\t` and other backslash escapes have the following meaning

Construct	Meaning
`\n`	Logical Newline (actual character is platform dependent)
`\f`	Formfeed
`\r`	Return
`\t`	Tab
`\l`	Lower case next letter
`\L`	Lower case all following letters until `\E`
`\u`	Upper case next letter
`\U`	Upper case all following letters until `\E`
`\Q`	Quote non-word characters by adding a backslash until `\E`
`\E`	End `\L`, `\U`, `\Q`

UTF-8

- Perl supports UTF-8 character encodings which give you access to non-ASCII characters
- The pragma

```
use utf8;
```

  allows you to use UTF-8 encoded characters in Perl scripts
- The function call

```
binmode(STDIN,  ":encoding(UTF-8)");
binmode(STDOUT, ":encoding(UTF-8)");
```

  ensures that UTF-8 characters are read correctly from STDIN and printed correctly to STDOUT
- The `Unicode::Normalize` module enables correct decomposition of strings containing UTF-8 encoded characters

```
use Unicode::Normalize;
```

UTF-8

Example:

```
binmode(STDOUT, ":utf8");
print "\x{4f60}\x{597d}\x{4e16}\x{754c}\n";     # chinese
print "\x{062d}\x{fef0}\n";                      # arabic
```

For further details see Schwartz et al., Appendix C

String operators and automatic conversion

- Two basic operations on strings are
  - string concatenation .
    ```
    "hello" . "world"          ⇝  "helloworld"
    "hello" . '␣' . "world"    ⇝  'hello␣world'
    "\Uhello" . '␣\LWORLD'     ⇝  'HELLO␣\LWORLD'
    ```
  - string repetition x:
    ```
    "hello␣" x 3  ⇝  "hello␣hello␣hello␣"
    ```
- These operations can be combined
  ```
  "hello␣" . "world␣" x 2  ⇝  "hello␣world␣world␣"
  ```
- Perl automatically converts between strings and numbers
  ```
  2 . "␣worlds"  ⇝  "2␣worlds"
  "2" * 3        ⇝  6
  2e-1 x 3       ⇝  "0.20.20.2" ("0.2" repeated three times)
  "hello"* 3     ⇝  0
  ```

'Booleans'

- Unlike Java, Perl does not have a boolean datatype
- Instead the values

```
0      # zero
''     # empty string
'0'    # string consisting of zero
undef  # undefined
()     # empty list
```

all represent *false* while all other values represent *true*

'Boolean operators'

- Perl offers the same short-circuit boolean operators as Java: `&&`, `||`, `!`
  Alternatively, `and`, `or`, `not` can be used

`A`	`B`	`(A && B)`
true	*true*	`B` (*true*)
true	*false*	`B` (*false*)
false	*true*	`A` (*false*)
false	*false*	`A` (*false*)

<table>
<tr><th><code>A</code></th><th><code>B</code></th><th><code>(A || B)</code></th></tr>
<tr><td>true</td><td>true</td><td><code>A</code> (true)</td></tr>
<tr><td>true</td><td>false</td><td><code>A</code> (true)</td></tr>
<tr><td>false</td><td>true</td><td><code>B</code> (true)</td></tr>
<tr><td>false</td><td>false</td><td><code>B</code> (false)</td></tr>
</table>

`A`	`(! A)`
true	`''` (*false*)
false	`1` (*true*)

- Note that this means that `&&` and `||` are not commutative, that is,
  `(A && B)` is not the same as `(B && A)`

```
($denom != 0) && ($num / $denom > 10)
```

Comparison operators

Perl distinguishes between numeric comparison and string comparison

Comparison	Numeric	String
Equal	`==`	`eq`
Not equal	`!=`	`ne`
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less than or equal to	`<=`	`le`
Greater than or equal to	`>=`	`ge`

Examples

```
   35  == 35.0       # true
 '35'  eq '35.0'     # false
 '35'  == '35.0'     # true
   35  <  35.0       # false
 '35'  lt '35.0'     # true
'ABC'  eq "\Uabc"    # true
```

Scalar variables

- Scalar variables start with $ followed by a Perl identifier
- A Perl identifier consists of letters, digits, and underscores, but cannot start with a digit
  Perl identifiers are case sensitive
- In Perl, a variable does not have to be declared before it can be used
- Scalar variables can store any scalar value
  (there are no 'integer variables' versus 'string variables')

Scalar variables

- A variable also does not have to be initialised before it can be used, although initialisation is a good idea

- Uninitialised variables have the special value `undef`
  However, `undef` acts
  like 0 for numeric variables and
  like '' for string variables
  if an uninitialised variable is used in an arithmetic or string operation

- To test whether a variable has value `undef` use the routine defined


```
$s1 = "";
print '$s1␣eq␣undef:␣',($s1 eq undef) ? 'TRUE':'FALSE',"\n";
print '$s1␣defined:␣␣',(defined($s1)) ? 'TRUE':'FALSE',"\n";
print '$s2␣defined:␣␣',(defined($s2)) ? 'TRUE':'FALSE',"\n";
```

```
$s1 eq undef: TRUE
$s1 defined:  TRUE
$s2 defined:  FALSE
```

Special Variables

- Perl has a lot of ‘pre-defined’ variables that have a particular meaning and serve a particular purpose

Variable	Explanation
`$_`	The default or implicit variable
`@_`	Subroutine parameters
`$a, $b`	sort comparison routine variables
`$&`	the string matched by the last successful pattern match
`$/`	input record separator, newline by default
`$\`	output record separator, `undef` by default
`$]`	version of Perl used

- For a full list see
`https://perldoc.perl.org/perlvar.html#SPECIAL-VARIABLES`

Constants

Perl offers three different ways to declare constants

- Using the `constant` pragma:

```
use constant PI => 3.14159265359;
```

  (A pragma is a module which influences some aspect of the compile time or run time behaviour of Perl)

- Using the `Readonly` module:

```
use Readonly;
Readonly $PI => 3.14159265359;
```

- Using the `Const::Fast` module:

```
use Const::Fast;
const $PI => 3.14159265359;
```

With our current Perl installation only `constant` works

⇝ variable interpolation with constants does not work

Assignments

- Just like Java, Perl uses the equality sign = for assignments:

```
$student_id = 200846369;
$name       = 'Jan␣Olsen';
$student_id = "E00481370";
```

But no type declaration is required and the same variable can hold a number at one point and a string at another

- An assignment also returns a value,
namely (the final value of) the variable on the left
⇝ enables us to use an assignment as an expressions

Example:

```
$b = ($a = 0) + 1;
# $a has value 0
# $b has value 1
```

Binary assignments

There are also binary assignment operators that serve as shortcuts for arithmetic and string operations

Binary assignment	Equivalent assignment
`$a += $b`	`$a = $a + $b`
`$a -= $b`	`$a = $a - $b`
`$a *= $b`	`$a = $a * $b`
`$a /= $b`	`$a = $a / $b`
`$a %= $b`	`$a = $a % $b`
`$a **= $b`	`$a = $a ** $b`
`$a .= $b`	`$a = $a . $b`

Example:

```
# Convert Fahrenheit to Celsius:
# Subtract 32, then multiply by 5, then divide by 9
$temperature = 105;             # temperature in Fahrenheit
($temperature -= 32) *= 5/9;    # converted to Celsius
```

Variable declarations

- In Perl, variables can be declared using the `my` function
  (Remember: This is not a requirement)

- The pragma

```
use strict;
```

enforces that all variables must be declared before their use, otherwise a compile time error is raised

Example:

```
use strict;
$studentsOnCOMP284 = 133;
```



```
Global symbol "$studentOnCOMP284" requires explicit
   package name at ./script line 2.
Execution of ./script aborted due to compilation errors.
```

```
use strict;
my $studentsOnCOMP281;
$studentsOnCOMP281 = 154;
my $studentsOnCOMP283 = 53;
```

Variable interpolation

Variable interpolation

Any scalar variable name in a double quoted string is (automatically) replaced by its current value

Example:


```
$actor = "Jeff␣Bridges";
$prize = "Academy␣Award␣for␣Best␣Actor";
$year  = 2010;
print "1:␣$actor␣won␣the␣$prize␣in␣$year\n";
print "2:␣",$actor,"␣won␣the␣",$prize,"␣in␣",$year,"\n";
```


Output:

```
1: Jeff Bridges won the Academy Award for Best Actor in 2010
2: Jeff Bridges won the Academy Award for Best Actor in 2010
```

Revision

Read

- Chapter 2: Scalar Data

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.
Harold Cohen Library: 518.579.86.S39 or e-book

COMP284 Scripting Languages

Lecture 3: Perl (Part 2)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Control structures: conditional statements

The general format of conditional statements is very similar to that in Java:

```
if (condition) {
    statements
} elsif (condition) {
    statements
} else {
    statements
}
```

- *condition* is an arbitrary expression
- the elsif-clauses is optional and there can be more than one
- the else-clause is optional but there can be at most one
- in contrast to Java, the curly brackets must be present even if *statements* consist only of a single statement

Control structures: conditional statements

- Perl also offers two shorter conditional statements:

```
statement if (condition);
```

and

```
statement unless (condition);
```


- In analogy to conditional statements
  Perl offers conditional expressions:

```
condition ? if_true_expr : if_false_expr
```


Examples:


```
$descr = ($distance < 50) ? "near" : "far";

$size  = ($width < 10) ? "small" :
         ($width < 20) ? "medium" :
                         "large";
```

Blocks

- A sequence of statements in curly brackets is a block
  ⇝ an alternative definition of conditional statements is


```
if (condition) block
elsif (condition) block
else block
```

- In


```
statement if (condition);
statement unless (condition);
```


only a single statement is allowed
but `do block` counts as a single statement,
so we can write

```
do block if (condition);
do block unless (condition);
```

Control structures: switch statement/expression

Starting with Perl 5.10 (released Dec 2007), the language includes a switch statement and corresponding switch expression

But these are considered experimental and need to be enabled explicitly

Example:


```
use feature "switch";

given ($month) {
  when ([1,3,5,7,8,10,12]) { $days = 31 }
  when ([4,6,9,11])        { $days = 30 }
  when (2)                 { $days = 28 }
  default                  { $days = 0  }
}
```


Note: No explicit break statement is needed

Control structures: while- and until-loops

- Perl offers while-loops and until-loops

```
while (condition) {
    statements
}

until (condition) {
    statements
}
```


- A 'proper' until-loop where the loop is executed at least once can be obtained as follows

```
do { statements } until (condition);
```

The same construct also works for `if`, `unless` and `while`

In case there is only a single statement it is also possible to write

```
statement until (condition);
```

Again this also works for `if`, `unless` and `while`

Control structures: for-loops

- for-loops in Perl take the form

```
for (initialisation; test; increment) {
    statements;
}
```


Again, the curly brackets are required even if the body of the loop only consists of a single statement

- Such a for-loop is equivalent to the following while-loop:

```
initialisation;
while (test) {
    statements;
    increment;
}
```

Lists and Arrays

- A list is an ordered collection of scalars
- An array (array variable) is a variable that contains a list

  - Array variables start with @ followed by a Perl identifier

    ```
    @identifier
    ```

    An array variable denotes the entire list stored in that variable

  - Perl uses

    ```
    $identifier[index]
    ```

    to denote the element stored at position *index* in @*identifier*

    The first array element has index 0

- Note that

  ```
  $identifier
  @identifier
  ```

  are two unrelated variables (but this situation should be avoided)

List literals

- A list can be specified by a list literal, a comma-separated list of values enclosed by parentheses

```
(1, 2, 3)
("adam", "ben", "colin", "david")
("adam", 1, "ben", 3)
( )
(1..10, 15, 20..30)
($start..$end)
```

- List literals can be assigned to an array:

```
@numbers = (1..10, 15, 20..30);
@names = ("adam", "ben", "colin", "david");
```

- Examples of more complex assignments, involving array concatenation:

```
@numbers = (1..10, undef, @numbers, ( ));
@names   = (@names,@numbers);
```

- Note that arrays do not have a pre-defined size/length

Size of an array

- There are three different ways to determine the size of an array

```
$arraySize = scalar(@array);
$arraySize = @array;
$arraySize = $#array + 1;
```

- One can access all elements of an array using indices in the range 0 to `$#array`
- But Perl also allows negative array indices:
  The expression `$array[-index]`
  is equivalent to `$array[scalar(@array)-index]`

  Example:
  `$array[-1]` is the same as `$array[scalar(@array)-1]`
  is the same as `$array[$#array]`
  that is the last element in `@array`

Array index out of bound

- Perl, in contrast to Java, allows you to access array indices that are out of bounds
- The value undef will be returned in such a case

```
@array = (0, undef, 22, 33);
print '$array[1]␣=␣',$array[1], ',␣which␣',
      (defined($array[1]) ? "IS␣NOT" : "IS", "undef\n";
print '$array[5]␣=␣',$array[5], ',␣which␣',
      (defined($array[5]) ? "IS␣NOT" : "IS", "undef\n";
```

```
$array[1] = , which IS undef
$array[5] = , which IS undef
```

- The function exists can be used to determine whether an array index is within bounds and has a value (including undef) associated with it

```
print '$array[1]␣exists:␣',exists($array[1]) ? "T":"F","\n";
print '$array[5]␣exists:␣',exists($array[5]) ? "T":"F","\n";
```

```
$array[1] exists: T
$array[5] exists: F
```

Scalar context versus list context

- Scalar context
  when an expression is used as an argument of an operation that requires a scalar value, the expression will be evaluated in a scalar context

  Example:

  ```
  $arraySize = @array;
  ```

  ⇝ @array stores a list, but returns the number of elements of @array in a scalar context

- List context
  when an expression is used as an argument of an operation that requires a list value, the expression will be evaluated in a list context

  Example:

  ```
  @sorted = sort 5;
  ```

  ⇝ A single scalar value is treated as a list with one element in a list context

Scalar context versus list context

Expressions behave differently in different contexts following these rules:

- Some operators and functions automatically return different values in different contexts


```
$line  = <IN>;      # return one line from IN
@lines = <IN>;      # return a list of all lines from IN
```


- If an expression returns a scalar value in a list context, then by default Perl will convert it into a list value with the returned scalar value being the one and only element

- If an expression returns a list value in a scalar context, then by default Perl will convert it into a scalar value by take the last element of the returned list value

List functions

Function	Semantics
`grep(`*expr*`,`*list*`)`	in a list context, returns those elements of *list* for which *expr* is true; in a scalar context, returns the number of times the expression was true
`join(`*string*`,`*list*`)`	returns a string that contains the elements of *list* connected through a separator *string*
`reverse(`*list*`)`	returns a list with elements in reverse order
`sort(`*list*`)`	returns a list with elements sorted in standard string comparison order
`split(/`*regexpr*`/,`*string*`)`	returns a list obtained by splitting *string* into substring using *regexpr* as separator
`(`*list*`) x `*number*	returns a list composed of *number* copies of *list*

Array functions: push, pop, shift, unshift

Perl has no stack or queue data structures,
but has stack and queue functions for arrays:

Function	Semantics
`push(@array1,value)` `push(@array1,list)`	appends an element or an entire list to the end of an array variable; returns the number of elements in the resulting array
`pop(@array1)`	extracts the last element from an array and returns it
`shift(@array1)`	shift extracts the first element of an array and returns it
`unshift(@array1,value)` `unshift(@array1,list)`	insert an element or an entire list at the start of an array variable; returns the number of elements in the resulting array

Array operators: push, pop, shift, unshift

Example:

```
@planets = ("earth");
unshift(@planets,"mercury","venus");
push(@planets,"mars","jupiter","saturn");
print "Array\@1:␣", join("␣",@planets),"\n";
$last = pop(@planets);
print "Array\@2:␣", join("␣",@planets),"\n";
$first = shift(@planets);
print "Array\@3:␣", join("␣",@planets),"\n";
print "␣␣␣␣␣\@4:␣",$first,"␣",$last,"\n";
```


Output:

```
Array@1: mercury venus earth mars jupiter saturn
Array@2: mercury venus earth mars jupiter
Array@3: venus earth mars jupiter
     @4: mercury saturn
```

Array operators: delete

- It is possible to delete array elements
- delete($*array*[*index*])
  - removes the value stored at *index* in @*array* and returns it
  - only if *index* equals $#*array* will the array's size shrink to the position of the highest element that returns true for exists()


```
@array = (0, 11, 22, 33);
delete($array[2]);
print '$array[2] exists: ',exists($array[2])?"T":"F", "\n";
print 'Size of $array: ',$#array+1,"\n";
delete($array[3]);
print '$array[3] exists: ',exists($array[3])?"T":"F", "\n";
print 'Size of $array: ',$#array+1,"\n";
```

```
$array[2] exists: F
Size of $array: 4
$array[3] exists: F
Size of $array: 2
```

Control structures: foreach-loop

Perl provides the foreach-construct to 'loop' through the elements of a list

```
foreach $variable (list) {
  statements
}
```

where $variable, the foreach-variable, stores a different element of the list in each iteration of the loop

Example:

```
@my_list = (1..5,20,11..18);
foreach $number (@my_list) {
  $max = $number if (!defined($max) || $number > $max);
}
print("Maximum number in ",join(',',@my_list)," is $max\n");
```

Output:

```
Maximum number in 1,2,3,4,5,20,11,12,13,14,15,16,17,18 is 20
```

Control structures: foreach-loop

Changing the value of the foreach-variable changes the element of the list that it currently stores

Example:

```
@my_list = (1..5,20,11..18);
print "Before:␣".join(",␣",@my_list)."\n";
foreach $number (@my_list) {
  $number++;
}
print "After:␣␣".join(",␣",@my_list)."\n";
```


Output:


```
Before: 1, 2, 3, 4, 5, 20, 11, 12, 13, 14, 15, 16, 17, 18
After:  2, 3, 4, 5, 6, 21, 12, 13, 14, 15, 16, 17, 18, 19
```

Note: If no variable is specified, then the special variable $_ will be used to store the array elements

Control structures: foreach-loop

An alternative way to traverse an array is

```
foreach $index (0..$#array) {
  statements
}
```

where an element of the array is then accessed using $array[$index] in *statements*

Example:

```
@my_list = (1..5,20,11..18);
foreach $index (0..$#my_list) {
  $max = $my_list[$index] if ($my_list[$index] > $max);
}
print("Maximum number in ",join(',',@my_list)," is $max\n");
```

Control structures: foreach-loop

- In analogy to while- and until-loops, there are the following variants of foreach-loops:

```
do { statements } foreach list;
```

```
statement foreach list;
```

In the execution of the statements within the loop, the special variable $_ will be set to consecutive elements of *list*

- Instead of foreach we can also use for:

```
do { statements } for list;
```

```
statement for list;
```


Example:

```
print "Hello $_!\n" foreach ("Peter","Paul",
                             "Mary");
```

Control structures: last and next

- The last command can be used in while-, until-, and foreach-loops and discontinues the execution of a loop

```
while ($value = shift($data)) {
  $written = print(FILE $value);
  if (!$written) { last; }
}
# Execution of 'last' takes us here
```


- The next command stops the execution of the current iteration of a loop and moves the execution to the next iteration

```
foreach $x (-2..2) {
   if ($x == 0) { next; }
   printf("10 / %2d = %3d\n",$x,(10/$x));
}
```

```
10 / -2 =  -5
10 / -1 = -10
10 /  1 =  10
10 /  2 =   5
```

Hashes

- A hash is a data structure similar to an array but it associates scalars with a string instead of a number
- Alternatively, a hash can be seen as a partial function mapping strings to scalars
- Remember that Perl can auto-magically convert any scalar into a string

- Hash variables start with a percent sign followed by a Perl identifier

```
%identifier
```

A hash variable denotes the entirety of the hash

- Perl uses

```
$identifier{key}
```

where *key* is a string, to refer to the value associated with *key*

Hashes

- Note that

```
$identifier
%identifier
```


are two unrelated variables (but this situation should be avoided)

- An easy way to print all key-value pairs of a hash %*hash* is the following


```
use Data::Dumper;
$Data::Dumper::Terse = 1;
print Dumper \%hash;
```


Note the use of \%*hash* instead of %*hash*
(\%*hash* is a reference to %*hash*)

`Data::Dumper` can produce string representations for arbitrary Perl data structures

Basic hash operations

- Initialise a hash using a list of key-value pairs

```
%hash = (key1, value1, key2, value2, ...);
```

- Initialise a hash using a list in big arrow notation

```
%hash = (key1 => value1, key2 => value2, ...);
```

- Associate a single value with a key

```
$hash{key} = value;
```

- Remember that undef is a scalar value

```
$hash{key} = undef;
```

extends a hash with another key but unknown value

Basic hash operations

- One can use the `exists` or `defined` function to check whether a key exists in a hash:

```
if (exists $hash{key}) { ... }
```

  Note that if `$hash{key} eq undef`, then `exists $hash{key}` is true

- The `delete` function removes a given key and its corresponding value from a hash:

```
delete($hash{key});
```

  After executing `delete($hash{key})`, `exists $hash{key}` will be false

- The `undef` function removes the contents and memory allocated to a hash:

```
undef %hash
```

Basic hash operations

- It is also possible to assign one hash to another

```
%hash1 = %hash2;
```


In contrast to C or Java this operation creates a copy of %hash2 that is then assigned to %hash1

Example:


```
%hash1 = ('a' => 1, 'b' => 2);
%hash2 = %hash1;
$hash1{'b'} = 4;
print "\$hash1{'b'} = $hash1{'b'}\n";
print "\$hash2{'b'} = $hash2{'b'}\n";
```


Output:

```
$hash1{'b'} = 4
$hash2{'b'} = 2
```

The each, keys, and values functions

`each %hash`	returns a 2-element list consisting of the key and value for the next element of `%hash`, so that one can iterate over it
`values %hash`	returns a list consisting of all the values of `%hash`, resets the internal iterator for `%hash`
`keys %hash`	returns a list consisting of all keys of `%hash`, resets the internal iterator for `%hash`

Examples:


```
while ( ($key,$value) = each %hash ) {
  statements
}
```

```
foreach $key (sort keys %hash) {
  $value = $hash{$key};
}
```

Example: Two-dimensional hash as a 'database'

```perl
use List::Util "sum";
$name{'200846369'} = 'Jan Olsen';
$marks{'200846369'}{'COMP201'} = 61;
$marks{'200846369'}{'COMP207'} = 57;
$marks{'200846369'}{'COMP213'} = 43;
$marks{'200846369'}{'COMP219'} = 79;

$average = sum(values($marks{'200846369'}))/
           scalar(values($marks{'200846369'}));
print("avg: $average\n");
```

Output:

```
avg: 60
```

Example: Frequency of words

```
# Establish the frequency of words in a string
$string = "peter paul mary paul jim mary paul";

# Split the string into words and use a hash
# to accumulate the word count for each word
++$count{$_} foreach split(/\s+/,$string);

# Print the frequency of each word found in the
# string
while ( ($key,$value) = each %count ) {
  print("$key => $value; ");
}
```


Output:

```
jim => 1; peter => 1; mary => 2; paul => 3
```

Revision

Read

- Chapter 3: Lists and Arrays
- Chapter 6: Hashes

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.
Harold Cohen Library: 518.579.86.S39 or e-book

COMP284 Scripting Languages
Lecture 4: Perl (Part 3)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Regular expressions: Motivation

Suppose you are testing the performance of a new sorting algorithm by measuring its runtime on randomly generated arrays of numbers of a given length:


```
Generating an unsorted array with 10000 elements took 1.250 seconds
Sorting took 7.220 seconds
Generating an unsorted array with 10000 elements took 1.243 seconds
Sorting took 10.486 seconds
Generating an unsorted array with 10000 elements took 1.216 seconds
Sorting took 8.951 seconds
```


Your task is to write a program that determines the average runtime of the sorting algorithm:


```
Average runtime for 10000 elements is 8.886 seconds
```

Solution: The regular expression `/^Sorting took (\d+\.\d+) seconds/` allows us to get the required information

⇝ Regular expressions are useful for information extraction

Regular expressions: Motivation

Suppose you have recently taken over responsibility for a company's website. You note that their HTML files contain a large number of URLs containing superfluous occurrences of '..', e.g.


```
http://www.myorg.co.uk/info/refund/../vat.html
```

Your task is to write a program that replaces URLs like these with equivalent ones without occurrences of '..':


```
http://www.myorg.co.uk/info/vat.html
```

while making sure that relative URLs like

```
../video/disk.html
```


are preserved

Solution: `s!/[^\/]+/\.\.!!;` removes a superfluous dot-segment

⇝ Substitution of regular expressions is useful for text manipulation

Regular expressions: Introductory example

```
\Ahttps?:\/\/[^\/]+\/.\w.\/(cat|dog)\/\1
```


- `\A` is an assertion or anchor
- `h`, `t`, `p`, `s`, `:`, `\/`, `c`, `a`, `t`, `d`, `o`, `g` are characters
- `?` and `+` are quantifiers
- `[^\/]` is a character class
- `.` is a metacharacter and `\w` is a special escape
- `(cat|dog)` is alternation within a capture group
- `\1` is a backreference to a capture group

Pattern match operation

- To match a regular expession *regexpr* against the special variable `$_` simply use one of the expressions `/`*regexpr*`/` or `m/`*regexpr*`/`
  - This is called a pattern match
  - `$_` is the target string of the pattern match
- In a scalar context a pattern match returns true (1) or false ('') depending on whether *regexpr* matches the target string

```
if (/\Ahttps?:\/\/[^\/]+\/.\w.\/(cat|dog)\/\1/) {
   ... }

if (m/\Ahttps?:\/\/[^\/]+\/.\w.\/(cat|dog)\/\1/) {
   ... }
```

Regular expressions: Characters

The simplest regular expression just consists of a sequence of

- alphanumberic characters and
- non-alphanumeric characters escaped by a backslash:

that matches exactly this sequence of characters occurring as a substring in the target string


```
$_ = "ababcbcdcde";
if (/cbc/) { print "Match\n"} else { print "No␣match\n" }
```

Output:

```
Match
```

```
$_ = "ababcbcdcde";
if (/dbd/) { print "Match\n"} else { print "No␣match\n" }
```

Output:

```
No match
```

Regular expressions: Special variables

- Often we do not just want to know whether a regular expession matches a target string, but retrieve additional information
- The special variable $-[0] can be used to retrieve the start position of the match

  Note that positions in strings are counted starting with 0
- The special variable $+[0] can be used to retrieve the first position after the match
- The special variable $& returns the match itself


```
$_ = "ababcbcdcde";
if (/cbc/) { print "Match found at position $-[0]: $&\n"}
```

Output:

```
Match found at position 4: cbc
```

Regular expressions: Special escapes

There are various special escapes and metacharacters that match more then one character:

`.`	Matches any character except `\n`
`\w`	Matches a 'word' character (alphanumeric plus '_', plus other connector punctuation characters plus Unicode characters
`\W`	Matches a non-'word' character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a decimal digit character
`\D`	Match a non-digit character
`\p{`*UnicodeProperty*`}`	Match *UnicodeProperty* characters
`\P{`*UnicodeProperty*`}`	Match non-*UnicodeProperty* characters

Regular expressions: Unicode properties

- Each unicode character has one or more properties, for example, which script it belongs it
- `\p{UnicodeProperty}` matches all characters that have a particular property
- `\P{UnicodeProperty}` matches those that do not
- Examples of unicode properties are

`Arabic`	Arabic characters
`ASCII`	ASCII characters
`Currency_Symbol`	Currency symbols
`Digit`	Digits in all scripts
`Greek`	Greek characters
`Han`	Chinese kanxi or Japanese kanji characters
`Space`	Whitespace characters

See `http://perldoc.perl.org/perluniprops.html` for a complete list

Regular expressions: Character class

- A character class, a list of characters, special escapes, metacharacters and unicode properties enclosed in square brackets, matches any single character from within the class;
  for example, `[ad\t\n\-\\09]`
- One may specify a range of characters with a hyphen -,
  for example, `[b-u]`
- A caret ^ at the start of a character class negates/complements it, that is, it matches any single character that is not from within the class,
  for example, `[^01a-z]`


```
$_ = "ababcbcdcde";
if (/[bc][b-e][^bcd]/) {
   print "Match␣at␣positions␣$-[0]␣to␣",$+[0]-1,":␣$&\n"};
```

Output:

```
Match at positions 8 to 10: cde
```

Quantifiers

- The constructs for regular expressions that we have so far are not sufficient to match, for example, natural numbers of arbitrary size
- Also, writing a regular expressions for, say, a nine digit number would be tedious

This is made possible with the use of quantifiers

*regexpr**	Match *regexpr* 0 or more times
regexpr+	Match *regexpr* 1 or more times
regexpr?	Match *regexpr* 1 or 0 times
regexpr{*n*}	Match *regexpr* exactly n times
regexpr{*n*,}	Match *regexpr* at least n times
regexpr{*n*,*m*}	Match *regexpr* at least n but not more than m times

Quantifiers are greedy by default and match the longest leftmost sequence of characters possible

Quantifiers

regexpr`*`	Match *regexpr* 0 or more times
regexpr`+`	Match *regexpr* 1 or more times
regexpr`?`	Match *regexpr* 1 or 0 times
regexpr`{`*n*`}`	Match *regexpr* exactly n times
regexpr`{`*n*`,}`	Match *regexpr* at least n times
regexpr`{`*n*`,`*m*`}`	Match *regexpr* at least n but not more than m times

Example:

```
$_ = "Sorting took 10.486 seconds";
if (/\d+\.\d+/) {
   print "Match at positions $-[0] to ",$+[0]-1,": $&\n"};
$_ = "E00481370";
if (/[A-Z]0{2}(\d+)/) {
   print "Match at positions $-[1] to ",$+[1]-1,": $1\n"};
```


Output:

```
Match at positions 13 to 18: 10.486
Match at positions 3 to 8: 481370
```

Quantifiers

Example:

```
$_ = "E00481370";
if (/\d+/) {
   print "Match␣at␣positions␣$-[0]␣to␣",$+[0]-1,":␣$&\n"};
```


Output:

```
Match at positions 1 to 8: 00481370
```


- The regular expression \d+ matches 1 or more digits
- As the example illustrates, the regular expression \d+
  - matches as early as possible
  - matches as many digits as possible

  ⇝ quantifiers are greedy by default

Revision

Read

- Chapter 7: In the World of Regular Expressions
- Chapter 8: Matching with Regular Expressions

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.

- `http://perldoc.perl.org/perlre.html`
- `http://perldoc.perl.org/perlretut.html`
- `http://www.perlfect.com/articles/regextutor.shtml`

COMP284 Scripting Languages
Lecture 5: Perl (Part 4)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Regular expressions: Capture groups and backreferences

- We often encounter situations where we want to identify the repetition of the same or similar text, for example, in HTML markup:


```
<strong> ... </strong>
<li> ... </li>
```

- We might also not just be interested in the repeating text itself, but the text between or outside the repetition

- We can characterise each individual example above using regular expressions:


```
<strong>.*<\/strong>
<li>.*<\/li>
```

but we cannot characterise both without losing fidelity, for example:

```
<\w+>.*<\/\w+>
```

does not capture the 'pairing' of HTML tags

Regular expressions: Capture groups

The solution are capture groups and backreferences

(*regexpr*)	creates a capture group
(?<*name*>*regexpr*)	creates a named capture group
(?:*regexpr*)	creates a non-capturing group
N, \g*N*, \g{*N*}	backreference to capture group *N* (where *N* is a natural number)
\g{*name*}	backreference to a named capture group

Examples:

```
/Sorting took (\d+\.\d+) seconds/
/<(\w+)>.*<\/\1>/
/([A-Z])0{2}(\d+)/
/(?<c1>\w)(?<c2>\w)\g{c2}\g{c1}/
/((?<c1>\w)(?<c2>\w)\g{c2}\g{c1})/
```

Regular expressions: Capture groups

Via capture variables the strings matched by a capture group are also available outside the pattern in which they are contained

$*N*	string matched by capture group *N* (where *N* is a natural number)
$+{*name*}	string matched by a named capture group

The matched strings are available until the end of the enclosing code block or until the next successful match

Example:


```
$_ = "Yabba␣dabba␣doo"
if (/((?<c1>\w)(?<c2>\w)\g{c2}\g{c1})/) {
  print "Match␣found:␣$1\n" }
```

Output:

```
Match found: abba
```

Regular expressions: Alternations

- The regular expression *regexpr1*|*regexpr2* matches if either *regexpr1* or *regexpr2* matches

  This type of regular expression is called an alternation

- Within a larger regular expression we need to enclose alternations in a capture group or non-capturing group:

  (*regexpr1*|*regexpr2*) or (?:*regexpr1*|*regexpr2*)

Examples:

```
/Mr|Ms|Mrs|Dr/
/cat|dog|bird/
/(?:Bill|Hillary) Clinton/
```

Regular expressions: Alternations

- The order of expressions in an alternation only matters if one expression matches a sub-expression of another

Example:

```
$_ = "cats␣and␣dogs";
if (/(cat|dog|bird)/) { print "Match␣1:␣$1\n" }
if (/(dog|cat|bird)/) { print "Match␣2:␣$1\n" }
if (/(dog|dogs)/) { print "Match␣3:␣$1\n" }
if (/(dogs|dog)/) { print "Match␣4:␣$1\n" }
```


Output:

```
Match 1: cat
Match 2: cat
Match 3: dog
Match 4: dogs
```

Regular expressions: Anchors

Anchors allow us to fix where a match has to start or end

\A	Match only at string start
^	Match only at string start (default) Match only at a line start (in //m)
\Z	Match only at string end modulo a preceding \n
\z	Match only at string end
$	Match only at string end modulo a preceding \n Match only at a line end (in //m)
\b	Match word boundary (between \w and \W)
\B	Match except at word boundary

Example:

```
$_ = "The girl who\nplayed with fire\n";
if (/fire\z/) { print "'fire' at string end\n" }
if (/fire\Z/) { print "'fire' at string end modulo \\n\n" }
```

```
'fire' at string end modulo \n
```

Regular expressions: Modifiers

Modifiers change the interpretation of certain characters in a regular expression or the way in which Perl finds a match for a regular expression

`/ /`	Default '`.`' matches any character except '`\n`' '`^`' matches only at string start '`$`' matches only at string end modulo preceding `\n`
`/ /s`	Treat string as a single long line '`.`' matches any character including '`\n`' '`^`' matches only at string start '`$`' matches only at string end modulo preceding `\n`
`/ /m`	Treat string as a set of multiple lines '`.`' matches any character except '`\n`' '`^`' matches at a line start '`$`' matches at a line end

Regular expressions: Modifiers

Modifiers change the interpretation of certain characters in a regular expression or the way in which Perl finds a match for a regular expression

`/ /sm`	Treat string as a single long line, but detect multiple lines '.' matches any character including '\n' '^' matches at a line start '$' matches at a line end
`/ /i`	perform a case-insensitive match

Example:

```
$_ = "bill\nClinton";
if (/(Bill|Hillary).Clinton)/smi) { print "Match:␣$1\n" }
```

Output:

```
Match: bill
Clinton
```

Regular expressions: Modifiers (/ /g and / /c)

Often we want to process all matches for a regular expression, but the following code has not the desired effect


```
$_ = "11␣22␣33";
while (/\d+/) { print "Match␣starts␣at␣$-[0]:␣$&\n" }
```

The code above does not terminate and endlessly prints out the same text:


```
Match starts at 0: 11
```

To obtain the desired behaviour of the while-loop we have to use the / /g modifier:

/ /g	In scalar context, successive invocations against a string will move from match to match, keeping track of the position in the string
	In list context, returns a list of matched capture groups, or if there are no capture groups, a list of matches to the whole regular expression

Regular expressions: Modifiers (/ /g and / /c)

With the / /g modifier our code works as desired:

```
$_ = "11␣22␣33";
while (/\d+/g) { print "Match␣starts␣at␣$-[0]:␣$&\n" }
```


Output:

```
Match starts at 0: 11
Match starts at 3: 22
Match starts at 6: 33
```


An example in a list context is the following:

```
$_ = "ab␣11␣cd␣22␣ef␣33";
@numbers = (/\d+/g);
print "Numbers:␣",join("␣|␣",@numbers),"\n";
```


Output:

```
Numbers: 11 | 22 | 33
```

Read / /g as: Start to look for a match from the position where the last match using / /g ended

Regular expressions: Modifiers (/ /g and / /c)

The current position in a string for a regular expression *regexpr* is associated with the string, not *regexpr*

- ⇝ different regular expressions for the same strings will move forward the same position when used with / /g
- ⇝ different strings have different positions and their respective positions move forward independently

Example:

```
$_ = "ab␣11␣cd␣22␣ef␣33";
if (/\d+/g)    { print "Match␣starts␣at␣$-[0]:␣$&\n" }
if (/[a-z]+/g) { print "Match␣starts␣at␣$-[0]:␣$&\n" }
if (/\d+/g)    { print "Match␣starts␣at␣$-[0]:␣$&\n" }
```


Output:

```
Match starts at 3: 11
Match starts at 6: cd
Match starts at 9: 22
```

Regular expressions: Modifiers (/ /g and / /c)

A failed match or changing the target string resets the position

```
$_ = "ab 11 cd 22 ef 33";
if (/\d+/g) { print "2: Match starts at $-[0]: $&\n" }
if (/ab/g)  { print "3: Match starts at $-[0]: $&\n" }
if (/\d+/g) { print "4: Match starts at $-[0]: $&\n" }
```

Output:

```
2: Match starts at 3: 11
4: Match starts at 3: 11
```

To prevent the reset, an additional modifier / /c can be used

```
$_ = "ab 11 cd 22 ef 33";
if (/\d+/g) { print "2: Match starts at $-[0]: $&\n" }
if (/ab/gc) { print "3: Match starts at $-[0]: $&\n" }
if (/\d+/g) { print "4: Match starts at $-[0]: $&\n" }
```

Output:

```
2: Match starts at 3: 11
4: Match starts at 9: 22
```

Generating regular expressions on-the-fly

The Perl parser will expand occurrences of $*variable* and @*variable* in regular expressions

⇝ regular expressions can be constructed at runtime

Example:

```
$_ = "Bart␣teases␣Lisa";
@keywords = ("bart","lisa","marge",'L\w+',"t\w+");
while ($keyword = shift(@keywords)) {
  print "Match␣found␣for␣$keyword:␣$&\n" if /$keyword/i;
}
```


Output:


```
Match found for bart: Bart
Match found for lisa: Lisa
Match found for L\w+: Lisa
Match found for t\w+: teases
```

Binding operator

Perl offers two binding operators for regular expressions

string =~ /*regexpr*/	true iff *regexpr* matches *string*
string !~ /*regexpr*/	true iff *regexpr* does not match *string*

- Note that these are similar to comparison operators not assignments
- Most of the time we are not just interested whether these expressions return true or false, but in the side effect they have on the special variables $*N* that store the strings matched by capture groups

Example:


```
$name = "Dr Ullrich Hustadt";
if ($name =~ /(Mr|Ms|Mrs|Dr)?\s*(\w+)/) {print "Hello $2\n"}
$name = "Dave Shield";
if ($name =~ /(Mr|Ms|Mrs|Dr)?\s*(\w+)/) {print "Hello $2\n"}
```

```
Hello Ullrich
Hello Dave
```

Pattern matching in a list context

- When a pattern match /*regexpr*/ is used in a list context, then the return value is
  - a list of the strings matched by the capture groups in *regexpr* if the match succeeds and *regexpr* contains capture groups, or
  - (a list containing) the value 1 if the match succeeds and *regexpr* contains no capture groups, or
  - an empty list if the match fails


```
$name = "Dr Ullrich Hustadt";
($t,$f,$l) = ($name =~ /(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/);
print "Name: $t, $f, $l\n";
$name = "Dave Shield";
($t,$f,$l) = ($name =~ /(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/);
print "Name: $t, $f, $l\n";
```

Output:

```
Name: Dr, Ullrich, Hustadt
Name: , Dave, Shield
```

Pattern matching in a list context

- When a pattern match /*regexpr*/g is used in a list context, then the return value is
  - a list of the strings matched by the capture groups in *regexpr* each time regex matches provided that *regexpr* contains capture groups, or
  - a list containing the string matched by *regexpr* each time *regexpr* matches provided that *regexpr* contains no capture groups, or
  - an empty list if the match fails


```
$string = "firefox: 10.3 seconds; chrome: 9.5 seconds";
%performance = ($string =~ /(\w+)\:\s+(\d+\.\d+)/g);
foreach $system (keys %performance) {
   print "$system -> $performance{$system}\n" }
```

Output:

```
firefox -> 10.3
chrome -> 9.5
```

Revision

Read

- Chapter 7: In the World of Regular Expressions
- Chapter 8: Matching with Regular Expressions

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011

- http://perldoc.perl.org/perlre.html
- http://perldoc.perl.org/perlretut.html
- http://www.perlfect.com/articles/regextutor.shtml

COMP284 Scripting Languages

Lecture 6: Perl (Part 5)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Substitutions

s/*regexpr*/*replacement*/

- Searches a variable for a match for *regexpr*, and if found, replaces that match with a string specified by *replacement*
- In both scalar context and list context returns the number of substitutions made (that is, 0 if no substitutions occur)
- If no variable is specified via one of the binding operators =~ or !~, the special variable $_ is searched and modified
- The binding operator !~ only negates the return value but does not affect the manipulation of the text

The delimiter / can be replaced by some other paired or non-paired character, for example:

s!*regexpr*!*replacement*! or s<*regexpr*>[*replacement*]

Substitutions

Example:

```
$text = "http://www.myorg.co.uk/info/refund/../vat.html";
$text =~ s!/[^\/]+/\.\.!!;
print "$text\n";
```

Output:

```
http://www.myorg.co.uk/info/vat.html
```

Example:

```
$_ = "Yabba dabba doo";
s/bb/dd/;
print $_,"\n";
```

Output:

```
Yadda dabba doo
```

Note: Only the first match is replaced

Substitutions: Capture variables

`s/regexpr/replacement/`

- Perl treats *replacement* like a double-quoted string

⇝ backslash escapes work as in a double-quoted string

`\n`	Newline
`\t`	Tab
`\l`	Lower case next letter
`\L`	Lower case all following letters until `\E`
`\u`	Upper case next letter
`\U`	Upper case all following letters until `\E`

⇝ variable interpolation is applied, including capture variables

`$N`	string matched by capture group *N* (where *N* is a natural number)
`$+{name}`	string matched by a named capture group

Substitutions: Capture variables

Example:

```
$name = "Dr␣Ullrich␣Hustadt";
$name =~ s/(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/\U$3\E, $2/;
print "$name\n";

$name = "Dave␣Shield";
$name =~ s/(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/\U$3\E, $2/;
print "$name\n";
```


Output:

```
HUSTADT, Ullrich
SHIELD, Dave
```

Substitutions: Modifiers

Modifiers for substitutions include the following:

s/ / /g	Match and replace globally, that is, all occurrences
s/ / /i	Case-insensitive pattern matching
s/ / /m	Treat string as multiple lines
s/ / /s	Treat string as single line
s/ / /e	Evaluate the right side as an expression

Combinations of these modifiers are also allowed

Example:

```
$_ = "Yabba␣dabba␣doo";
s/bb/dd/g;
print $_,"\n";
```

Output:

```
Yadda dadda doo
```

Substitutions: Modifiers

Modifiers for substitutions include the following:

`s/ / /e`	Evaluate the right side as an expression

Example:

```
$text = "The␣temperature␣is␣105␣degrees␣Fahrenheit";
$text =~ s!(\d+) degrees Fahrenheit!
         (($1-32)*5/9)."␣degrees␣Celsius"!e;
print "$text\n";
$text =~ s!(\d+\.\d+)!sprintf("%d",$1+0.5)!e;
print "$text\n";
```



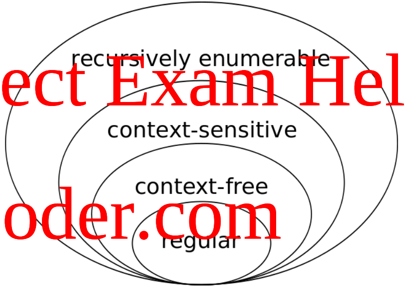
```
The temperature is 40.5555555555556 degrees Celsius
The temperature is 41 degrees Celsius
```


Better:

```
$text = "The␣temperature␣is␣105␣degrees␣Fahrenheit";
$text =~ s!(\d+) degrees Fahrenheit!
           sprintf("%d",(($1-32)*5/9)+0.5).
           "␣degrees␣Celsius"!e;
```

Regular Expressions and the Chomsky Hierarchy

- In Computer Science, formal languages are categorised according to the type of grammar needed to generate them (or the type of automaton needed to recognise them)
- Perl regular expressions can at least recognise all context-free languages



Chomsky Hiearchy of Formal Languages

- Howerver, this does not mean regular expression should be used for parsing context-free languages
- Instead there are packages specifically for parsing context-free languages or dealing with specific languages, e.g. HTML, CSV

Java methods versus Perl subroutines

- Java uses methods as a means to encapsulate sequences of instructions
- In Java you are expected
  - to declare the type of the return value of a method
  - to provide a list of parameters, each with a distinct name, and to declare the type of each parameter


```
public static int sum2( int f, int s) {
  f = f+s;
  return f;
}

public static void main(String[] args) {
  System.out.println("Sum␣of␣3␣and␣4␣is␣" + sum2(3, 4));
}
```

- Instead of methods, Perl uses subroutines

Subroutines

Subroutines are defined as follows in Perl:

```
sub identifier {
  statements
}
```


- Subroutines can be placed anywhere in a Perl script but preferably they should all be placed at start of the script (or at the end of the script)
- All subroutines have a return value (but no declaration of its type)
  - The statement
    `return value`
    can be used to terminate the execution of a subroutine and to make *value* the return value of the subroutine
  - If the execution of a subroutine terminates without encountering a `return` statement, then the value of the last evaluation of an expression in the subroutine is returned

The return value does not have to be scalar value, but can be a list

Parameters and Arguments

Subroutines are defined as follows in Perl:

```
sub identifier {
  statements
}
```


- In Perl there is no need to declare the parameters of a subroutine (or their types)
  ⇝ there is no pre-defined fixed number of parameters
- Arguments are passed to a subroutine via a special array `@_`
- Individual arguments are accessed using `$_[0]`, `$_[1]` etc
- Is is up to the subroutine to process arguments as is appropriate
- The array `@_` is private to the subroutine
  ⇝ each nested subroutine call gets its own `@_` array

Parameters and Arguments: Examples

- The Java method

```
public static int sum2( int f, int s) {
  f = f+s;
  return f;
}
```

could be defined as follows in Perl:

```
sub sum2 {
  return $_[0] + $_[1];
}
```

- A more general solution, taking into account that a subroutine can be given arbitrarily many arguments, is the following:

```
sub sum {
  return undef if (@_ < 1);
  $sum = shift(@_);
  foreach (@_) { $sum += $_ }
  return $sum;
}
```

Private variables

```
sub sum {
  return undef if (@_ < 1);
  $sum = shift(@_);
  foreach (@_) { $sum += $_ }
  return $sum;
}
```

The variable $sum in the example above is global:

```
$sum = 5;
print "Value of \$sum before call of sum: ",$sum,"\n";
print "Return value of sum: ",&sum(5,4,3,2,1),"\n";
print "Value of \$sum after  call of sum: ",$sum,"\n";
```

produces the output

```
Value of $sum before call of sum: 5
Return value of sum: 15
Value of $sum after  call of sum: 15
```

This use of global variables in subroutines is often undesirable
⇝ we want $sum to be private/local to the subroutine

Private variables

- The operator my declares a variable or list of variables to be private:

```
my $variable;
my ($variable1,$variable2);
my @array;
```


- Such a declaration can be combined with a (list) assignment:

```
my $variable = $_[0];
my ($variable1,$variable2) = @_;
my @array = @_;
```


- Each call of a subroutine will get its own copy of its private variables

Example:


```
sub sum {
  return undef if (@_ < 1);
  my $sum = shift(@_);
  foreach (@_) { $sum += $_ }
  return $sum;
}
```

Calling a subroutine

A subroutine is called by using the subroutine name with an ampersand &
in front possibly followed by a list of arguments
The ampersand is optional if a list of arguments is present

```
sub identifier {
  statements
}

... &identifier ...
... &identifier(arguments) ...
... identifier(arguments) ...
```

Examples:

```
print "sum0:␣",&sum,"\n";
print "sum0:␣",sum(),"\n";
print "sum1:␣",sum(5),"\n";
print "sum2:␣",sum(5,4),"\n";
print "sum5:␣",&sum(5,4,3,2,1),"\n";
$total = sum(9,8,7,6)+sum(5,4,3,2,1);
sum(1,2,3,4);
```

Persistent variables

- Private variables within a subroutine are forgotten once a call of the subroutine is completed
- In Perl 5.10 and later versions we can make a variable both private and persistent using the `state` operator
  - ⇝ the value of a persistent variable will be retained between independent calls of a subroutine

Example:

```
use 5.010;

sub running_sum {
  state $sum;
  foreach (@_) { $sum += $_ }
  return $sum;
}
```

Persistent variables

Example:

```
use 5.010;

sub running_sum {
  state $sum;
  foreach (@_) { $sum += $_ }
  return $sum;
}

print "running_sum():\t\t",   running_sum(),      "\n";
print "running_sum(5):\t",    running_sum(5),     "\n";
print "running_sum(5,4):\t",  running_sum(5,4),   "\n";
print "running_sum(3,2,1):\t",running_sum(3,2,1), "\n";
```


Output:

```
running_sum():
running_sum(5):         5
running_sum(5,4):       14
running_sum(3,2,1):     20
```

Nested subroutine definitions

- Perl allows nested subroutine definitions (unlike C or Java)

```
sub outer_sub {
   sub inner_sub { ... }
}
```


- Normally, nested subroutines are a means for information hiding
  ⇝ the inner subroutine should only be visible and executable from inside the outer subroutine

- However, Perl allows inner subroutines to be called from anywhere (within the package in which they are defined)


```
sub outer_sub {
   sub inner_sub { ... }
}
inner_sub();
```

Nested subroutine definitions

If an inner subroutine uses a local variable of an outer subroutine, then it refers to the instance of that local variable created the first time the outer subroutine was called


```
sub outer {
  my $x = $_[0];
  sub inner { return $x }
  return inner();            # returns $_[0]?
}
print "1:␣",outer(10),"\n";
print "2:␣",outer(20),"\n";
```



```
1: 10
2: 10 # not 20!
```

⇝ Do not refer to local variables of an outer subroutine, pass information via arguments instead

Nested subroutine definitions: Example

```perl
sub sqrt2 {
  my $x = shift(@_);
  my $precision = 0.001;

  sub sqrtIter {
    my ($guess,$x) = @_;
    if (isGoodEnough($guess,$x)) {
      return int($guess/$precision+0.5)*$precision;
    } else { sqrtIter(improveGuess($guess, $x), $x) } }

  sub improveGuess {
    my ($guess,$x) = @_;
    return ($guess + $x / $guess) / 2; }

  sub isGoodEnough {
    my ($guess,$x) = @_;
    return (abs($guess * $guess - $x) < $precision); }

  return sqrtIter(1.0,$x);
}
```

Revision

Read

- Chapter 9: Processing Text with Regular Expressions
- Chapter 4: Subroutines

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.

- `http://perldoc.perl.org/perlsub.html`

COMP284 Scripting Languages

Lecture 7: Perl (Part 6)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

I/O Connections

- Perl programs interact with their environment via I/O connections
- A filehandle is the name in a Perl program for such an I/O connection, given by a Perl identifier
  Beware: Despite the terminology, no files might be involved
- There are six pre-defined filehandles

`STDIN`	Standard Input, for user input, typically the keyboard
`STDOUT`	Standard Output, for user output, typically the terminal
`STDERR`	Standard Error, for error output, typically defaults to the terminal
`DATA`	Input from data stored after `__END__` at the end of a Perl program
`ARGV`	Iterates over command-line filenames in `@ARGV`
`ARGVOUT`	Points to the currently open output file when doing edit-in-place processing with `-i` `perl -pi -e 's/cat/dog/' file`

I/O Connections

Except for the six predefined I/O connections, all other I/O connections

- need to be opened before they can be used
  `open filehandle, mode, expr`
- should be closed once no longer needed
  `close filehandle`
- can be used to read from
  `<filehandle>`
- can be used to write to
  `print filehandle list`
  `printf filehandle list`
- can be selected as default output
  `select filehandle`

I/O Connections

Example:

```
open INPUT, "<", "oldtext.txt" or die "Cannot open file";
open OUTPUT, ">", "newtext.txt";
while (<INPUT>) {
   s!(\d+) degrees Fahrenheit!
     sprintf("%d",(($1-32)*5/9)+0.5)." degrees Celsius"!e;
   print OUTPUT;
}
close(INPUT);
close(OUTPUT);
```

`oldtext.txt`:

```
105 degrees Fahrenheit is quite warm
```

`newtext.txt`:

```
41 degrees Celcius is quite warm
```

Opening a filehandle

```
open filehandle, expr
open filehandle, mode, expr
```

- Opens an I/O connection specified by *mode* and *expr* and associates it with *filehandle*
- *expr* specifies a file or command
- *mode* is one of the following

<table>
<tr><th>Mode</th><th>Operation</th><th>Create</th><th>Truncate</th></tr>
<tr><td><</td><td>read file</td><td></td><td></td></tr>
<tr><td>></td><td>write file</td><td>yes</td><td>yes</td></tr>
<tr><td>>></td><td>append file</td><td>yes</td><td></td></tr>
<tr><td>+<</td><td>read/write file</td><td></td><td></td></tr>
<tr><td>+></td><td>read/write file</td><td>yes</td><td>yes</td></tr>
<tr><td>+>></td><td>read/append file</td><td>yes</td><td></td></tr>
<tr><td>|-</td><td>write to command</td><td>yes</td><td></td></tr>
<tr><td>-|</td><td>read from command</td><td>yes</td><td></td></tr>
</table>

Closing a filehandle

`close`

`close` *filehandle*

- Flushes the I/O buffer and closes the I/O connection associated with *filehandle*
- Returns true if those operations succeed
- Closes the currently selected filehandle if the argument is omitted

Reading

<*filehandle*>

- In a scalar context, returns a string consisting of all characters from *filehandle* up to the next occurrence of $/ (the input record separator)
- In a list context, returns a list of strings representing the whole content of *filehandle* separated into string using $/ as a separator (Default value of $/: newline \n)


```
open INPUT, "<", "oldtext.txt" or die "Cannot open file";
$first_line = <INPUT>;
while ($other_line = <INPUT>) { ... }
close INPUT;

open LS, "-|", "ls -1";
@files = <LS>;
close LS;
foreach $file (@files) { ... }
```

Selecting a filehandle as default output

`select`

`select` *filehandle*

- If *filehandle* is supplied, sets the new current default filehandle for output
  - ⇝ `write` or `print` without a filehandle default to *filehandle*
  - ⇝ References to variables related to output will refer to *filehandle*
- Returns the currently selected filehandle

Printing

```
print filehandle list
print filehandle
print list
print
```

- Print a string or a list of strings to *filehandle*
- If *filehandle* is omitted, prints to the last selected filehandle
- If *list* is omitted, prints `$_`
- The current value of `$,` (if any) is printed between each *list* item (Default: `undef`)
- The current value of `$\` (if any) is printed after the entire *list* has been printed
  (Default: `undef`)

Printing: Formatting

`sprintf(`*`format`*`, `*`list`*`)`

- Returns a string formatted by the usual printf conventions of the C library function sprintf (but does not by itself print anything)

```
sprintf "(%10.3f)" 1234.5678
```

format a floating-point number with minimum width 10 and precision 3 and put the result in parentheses:

```
(  1234.568)
```

See http://perldoc.perl.org/functions/sprintf.html for further details

Printing: Formatting

```
printf filehandle format, list
printf format, list
```


- Equivalent to
  `print filehandle sprintf(format, list)`
  except that `$\` (the output record separator) is not appended

Printing: Formatting

Format strings can be stored in variables and can be constructed on-the-fly:

```
@list = qw(wilma dino pebbles);
$format = "The items are:\n" . ("%10s\n" x @list);
printf $format, @list;
```

Output:

```
The items are:
     wilma
      dino
   pebbles
```

(The code above uses the 'quote word' function qw() to generate a list of words. See http://perlmeme.org/howtos/perlfunc/qw_function.html for details)

Here documents

- A here document is a way of specifying multi-line strings in a scripting or programming language
- The basic syntax is

```
<<identifier
here document
identifier
```

  - *identifier* declares the terminating string that will indicate where the here document ends
  - *identifier* might optionally be surrounded by double-quotes, single-quotes or backticks
    An unquoted identifier works like a double-quoted one
  - The here document starts on the following line
  - The terminating string *identifier* must appear by itself (unquoted and with no surrounding whitespace) after the last line of the here document

Here documents: Double-quotes

```
$title = "My␣HTML␣document"
print <<"END";
Content-type: text/html

<!DOCTYPE html>
<HTML>
<HEADER><TITLE>$title</TITLE></HEADER>
<BODY>
  <H1>$title</H1>
  Lots of HTML markup here
</BODY>
</HTML>
END
```

```
Content-type: text/html

<!DOCTYPE html>
<HTML>
<HEADER><TITLE>My HTML document</TITLE></HEADER>
<BODY>
  <H1>My HTML document</H1>
  Lots of HTML markup here
</BODY>
</HTML>
```

The double-quotes in "END" indicate that everything between the opening "END" and the closing END should be treated like a double-quoted string

Here documents: Single-quotes

```
$title = "My HTML document"
print <<'END';
Content-type: text/html

<!DOCTYPE html>
<HTML><HEADER><TITLE>$title</TITLE></HEADER>
<BODY></BODY></HTML>
END
```


The single-quotes in 'END' indicate that everything between 'END' and END should be treated like a single-quoted string

⇝ no variable interpolation is applied

⇝ `$title` will not be expanded

```
Content-type: text/html

<!DOCTYPE html>
<HTML><HEADER><TITLE>$title</TITLE></HEADER>
<BODY></BODY></HTML>
END
```

Here documents: Backticks

```
$command = "ls";
print <<`END`;
$command -1
END
```


The backticks in `END` tell Perl to run the here document as a shell script (with the here document treated like a double-quoted string)


```
handouts.aux
handouts.log
handouts.pdf
handouts.tex
```

Here documents: Variables

Here documents can be assigned to variables and manipulated using string operations

```
$header = <<"HEADER";
Content-type: text/html

<!DOCTYPE html>
<HTML><HEADER><TITLE>$title</TITLE></HEADER>
HEADER

$body = <<"BODY";
<BODY>
  <H1>$title</H1>
  Lots of HTML markup here
</BODY>
</HTML>
BODY

$html = $header.$body;
print $html;
```

Invocation Arguments

- Another way to provide input to a Perl program are invocation arguments (command-line arguments)

```
./perl_program arg1 arg2 arg3
```


- The invocation arguments given to a Perl program are stored in the special array `@ARGV`

perl_program1:


```
print "Number of arguments: ",$#ARGV+1,"\n";
for ($index=0; $index <= $#ARGV; $index++) {
  print "Argument $index: $ARGV[$index],"\n";
}
```



```
./perl_program1 ada 'bob' 2
```

Output:

```
Number of arguments: 3
Argument 0: ada
Argument 1: bob
Argument 2: 2
```

Options

- There are various Perl modules that make it easier to process command-line options

  `--scale=5 --debug --file='image.png'`

- One such module is `Getopt::Long`:
  `http://perldoc.perl.org/Getopt/Long.html`
- The module provides the `GetOptions` function
- `GetOptions` parses the command line arguments that are present in `@ARGV` according to an option specification
- Arguments that do not fit to the option specification remain in `@ARGV`
- `GetOptions` returns true if `@ARGV` can be processed successfully

Options: Example

perl_program2:

```perl
use Getopt::Long;
my $file = "photo.jpg";
my $scale = 2;
my $debug = 0;

$result = GetOptions ("debug"   => \$debug,  # flag
                      "scale=i" => \$scale,  # numeric
                      "file=s"  => \$file);  # string

print "Debug: $debug; Scale: $scale; File: $file\n";
print "Number of arguments: ",$#ARGV+1,"\n";
print "Arguments: ",join(", ",@ARGV),"\n";
```

```
./perl_program2 --scale=5 --file='image.png' arg1 arg2
```

```
Debug: 0; Scale: 5; File: image.png
Number of arguments: 2
Arguments: arg1, arg2
```

Revision

Read

- Chapter 5: Input and Output

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.

- `http://perldoc.perl.org/perlop.html#I%2fO-Operators`
- `http://perldoc.perl.org/perlop.html#Quote-Like-Operators`
- `http://perldoc.perl.org/Getopt/Long.html`

COMP284 Scripting Languages

Lecture 8: Perl (Part 7)

Handouts

Ullrich Hustadt

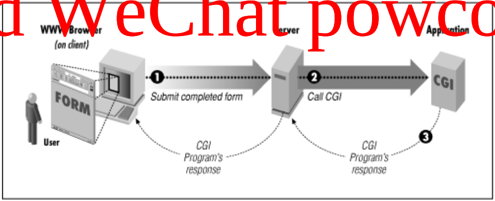
Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Common Gateway Interface — CGI

The Common Gateway Interface (CGI) is a standard method for web servers to use an external application, a CGI program, to dynamically generate web pages

1. A web client generates a client request, for example, from a HTML form, and sends it to a web server
2. The web server selects a CGI program to handle the request, converts the client request to a CGI request, executes the program
3. The CGI program then processes the CGI request and the server passes the program's response back to the client



Client requests

In the following we focus on client requests that are generated using HTML forms


```
<!DOCTYPE html>
<html>
<head><title>My HTML Form</title></head>
<body>
<form action=
 "http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/ullrich/demo"
 method="post">
<label>Enter your user name:
    <input type="text" name="username"></label><br>
<label>Enter your full name:
    <input type="text" name="fullname"></label><br>
<input type="submit" value="Click for response">
</form>
</body>
</html>
```

Client requests

In the following we focus on client requests that are generated using HTML forms


```
<!DOCTYPE html>
<html>
<head><title>My HTML Form</title></head>
<body>
<form action="http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/ullrich/demo"
 method="post">
<label>Enter your user name: <input type="text" name="username"></label><br>
<label>Enter your full name: <input type="text" name="fullname"></label><br>
<input type="submit" value="Click␣for␣response">
</form>
</body>
</html>
```


Encoding of input data

- Input data from an HTML form is sent URL-encoded as sequence of key-value pairs: `key1=value1&key2=value2&...`

  Example:

  `username=dave&fullname=David%20Davidson`

- All characters except A-Z, a-z, 0-9, -, _, ., ~ (unreserved characters) are encoded
- ASCII characters that are not unreserved characters are represented using ASCII codes (preceded by %)
  - A space is represented as %20 or +
  - + is represented as %2B
  - % is represented as %25

  Examples:

  `username=cath&fullname=Catherine+O%27Donnell`

Request methods: GET versus POST

The two main request methods used with HTML forms are `GET` and `POST`:

- `GET`:
  - Form data is appended to the URI in the request

    ```
    <scheme> "://" <server-name> ":" <server-port>
    <script-path> <extra-path> "?" <query-string>
    ```


  - Form data is accessed by the CGI program via environment variables

Example:


```
GET /cgi-bin/cgiwrap/ullrich/demo?username=dave&
fullname=David+Davidson HTTP/1.1
Host: cgi.csc.liv.ac.uk
```

Request methods: GET versus POST

The two main request methods used with HTML forms are GET and POST:

- POST:
  - Form data is appended to end of the request (after headers and blank line)
  - Form data can be accessed by the CGI program via standard input
  - Form data is not necessarily URL-encoded (but URL-encoding is the default)

Example:

```
POST /cgi-bin/cgiwrap/ullrich/demo HTTP/1.1
Host: cgi.csc.liv.ac.uk

username=dave&fullname=David+Davidson
```

Environment variables: GET

Env variable	Meaning
QUERY_STRING	The query information passed to the program
REQUEST_METHOD	The request method that was used
PATH_INFO	Extra path information passed to a CGI program
PATH_TRANSLATED	Translation of PATH_INFO from virtual to physical path
SCRIPT_NAME	The relative virtual path of the CGI program
SCRIPT_FILENAME	The physical path of the CGI program

Example (1):

```
GET http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/ullrich/demo/more/dirs?
    username=dave&fullname=David+Davidson
```

```
QUERY_STRING        username=dave&fullname=David+Davidson
REQUEST_METHOD      GET
PATH_INFO           /more/dirs
PATH_TRANSLATED     /users/www/external/docs/more/dirs
SCRIPT_NAME         /cgi-bin/cgiwrap/ullrich/demo
SCRIPT_FILENAME     /users/loco/ullrich/public_html/cgi-bin/demo

STDIN
# empty
```

Environment variables: GET

Env variable	Meaning
QUERY_STRING	The query information passed to the program
REQUEST_METHOD	The request method that was used
PATH_INFO	Extra path information passed to a CGI program
PATH_TRANSLATED	Translation of PATH_INFO from virtual to physical path
SCRIPT_NAME	The relative virtual path of the CGI program
SCRIPT_FILENAME	The physical path of the CGI program

Example (2):

```
GET http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/ullrich/demo/more/dirs?
    username=2%60n+d%2Bt+e+s%27t&fullname=Peter+Newton
```

```
QUERY_STRING       username=2%60n+d%2Bt+e+s%27t&fullname=Peter+Newton
REQUEST_METHOD     GET
PATH_INFO          /more/dirs
PATH_TRANSLATED    /users/www/external/docs/more/dirs
SCRIPT_NAME        /cgi-bin/cgiwrap/ullrich/demo
SCRIPT_FILENAME    /users/loco/ullrich/public_html/cgi-bin/demo

STDIN
# empty
```

Environment variables: POST

Env variable	Meaning
QUERY_STRING	The query information passed to the program
REQUEST_METHOD	The request method that was used
SCRIPT_NAME	The relative virtual path of the CGI program
SCRIPT_FILENAME	The physical path of the CGI program

Example:

```
POST /cgi-bin/cgiwrap/ullrich/demo
Host: cgi.csc.liv.ac.uk

username=2%60n+d%2Bt+e+s%27t&fullname=Peter+Newton
```

```
QUERY_STRING
# empty
REQUEST_METHOD     POST
SCRIPT_NAME        /cgi-bin/cgiwrap/ullrich/demo
SCRIPT_FILENAME    /users/loco/ullrich/public_html/cgi-bin/demo

STDIN              username=2%60n+d%2Bt+e+s%27t&fullname=Peter+Newton
```

More environment variables

Env variable	Meaning
HTTP_ACCEPT	A list of the MIME types that the client can accept
HTTP_REFERER	The URL of the document that the client points to before accessing the CGI program
HTTP_USER_AGENT	The browser the client is using to issue the request
REMOTE_ADDR	The remote IP address of the user making the request
REMOTE_HOST	The remote hostname of the user making the request
SERVER_NAME	The server's hostname
SERVER_PORT	The port number of the host on which the server is running
SERVER_SOFTWARE	The name and version of the server software

CGI programs and Perl

- CGI programs need to process input data from environment variables and STDIN, depending on the request method
  ⇝ preferably the input data would be accessible by the program in a uniform way
- CGI programs need to process input data that is encoded
  ⇝ preferably, the input data would be available in decoded form
- CGI programs need to produce HTML markup/documents as output
  ⇝ preferably, there would be an easy way to produce HTML markup

In Perl all this can be achieved with the use of the `CGI.pm` module
`http://perldoc.perl.org/CGI.html`

CGI.pm HTML shortcuts

- CGI.pm provides so-called HTML shortcuts that create HTML tags

a	address	applet	b	body	br	center	code
dd	div	dl	dt	em	font	form	
h1	h2	h3	h4	h5	h6	head	header
html	hr	img	li	ol	p	pre	strong
sup	table	td	th	tr	title	tt	ul

- HTML tags have attributes and contents

```
<p align="right">This is a paragraph</p>
```

- HTML shortcuts are given
  - HTML attributes in the form of a hash reference as the first argument
  - the contents as any subsequent arguments

```
p({-align=>right},"This is a paragraph")
```

CGI.pm HTML shortcuts: Examples

```
Code:     print p();
Output:   <p />
```

```
Code:     print p('');
Output:   <p></p>
```

```
Code:     print p({-align=>right},"Hello␣world!");
Output:   <p align="right">Hello world!</p>
```

```
Code:     print p({-class=>right_para,-id=>p1},"Text");
Output:   <p class="right_para" id="p1">Text</p>
```

CGI.pm HTML shortcuts: Nesting vs Start/End

- Nested HTML tags using nested HTML shortcuts

```
Code:   print p(em("Emphasised")." Text"), "\n";
```

```
Output: <p><em>Emphasised</em> Text</p>
```


- Nested HTML tags using start_*tag* and end_*tag*:

```
use CGI qw(-utf8 :all *em *p);

print start_p(), start_em(), "Emphasised", end_em(),
      " Text", end_p(), "\n";
```

```
Output: <p><em>Emphasised</em> Text</p>
```


The following start_*tag*/end_*tag* HTML shortcuts are generated automatically by CGI.pm:

```
start_html(), start_form(), start_multipart_form()
  end_html(),   end_form()     end_multipart_form()
```

All others need to be requested by adding **tag* to the CGI.pm import list

CGI.pm Forms

- HTML forms are created using `start_form` and `end_form`

```
print start_form({-method=>request_method,
                  -action=>uri});
form elements
print end_form;
```


- HTML form elements are again created using HTML shortcuts

textfield	textarea	password_field
filefield	hidden	scrolling_list
popup_menu	optgroup	
image_button	checkbox	checkbox_group
radio_group	reset	submit

- `optgroup` creates an option group within a popup menu
  ⇝ `optgroup` occurs nested inside `popup_menu`
- All other HTML shortcuts for HTML form elements will occur independently of each other within a form

CGI.pm Forms: Examples

```
print textfield({-name=>'username',
                 -value=>'dave',
                 -size=>100,
                 -maxlength=>500});
```


- `-name` specifies the name of the text field and is the only required argument of `textfield`
- `-value` specifies a default value that will be shown in the text field
- `-size` is the size of the text field in characters
- `-maxlength` is the maximum number of characters that the text field will accept

Output:

```
<input type="text" name="username"
       value="dave" size="100" maxlength="500" />
```

CGI.pm Forms: Examples

```
print submit({-name=>'submit',
              -label=>'Click␣for␣response'});
```


- `-name` is an optional argument that allows to distinguish submit buttons from each other
- `-label` or `-value` is an optional argument that determines the label shown to the user and the value passed to the CGI program

Output:

```
<input type="submit" name="submit"
       value="Click␣for␣response" />
```

CGI.pm Forms: Example

```perl
#!/usr/bin/perl

use CGI qw(-utf8 :all);

print header(-charset=>'utf-8'),
      start_html({-title=>'My HTML Form',
                  -author=>'u.hustadt@liverpool.ac.uk',
                  -style=>'style.css'});

print start_form({-method=>"GET",
                  -action=>"http://cgi.csc.liv.ac.uk/".
                           "cgi-bin/cgiwrap/ullrich/demo"});
print textfield({-name=>'username',
                 -value=>'dave',
                 -size=>100});
print br();
print textfield({-name=>'fullname',
                 -value=>'Please enter your name',
                 -size=>100});
print br();
print submit({-name=>'submit',
              -value=>'Click for response'});
print end_form, end_html;
```

Making it work

For CGI programs to work on our systems you must proceed as follows:

1. Your home directory must be ‘world executable’
2. You must have a directory

   $HOME/public_html/cgi-bin/

   Your public_html and cgi-bin directory must be both readable and executable by everyone
3. Your CGI script must be placed in

   $HOME/public_html/cgi-bin/

   and must be executable by everyone
4. The CGI script can then be accessed using the URL

   http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/<user>/<script>
   or http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrapd/<user>/<script>

   where <user> is your user name
   and <script> is the filename of the script
   (cgiwrapd provides debugging output, but does not reveal all errors)

Accessing and processing data

- Perl provides a hash %ENV that stores the information stored in environment variables
- Processing %ENV is done in the standard way for hashes


```
print "The request method used is ",
      $ENV{'REQUEST_METHOD'}, br(), "\n";

foreach $key (keys %ENV) {
  print "The value of $key is $ENV{$key}", br(), "\n";
}
```


Output:


```
The request method used is GET
The value of SCRIPT_NAME is /cgi-bin/cgiwrap/ullrich/demo
The value of SERVER_NAME is cgi.csc.liv.ac.uk
The value of SERVER_ADMIN is root@localhost
The value of HTTP_ACCEPT_ENCODING is gzip,deflate
The value of HTTP_CONNECTION is keep-alive
The value of REQUEST_METHOD is GET
```

Accessing and processing data

- `CGI.pm` provides the `param` routine to access the input data of HTML forms
- For a sequence of key-value pairs

  key1`=`*value1*`&`*key2*`=`*value2*`&`*key3*`=`*value3*`&...`

  representing the input data of a HTML form

  `param('key1')` `param('key2')` `param('key3')` ...

  will return

  value1 *value2* *value3*

  while `param()` returns the list `('key1','key2','key3',...)`
- The values returned by `param` have already been decoded
- `param('key')` returns the empty string if *value* is empty
- `param('key')` returns `undef` if *key* is not among the key-value pairs of the request
- This does not depend on whether the request method is GET or POST

Accessing and processing data

- `CGI.pm` provides the `param` routine to access the input data of HTML forms


```
print "The␣value␣of␣username␣is␣",
      param('username'), br(), br(), "\n";
print "The␣value␣of␣fullname␣is␣",
      param('fullname'), br(), br(), "\n";

foreach $key (param()) {
   print "The␣value␣of␣$key␣is␣",param($key), br(), "\n";
}
```


Output:


```
The value of username is dave
The value of fullname is David Davidson

The value of submit is Click for response
The value of username is dave
The value of fullname is David Davidson
```

Accessing and processing data: UFT-8

- The pragma -utf8 in

```
use CGI qw(-utf8 :all);
```


makes CGI.pm treat all param() values as UTF-8 strings

- Alternatively, specific param() values can be decoded using the decode subroutine of the Encode module

```
use Encode;
my $fullname = decode("utf8",param('fullname'));
```


- With

```
binmode(STDOUT,":encoding(utf-8)");
print header(-charset=>'utf-8');
```


we ensure that the web page we produce is sent to the brower using UTF-8 encoding

Accessing and processing data: Security

- Do not trust any data accessed via `param` (beware code injection)

  Example:

  ```
  print "The value of username is ",param('username'),"\n";
  ```

  together with input

  ```
  <script>window.location="http://malware_site/"</script>
  ```

  for username, would redirect the browser to `malware_site`.

- Check whether the data has the format expected

  ```
  if (param('username') !~ /^[a-zA-Z0-9]+$/s) {
    print "Not a valid user name"
  } else {
    print "The value of username is ",param('username'),"\n";
  }
  ```


  or sanitise the input using the CGI.pm routine `escapeHTML`:

  ```
    print "The value of username is ",
          escapeHTML(param('username')),"\n";
  ```

  or even better, do both

CGI.pm Scripts: Example (Part 1)


```
use CGI qw(-utf-8 :all *table);
binmode(STDOUT, ":encoding(utf-8)");

print header(-charset=>'utf-8'), "\n",
      start_html({-title=>'Form␣Processing',
                  -author=>'u.hustadt@liverpool.ac.uk'});

if (!defined(param('username'))) {
  # This branch is executed if the user first visits this page/script
  print start_form({-method=>"POST"});
  print textfield({-name=>'username', -value=>'dave',
                   -size=>100}), "\n";
  print br(), "\n";
  print textfield({-name=>'fullname',
                   -value=>'Please␣enter␣your␣name',
                   -size=>100}), "\n";
  print br(), "\n";
  print submit({-name=>'submit',
                -value=>'Click␣for␣response'}), "\n";
  print end_form;
} else {
  # This branch is executed if the client request is generated
  # by the form
```

CGI.pm Scripts: Example (Part 2)

```
  # (We are in the else-branch now)

  print start_table({-border=>1});
  print caption("Inputs");
  foreach $key (param()) {
    print Tr(td('PARAM'),td($key),td(escapeHTML(param($key))));
   }
  foreach $key (keys %ENV) {
    print Tr(td('ENV'),td($key),td(escapeHTML($ENV{$key})));
  }
  print end_table;
}
print end_html;
```

CGI.pm Scripts: Example (Part 3)

Page produced on the first visit

Page produced on submission of the form

Inputs

PARAM	username	dave
PARAM	fullname	David Davidson
PARAM	submit	Click for response
ENV	REQUEST_METHOD	POST
ENV	QUERY_STRING	
ENV	SCRIPT_FILENAME	/users/loco/ullrich/public_html/cgi-bin/lect09.pl
ENV	SERVER_NAME	cgi.csc.liv.ac.uk
ENV	HTTP_REFERER	http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/ullrich/lect09.pl

Revision

Read

- Chapter 11: Perl Modules

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.

- `http://perldoc.perl.org/CGI.html`

COMP284 Scripting Languages

Lecture 9: PHP (Part 1)

Handouts

Ullrich Hustadt

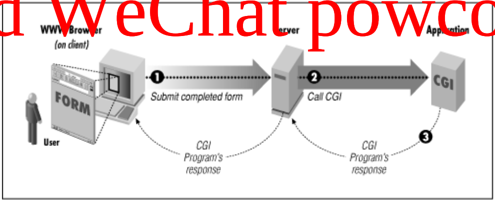
Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Common Gateway Interface — CGI

The Common Gateway Interface (CGI) is a standard method for web servers to use external applications, a CGI program, to dynamically generate web pages

1. A web client generates a client request, for example, from a HTML form, and sends it to a web server
2. The web server selects a CGI program to handle the request, converts the client request to a CGI request, executes the program
3. The CGI program then processes the CGI request and the server passes the program's response back to the client



Disadvantages of CGI/Perl

- A distinction is made between static web pages and dynamic web pages created by an external program
- Using Perl scripting it is difficult to add 'a little bit' of dynamic content to a web page
  – can be alleviated to some extent by using here documents
- Use of an external program requires
  - starting a separate process every time an external program is requested
  - exchanging data between web server and external program

  ⇝ resource-intensive

If our main interest is the creation of dynamic web pages, then the scripting language we use

- should integrate well with HTML
- should not require a web server to execute an external program

PHP

- PHP is (now) a recursive acronym for PHP: Hypertext Preprocessor
- Development started in 1994 by Rasmus Lerdorf
- Originally designed as a tool for tracking visitors at Lerdorf's website
- Developed into full-featured, scripting language for server-side web programming
- Inherits a lot of the syntax and features from Perl
- Easy-to-use interface to databases
- Free, open-source
- Probably the most widely used server-side web programming language
- Negatives: Inconsistent, muddled API; no scalar objects

The departmental web server uses PHP 5.6.25 (released August 2014)
PHP 7 was released in December 2015 (PHP 6 was never released)

PHP processing

- Server plug-ins exist for various web servers
  ⇝ avoids the need to execute an external program

- PHP code is embedded into HTML pages using tags
  ⇝ static web pages can easily be turned into dynamic ones

PHP satisfies the criteria we had for a good web scripting language

Processing proceeds as follows:

1. The web server receives a client request
2. The web server recognizes that the client request is for a HTML page containing PHP code

3. The server executes the PHP code, substitutes output into the HTML page, the resulting page is then send to the client

As in the case of Perl, the client never sees the PHP code, only the HTML web page that is produced

PHP: Applications

- Applications written using PHP
  - activeCollab – Project Collaboration Software
    `http://www.activecollab.com/`
  - Drupal – Content Management System (CMS)
    `http://drupal.org/home`
  - Magento – eCommerce platform
    `http://www.magentocommerce.com/`
  - MediaWiki – Wiki software
    `http://www.mediawiki.org/wiki/MediaWiki`
  - Moodle – Virtual Learning Environment (VLE)
    `http://moodle.org/`
  - Sugar – Customer Relationship Management (CRM) platform
    `http://www.sugarcrm.com/crm/`
  - WordPress – Blogging tool and CMS
    `http://wordpress.org/`

PHP: Websites

- Websites using PHP:
  - Delicious – social bookmarking
    `http://delicious.com/`
  - Digg – social news website
    `http://digg.com`
  - Facebook – social networking
    `http://www.facebook.com`
  - Flickr – photo sharing
    `http://www.flickr.com`
  - Friendster – social gaming
    `http://www.friendster.com`
  - SourceForge – web-based source code repository
    `http://sourceforge.net/`
  - Wikipedia – collaboratively built encyclopedia
    `http://www.wikipedia.org`

Recommended texts

- R. Nixon:
  Learning PHP, MySQL, and JavaScript.
  O'Reilly, 2009.

  Harold Cohen Library: 518.561.N73 or e-book
  (or later editions of this book)

- M. Achour, F. Betz, A. Dovgal, N. Lopes,
  H. Magnusson, G. Richter, D. Seguy, J. Vrana, et al.:
  PHP Manual.
  PHP Documentation Group, 2018.

  `http://www.php.net/manual/en/index.php`

PHP: Hello World!

```
<html>
<head><title>Hello World</title></head>
<body>
<p>Our first PHP script</p>
<?php
  print ("<p><b>Hello World!</b></p>\n");
?>
</body></html>
```


- PHP code is enclosed between <?php and ?>
- File must be stored in a directory accessible by the web server, for example $HOME/public_html, and be readable by the web server
- File name must have the extension .php, e.g. hello_world.php

PHP: Hello World!

Since version 4.3.0, PHP also has a command line interface

```
#!/usr/bin/php
<?php
  /* Author: Ullrich Hustadt
     A "Hello World" PHP script. */
  print ("Hello World!\n");
  // A single-line comment
?>
```


- PHP code still needs to be enclosed between <?php and ?>
- Code must be stored in an executable file
- File name does not need to have any particular format

⇝ PHP can be used as scripting language outside a web programming context

Output:

```
Hello World!
```

PHP: Hello World!

```
<html>
<head><title>Hello World</title></head>
<body><p>Our first PHP script</p>
<?php
  print ("<p><b>Hello␣World!</b></p>\n");
?>
</body></html>
```


- Can also 'executed' using

```
php filename
```

- File does not need to exectuable, only readable for the user

Output:

```
<html>
<head><title>Hello World</title></head>
<body><p>Our first PHP script</p>
<p><b>Hello World!</b></p>
</body></html>
```

PHP scripts

- PHP scripts are typically embedded into HTML documents and are enclosed between <?php and ?> tags
- A PHP script consists of one or more statements and comments
  ⇝ there is no need for a main function (or classes)
  - Statements end in a semi-colon
  - Whitespace before and in between statements is irrelevant
    (This does not mean its irrelevant to someone reading your code)
  - One-line comments start with // or # and run to the end of the line or ?>
  - Multi-line comments are enclosed in /* and */

Types

PHP has eight primitive types

- Four scalar types:
  - `bool` – booleans
  - `int` – integers
  - `float` – floating-point numbers
  - `string` – strings
- Two compound types:
  - `array` – arrays
  - `object` – objects
- Two special types:
  - `resource`
  - `NULL`

- Integers, floating-point numbers, and strings do not differ significantly from the corresponding Perl scalars, including the pecularities of single-quoted versus double-quoted strings
- In contrast to Perl, PHP does distinguish between different types including between the four scalar types

Variables

- All PHP variable names start with $ followed by a PHP identifier
- A PHP identifier consists of letters, digits, and underscores, but cannot start with a digit
  PHP identifiers are case sensitive

- In PHP, a variable does not have to be declared before it can be used

- A variable also does not have to be initialised before it can be used, although initialisation is a good idea
- Uninitialized variables have a default value of their type depending on the context in which they are used

Type	Default	Type	Default
`bool`	FALSE	`string`	empty string
`int`/`float`	0	`array`	empty array

If there is no context, then the default value is `NULL`

Assignments

- Just like Java and Perl, PHP uses the equality sign = for assignments

```
$student_id = 200846369;
```

As in Perl, this is an assignment expression

- The value of an assignment expression is the value assigned

```
$b = ($a = 0) + 1;
// $a has value 0
// $b has value 1
```

Binary assignments

PHP also supports the standard binary assignment operators:

Binary assignment	Equivalent assignment
`$a += $b`	`$a = $a + $b`
`$a -= $b`	`$a = $a - $b`
`$a *= $b`	`$a = $a * $b`
`$a /= $b`	`$a = $a / $b`
`$a %= $b`	`$a = $a % $b`
`$a **= $b`	`$a = $a ** $b`
`$a .= $b`	`$a = $a . $b`

Example:

```
// Convert Fahrenheit to Celsius:
// Subtract 32, then multiply by 5, then divide by 9
$temperature = 105;           // temperature in Fahrenheit
$temperature -= 32;
$temperature *= 5/9;          // converted to Celsius
```

Constants

- `bool define(string, expr [, case_insensitive])`
  - defines a constant that is globally accessible within a script
  - *string* should be a string consisting of a PHP identifier (preferably all upper-case)
    The PHP identifier is the name of the constant
  - *expr* is an expression that should evaluate to a scalar value
  - *case_insensitive* is an optional boolean argument, indicating whether the name of the constant is case-insensitive (default is FALSE)
  - returns TRUE on success or FALSE on failure

```
define("PI",3.14159);
define("SPEED_OF_LIGHT",299792458,true);
```

Constants

- To use a constant we simply use its name

```
define("PI",3.14159);
define("SPEED_OF_LIGHT",299792458,true);
$circumfence = PI * $diameter;
$distance     = speed_of_light * $time;
```

- Caveat: PHP does not resolve constants within double-quoted strings (or here documents)

```
print "1 - Value of PI: PI\n";
print "2 - Value of PI: ".PI."\n";
```

```
1 - Value of PI: PI
2 - Value of PI: 3.14159
```

Values, Variables and Types

PHP provides several functions that explore the type of an expression:

string gettype(*expr*)	returns the type of *expr* as string
bool is_*type*(*expr*)	checks whether *expr* is of type *type*
void var_dump(*expr*)	displays structured information about *expr* that includes its type and value

```
<?php print "Type of 23:   ".gettype(23)."\n";
      print "Type of 23.0: ".gettype(23.0)."\n";
      print "Type of \"23\": ".gettype("23")."\n";

      if (is_int(23)) { echo "23 is an integer\n"; }
         else { echo "23 is not an integer\n"; }
?>
```

```
Type of 23:   integer
Type of 23.0: double
Type of "23": string
23 is an integer
```

Type juggling and Type casting

- PHP automatically converts a value to the appropriate type as required by the operation applied to the value (type juggling)

```
2 . "␣worlds"          ⇝  "2␣worlds"
"2" * 3                ⇝  6
"1.23e2" + 0           ⇝  123
"hello" * 3            ⇝  0
"10hello5" + 5         ⇝  15
```

- PHP also supports explicit type casting via (*type*)

```
(int) "12"             ⇝  12       (bool) "0"          ⇝  FALSE
(int) "1.23e2"         ⇝  1        (bool) "foo"        ⇝  TRUE
(int) ("1.23e2" + 0)   ⇝  123      (float) "1.23e2"    ⇝  123
(int) "10hello5"       ⇝  10
(int) 10.5             ⇝  10
(array) "foo"          ⇝  array(0 => "foo")
```

Comparison operators

Type juggling also plays a role in the way PHP comparison operators work:

expr1 == *expr2*	Equal	TRUE iff *expr1* is equal to *expr2* after type juggling
expr1 != *expr2*	Not equal	TRUE iff *expr1* is not equal to *expr2* after type juggling
expr1 <> *expr2*	Not equal	TRUE iff *expr1* is not equal to *expr2* after type juggling
expr1 === *expr2*	Identical	TRUE iff *expr1* is equal to *expr2*, and they are of the same type
expr1 !== *expr2*	Not identical	TRUE iff *expr1* is not equal to *expr2*, or they are not of the same type

Note: For ==, !=, and <>, numerical strings are converted to numbers and compared numerically

```
"123" == 123             ⇝  TRUE
"123" != 123             ⇝  FALSE
"1.23e2" == 123          ⇝  TRUE
"1.23e2" == "12.3e1"     ⇝  TRUE
5 == TRUE                ⇝  TRUE
```

```
"123" === 123            ⇝  FALSE
"123" !== 123            ⇝  TRUE
1.23e2 === 123           ⇝  FALSE
"1.23e2" === "12.3e1"    ⇝  FALSE
5 === TRUE               ⇝  FALSE
```

Comparison operators

Type juggling also plays a role in the way PHP comparison operators work:

expr1 < *expr2*	Less than	TRUE iff *expr1* is strictly less than *expr2* after type juggling
expr1 > *expr2*	Greater than	TRUE iff *expr1* is strictly greater than *expr2* after type juggling
expr1 <= *expr2*	Less than or equal to	TRUE iff *expr1* is less than or equal to *expr2* after type juggling
expr1 >= *expr2*	Greater than or equal to	TRUE iff *expr1* is greater than or equal to *expr2* after type juggling

```
'35.5' > 35           ⇝  TRUE        '35.5' >= 35           ⇝  TRUE
'ABD' > 'ABC'         ⇝  TRUE        'ABD' >= 'ABC'         ⇝  TRUE
'1.23e2' > '12.3e1'   ⇝  FALSE       '1.23e2' >= '12.3e1'   ⇝  TRUE
"F1" < "G0"           ⇝  TRUE        "F1" <= "G0"           ⇝  TRUE
TRUE > FALSE          ⇝  TRUE        TRUE >= FALSE          ⇝  TRUE
5 > TRUE              ⇝  FALSE       5 >= TRUE              ⇝  TRUE
```

Revision

Read

- Chapter 3: Introduction to PHP

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

Also read

- `http://uk.php.net/manual/en/language.types.intro.php`
- `http://uk.php.net/manual/en/language.types.type-juggling.php`
- `http://uk.php.net/manual/en/language.operators.comparison.php`
- `http://uk.php.net/manual/en/types.comparisons.php`

COMP284 Scripting Languages
Lecture 10: PHP (Part 2)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Integers and Floating-point numbers

- PHP distinguishes between
  - integer numbers `0` `2012` `-40` `1263978`
  - floating-point numbers `1.25` `256.0` `-12e19` `2.4e-10`
- PHP supports a wide range of pre-defined mathematical functions

`abs(number)`	absolute value
`ceil(number)`	round fractions up
`floor(number)`	round fractions down
`round(number [,prec,mode])`	round fractions
`log(number [,base])`	logarithm
`rand(min,max)`	generate an integer random number
`sqrt(number)`	square root

- PHP provides a range of pre-defined number constants including

`M_PI`	3.14159265358979323846
`NAN`	‘not a number’
`INF`	‘infinity’

Integers and Floating-point numbers: NAN and INF

The constants `NAN` and `INF` are used as return values for some applications of mathematical functions that do not return a number

- `log(0)` returns `-INF` (negative 'infinity')
- `sqrt(-1)` returns `NAN` ('not a number')

In contrast

- 1/0 returns `FALSE` and produces an error message
- 0/0 returns `FALSE` and produces an error message

and execution of the script continues!

In PHP 7

- 1/0 returns `INF` and produces an error message
- 0/0 returns `NAN` and produces an error message

and execution of the script continues!

Integers and Floating-point numbers: NAN and INF

NAN and INF can be compared with each other and other numbers using equality and comparison operators:

`NAN == NAN` ⇝ FALSE	`NAN === NAN` ⇝ FALSE	`NAN == 1` ⇝ FALSE
`INF == INF` ⇝ FALSE	`INF === INF` ⇝ TRUE	`INF == 1` ⇝ FALSE
`NAN < NAN` ⇝ TRUE	`INF < INF` ⇝ TRUE	`1 < INF` ⇝ TRUE
`NAN < INF` ⇝ TRUE	`INF < NAN` ⇝ TRUE	`INF < 1` ⇝ FALSE
`NAN < 1` ⇝ TRUE	`1 < NAN` ⇝ TRUE	

In PHP 5.3 and earlier versions, `INF == INF` returns `FALSE`
In PHP 5.4 and later versions, `INF == INF` returns `TRUE`

Integers and Floating-point numbers: NAN and INF

- PHP provides three functions to test whether a value is or is not `NAN`, `INF` or `-INF`:
  - `bool is_nan(value)`
    returns TRUE iff *value* is `NAN`
  - `bool is_infinite(value)`
    returns TRUE iff *value* is `INF` or `-INF`
  - `bool is_finite(value)`
    returns TRUE iff *value* is neither `NAN` nor `INF`/`-INF`
- In conversion to a boolean value,
  both `NAN` and `INF` are converted to `TRUE`
- In conversion to a string,
  `NAN` converts to `'NAN'` and `INF` converts to `'INF'`

Exceptions and error handling

PHP distinguishes between exceptions and errors

- A possible way to perform exception handling in PHP is as follows:

```
try { ... run code here ...           // try
} catch (Exception $e) {
      ... handle the exception here using $e  // catch
}
```

- Errors must be dealt with by an error handling function
  ('Division by zero' produces an error not an exception)

  One possible approach is to let the error handling function
  turn errors into exceptions

```
function exception_error_handler($errno, $errstr,
  $errfile, $errline ) {
  throw new ErrorException($errstr, $errno,
                          0, $errfile, $errline); }
set_error_handler("exception_error_handler");
```

`http://www.php.net/manual/en/class.errorexception.php`

Booleans

- Unlike Perl, PHP does have a boolean datatype
  with constants `TRUE` and `FALSE` (case insensitive)
- PHP offers the same short-circuit boolean operators as Java and Perl:

  `&&` (conjunction) `||` (disjunction) `!` (negation)

  - Alternatively, `and` and `or` can be used instead of `&&` and `||`, respectively
  - However, `not` is not a PHP operator
- The truth tables for these operators are the same as for Perl
- Remember that `&&` and `||` are not commutative, that is,
  `(A && B)` is not the same as `(B && A)`
  `(A || B)` is not the same as `(B || A)`

Type conversion to boolean

When converting to boolean, the following values are considered `FALSE`:

- the boolean `FALSE` itself
- the integer 0 (zero)
- the float 0.0 (zero)
- the empty string, and the string '0'
- an array with zero elements
- an object with zero member variables (PHP 4 only)
- the special type `NULL` (including `unset` variables)
- SimpleXML objects created from empty tags

Every other value is considered `TRUE` (including any resource)

Strings

- PHP supports both single-quoted and double-quoted strings
- PHP also supports heredocs as a means to specify multi-line strings
  The only difference to Perl is the use of `<<<` instead of `<<` in their definition:

```
<<<identifier
here document
identifier
```

  - *identifier* might optionally be surrounded by double-quotes
  - *identifier* might also be surrounded by single-quotes,
    making the string a nowdoc in PHP terminology


```
print '<html>
<head><title>Multi-line String</title></head>';
print <<<EOF
<body>Some text</body>
</html>
EOF;
```

Strings

- Variable interpolation is applied to double-quoted strings (with slight differences to Perl)
- The string concatenation operator is denoted by '.' (as in Perl)
- Instead of Perl's string multiplication operator 'x' there is
  string str_repeat(*string_arg*, *number*)
- There are no built-in HTML shortcuts in PHP


```
$title  = "String␣Multiplication";
$string = "<p>I␣shall␣not␣repeat␣myself.<p>\n";
print "<!DOCTYPE␣html>\n<html><head><title>$title</title>
</head><body>".str_repeat($string,3).'</body></html>';
```

```
<!DOCTYPE html>
<html><head><title>String Multiplication</title>
</head><body><p>I shall not repeat myself.<p>
<p>I shall not repeat myself.<p>
<p>I shall not repeat myself.<p>
</body></html>
```

Arrays

- PHP only supports associative arrays (hashes), simply called arrays
- PHP arrays are created using the array construct or, since PHP 5.4 [...]:


```
array( key => value , ... )
[key => value, ...]
```

where *key* is an integer or string and *value* can be of any type, including arrays


```
$arr1 = [1 => "Peter", 3 => 2009, "a" => 101];
$arr2 = array(200846369 => array("name" => "Jan Olsen",
                                 "COMP101" => 69,
                                 "COMP102" => 52));
```


- The size of an array can be determined using the count function:

int count(*array* [, *mode*])

```
print count($arr1);      // prints 3
print count($arr2);      // prints 1
print count($arr2,1);    // prints 4
```

Arrays

- It is possible to omit the keys when using the `array` construct:

```
$arr3 = array("Peter", "Paul", "Mary");
```

  The values given in `array` will then be associated with the natural numbers 0, 1, . . .

- All the keys of an array can be retrieved using
  `array_keys($array1)`
  ⇝ returns a natural number-indexed array containing the keys of `$array1`
- All the values of an array can be retrieved using
  `array_values($array1)`
  ⇝ returns a natural number-indexed array containing the values stored in `$array1`

Arrays

- An individual array element can be accessed via its key
- Accessing an undefined key produces an error message and returns NULL


```
$arr1 = array(1 => "Peter", 3 => 2009, "a"=> 101);

print "'a':␣".$arr1["a"]."\n";
'a': 101
print "'b':␣".$arr1["b"]."\n";
PHP Notice: Undefined index: b in <file> on line <lineno>
'b':          // $arr1["b"] returns NULL
$arr1['b'] = 102;
print "'b':␣".$arr1["b"]."\n";
'b': 102
```

Arrays

- PHP allows the construct

```
$array[] = value;
```


PHP will determine the maximum value M among the integer indices in $array and use the key $K = M + 1$; if there are no integer indices in $array, then $K = 0$ will be used

$\leadsto$ auto-increment for array keys


```
$arr4[] = 51; // 0 => 51
$arr4[] = 42; // 1 => 42
$arr4[] = 33; // 2 => 33
```


- A key-value pair can be removed from an array using the unset function:

```
$arr1 = array(1 => "Peter", 3 => 2009, "a" => 101);
unset($arr1[3]);    // Removes the pair 3 => 2009
unset($arr1);       // Removes the whole array
```

Arrays: foreach-loop

- PHP provides a foreach-loop construct to 'loop' through the elements of an array
- Syntax and semantics is slightly different from that of the corresponding construct in Perl


```
foreach (array as $value)
    statement

foreach (array as $key => $value)
    statement
```


- *array* is an array expression
- *$key* and *$value* are two variables, storing a different key-value pair in *array* at each iteration of the foreach-loop
- We call *$value* the foreach-variable
- foreach iterates through an array in the order in which elements were defined

Arrays: foreach-loop

`foreach` iterates through an array in the order in which elements were defined

Example 1:

```
foreach (array("Peter", "Paul", "Mary") as $key => $value)
   print "The␣array␣maps␣$key␣to␣$value\n";
```

```
The array maps 0 to Peter
The array maps 1 to Paul
The array maps 2 to Mary
```


Example 2:

```
$arr5[2] = "Marry";
$arr5[0] = "Peter";
$arr5[1] = "Paul";
// 0 => 'Peter', 1 => 'Paul', 2 => 'Marry'
foreach ($arr5 as $key => $value)
   print "The␣array␣maps␣$key␣to␣$value\n";
```

```
The array maps 2 to Mary
The array maps 0 to Peter
The array maps 1 to Paul
```

Arrays: foreach-loop

Does changing the value of the foreach-variable change the element of the list that it currently stores?

Example 3:

```
$arr6 = array("name" => "Peter", "year" => 2009);

foreach ($arr6 as $key => $value) {
   print "The␣array␣maps␣$key␣to␣$value\n";
   $value .= "␣-␣modified"; // Changing $value
}
print "\n";

foreach ($arr6 as $key => $value)
   print "The␣array␣now␣maps␣$key␣to␣$value\n";
```



```
The array maps name to Peter
The array maps year to 2009

The array now maps name to Peter
The array now maps year to 2009
```

Arrays: foreach-loop

- In order to modify array elements within a foreach-loop we need use a reference

```
foreach (array as &$value)
    statement
unset($value);

foreach (array as $key => &$value)
    statement
unset($value);
```

  - In the code schemata above, &$value is a variable whose value is stored at the same location as an array element
  - Note that PHP does not allow the key to be a reference
  - The unset statement is important to return $value to being a 'normal' variable

Arrays: foreach-loop

In order to modify array elements within a `foreach`-loop we need use a reference

Example:

```
$arr6 = array("name" => "Peter", "year" => 2009);
foreach ($arr6 as $key => &$value) { // Note: reference!
   print "The␣array␣maps␣$key␣to␣$value\n";
   $value .= "␣-␣modified";
}
unset($value); // Remove the reference from $value
print "\n";

foreach ($arr6 as $key => $value)
   print "The␣array␣now␣maps␣$key␣to␣$value\n";
```

```
The array maps name to Peter
The array maps year to 2009

The array now maps name to Peter - modified
The array now maps year to 2009 - modified
```

Array functions

PHP has no stack or queue data structures,
but has stack and queue functions for arrays:

- `array_push($array, value1, value2, ...)`
  appends one or more elements at the end of the end of an array variable; returns the number of elements in the resulting array

- `array_pop($array)`
  extracts the last element from an array and returns it
- `array_shift($array)`
  shift extracts the first element of an array and returns it

- `array_unshift($array, value1, value2, ...)`
  inserts one or more elements at the start of an array variable; returns the number of elements in the resulting array

Note: `$array` needs to be a variable

Printing

In PHP, the default command for generating output is `echo`

- `void echo(arg1)`
  `void echo arg1, arg2, ...`
  - Outputs all arguments
  - No parentheses are allowed if there is more than one argument
  - More efficient than print (and therefore preferred)

Additionally, PHP also provides the functions `print`, and `printf`:

- `int print(arg)`
  - Outputs its argument
    Only one argument is allowed!
  - Returns value 1
  - Parentheses can be omitted

Printing

- `string sprintf(format, arg1, arg2, ....)`
  - Returns a string produced according to the formatting string *format*
  - Parentheses are necessary

  See http://www.php.net/manual/en/function.sprintf.php for details

- `int printf(format, arg1, arg2, ....)`
  - Produces output according to *format*
  - Parentheses are necessary
  - Returns the length of the outputted string

- Important: In contrast to Perl, a PHP array cannot take the place of a list of arguments

```
printf("%2d apples %2d oranges\n",array(5,7));
```

produces an error message

Printing

- `string vsprintf(format, array)`
  - Returns a string produced according to the formatting string *format*
  - Identical to `sprintf` but accepts an array as argument
  - Parentheses are necessary

- `int vprintf(format, array)`
  - Produces output according to *format*
  - Identical to `printf` but accepts an array as argument
  - Parentheses are necessary


```
vprintf("%2d␣apples␣%2d␣oranges\n",array(5,7));
```

```
5 apples 7 oranges
```

Revision

Read

- Chapter 6: PHP Arrays

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

- http://uk.php.net/manual/en/language.types.boolean.php
- http://uk.php.net/manual/en/language.types.integer.php
- http://uk.php.net/manual/en/language.types.float.php
- http://uk.php.net/manual/en/language.types.string.php
- http://uk.php.net/manual/en/language.types.array.php
- http://uk.php.net/manual/en/control-structures.foreach.php

COMP284 Scripting Languages

Lecture 11: PHP (Part 3)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

NULL

- `NULL` is both a special type and a value
- `NULL` is the only value of type `NULL`
  and the name of this constant is case insensitive

- A variable has both type `NULL` and value `NULL` in the following three situations:
  1. The variable has not yet been assigned a value (not equal to `NULL`)
  2. The variable has been assigned the value `NULL`
  3. The variable has been unset using the `unset` operation

- There are a variety of functions that can be used to test whether a variable is `NULL` including:
  - `bool isset($variable)`
    `TRUE` iff `$variable` exists and does not have value `NULL`
  - `bool is_null(expr)`
    `TRUE` iff *expr* is identical to `NULL`

NULL

Warning: Using `NULL` with == may lead to counter-intuitive results

```
$d = array();
echo var_dump($d), "\n";
array(0) {
}
echo 'is_null($d): ', (is_null($d)) ? "TRUE\n": "FALSE\n";
is_null($d): FALSE
echo '$d === null: ', ($d === null) ? "TRUE\n": "FALSE\n";
$d === null: FALSE
echo '$d  == null: ', ($d == null) ? "TRUE\n": "FALSE\n";
$d  == null: TRUE
```

Type juggling means that an empty array is (loosely) equal to `NULL` but not identical (strictly equal) to `NULL`

Resources

A resource is a reference to an external resource and corresponds to a Perl filehandle

- `resource fopen(filename, mode)`
  Returns a file pointer resource for *filename* access using *mode* on success, or `FALSE` on error

Mode	Operation	Create	Truncate
'r'	read file		
'r+'	read/write file		
'w'	write file	yes	yes
'w+'	read/write file	yes	yes
'a'	append file	yes	
'a+'	read/append file	yes	
'x'	write file	yes	
'x+'	read/write file	yes	

See `http://www.php.net/manual/en/resource.php` for further details

Resources

- `bool fclose(resource)`
  - Closes the resource
  - Returns TRUE on success

- `string fgets(resource [, length])`
  - Returns a line read from *resource* and returns FALSE if there is no more data to be read
  - With optional argument *length*, reading ends when *length* − 1 bytes have been read, or a newline or on EOF (whichever comes first)

- `string fread(resource, length)`
  - Returns *length* characters read from *resource*


```
$handle = fopen('somefile.txt', 'r');
while ($line = fgets($handle)) {
  // processing the line of the file
}
fclose($handle);
```

Resources

- `int fwrite(resource, string [, length])`
  - Writes a string to a resource
  - If *length* is given, writing stops after *length* bytes have been written or the end of string is reached, whichever comes first
- `int fprintf(resource, format, arg1, arg2, ...)`
  - Writes a list of arguments to a resource in the given format
  - Identical to `fprintf` with output to *resource*
- `int vfprintf (resource, format, array)`
  - Writes the elements of an array to a resource in the given format
  - Identical to `vprintf` with output to *resource*

```
$handle = fopen('somefile.txt', 'w');
fwrite($handle,"Hello World!".PHP_EOL); // `logical newline'
fclose($handle);
```

In contrast to Perl, in PHP `\n` always represents the character with ASCII code 10 not the platform dependent newline ⇝ use `PHP_EOL` instead

Control structures: conditional statements

The general format of conditional statements is very similar but not identical to that in Java and Perl:

```
if (condition) {
    statements
} elseif (condition) {
    statements
} else {
    statements
}
```


- the elseif-clauses is optional and there can be more than one
  Note: `elseif` instead of `elsif`!
- the else-clause is optional but there can be at most one
- in contrast to Perl, the curly brackets can be omitted if there is only a single statement in a clause

Control structures: conditional statements/expressions

- PHP allows to replace curly brackets with a colon : combined with an endif at the end of the statement:

```
if (condition):
    statements
elseif (condition):
    statements
else:
    statements
endif
```

This also works for the switch statement in PHP

However, this syntax becomes difficult to parse when nested conditional statements are used and is best avoided

- PHP also supports conditional expressions

```
condition ? if_true_expr : if_false_expr
```

Control structures: switch statement

A switch statement in PHP takes the following form

```
switch (expr) {
   case expr1:
         statements
         break;
   case expr2:
         statements
         break;
   default:
         statements
         break;
}
```

- there can be arbitrarily many case-clauses
- the default-clause is optional but there can be at most one
- *expr* is evaluated only once and then compared to *expr1*, *expr2*, etc using (loose) equality ==
- once two expressions are found to be equal the corresponing clause is executed
- if none of *expr1*, *expr2*, etc are equal to *expr*, then the default-clause will be executed
- `break` 'breaks out' of the switch statement
- if a clause does not contain a `break` command, then execution moves to the next clause

Control structures: switch statement

Example:

```
switch ($command) {
  case "North":
     $y += 1; break;
  case "South":
     $y -= 1; break;
  case "West";
     $x -= 1; break;
  case "East":
     $x += 1; break;
  case "Search":
     if (($x = 5) && ($y = 3))
        echo "Found a treasure\n";
      else
        echo "Nothing here\n";
     break;
  default:
     echo "Not a valid command\n"; break;
}
```

Control structures: switch statement

Not every case-clause needs to have associated statements

Example:

```
switch ($month) {
  case 1:   case 3:    case 5:   case 7:
  case 8:   case 10:   case 12:
     $days = 31;
     break;
  case 4:   case 6:    case 9:   case 11:
     $days = 30;
     break;
  case 2:
     $days = 28;
     break;
  default:
     $days = 0;
     break;
}
```

Control structures: while- and do while-loops

- PHP offers while-loops and do while-loops

```
while (condition) {
    statements
}

do {
    statements
} while (condition);
```

- As usual, curly brackets can be omitted if the loop consists of only one statement

Example:

```
// Compute the factorial of $number
$factorial = 1;
do {
    $factorial *= $number--;
} while ($number > 0);
```

Control structures: for-loops

- for-loops in PHP take the form

```
for (initialisation; test; increment) {
    statements
}
```

  Again, the curly brackets are not required if the body of the loop only consists of a single statement

- In PHP *initialisation* and *increment* can consist of more than one statement, separated by commas instead of semicolons

Example:

```
for ($i = 3, $j = 3; $j >= 0; $i++, $j--)
   echo "$i - $j - ", $i*$j, "\n";
```

```
3 - 3 - 9
4 - 2 - 8
5 - 1 - 5
6 - 0 - 0
```

Control structures: break and continue

- The break command can also be used in while-, do while-, and for-loops and discontinues the execution of the loop

```
while ($value = array_shift($data)) {
  $written = fwrite($resource,$value);
  if (!$written) break;
}
```

- The continue command stops the execution of the current iteration of a loop and moves the execution to the next iteration

```
for ($x = -2; $x <= 2; $x++) {
   if ($x == 0) continue;
   printf("10 / %2d = %3d\n",$x,(10/$x));
}
```

```
10 / -2 =  -5
10 / -1 = -10
10 /  1 =  10
10 /  2 =   5
```

Functions

Functions are defined as follows in PHP:

```
function identifier($param1,&$param2, ...) {
  statements
}
```


- Functions can be placed anywhere in a PHP script but preferably they should all be placed at start of the script (or at the end of the script)
- Function names are case-insensitive
- The function name must be followed by parentheses
- A function has zero, one, or more parameters that are variables
- Parameters can be given a default value using

  `$param = const_expr`

- When using default values, any defaults must be on the right side of any parameters without defaults

Functions

Functions are defined as follows in PHP:

```
function identifier($param1,&$param2, ...) {
  statements
}
```


- The return statement

  ```
  return value
  ```

  can be used to terminate the execution of a function and to make *value* the return value of the function

- The return value does not have to be scalar value
- A function can contain more than one return statement
- Different return statements can return values of different types

Calling a function

A function is called by using the function name followed by a list of arguments in parentheses

```
function identifier($param1, &$param2, ...) {
  ...
}
... identifier(arg1, arg2,...) ...
```


- The list of arguments can be shorter as well as longer as the list of parameters
- If it is shorter, then default values must have been specified for the parameters without corresponding arguments

Example:

```
function sum($num1,$num2) {
  return $num1+$num2;
}
echo "sum: ",sum(5,4),"\n";
$sum = sum(3,2);
```

Variables

PHP distinguishes three categories of variables:

- Local variables are only accessible in the part of the code in which they are introduced
- Global variables are accessible everywhere in the code
- Static variables are local variables within a function that retain their value between separate calls of the function

By default, variables in PHP are local but not static
(Variables in Perl are by default global)

PHP functions: Example

```
function bubble_sort($array) {
  // $array, $size, $i, $j are all local
  if (!is_array($array))
    trigger_error("Argument not an array\n", E_USER_ERROR);
  $size = count($array);
  for ($i=0; $i<$size; $i++) {
    for ($j=0; $j<$size-1-$i; $j++) {
      if ($array[$j+1] < $array[$j]) {
        swap($array, $j, $j+1);  }  }  }
  return $array;
}

function swap(&$array, $i, $j) {
  // swap expects a reference (to an array)
  $tmp = $array[$i];
  $array[$i] = $array[$j];
  $array[$j] = $tmp;
}
```

PHP functions: Example

```
function bubble_sort($array) {
  ... swap($array, $j, $j+1); ...
  return $array;
}

function swap(&$array, $i, $j) {
  $tmp = $array[$i];
  $array[$i] = $array[$j];
  $array[$j] = $tmp; }

$array = array(2,4,3,9,6,8,5,1);
echo "Before sorting ", join(", ",$array), "\n";
$sorted = bubble_sort($array);
echo "After  sorting ", join(", ",$array), "\n";
echo "Sorted array   ", join(", ",$sorted), "\n";
```

```
Before sorting 2, 4, 3, 9, 6, 8, 5, 1
After  sorting 2, 4, 3, 9, 6, 8, 5, 1
Sorted array   1, 2, 3, 4, 5, 6, 8, 9
```

Functions and global variables

- A variable is declared to be global using the keyword `global`

```
function echo_x($x) {
  echo $x," ";
  global $x;
  echo $x;
}

$x = 5;  // this is a global variable called $x
echo_x(10); // prints first '10' then '5'
```

  - ⇝ an otherwise local variable is made accessible outside its normal scope using `global`
  - ⇝ all global variables with the same name refer to the same storage location/data structure
  - ⇝ an `unset` operation removes a specific variable, but leaves other (global) variables with the same name unchanged

PHP functions and Global variables


```
function modify_or_destroy_var($arg) {
  global $x, $y;
  if (is_bool($arg) && !$arg) { $x = $x * $y; }
  if (is_bool($arg) && $arg) { unset($x); echo $x; }
}
$x = 2; $y = 3; $z = 4;
echo "1: \$x = $x, \$y = $y, \$z = $z\n";
```

```
1: $x = 2, $y = 3, $z = 4
```

```
unset($z);
echo "2: \$x = $x, \$y = $y, \$z = $z\n";
```

```
PHP Notice:  Undefined variable: z in script on line 9
2: $x = 2, $y = 3, $z =
```

```
modify_or_destroy_var(false);
echo "3: \$x = $x, \$y = $y\n";
```

```
3: $x = 6, $y = 3
```

```
modify_or_destroy_var(true);
echo "4: \$x = $x, \$y = $y\n";
```

```
PHP Notice:  Undefined variable: x in script on line 4
4: $x = 6, $y = 3
```

PHP functions and Static variables

- A variable is declared to be static using the keyword `static` and should be combined with the assignment of an initial value (initialisation)

```
function counter() { static $count = 0; return $count++; }
```


⇝ static variables are initialised only once

```
function counter() { static $count = 0; return $count++; }
$count = 5;
echo "1: global \$count = $count\n";
echo "2: static \$count = ",counter(),"\n";
echo "3: static \$count = ",counter(),"\n";
echo "4: global \$count = $count\n";
```



```
1: global $count = 5
2: static $count = 0
3: static $count = 1
4: global $count = 5
```

Functions and HTML

- It is possible to include HTML markup in the body of a function definition
- The HTML markup can in turn contain PHP scripts
- A call of the function will execute the PHP scripts, insert the output into the HTML markup, then output the resulting HTML markup


```
<?php
function print_form($fn, $ln) {
?>
<form action="process_form.php" method=POST">
<label>First Name: <input type="text" name="f" value="<?php echo $fn?>"></label><br>
<label>Last Name<b>*</b>:<input type="text" name="l" value="<?php echo $ln?>"></label><br>
<input type="submit" name="submit" value="Submit"> <input type=reset>
</form>
<?php
}
print_form("Ullrich","Hustadt");
?>
```

```
<form action="process_form.php" method=POST">
<label>First Name: <input type="text" name="f" value="Ullrich"></label><br>
<label>Last Name<b>*</b>:<input type="text" name="l" value="Hustadt"></label><br>
<input type="submit" name="submit" value="Submit"> <input type=reset>
</form>
```

Functions with variable number of arguments

The number of arguments in a function call is allowed to exceed the number of its parameters

⇝ the parameter list only specifies the minimum number of arguments

- `int func_num_args()`
  returns the number of arguments passed to a function

- `mixed func_get_arg(`*arg_num*`)`
  returns the specified argument, or FALSE on error
- `array func_get_args()`
  returns an array with copies of the arguments passed to a function


```
function sum() { // no minimum number of arguments
  if (func_num_args() < 1) return null;
  $sum = 0;
  foreach (func_get_args() as $value) { $sum += $value; }
  return $sum;
}
```

Including and requiring files

- It is often convenient to build up libraries of function definitions, stored in one or more files, that are then reused in PHP scripts
- PHP provides the commands `include`, `include_once`, `require` and `require_once` to incorporate the content of a file into a PHP script

```
include 'mylibrary.php';
```

- PHP code in a library file must be enclosed within a PHP start tag `<?php` and an end PHP tag `?>`
- The incorporated content inherits the scope of the line in which an `include` command occurs
- If no absolute or relative path is specified, PHP will search for the file
  - first, in the directories in the include path `include_path`
  - second, in the script's directory
  - third, in the current working directory

Including and requiring files

- Several `include` or `require` commands for the same library file results in the file being incorporated several times
  ⇝ defining a function more than once results in an error
- Several `include_once` or `require_once` commands for the same library file results in the file being incorporated only once
- If a library file requested by `include` and `include_once` cannot be found, PHP generates a warning but continues the execution of the requesting script
- If a library file requested by `require` and `require_once` cannot be found, PHP generates a error and stops execution of the requesting script

PHP Libraries: Example

mylibrary.php

```
<?php
function bubble_sort($array) {
  ... swap($array, $j, $j+1); ...
  return $array;
}

function swap(&$array, $i, $j) {
  ...
}
?>
```

example.php

```
<?php
require_once 'mylibrary.php';
$array = array(2,4,3,9,6,8,5,1);
$sorted = bubble_sort($array);
?>
```

Revision

Read

- Chapter 4: Expressions and Control Flow in PHP
- Chapter 5: PHP Functions and Objects
- Chapter 7: Practical PHP

of

R. Nixon:

Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

- `http://uk.php.net/manual/en/language.control-structures.php`
- `http://uk.php.net/manual/en/language.functions.php`
- `http://uk.php.net/manual/en/function.include.php`
- `http://uk.php.net/manual/en/function.include-once.php`
- `http://uk.php.net/manual/en/function.require.php`
- `http://uk.php.net/manual/en/function.require-once.php`

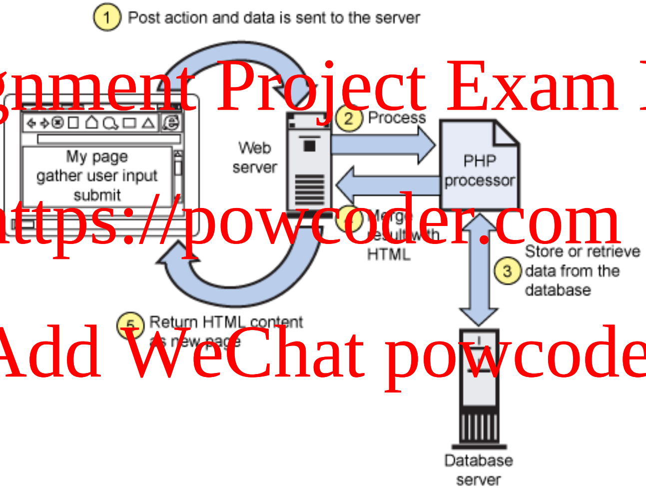
COMP284 Scripting Languages
Lecture 12: PHP (Part 4)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Web applications using PHP



IBM: Build Ajax-based Web sites with PHP, 2 Sep 2008.
`https://www.ibm.com/developerworks/library/wa-aj-php/` [accessed 6 Mar 2013]

HTML forms

When considering Perl CGI programming we have used HTML forms that generated a client request that was handled by a Perl CGI program:

```
<form action=
"http://cgi.csc.liv.ac.uk/cgi-bin/cgiwrap/ullrich/demo"
 method="post">
...
</form>
```

Now we will use a PHP script instead:

```
<form action="http://cgi.csc.liv.ac.uk/~ullrich/demo.php"
 method="post">
...
</form>
```

- The PHP script file must be stored in a directory accessible by the web server, for example `$HOME/public_html`, and be readable by the web server
- The PHP script file name must have the extension `.php`, e.g. `demo.php`

Information available to PHP scripts

- Information about the PHP environment
- Information about the web server and client request
- Information stored in files and datbases
- Form data
- Cookie/Session data
- Miscellaneous
  - `string date(format)`
    returns the current date/time presented according to *format*
    for example `date('H:i l, j F Y')`
    results in `12:20 Thursday, 8 March 2012`
    (See `http://www.php.net/manual/en/function.date.php`)
  - `int time()`
    returns the current time measured in the number of seconds since January 1 1970 00:00:00 GMT

PHP environment

- `phpinfo()` displays information about the PHP installation and EGPCS data (Environment, GET, POST, Cookie, and Server data) for the current client request

- `phpinfo(`*part*`)` displays selected information

```
<html><head></head><body>
<?php
  phpinfo();                    // Show all information
  phpinfo(INFO_VARIABLES);      // Show only info on EGPCS data
?>
</body></html>
```


http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/phpinfo.php

INFO_GENERAL	The configuration, php.ini location, build date, web server
INFO_CONFIGURATION	Local and master values for PHP directives
INFO_MODULES	Loaded modules
INFO_VARIABLES	All EGPCS data

Manipulating the PHP configuration

The following functions can be used to access and change the configuation of PHP from within a PHP script:

- `array ini_get_all()`
  - returns all the registered configuration options

- `string ini_get(`*option*`)`
  - returns the value of the configuration option on success
- `string ini_set(`*option*`,` *value*`)`
  - sets the value of the given configuration option to a new value
  - the configuration option will keep this new value during the script's execution and will be restored afterwards

- `void ini_restore(`*option*`)`
  - restores a given configuration option to its original value

Server variables

The $_SERVER array stores information about the web server and the client request

↝ Similar to %ENV for Perl CGI programs

```
<html><head></head><body>
<?php
echo 'Server software: ',$_SERVER['SERVER_SOFTWARE'],'<br />';
echo 'Remote address: ',$_SERVER['REMOTE_ADDR'],'<br />';
echo 'Client browser: ',$_SERVER['HTTP_USER_AGENT'],'<br />';
echo 'Request method: ',$_SERVER['REQUEST_METHOD'];
?></body></html>
```


http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/server.php


```
Server software: Apache/2.2.22 (Fedora)
Remote address: 10.128.0.215
Client browser: Mozilla/5.0 ... Chrome/41.0.2272.53 ...
Request method:
```

See http://php.net/manual/en/reserved.variables.server.php for a list of keys

Form data

- Form data is passed to a PHP script via the three arrays:

`$_POST`	Data from POST client requests
`$_GET`	Data from GET client requests
`$_REQUEST`	Combined data from POST and GET client requests (derived from `$_POST` and `$_GET`)

⇝ Accessing `$_REQUEST` is the equivalent in PHP to using the `param` routine in Perl


```
<form action="process.php" method="post">
<label>Enter your user name:
      <input type="text" name="username"></label><br>
<label>Enter your full name:
      <input type="text" name="fullname"></label><br>
<input type="submit" value="Click␣for␣response"></form>
```

`$_REQUEST['username']`	Value entered into field with name ‘username’
`$_REQUEST['fullname']`	Value entered into field with name ‘fullname’

Forms in PHP: Example (1)

- Create a web-based system that asks the user to enter the URL of a file containing bibliographic information
- Bibliographic information will have the following form:

```
@entry{
 name={Jonas Lehner},
 name={Andreas Schoknecht},
 title={<strong>You only live twice</strong>},
}
@entry{
 name={Andreas Schoknecht},
 name={Eva Eggeling},
 title={No End in Sight?}
}
```

- The system should extract the names, count them, and create a table of names and their frequency, ordered from most frequent to least frequent

Forms in PHP: Example (1)

extract_names.php

```
<!DOCTYPE html>
<html><head><title>Name Extraction</title></head><body>
<?php
require_once 'extraction.php';
 if (isset($_SERVER['REQUEST_METHOD']) &&
     $_SERVER['REQUEST_METHOD'] == 'POST' &&
     isset($_REQUEST['url'])) {
   $extracted_names = extract_names($_REQUEST['url']);
   echo "<p>The names occurring in <br>", htmlspecialchars($_REQUEST['url']),
        "<br>are</p>$extracted_names\n";
 } else {
   echo <<<FORM
   <form method="post">
     <label>Enter a URL:
       <input type="text" name="url" size="100"
        value="http://cgi.csc.liv.ac.uk/~ullrich/COMP284/tests/a1test1.txt">
     </label><br><br>
     <input type="submit" value="Extract Names">
   </form>
FORM;
 }
?>
</body></html>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/extract_names.php

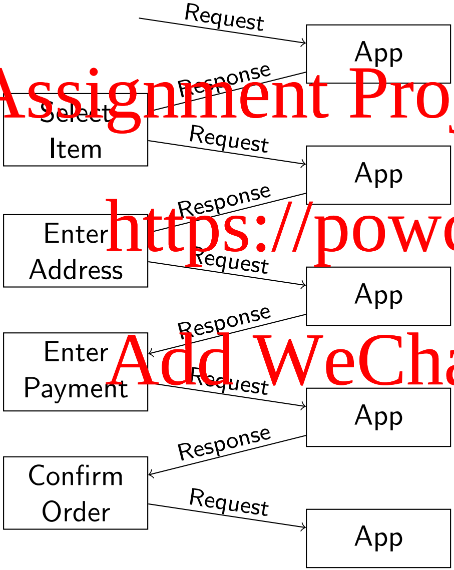
Forms in PHP: Example (1)

extraction.php

```
<?php
function extract_names($url) {
 $text = file_get_contents($url);
 if ($text === false)
   return "ERROR: INVALID URL!";
 else {
   $correct = preg_match_all("/name={([^\}]+)}/",
                 $text,$matches,PREG_PATTERN_ORDER);
   if ($correct == 0) return "ERROR: NO NAMES FOUND";
   $count = array_count_values($matches[1]);
   arsort($count);
   foreach ($count as $name => $number) {
     $table .= "<tr><td>$name</td><td>$number</td></tr>";
   }
   $table = "<table><thead><tr><th>Name</th><th>No of occur".
   "rences</th></tr></thead><tbody>".$table."</tbody></table>";
   return $table;
} }
?>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/extraction.php

Web Applications Revisited



- An interaction between a user and a server-side web application often requires a sequence of requests and responses
- For each request, the application starts from scratch
  - it does not maintain a state between consecutive requests
  - it does not know whether the requests come from the same user or different users

$\leadsto$ data needs to be transferred from one execution of the application to the next

Transfer of Data: Example

- Assume for a sequence of requests we do not care whether they come from the same user or different users
- Then hidden inputs can be used for the transfer of data from one request / page to the next

form1.php

```
<form action="form2.php" method="post">
  <label>Name: <input type="text" name="name"></label>
</form>
```

form2.php

```
<form action="process.php" method="post">
  <label>Address: <input type="text" name="address"></label>
  <input type="hidden" name="name"
         value="<?php echo $_REQUEST[’name’] ?>">
</form>
```

process.php

```
<?php
 echo $_REQUEST[’name’];    echo $_REQUEST[’address’];
?>
```

Sessions

- By default, HTML and web servers do not keep track whether several client requests come from the same user or different users

  Thus, a process that spans several pages, for example, placing an order, requires additional mechanisms

- Sessions help solve this problem by associating client requests with specific users and maintaining data during a user's visit

- Sessions are often linked to user authentication but session can be used without user authentication, for example, eCommerce websites maintain a 'shopping basket' without requiring user authentication first

  However, sessions are the mechanism that is typically used to allow or deny access to web pages based on a user having been authenticated

Sessions

- Servers keep track of a user's sessions by using a session identifier, which
  - is generated by the server when a session starts and
  - is then used by the browser when the user requests a page from the server

  The session identifier can be sent through a cookie or by passing the session identifier in client requests

- In addition, one can use session variables for storing information to relate to a user and her session (session data), for example, the items of an order

- Sessions only store information temporarily

  If one needs to preserve information between visits by the same user, one needs to consider a method such as using a cookie or a database to store such information

Cookies

Browser ⟶ Server

```
GET /index.html HTTP/1.1
Host: intranet.csc.liv.ac.uk
```

Browser ⟵ Server

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: name1=value1
Set-Cookie: name2=value2; Expires= Thu, 20 Mar 2014, 14:00 GMT
(content of index.html)
```

Browser ⟶ Server

```
GET /teaching.html HTTP/1.1
Host: intranet.csc.liv.ac.uk
Cookie: name1=value1; name2=value2
Accept: */*
```

Browser ⟵ Server

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: name1=value3
Set-Cookie: name2=value4; Expires= Fri, 21 Mar 2014, 14:00 GMT
Set-Cookie: name3=value5; Expires= Fri, 28 Mar 2014, 20:00 GMT
(content of teaching.html)
```


Wikipedia Contributors: HTTP Cookie. Wikipedia, The Free Encyclopedia, 5 March 2014 20:50. `http://en.wikipedia.org/wiki/HTTP_cookie` [accessed 6 Mar 2014]

PHP sessions

Sesssions proceed as follows

1. Start a PHP session
   - `bool session_start()`
   - `string session_id([id])`
   - `bool session_regenerate_id([delete_old])`
2. Maintain session data
   - `bool session_start()`
   - `$_SESSION` array
   - `bool isset($_SESSION[key])`
   - (interacting with a database)
3. End a PHP session
   - `bool session_destroy()`
   - `void session_unset()`
   - `bool setcookie(name, value, expires, path)`

Start a session

- `bool session_start()`
  - creates a session
  - creates a session identifier (session id) when a session is created
  - sets up `$_SESSION` array that stores session variables and session data
  - the function must be executed before any other header calls or output is produced
- `string session_id([id])`
  - get or set the session id for the current session
  - the constant SID can also be used to retrieve the current name and session id as a string suitable for adding to URLs
- `string session_name([name])`
  - returns the name of the current session
  - if a name is given, the current session name will be replaced with the given one and the old name returned

Start a PHP session

- **bool session_regenerate_id([*delete_old*])**
  - replaces the current session id with a new one
  - by default keeps the current session information stored in $_SESSION
  - if the optional boolean agument is TRUE, then the current session information is deleted

⇝ regular use of this function alleviates the risk of a session being 'hijacked'

```
<?php
 session_start();
 echo "Session id: ",session_id(),"<br />";
 echo "Session name: ",session_name(),"<br />";

 session_regenerate_id();
 echo "Session id: ",session_id(),"<br />";     // changed
 echo "Session name: ",session_name(),"<br />"; // unchanged
?>
```

Maintain session data

- `bool session_start()`
  - resumes the current session based on a session identifier passed via a GET or POST request, or passed via a cookie
  - restores session variables and session data into `$_SESSION`
  - the function must be executed before any other header calls or output is produced
- `$_SESSION` array
  - an associative array containing session variables and session data
  - you are responsible for choosing keys (session variables) and maintaining the associated values (session data)
- `bool isset($_SESSION[`*key*`])`
  returns TRUE iff `$_SESSION[`*key*`]` has already been assigned a value

Maintain session data

- bool session_start()
- $_SESSION array
- bool isset($_SESSION[*key*])


```
<?php
// Counting the number of page requests in a session
// Each web page contains the following PHP code
session_start();
if (!isset($_SESSION['requests']))
   $_SESSION['requests'] = 1;
else
   $_SESSION['requests']++;
echo "#Requests in this session so far: ",
     $_SESSION['requests'],"<br />\n";
?>
```

End a PHP session

- `bool session_destroy()`
  - destroys all of the data associated with the current session
  - it does not unset any of the global variables associated with the session, or unset the session cookie
- `void session_unset()`
  - frees all session variables currently registered
- `bool setcookie(`*`name`*`, `*`value`*`, `*`expires`*`, `*`path`*`)`
  - defines a cookie to be sent along with the rest of the HTTP headers
  - must be sent before any output from the script
  - the first argument is the name of the cookie
  - the second argument is the value of the cookie
  - the third argument is time the cookie expires (as a Unix timestamp), and
  - the fourth argument is the parth on the server in which the cookie will be available

End a PHP session

- bool session_destroy()
  - destroys all of the data associated with the current session
- void session_unset()
  - frees all session variables currently registered
- bool setcookie(*name*, *value*, *expires*, *path*)
  - defines a cookie to be sent along with the rest of the HTTP headers


```php
<?php
session_start();
session_unset();
if (session_id() != "" || isset($_COOKIE[session_name()]))
  // force the cookie to expire
  setcookie(session_name(),session_id(),time()-2592000,'/');
session_destroy();
?>
```

Note: Closing your web browser will also end a session

More on session management

The following code tracks whether a session is active and ends the session if there has been no activity for more then 30 minutes


```
if (isset($_SESSION['LAST_ACTIVITY']) &&
   (time() - $_SESSION['LAST_ACTIVITY'] > 1800)) {
   // last request was more than 30 minates ago
   session_destroy(); // destroy session data in storage
   session_unset();   // unset session variables
   if (session_id() != "" || isset($_COOKIE[session_name()]))
      setcookie(session_name(), session_id(), time()-2592000,'/');
} else {
   // update last activity time stamp
   $_SESSION['LAST_ACTIVITY'] = time();
}
```


The following code generates a new session identifier every 30 minutes


```
if (!isset($_SESSION['CREATED'])) {
   $_SESSION['CREATED'] = time();
} else if (time() - $_SESSION['CREATED'] > 1800) {
         // session started more than 30 minates ago
         session_regenerate_id(true);
         $_SESSION['CREATED'] = time();
}
```

http://stackoverflow.com/questions/520237/how-do-i-expire-a-php-session-after-30-minutes

PHP sessions: Example

mylibrary.php:

```
<?php
session_start();

function destroy_session_and_data() {
 session_unset();
 if (session_id() != "" || isset($_COOKIE[session_name()]))
   setcookie(session_name(),session_id(),time()-2592000,'/');
 session_destroy();
}

function count_requests() {
 if (!isset($_SESSION['requests']))
    $_SESSION['requests'] = 1;
 else $_SESSION['requests']++;
 return $_SESSION['requests'];
}
?>
```

PHP sessions: Example

page1.php:

```
<?php
 require_once 'mylibrary.php';
 echo "<html><head></head><body>\n";
 echo "Hello visitor!<br />This is your page request no ";
 echo count_requests()." from this site.<br />\n";
 echo '<a href="page1.php">Continue</a> |
       <a href="finish.php">Finish</a></body>';
?>
```


finish.php:

```
<?php
 require_once 'mylibrary.php';
 destroy_session_and_data();
 echo "<html><head></head><body>\n";
 echo "Goodbye visitor!<br />\n";
 echo '<a href="page1.php">Start again</a></body>';
?>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/page1.php

PHP and Cookies

Cookies can survive a session and transfer information from one session to the next

cmylibrary.php:


```
<?php
session_start();
function destroy_session_and_data() { // unchanged }

function count_requests() {
  if (!isset($_COOKIE['requests'])) {
    setcookie('requests', 1, time()+31536000, '/');
    return 1;
  } else {
    // $_COOKIE['requests']++ would not survive, instead use
    setcookie('requests', $_COOKIE['requests']+1,
              time()+31536000, '/'); // valid for 1 year
    return $_COOKIE['requests']+1;
} }
?>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/cpage1.php

PHP Sessions and Authentication

- Sessions are the mechanism that is typically used to allow or deny access to web pages based on a user having been authenticated

- Outline solution
  - We want to protect a page content.php from unauthorised use
  - Before being allowed to access content.php, users must first authenticate themselves by providing a username and password on the page login.php
  - The system maintains a list of valid usernames and passwords in a database and checks usernames and passwords entered by the user against that database
    If the check succeeds, a session variable is set
  - The page content.php checks whether this session variable is set
    If the session variable is set, the user will see the content of the page
    If the session variable is not set, the user is redirected to login.php
  - The system also provides a logout.php page to allow the user to log out again

PHP Sessions and Authentication: Example

Second part of login.php:

```
<!DOCTYPE html>
<html>
<head><title>Login</title></head>
<body>
 <h1>Login</h1>
 <form action="" method="post">
  <label>Username:
  <input name="user" placeholder="username" type="text">
  </label>
  <label>
  Password:
  <input name="passwd" placeholder="**" type="password">
  </label>
  <input name="submit" type="submit" value="login ">
  <span><?php echo $error; ?></span>
 </form>
</body>
</html>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/login.php

PHP Sessions and Authentication: Example

First part of login.php:

```
<?php
session_start();

function checkCredentials($user,$passwd) {
  // Check whether $user and $passwd are non-empty
  // and match an entry in the database
}

$error='';
if (isset($_POST['submit'])) {
   if (checkCredentials($_REQUEST['user'],$_REQUEST['passwd'])) {
      $_SESSION['user']=$_REQUEST['user'];
      header("location:content.php"); // Redirecting to Content
   } else {
      $error = "Username or Password is invalid. Try Again";
   }
}
if (isset($_SESSION['user'])){
   header("location:content.php");
}
?>
```

PHP Sessions and Authentication: Example

content.php:

```
<?php
session_start();
if (!isset($_SESSION['user'])) {
   // User is not logged in, redirecting to login page
   header('Location:login.php');
}
?>
<!DOCTYPE html>
<html>
<head><title>Content that requires login</title></head>
<body>
<h1>Protected Content</h1>
<b>Welcome <i><?php echo $_SESSION['user'] ?></i></b><br />
<b><a href="logout.php">Log Out</a></b>
</body>
</html>
```


http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/content.php

PHP Sessions and Authentication: Example

logout.php:

```
<?php
session_start();
$user = $_SESSION['user'];
session_unset();
session_destroy();
?>
<!DOCTYPE html>
<html>
<head>
<title>Logout</title>
</head>
<body>
<h1>Logout</h1>
<b>Goodbye <i><?php echo $user ?></i></b><br />
<b><a href="login.php">Login</a></b>
</form>
</body>
```


`http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/logout.php`

Revision

Read

- Chapter 10: Accessing MySQL Using PHP
- Chapter 11: Form Handling
- Chapter 13: Cookies, Sessions, and Authentication

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

COMP284 Scripting Languages
Lecture 13: PHP (Part 5)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Defining and Instantiating a Class

- PHP is an object-oriented language with classes
- A class can be defined as follows:


```
class identifier {
    property_definitions
    function_definitions
}
```


  - The class name *identifier* is case-sensitive
  - The body of a class consists of property definitions and function definitions
  - The function definitions may include the definition of a constructor

- An object of a class is created using

```
new identifier(arg1,arg2,...)
```

where `arg1,arg2,...` is a possibly empty list of arguments passed to the constructor of the class *identifier*

A Closer Look at Class Definitions

In more detail, the definition of a class typically looks as follows

```
class identifier {
  # Properties
  vis $attrib1
  ...
  vis $attribN = value

  # Constructor
  function __construct(p1,...) {
     statements
  }
  # Methods
  vis function method1(p1,...) {
    statements
  }
  vis function methodN(p1,...) {
    statements
  }
}
```

- Every instance `obj` of this class will have attributes *attrib1*,... and methods *method1*(), ... accessible as `obj->`*attrib1* and `obj->`*method1*(*a1*...)
- `__construct` is the constructor of the class and will be called whenever `new` *identifier*(*a1*,...) is executed
- *vis* is a declaration of the visibility of each attribute and method

A Closer Look at Class Definitions

- The pseudo-variable `$this` is available when a method is called from within an object context and is a reference to the calling object
- Inside method definitions, `$this` can be used to refer to the properties and methods of the calling object
- The object operator `->` is used to access methods and properties of the calling object


```
class Rectangle {
  protected $height;
  protected $width;

  function __construct($height,$width) {
     $this->width  = $width;
     $this->height = $height;
  }
}
```

Visibility

- Properties and methods can be declared as

`public`	accessible everywhere
`private`	accessible only within the same class
`protected`	accessible only within the class itself and by inheriting and parent classes

- For properties, a visibility declaration is required
- For methods, a visibility declaration is optional
  ⇝ by default, methods are `public`
- Accessing a private or protected property / method outside its visibility is a fatal error

```
class Vis {
 public    $public    = 1;
 private   $private   = 2;
 protected $protected = 3;
 protected function proFc() {}
 private   function priFc() {}
}
$v = new Vis();
echo $v->public;    # prints 1
echo $v->private;   # Fatal Error
echo $v->protected; # Fatal Error
echo $v->priFc();   # Fatal Error
echo $v->proFc();   # Fatal Error
```

Constants

- Classes can have their own constants and
  constants can be declared to be public, private or protected
  ⤳ by default, class constants are public

```
vis const identifier = value;
```

- Accessing a private or protected constant outside its visibility is a fatal error ⤳ execution of the script stops
- Class constants are allocated once per class,
  and not for each class instance
- Class constants are accessed using the scope resolution operator ::

```
class MyClass {
  const SIZE = 10;
}
echo MyClass::SIZE;   # prints 10
$o = new MyClass();
echo $o::SIZE;        # prints 10
```

Static Properties and Methods

- Class properties or methods can be declared static
- Static class properties and methods are accessed (via the class) using the scope resolution operator ::

- Static class properties cannot be accessed via an instantiated class object, but static class methods can
- Static class method have no access to $this


```
class Employee {
  static $totalNumber = 0;
  public $name;

  function __construct($name) {
    $this->$name = $name;
    Employee::$totalNumber++;
} }
$e1 = new Employee("Ada");
$e2 = new Employee("Ben");
echo Employee::$totalNumber  # prints 2
```

Destructors

- A class can have a destructor method `__destruct` that will be called as soon as there are no other references to a particular object


```
class Employee {
  static $totalNumber = 0;
  public $name;

  function __construct($name) {
    $this->name = $name;
    Employee::$totalNumber++;
  }
  function __destruct() {
    Employee::$totalNumber--;
  }
}
$e1 = new Employee("Ada");
$e2 = new Employee("Ben");
echo Employee::$totalNumber  # prints 2
$e1 = null;
echo Employee::$totalNumber  # prints 1
```

Inheritance

- In a class definition it is possible to specify one parent class from which a class inherits constants, properties and methods:

```
class identifier1 extends identifier2 { ... }
```


- The constructor of the parent class is not automatically called it must be called explicitly from the child class
- Inherited constants, properties and methods can be overridden by redeclaring them with the same name defined in the parent class

- The declaration `final` can be used to prevent a method from being overriden

- Using `parent::` it is possible to access overridden methods or static properties of the parent class
- Using `self::` it is possible to access static properties and methods of the current class

Inheritance: Example

```
class Rectangle {
  protected $height;
  protected $width;

  function __construct($height,$width) {
     $this->width  = $width;
     $this->height = $height;
  }
  function area() {
     return $this->width * $this->height;
} }

class Square extends Rectangle {
  function __construct($size) {
    parent::__construct($size,$size);
} }

$rt1 = new Rectangle(3,4);
echo "\$rt1 area = ",$rt1->area(),"\n";
$sq1 = new Square(5);
echo "\$sq1 area = ",$sq1->area(),"\n";
```

```
$rt1 area = 12
$sq1 area = 15
```

Interfaces

- Interfaces specify which methods a class must implement without providing an implementation
- Interfaces are defined in the same way as a class with the keyword `class` replaced by `interface`
- All methods in an interface must be declared public
- A class can declare that it implements one ore more interfaces using the `implements` keyword

```
interface Shape {
  public function area();
}
class Rectangle implements Shape {
  ...
}
```

Introspection Functions

There are functions for inspecting objects and classes:

`bool class_exists(string class)`
returns `TRUE` iff a class *`class`* exists
`class_exists('Rectangle')    # returns TRUE`
`string get_class(object obj)`
returns the name of the class to which an object belongs
`get_class($sq1)    # returns 'Square'`
`bool is_a(object obj, string class)`
returns `TRUE` iff *`obj`* is an instance of class named *`class`*
`is_a($sq1,'Rectangle')    # returns TRUE`
`bool method_exists(object obj,string method)`
returns `TRUE` iff *`obj`* has a method named *`method`*
`method_exists($sq1,'area')    # returns TRUE`

Introspection Functions

There are functions for inspecting objects and classes:

`bool property_exists(object obj,string property)`
returns `TRUE` iff `object` has a property named `property`
`property_exists($sq1,'size') # returns FALSE`
`get_object_vars(object)`
returns an array with the accessible non-static properties of `object` mapped to their values
`get_object_vars($e2)` `# returns ["name" => "Ben"]`
`get_class_methods(class)`
returns an array of method names defined for `class`
`get_class_methods('Square')` `# returns ["__construct", "area"]`

The PDO Class

- The PHP Data Objects (PDO) extension defines an interface for accessing databases in PHP
- Various PDO drivers implement that interface for specific database management systems
  - `PDO_MYSQL` implements the PDO interface for MySQL 3.x to 5.x
  - `PDO_SQLSRV` implements the PDO interface for MS SQL Server and SQL Azure

Connections

- Before we can interact with a DBMS we need to establish a connection to it
- A connection is established by creating an instance of the PDO class
- The constructor for the PDO class accepts arguments that specify the database source (DSN), username, password and additional options

```
$pdo = new PDO(dsn,username,password,options);
```

- Upon successful connection to the database, the constructor returns an instance of the PDO class
- The connection remains active for the lifetime of that PDO object
- Assigning NULL to the variable storing the PDO object destroys it and closes the connection

```
$pdo = NULL
```

Connections: Example


```
# Connection information for the Departmental MySQL Server
$host     = "mysql";
$user     = "ullrich";
$passwd   = "--------";
$db       = "ullrich";
$charset = "utf8mb4";
$dsn      = "mysql:host=$host;dbname=$db;charset=$charset";

# Useful options
$opt = array(
  PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
  PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
  PDO::ATTR_EMULATE_PREPARES   => false
);

try {
 $pdo = new PDO($dsn,$user,$passwd,$opt);
} catch (PDOException $e) {
 echo 'Connection failed: ',$e->getMessage();
}
```

Queries

- The `query()` method of PDO objects can be used to execute an SQL query

```
$result = $pdo->query(statement)
$result = $pdo->query("SELECT␣*␣FROM␣meetings")
```

- `query()` returns the result set (if any) of the SQL query as a PDOStatement object

- The `exec()` method of PDO objects executes an SQL statement, returning the number of rows affected by the statement

```
$rowNum = $pdo->exec(statement)
$rowNum = $pdo->exec("DELETE␣*␣FROM␣meetings")
```

Processing Result Sets

- To get a single row as an array from a result set stored in a PDOStatement object, we can use the fetch() method
- By default, PDO returns each row as an array indexed by the column name and 0-indexed column position in the row


```
$row = $result->fetch()
```

```
array('slot' => 1,
      'name' => 'Michael North',
      'email' => 'M.North@student.liverpool.ac.uk',
      0 => 1,
      1 => 'Michael North',
      2 => 'M.North@student.liverpool.ac.uk')
```


- After the last call of fetch() the result set should be released using

```
$rows = $result->closeCursor()
```

- The get all rows as an array of arrays from a result set stored in a PDOStatement object, we can use the fetchAll() method

```
$rows = $result->fetchAll()
```

Processing Result Sets

- We can use a while-loop together with the `fetch()` method to iterate over all rows in a result set

```
while ($row = $result->fetch()) {
  echo "Slot:  ",$row["slot"], "<br>\n";
  echo "Name:  ",$row["name"], "<br>\n";
  echo "Email: ",$row["email"],"<br><br>\n";
}
```


- Alternatively, we can use a foreach-loop

```
foreach($result as $row) {
  echo "Slot:  ",$row["slot"], "<br>\n";
  echo "Name:  ",$row["name"], "<br>\n";
  echo "Email: ",$row["email"],"<br><br>\n";
}
```

Processing Result Sets

- Using `bindColumn()` we can bind a variable a particular column in the result set from a query
  - columns can be specified by number (starting with 1!)
  - columns can be specified by name (matching case)
- Each call to `fetch()` and `fetchAll()` will then update all the variables that are bound to columns
- The binding needs to be renewed after each query execution


```
$result->bindColumn(1, $slot);          # bind by column no
$result->bindColumn(2, $name);
$result->bindColumn('email', $email);   # bind by column name
while ($row = $result->fetch(PDO::FETCH_BOUND)) {
  echo "Slot:  ",$slot, "<br>\n";
  echo "Name:  ",$name, "<br>\n";
  echo "Email: ",$email,"<br><br>\n";
}
```

Prepared Statements

- The use of parameterised prepared statements is preferable over queries
- Prepared statements are are parsed, analysed, compiled and optimised only once
- Prepared statements can be executed repeatedly with different arguments
- Arguments to prepared statements do not need to be quoted and binding of parameters to arguments will automatically prevent SQL injection
- PDO can emulate prepared statements for a DBMS that does not support them
- MySQL supports prepared statements natively, so PDO emulation should be turned off

```
$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, FALSE);
```

Prepared Statements: SQL Templates

- An SQL template is an SQL query (as a string) possibly containing either
  - named parameters of the form :*name* where *name* is a PHP identifier, or

  - question marks ?

  for which values will be substituted when the query is executed

```
$tpl1 = "select slot from meetings where
      name=:name and email=:email";
$tpl2 = "select slot from meetings where name=?";
```


- The PDO method `prepare()` turns an SQL template into prepared statement (by asking the DBMS to do so)

  - on success, a PDOStatement object is returned
  - on failure, `FALSE` or an error will be returned

```
$stmt1 = $pdo->prepare($tpl1);
$stmt2 = $pdo->prepare("select * from fruit where col=?");
```

Prepared Statements: Binding

- We can bind the parameters of a PDOStatement object to a value using the bindValue() method
  - Named parameters are bound by name
  - Question mark parameters are bound by position (starting from 1!)
  - the datatype of the value can optionally be declared (to match that of the corresponding database field)
  - the value is bound to the parameter at the time bindValue() is executed

```
$stmt1->bindValue(':name','Ben',PDO::PARAM_STR);
$email = 'bj1@liv.ac.uk';
$stmt1->bindValue(':email',$email);
$stmt2->bindValue(1,20,PDO::PARAM_INT);
```

Prepared Statements: Binding

- We can bind the parameters of a PDOStatement object to a variable using the `bindParam()` method
  - Named parameters are bound by name
  - Question mark parameters are bound by position (starting from 1!)
  - the datatype of the value can optionally be declared (to match that of the corresponding database field)
  - the variable is bound to the parameter as a reference
  - a value is only substituted when the statement is executed

```
$name  = 'Ben';
$stmt1->bindParam(':name',$name,PDO::PARAM_STR);
$stmt1->bindParam(':email',$email);
$email = 'bj1@liv.ac.uk';
$slot  = 20;
$stmt2->bindParam(1,$slot,PDO::PARAM_INT);
```

- It is possible to mix `bindParam()` and `bindValue()`

Prepared Statements: Execution

- Prepared statements are executed using `execute()` method
- Parameters must
  - previously have been bound using `bindValue()` or `bindParam()`, or
  - be given as an array of values to `execute`
    - ⇝ take precedence over previous bindings
    - ⇝ are bound using `bindValue()`
- `execute()` returns `TRUE` on success or `FALSE` on failure
- On success, the `PDOStatement` object stores a result set (if appropriate)

```
$stmt1->execute();
$stmt1->execute(array(':name' => 'Eve', ':email' => $email));
$stmt2->execute(array(10));
```

Transactions

- There are often situations where a single 'unit of work' requires a sequence of database operations
  ⇝ e.g., bookings, transfers
- By default, PDO runs in "auto-commit" mode
  ⇝ successfully executed SQL statements cannot be 'undone'
- To execute a sequence of SQL statements whose changes are
  - only committed at the end once all have been successful or
  - rolled back otherwise,

  PDO provides the methods
  - `beginTransaction()`
  - `commit()`
  - `rollBack()`

Transactions

To support transactions, PDO provides the methods

`beginTransaction()`
- turns off auto-commit mode; changes to the database are not committed until `commit()` is called
- returns `TRUE` on success or `FALSE` on failure
- throws an exception if another transaction is already active

`commit()`
- changes to the database are made permanent; auto-commit mode is turned on
- returns `TRUE` on success or `FALSE` on failure
- throws an exception if no transaction is active

`rollBack()`
- discard changes to the database; auto-commit mode is restored
- returns `TRUE` on success or `FALSE` on failure
- throws an exception if no transaction is active

Transactions: Example

```
$pdo = new PDO('mysql:host=...;dbname=...','...','...',
       array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
             PDO::ATTR_EMULATE_PREPARES => false));
$pdo->beginTransaction();
try{
  $userId = 1;      $paymentAmount = 10.50;

  //Query 1: Attempt to insert a payment record
  $sql = "INSERT INTO payments (user_id, amount) VALUES (?, ?)";
  $stmt = $pdo->prepare($sql);
  $stmt->execute(array($userId,$paymentAmount));

  //Query 2: Attempt to update the user's account
  $sql = "UPDATE accounts SET balance = balance + ? WHERE id = ?";
  $stmt = $pdo->prepare($sql);
  $stmt->execute(array($paymentAmount,$userId));

  // Commit the transaction
  $pdo->commit();
} catch(Exception $e){
  echo $e->getMessage();
  //Rollback the transaction
  $pdo->rollBack();
}
```

Based on http://thisinterestsme.com/php-pdo-transaction-example/

Revision

Read

- Language Reference: Classes and Objects
  http://php.net/manual/en/language.oop5.php
- The PDO Class
  http://php.net/manual/en/class.pdo.php

of M. Achour, F. Betz, A. Dovgal, et al. PHP Manual. The PHP Group, 2017. http://uk.php.net/manual/en [accessed 07 Dec 2017]

COMP284 Scripting Languages

Lecture 14: JavaScript (Part 1)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

JavaScript: Motivation

- PHP and Perl both allow us to create dynamic web pages
- In web applications, PHP and Perl code is executed on the web server (server-side scripting)

  - allows to use a website template that is instantiated using data stored in a database
  - 'business logic' is hidden from the user:
    the code of an application is not visible to the user/client;
    the user/client only has access to the HTML produced by the code

  - not ideal for interactive web applications:
    too slow to react and too much data needs to be transferred

  - operations that refer to the location of the user/client are difficult, for example, displaying the local time

```
echo date('H:i l, j F Y');
```

displays the local time on the server not the local time for the user

JavaScript

- JavaScript is a language for client-side scripting
  - script code is embedded in a web page (as for PHP), but delivered to the client as part of the web page and executed by the user's web browser
    $\rightsquigarrow$ code is visible to the user/client
  - allows for better interactivity as reaction time is improved and data exchange with the server can be minimised
  - a web browser may not support JavaScript or the user may have disallowed the execution of JavaScript code
  - different JavaScript engines may lead to different results, in particular, results not anticipated by the developer of JavaScript code
  - performance relies on the efficiency of the JavaScript engine and the client's computing power (not the server's)
  - operations that refer to the location of the client are easy:

```
document.write("Local time: " + (new Date).toString());
```

JavaScript: History

- originally developed by Brendan Eich at Netscape under the name Mocha
- first shipped together with Netscape browser in September 1995 under the name LiveScript
- obtained its current name in December 1995 under a deal between Netscape and Sun Microsystems, the company behind Java, in December 1995
- does not have a particularly close relationship to Java, it mixes aspects of Java with aspects of PHP and Perl and its own peculiarities
- is a dialect of ECMAScript, a scripting language standardised in the ECMA-262 specification and ISO/IEC 16262 standard since June 1997
- other dialects include Microsoft's JScript and TypeScript and Adobe's ActionScript

Websites and Programming Languages

Website	Client-Side	Server-Side	Database
Google	JavaScript	C, C++, Go, Java, Python, ~~PHP~~	BigTable, MariaDB
Facebook	JavaScript	Hack, PHP, Python, C++, Java, . . .	MariaDB, MySQL, HBase Cassandra
YouTube	~~Flash~~, JavaScript	C, C++, Python, Java, Go	BigTable, MariaDB
Yahoo	JavaScript	PHP	MySQL, PostgreSQL
Amazon	JavaScript	Java, C++, Perl	Oracle Database
Wikipedia	JavaScript	PHP, Hack	MySQL, MariaDB
Twitter	JavaScript	C++, Java, Scala	MySQL
Bing	JavaScript	ASP.NET	MS SQL Server

Wikipedia Contributors: Programming languages used in most popular websites. Wikipedia, The Free Encyclopedia, 20 October 2017, at 11:28. `http://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites` [accessed 23 October 2017]

JavaScript: Hello World!

```
<html><head><title>Hello World</title></head>
<body>
<p>Our first JavaScript script</p>
<script type="text/javascript">
  document.writeln("<p><b>Hello␣World!</b></p>")
</script>
<noscript>
JavaScript not supported or disabled
</noscript>
</body></html>
```


- JavaScript code is enclosed between `<script>` and `</script>`
- Alternative HTML markup that is to be used in case JavaScript is not enabled or supported by the web browser, can be specified between `<noscript>` and `</noscript>`
- File must be stored in a directory accessible by the web server, for example `$HOME/public_html`, and be readable by the web server
- No particular file name extension is required

JavaScript scripts

- JavaScript scripts are embedded into HTML documents and are enclosed between `<script` and `</script>` tags
- A JavaScript script consists of one or more statements and comments
  ⇝ there is no need for a main function (or classes)
  - Statements do not have to end in a semi-colon but they can
    ⇝ stick to one convention in your code
  - Whitespace before and in-between statements is irrelevant
    (This does not mean it is irrelevant to someone reading your code)
  - One-line comments start with // and run to the end of the line
  - Multi-line comments are enclosed in /* and */
  - Comments should precede the code they are referring to

Types

- JavaScript is a loosely typed language — like PHP and Perl
- JavaScript distinguished five main types:
  - `boolean` – booleans
  - `number` – integers and floating-point numbers
  - `string` – strings
  - `function` – functions
  - `object` – objects (including arrays)
- Integers, floating-point numbers, and strings do not differ significantly from the corresponding Perl scalars, including the pecularities of single-quoted versus double-quoted strings
- JavaScript distinguishes between these five types including between the three primitive types boolean, number and string

Variables

- JavaScript variable names do not start with a particular character
- A JavaScript variable name may consist of letters, digits, the $ symbol, and underscore, but cannot start with a digit
  - ⇝ you can still stick to the PHP and Perl 'convention' that (some) variable names start with a $ symbol
- JavaScript variable names are case sensitive

Variables

- Variables can be declared using one of the following statements:

```
var variable1, variable2, ...
var variable1 = value1, variable2 = value2, ...
```


  - The second statement also initialises the variables
  - Used inside a function definition, a declaration creates a local variable (only accessible within the function)

  - Used outside a function definition, a declaration creates a global variable

- A variable can be inialised without a declaration by assigning a value to it:

```
variable = value
```


  - Both inside and outside a function definition, initialising an undeclared variable creates a global variable

- Note: A declaration does not specify the type of a variable
  only assigning a value of a certain type gives a variable a type

Variables

- In JavaScript, the use of the value of a variable that is neither declared nor initialised will result in a reference error and script execution stops
- A declared but uninitialised variable has the default value `undefined` and has no specific type
- JavaScript automatically converts a value to the appropriate type as required by the operation applied to the value (type coercion)
- The value `undefined` is converted as follows:

Type	Default	Type	Default	Type	Default
`bool`	`false`	`string`	`'undefined'`	`number`	`NaN`

```
myVar1++                // reference error
var myVar2
myVar2++                // myVar2 has value NaN
var myVar3
myVar3 = myVar3 + '!' // myVar3 has value 'undefined!'
```

Assignments

- JavaScript uses the equality sign = for assignments

```
student_id = 200846369;
```


As in PHP and Perl, this is an assignment expression

- The value of an assignment expression is the value assigned

```
b = (a = 0) + 1;  // a has value 0, b has value 1
```


- JavaScript supports most of the standard binary assignment operators:

Binary assignment	Equivalent assignment
var += *expr*	*var* = *var* + *expr*
var -= *expr*	*var* = *var* - *expr*
var *= *expr*	*var* = *var* * *expr*
var /= *expr*	*var* = *var* / *expr*
var %= *expr*	*var* = *var* % *expr*

Note: **= is not supported

Constants

- Some JavaScript dialects allow the definition of constants using

```
const variable1 = value1, variable2 = value2, ...
```


  - defines one or more constants
  - constants follow the same scope rules as variables

- However, this construct is not supported by Internet Explorer 6–10 and does not have the desired effect in Safari before version 5.1.7 nor Opera before version 12

Values, Variables and Types

- `string typeof value`

  returns a string representation of the type of *value*

Boolean	"boolean"	Number	"number"
String	"string"	Object	"object"
undefined	"undefined"	null	"object"
NaN	"number"	Infinity	"number"

Future versions of JavaScript may have an option to change `typeof null` to `"null"` (as in PHP)

```
document.writeln("Type of 23.0:   " + typeof(23.0) + "<br />"
document.writeln("Type of \"23\":  " + typeof("23") +"<br />"
var a
document.writeln("Type of a:     " + typeof(a) + "<br />"
```

```
Type of 23.0: number<br />
Type of "23": string<br />
Type of a:    undefined<br />
```

Typecasting

JavaScript provides several ways to explicitly type cast a value

- Apply an identity function of the target type to the value

`"12" * 1`	⇝	`12`	`!!"1"`	⇝	`true`
`12 + ""`	⇝	`"12"`	`!!"0"`	⇝	`true`
`false + ""`	⇝	`"false"`	`!!""`	⇝	`false`
`[12,[3,4]] + ""`	⇝	`"12,3,4"`	`!!1`	⇝	`true`
			`[12,13] * 1`	⇝	`NaN`
			`[12] * 1`	⇝	`12`

Typecasting

JavaScript provides several ways to explicitly type cast a value

- Wrap a value of a primitive type into an object
  ⇝ JavaScript has objects Number, String, and Boolean with unary constructors/wrappers for values of primitive types (JavaScript does not have classes but prototypical objects)

`Number("12")`	⇝	`12`	`Boolean("0")`	⇝	`true`
`String(12)`	⇝	`"12"`	`Boolean(1)`	⇝	`true`
`String(false)`	⇝	`"false"`	`Number(true)`	⇝	`1`

- Use parser functions `parseInt` or `parseFloat`

`parseInt("12")`	⇝	`12`	`parseFloat("2.5")`	⇝	`2.5`
`parseInt("2.5")`	⇝	`2`	`parseFloat("2.5e1")`	⇝	`25`
`parseInt("E52")`	⇝	`NaN`	`parseFloat("E5.2")`	⇝	`NaN`
`parseInt("␣42")`	⇝	`42`	`parseFloat("␣4.2")`	⇝	`4.2`
`parseInt("2014Mar")`	⇝	`2014`	`parseFloat("4.2end")`	⇝	`4.2`

Comparison operators

JavaScript distinguishes between (loose) equality ==
and strict equality === in the same way as PHP:

expr1 == *expr2*	Equal	TRUE iff *expr1* is equal to *expr2* after type coercion
expr1 != *expr2*	Not equal	TRUE iff *expr1* is not equal to *expr2* after type coercion

- When comparing a number and a string, the string is converted to a number
- When comparing with a boolean, the boolean is converted to 1 if `true` and to 0 if `false`
- If an object is compared with a number or string, JavaScript uses the `valueOf` and `toString` methods of the objects to produce a primitive value for the object
- If two objects are compared, then the equality test is true only if both refer to the same object

Comparison operators

JavaScript distinguishes between (loose) equality ==
and strict equality === in the same way as PHP:

expr1 === *expr2*	Strictly equal	TRUE iff *expr1* is equal to *expr2*, and they are of the same type
expr1 !== *expr2*	Strictly not equal	TRUE iff *expr1* is not equal to *expr2*, or they are not of the same type

```
"123" == 123              ⇝  true
"123" != 123              ⇝  false
"1.23e2" == 123           ⇝  true
"1.23e2" == "12.3e1"      ⇝  false
5 == true                 ⇝  false
```

```
"123" === 123             ⇝  false
"123" !== 123             ⇝  true
1.23e2 === 123            ⇝  false
"1.23e2" === "12.3e1"     ⇝  false
5 === true                ⇝  false
```

Comparison operators

JavaScript's comparison operators also applies type coercion to their operands and do so following the same rules as equality ==:

expr1 < *expr2*	Less than	true iff *expr1* is strictly less than *expr2* after type coercion
expr1 > *expr2*	Greater than	true iff *expr1* is strictly greater than *expr2* after type coercion
expr1 <= *expr2*	Less than or equal to	true iff *expr1* is less than or equal to *expr2* after type coercion
expr1 >= *expr2*	Greater than or equal to	true iff *expr1* is greater than or equal to *expr2* after type coercion

`'35.5' > 35`	⇝	`true`	`'35.5' >= 35`	⇝	`true`
`'ABD' > 'ABC'`	⇝	`true`	`'ABD' >= 'ABC'`	⇝	`true`
`'1.23e2' > '12.3e1'`	⇝	`false`	`'1.23e2' >= '12.3e1'`	⇝	`false`
`"F1" < "G0"`	⇝	`true`	`"F1" <= "G0"`	⇝	`true`
`true > false`	⇝	`true`	`true >= false`	⇝	`true`
`5 > true`	⇝	`true`	`5 >= true`	⇝	`true`

Equality

Why do we care whether `5 == true` is true or false?

⇝ it influences how our scripts behave

⇝ it influences whether more complex objects are equal or not

PHP:

```
if (5) print("5 is true");
else   print("5 is not true");
print(" and ");
if (5 == true) print("5 is equal to true");
          else print("5 is not equal to true");
```


Output: `5 is true and 5 is equal to true`

JavaScript:

```
if (5) document.writeln("5 is true");
  else document.writeln("5 is not true")
document.writeln(" and ")
if (5 == true) document.writeln("5 is equal to true")
          else document.writeln("5 is not equal to true")
```

Output: `5 is true and 5 is not equal to true`

Equality

Why do we care whether `5 == true` is true or false?

⇝ it influences how our scripts behave

⇝ it influences whether more complex objects are equal or not

PHP:

```
$array3 = array("1.23e2",5);
$array4 = array("12.3e1",true);
if (($array3[1] == $array4[1]) && ($array3[2] == $array4[2]))
    print("The two arrays are equal");
else print("The two arrays are not equal");
```


Output: `The two arrays are equal`

JavaScript:

```
$array3 = ["1.23e2",5]
$array4 = ["12.3e1",true]
if (($array3[1] == $array4[1]) && ($array3[2] == $array4[2]))
    document.writeln("The two arrays are equal")
else document.writeln("The two arrays are not equal")
```

Output: `The two arrays are not equal`

Equality

Note: The way in which more complex data structures are compared also differs between PHP and JavaScript

PHP:

```
$array3 = array("1.23e2",5);
$array4 = array("12.3e1",true);
if ($array3 == $array4)
     print("The two arrays are equal");
else print("The two arrays are not equal");
```


Output: The two arrays are equal

JavaScript:

```
$array3 = ["1.23e2",5]
$array5 = ["1.23e2",5]
if ($array3 == $array5)
     document.writeln("The two arrays are equal")
else document.writeln("The two arrays are not equal")
```

Output: The two arrays are not equal

Revision

Read

- Chapter 14: Exploring JavaScript

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

COMP284 Scripting Languages
Lecture 15: JavaScript (Part 2)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Integers and Floating-point numbers

- The JavaScript datatype `number` covers both
  - integer numbers `0` `2012` `-40` `1263978`
  - floating-point numbers `1.25` `256.0` `-12e19` `2.4e-10`

- The Math object provides a wide range of mathematical functions

`Math.abs(number)`	absolute value
`Math.ceil(number)`	round fractions up
`Math.floor(number)`	round fractions down
`Math.round(number)`	round fractions
`Math.log(number)`	natural logarithm
`Math.random()`	random number between 0 and 1
`Math.sqrt(number)`	square root

- There are also some pre-defined number constants including

`Math.PI`	(case sensitive)	3.14159265358979323846
`NaN`	(case sensitive)	'not a number'
`Infinity`	(case sensitive)	'infinity'

Numbers: NaN and Infinity

- The constants `NaN` and `Infinity` are used as return values for applications of mathematical functions that do not return a number
  - `Math.log(0)` returns `-Infinity` (negative 'infinity')
  - `Math.sqrt(-1)` returns `NaN` ('not a number')
  - `1/0` returns `Infinity` (positive 'infinity')
  - `0/0` returns `NaN` ('not a number')
- Equality and comparison operators produce the following results for `NaN` and `Infinity`:

`NaN == NaN`	⇝ `false`	`NaN === NaN`	⇝ `false`
`Infinity == Infinity`	⇝ `true`	`Infinity === Infinity`	⇝ `true`
`NaN == 1`	⇝ `false`	`Infinity == 1`	⇝ `false`
`NaN < NaN`	⇝ `false`	`Infinity < Infinity`	⇝ `false`
`1 < Infinity`	⇝ `true`	`1 < NaN`	⇝ `false`
`Infinity < 1`	⇝ `false`	`NaN < 1`	⇝ `false`
`NaN < Infinity`	⇝ `false`	`Infinity < NaN`	⇝ `false`

Integers and Floating-point numbers: NaN and Infinity

- JavaScript provides two functions to test whether a value is or is not `NaN`, `Infinity` or `-Infinity`:
  - `bool isNaN(value)`
    returns TRUE iff *value* is `NaN`
  - `bool isFinite(value)`
    returns TRUE iff *value* is neither `NaN` nor `Infinity`/`-Infinity`

  There is no `isInfinite` function
- In conversion to a boolean value,
  - `NaN` converts to `false`
  - `Infinity` converts to `true`
- In conversion to a string,
  - `NaN` converts to `'NaN'`
  - `Infinity` converts to `'Infinity'`

Booleans

- JavaScript has a boolean datatype
  with constants `true` and `false` (case sensitive)

- JavaScript offers the same short-circuit boolean operators
  as Java, Perl and PHP:

  `&&` (conjunction) `||` (disjunction) `!` (negation)

  But `and` and `or` cannot be used instead of `&&` and `||` respectively

- The truth tables for these operators are the same as for Perl and PHP,
  taking into account that the conversion of non-boolean values to
  boolean values differs

- Remember that `&&` and `||` are not commutative, that is,
  `(A && B)` is not the same as `(B && A)`
  `(A || B)` is not the same as `(B || A)`

Type conversion to boolean

When converting to boolean, the following values are considered `false`:

- the boolean `false` itself
- the number 0 (zero)
- the empty string, but not the string '0'
- `undefined`
- `null`
- `NaN`

Every other value is converted to `true` including

- `Infinity`
- `'0'`
- functions
- objects, in particular, arrays with zero elements

Strings

- JavaScript supports both single-quoted and double-quoted strings
- JavaScript uses + for string concatenation
- Within double-quoted strings JavaScript supports the following escape characters

\b (backspace)	\f (form feed)	\n (newline)
\r (carriage return)	\t (tab)	\\ (backslash)
\' (single quote)	\" (double quote)	

- JavaScript does not support variable interpolation
- JavaScript also does not support heredocs, but multi-line strings are possible

```
document.writeln("Your\
        name is " + name + "and\
        you are studying " + degree + "\
        at " + university);
```

Arrays

- An array is created by assigning an array value to a variable

```
var arrayVar = []
var arrayVar = [elem0, elem1, ... ]
```


- JavaScript uses

```
arrayVar[index]
```


to denote the element stored at position *index* in *arrayVar*
The first array element has index 0

- Arrays have no fixed length and it is always possible to add more elements to an array

- Accessing an element of an array that has not been assigned a value yet returns `undefined`
- For an array *arrayVar*, *arrayVar*.`length` returns the maximal index *index* such that *arrayVar*[*index*] has been assigned a value (including the value `undefined`) plus one

Arrays

- It is possible to assign a value to *arrayVar*.`length`
  - if the assigned value is greater than the previous value of *arrayVar*.`length`, then the array is 'extended' by additional `undefined` elements
  - if the assigned value is smaller than the previous value of *arrayVar*.`length`, then array elements with greater or equal index will be deleted

- Assigning an array to a new variable creates a reference to the original array

  ⇝ changes to the new variable affect the original array

- Arrays are also passed to functions by reference
- The `slice` function can be used to create a proper copy of an array:

  `object` *arrayVar*.`slice`(*start*, *end*)

  returns a copy of those elements of array *variable* that have indices between *start* and *end*

Arrays: Example

```
var array1 = ['hello', [1, 2], function() {return 5}, 43]
document.writeln("1:␣array1.length␣=␣"+array1.length+"<br>")
1: array1.length = 4<br>
document.writeln("2:␣array1[3]␣="+array1[3]+"<br>")
2: array1[3] = 43<br>
array1[5] = 'world'
document.writeln("3:␣array1.length␣=␣"+array1.length+"<br>")
3: array1.length = 6<br>
document.writeln("4:␣array1[4]␣="+array1[4]+"<br>")
4: array1[4] = undefined<br>
document.writeln("5:␣array1[5]␣="+array1[5]+"<br>")
5: array1[5] = world<br>
array1.length = 4
document.writeln("6:␣array1[5]␣="+array1[5]+"<br>")
6: array1[5] = undefined<br>
var array2 = array1
array2[3] = 7
document.writeln("7:␣array1[3]␣="+array1[3]+"<br>")
7: array1[3] = 7<br>
```

forEach-method

- The recommended way to iterate over all elements of an array is a for-loop

```
for (index = 0; index < arrayVar.length; index++) {
  ... arrayVar[index] ...
}
```


- An alternative is the use of the forEach method:

```
var callback = function (elem, index, arrayArg) {
    statements
}
array.forEach(callback);
```

  - The forEach method takes a function as an argument
  - It iterates over all indices/elements of an array
  - It passes the current array element (*elem*), the current index (*index*) and a pointer to the array (*arrayArg*) to the function
  - Return values of that function are ignored, but the function may have side effecs

forEach-method: Example


```
var myArray = ['Michele Zito','Ullrich Hustadt'];

var rewriteNames = function (elem, index, arr) {
  arr[index] = elem.replace(/(\w+)\s(\w+)/, "$2, $1");
}

myArray.forEach(rewriteNames);

for (i=0; i<myArray.length; i++) {
  document.write('['+i+'] = '+myArray[i]+" ");
}
document.writeln("<br>");
```

```
[0] = Zito, Michele [1] = Hustadt, Ullrich <br>
```

Array operators

JavaScript has no stack or queue data structures,
but has stack and queue functions for arrays:

- `number` *`array`*`.push(`*`value1`*`, `*`value2`*`,...)`
  appends one or more elements at the end of an array;
  returns the number of elements in the resulting array

- `mixed` *`array`*`.pop()`
  extracts the last element from an array and returns it
- `mixed` *`array`*`.shift()`
  shift extracts the first element of an array and returns it

- `number` *`array`*`.unshift(`*`value1`*`, `*`value2`*`,...)`
  inserts one or more elements at the start of an array variable;
  returns the number of elements in the resulting array

Note: In contrast to PHP and Perl, *array* does not need to be a variable

Array operators: push, pop, shift, unshift

```
planets = ["earth"]
planets.unshift("mercury","venus")
planets.push("mars","jupiter","saturn");
document.writeln("planets\@1:␣"+planets.join("␣")+"␣<br>")
planets@1: mercury venus earth mars jupiter saturn <br>
last = planets.pop()
document.writeln("planets\@2:␣"+planets.join("␣")+"␣<br>")
planets@2: mercury venus earth mars jupiter <br>
first = planets.shift()
document.writeln("planets\@3:␣"+planets.join("␣")+"␣<br>")
planets@3: venus earth mars jupiter <br>
document.writeln("␣␣␣␣␣␣␣\@4:␣"+first+"␣"+last+"␣<br>")
       @4: mercury saturn <br>
home = ["mercury","venus","earth"].pop()
document.writeln("␣␣␣␣␣␣␣\@5:␣"+ home + "␣<br>")
       @5: earth <br>
number = ["earth"].push("mars");
document.writeln("␣␣␣␣␣␣␣\@6:␣"+ number + "␣<br>")
       @6: 2 <br>
```

Control structures

JavaScript control structures

- conditional statements
- switch statements
- while- and do while-loops
- for-loops
- break and continue

are identical to those of PHP except for conditional statements

Control structures: conditional statements

JavaScript conditional statements do not allow for `elsif`- or `elseif`-clauses, but conditional statements can be nested:

```
if (condition) {
    statements
} else if (condition) {
    statements
} else {
    statements
}
```

- The else-clause is optional but there can be at most one
- Curly brackets can be omitted if there is only a single statement in a clause

JavaScript also supports conditional expressions

```
condition ? if_true_expr : if_false_expr
```

Control structures: switch statement

Switch statements in JavaScript take the same form as in PHP:

```
switch (expr) {
  case expr1:
        statements
        break;
  case expr2:
        statements
        break;
  default:
        statements
        break;
}
```

- there can be arbitrarily many case-clauses
- the default-clause is optional but there can be at most one
- *expr* is evaluated only once and then compared to *expr1*, *expr2*, etc using (loose) equality ==
- once two expressions are found to be equal the corresponing clause is executed
- if none of *expr1*, *expr2*, etc are equal to *expr*, then the default-clause will be executed
- break 'breaks out' of the switch statement
- if a clause does not contain a break command, then execution moves to the next clause

Control structures: switch statement

Not every case-clause needs to have associated statements

Example:

```
switch (month) {
  case 1:   case 3:    case 5:   case 7:
  case 8:   case 10:   case 12:
     days = 31;
     break;
  case 4:   case 6:    case 9:   case 11:
     days = 30;
     break;
  case 2:
     days = 28;
     break;
  default:
     days = 0;
     break;
}
```

Control structures: while- and do while-loops

- JavaScript offers while-loops and do while-loops

```
while (condition) {
    statements
}

do {
    statements
} while (condition);
```


- As usual, curly brackets can be omitted if the loop onsists of only one statement

Example:

```
// Compute the factorial of a given number
factorial = 1;
do {
    factorial *= number--;
} while (number > 0);
```

Control structures: for-loops

- for-loops in JavaScript take the form

```
for (initialisation; test; increment) {
    statements
}
```


  Again, the curly brackets are not required if the body of the loop only consists of a single statement

- In JavaScript, as in PHP, *initialisation* and *increment* can consist of more than one statement, separated by commas instead of semicolons

  Example:


```
for (i = 3, j = 3; j >= 0; i++, j--)
   document.writeln(i + " - " + j + " - " + i*j)
   // Indentation has no 'meaning' in JavaScript,
   // the next line is not part of the loop
   document.writeln("After loop: " + i + " - " + j)
```

- Note: Variables introduced in a for-loop are still global even if declared using `var`

Control structures: break and continue

- The break command can also be used in while-, do while-, and for-loops and discontinues the execution of the loop

```
while (value < 100) {
  if (value == 0) break;
  value++
}
```

- The continue command stops the execution of the current iteration of a loop and moves the execution to the next iteration

```
for (x = -2; x <= 2; x++) {
   if (x == 0) continue;
   document.writeln("10 / " + x + " = " + (10/x));
}
```

```
10 / -2 = -5
10 / -1 = -10
10 / 1 = 10
10 / 2 = 5
```

Revision

Read

- Chapter 15: Expressions and Control Flow in JavaScript
- Chapter 16: JavaScript Functions, Objects, and Arrays

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

COMP284 Scripting Languages

Lecture 16: JavaScript (Part 3)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Functions

Function definitions can take several different forms in JavaScript including:

```
function identifier(param1,param2, ...) {
  statements }

var identifier = function(param1,param2, ...) {
  statements }
```


- Such function definitions are best placed in the head section of a HTML page or in a library that is then imported
- Function names are case-sensitive
- The function name must be followed by parentheses
- A function has zero, one, or more parameters that are variables
- Parameters are not typed
- *identifier*`.length` can be used inside the body of the function to determine the number of parameters

Functions

Function definitions can take several different forms in JavaScript including:

```
function identifier(param1,param2, ...) {
  statements }

var identifier = function(param1,param2, ...) {
  statements }
```


- The return statement

  ```
  return value
  ```

  can be used to terminate the execution of a function and to make *value* the return value of the function
- The return value does not have to be of a primitive type
- A function can contain more than one return statement
- Different return statements can return values of different types
  ⇝ there is no return type for a function

Calling a function

A function is called by using the function name followed by a list of arguments in parentheses


```
function identifier(param1, param2, ...) {
  ...
}
... identifier(arg1, arg2,...) ... // Function call
```


- The list of arguments can be shorter as well as longer as the list of parameters
- If it is shorter, then any parameter without corresponding argument will have value `undefined`


```
function sum(num1,num2) { return num1 + num2 }

sum1 = sum(5,4)       // sum1 = 9
sum2 = sum(5,4,3)     // sum2 = 9
sum3 = sum(5)         // sum3 = NaN
```

'Default values' for parameters

- JavaScript does not allow to specify default values for function parameters
- Instead a function has to check whether a parameter has the value `undefined` and take appropriate action


```
function sum(num1,num2) {
  if (num1 == undefined) num1 = 0
  if (num2 == undefined) num2 = 0
  return num1 + num2
}

sum3 = sum(5)          // sum3 = 5
sum4 = sum()           // sum4 = 0
```

Variable-length argument lists

- Every JavaScript function has a property called `arguments`
- The `arguments` property consists of an array of all the arguments passed to a function
- As for any JavaScript array, `arguments.length` can be used to determine the number of arguments


```
function sumAll() { // no minimum number of arguments
  if (arguments.length < 1) return null
  sum = 0
  for (var i=0; i<arguments.length; i++)
     sum = sum + arguments[i]
  return sum
}

sum0 = sumAll()          // sum0 = null
sum1 = sumAll(5)         // sum1 = 5
sum2 = sumAll(5,4)       // sum2 = 9
sum3 = sumAll(5,4,3)     // sum3 = 12
```

JavaScript functions and Static variables

- JavaScript does not have a `static` keyword to declare a variable to be static and preserve its value between different calls of a function
- The solution is to use a function property instead


```
function counter() {
  counter.count = counter.count || 0  // function property
  counter.count++
  return counter.count
}
document.writeln("1: static count = "+counter())
document.writeln("2: static count = "+counter())
document.writeln("3: global counter.count = "+counter.count)
```



```
1: static count = 1
2: static count = 2
3: global counter.count = 2
```

- As the example shows the function property is global/public
- Private static variables require more coding effort

JavaScript functions: Example

```
function bubble_sort(array) {
  if (!(array && array.constructor == Array))
    throw("Argument␣not␣an␣array")
  for (var i=0; i<array.length; i++) {
    for (var j=0; j<array.length-i; j++) {
      if (array[j+1] < array[j]) {
        // swap can change array because array is
        // passed by reference
        swap(array, j, j+1)
  } } }
  return array
}

function swap(array, i, j) {
  var tmp  = array[i]
  array[i] = array[j]
  array[j] = tmp
}
```

JavaScript functions: Example

```
function bubble_sort(array) { ... }
function swap(array, i, j)  { ... }

array = [2,4,3,9,6,8,5,1]
document.writeln("array  before sorting         "+
                 array.join(", ")+" <br>")
```

```
array  before sorting          2, 4, 3, 9, 6, 8, 5, 1 <br>
```

```
sorted = bubble_sort(array.slice(0)) // slice creates copy
document.writeln("array  after  sorting of copy   "+
                 array.join(", ")+"< br>")
```

```
array  after  sorting of copy  2, 4, 3, 9, 6, 8, 5, 1 <br>
```

```
sorted = bubble_sort(array)
document.writeln("array  after  sorting of itself "+
                 array.join(", ")+" <br>")
```

```
array  after  sorting of itself 1, 2, 3, 4, 5, 6, 8, 9 <br>
```

```
document.writeln("sorted array                    "+
                 sorted.join(", ")+" <br>")
```

```
sorted array                   1, 2, 3, 4, 5, 6, 8, 9 <br>
```

Nested function definitions

- Function definitions can be nested in JavaScript
- Inner functions have access to the variables of outer functions
- By default, inner functions can not be invoked from outside the function they are defined in


```
function bubble_sort(array) {
  function swap(i, j) {
    // swap can change array because array is
    // a local variable of the outer function bubble_sort
    var tmp = array[i]; array[i] = array[j]; array[j] = tmp;
  }
  if (!(array && array.constructor == Array))
     throw("Argument␣not␣an␣array")
  for (var i=0; i<array.length; i++) {
    for (var j=0; j<array.length-i; j++) {
      if (array[j+1] < array[j]) swap(j, j+1)
  } }
  return array }
```

JavaScript libraries

- Collections of JavaScript functions (and other code), libraries, can be stored in one or more files and then be reused
- By convention, files containing a JavaScript library are given the file name extension .js
- <script>-tags are not allowed to occur in the file
- A JavaScript library is imported using

```
<script type="text/javascript" src="url"></script>
```

where url is the (relative or absolute) URL for library

```
<script type="text/javascript"
 src="http://cgi.csc.liv.ac.uk/~ullrich/jsLib.js"></script>
```

- One such import statement is required for each library
- Import statements are typically placed in the head section of a page or at the end of the body section
- Web browers typically cache libraries

JavaScript libraries: Example

~ullrich/public_html/sort.js

```
function bubble_sort(array) {
  ... swap(array, i, j+1) ...
  return array
}

function swap(array, i, j) { ... }
```

example.html

```
<html><head><title>Sorting example</title>
<script type="text/javascript"
 src="http://cgi.csc.liv.ac.uk/~ullrich/sort.js">
</script></head>
<body>
<script type="text/javascript">
array  = [2,4,3,9,6,8,5,1];
sorted = bubble_sort(array.slice(0))
</script>
</body></html>
```

Object Literals

- JavaScript is an object-oriented language, but one without classes
- Instead of defining a class,
  we can simply state an object literal

  `{ property1: value1, property2: value2, ... }`

  where *property1*, *property2*, ... are variable names
  and *value1*, *value2*, ... are values (expressions)


```
var person1 = {
  age:       (30 + 2),
  gender:    'male',
  name:      { first : 'Bob', last : 'Smith' },
  interests: ['music', 'skiing'],
  hello: function() { return 'Hi! I\'m ' + this.name.first + '.' }
};
```

```
person1.age              --> 32        // dot notation
person1['gender']        --> 'male'    // bracket notation
person1.name.first       --> 'Bob'
person1['name']['last']  --> 'Smith'
```

Object Literals

```
var person1 = {
  ...
  name:   { first : 'Bob', last : 'Smith' },
  hello: function() { return 'Hi! I\'m ' + this.name.first + '.' }
};
person1.hello()     --> "Hi!␣I'm␣Bob."
```


- Every part of a JavaScript program is executed in a particular execution context

- Every execution context offers a keyword `this` as a way of referring to itself
- In `person1.hello()` the execution context of `hello()` is `person1`
  $\leadsto$ `this.name.first` is `person1.name.first`

Object Literals

```
var person1 = {
  name:  { first : 'Bob', last : 'Smith' },
  greet: function() { return 'Hi! I\'m ' + name.first + '.' },
  full1: this.name.first + " " + this.name.last,
  full2:      name.first + " " +      name.last
};
```

```
person1.greet()     --> "Hi!␣I'm␣undefined."
person1.full1       --> "undefined␣undefined"
person1.full2       --> "undefined␣undefined"
```

- In `person1.greet()` the execution context of `greet()` is `person1`
  - ⤳ but `name.first` does not refer to `person1.name.first`
- In the (construction of the) object literal itself, `this` does not refer to `person1` but its execution context (the window object)
  - ⤳ none of `name.first`, `name.last`, `this.name.first`, and `this.name.last` refers to properties of this object literal

Objects Constructors

- JavaScript is an object-oriented language, but one without classes
- Instead of defining a class,
  we can define a function that acts as object constructor
  - variables declared inside the function will be instance variables of the object
    $\rightsquigarrow$ each object will have its own copy of these variables
  - it is possible to make such variables private or public
  - inner functions will be methods of the object
  - it is possible to make such functions/methods private or public
  - private variables/methods can only be accessed inside the function
  - public variables/methods can be accessed outside the function
- Whenever an object constructor is called,
  prefixed with the keyword `new`, then
  - a new object is created
  - the function is executed with the keyword `this` bound to that object

Objects: Definition and use


```
function SomeObj() {
  instVar2       = 'B'        // private variable
  var instVar3   = 'C'        // private variable

  this.instVar1 = 'A'         // public variable

  this.method1 = function() {  // public method
    // use of a public variable, e.g. 'instVar1', must be preceded by 'this'
    return 'm1[' + this.instVar1 + ']' + method3() }

  this.method2 = function() {   // public method
    // calls a public method, e.g. 'method1', must be preceded by 'this'
    return 'm2[' + this.method1() + ']' }

  method3 = function() {        // private method
    return ' m3[' + instVar2 + ']' + method4()     }

  var method4 = function() {   // private method
    return ' m4[' + instVar3 + ']' }
}
obj = new SomeObj()                          // creates a new object
```

```
obj.instVar1     --> "A"
obj.instVar2     --> undefined
obj.instVar3     --> undefined
obj.method1()    --> "m1[A] m3[B] m4[C]"
obj.method2()    --> "m2[m1[A] m3[B] m4[C]]"
obj.method3()    --> error
obj.method4()    --> error
```

Objects: Definition and use

```
function SomeObj() {
  this.instVar1 = 'A'          // public variable
  instVar2 = 'B'               // private variable
  var instVar3 = 'C'           // private variable

  this.method1 = function() { ... }  // public method
  this.method2 = function() { ... }  // public method

  method3 = function() { ... }       // private method
  var method4 = function() { ... }   // private method
}
```


- Note that all of `instVar1` to `instVar3`, `method1` to `method4` are instance variables (properties, members) of `someObj`
- The only difference is that `instVar1` to `instVar3` store strings while `method1` to `method4` store functions

⇝ every object stores its own copy of the methods

Objects: Prototype property

- All functions have a `prototype` property that can hold shared object properties and methods
  ⇝ objects do not store their own copies of these properties and methods but only store references to a single copy


```
function SomeObj() {
  this.instVar1 = 'A'         // public variable
  instVar2      = 'B'         // private variable
  var instVar3  = 'C'         // private variable

  SomeObj.prototype.method1 = function() { ... }  // public
  SomeObj.prototype.method2 = function() { ... }  // public

  method3 = function() { ... }       // private method
  var method4 = function() { ... }   // private method
}
```


Note: `prototype` properties and methods are always public!

Objects: Prototype property

- The `prototype` property can be modified 'on-the-fly'
  - ⇝ all already existing objects gain new properties / methods
  - ⇝ manipulation of properties / methods associated with the `prototype` property needs to be done with care


```
function SomeObj() { ... }
obj1 = new SomeObj()
obj2 = new SomeObj()
document.writeln(obj1.instVar4)   // undefined
document.writeln(obj2.instVar4)   // undefined

SomeObj.prototype.instVar4 = 'A'
document.writeln(obj1.instVar4)   // 'A'
document.writeln(obj2.instVar4)   // 'A'

SomeObj.prototype.instVar4 = 'B'
document.writeln(obj1.instVar4)   // 'B'
document.writeln(obj2.instVar4)   // 'B'

obj1.instVar4 = 'C' // creates a new instance variable for obj1
SomeObj.prototype.instVar4 = 'D'
document.writeln(obj1.instVar4)   // 'C' !!
document.writeln(obj2.instVar4)   // 'D' !!
```

Objects: Prototype property

- The prototype property can be modified 'on-the-fly'
  - ⇝ all already existing objects gain new properties / methods
  - ⇝ manipulation of properties / methods associated with the prototype property needs to be done with care


```
function SomeObj() { ... }
obj1 = new SomeObj()
obj2 = new SomeObj()

SomeObj.prototype.instVar5 = 'E'

SomeObj.prototype.setInstVar5 = function(arg) {
  this.instVar5 = arg
}

obj1.setInstVar5('E')
obj2.setInstVar5('F')

document.writeln(obj1.instVar5) // 'E' !!
document.writeln(obj2.instVar5) // 'F' !!
```

'Class' variables and 'Class' methods

Function properties can be used to emulate Java's class variables (static variables shared among instances) and class methods

```
function Circle(radius) { this.r = radius }

// 'class variable' - property of the Circle constructor function
Circle.PI = 3.14159

// 'instance method'
Circle.prototype.area = function () {
   return Circle.PI * this.r * this.r; }

// 'class method'    - property of the Circle constructor function
Circle.max = function (cx,cy) {
   if (cx.r > cy.r) { return cx } else { return cy }
}

c1      = new Circle(1.0)     // create an instance of the Circle class
c1.r    = 2.2;                // set the r instance variable
c1_area = c1.area();          // invoke the area() instance method
x       = Math.exp(Circle.PI) // use the PI class variable in a computation
c2      = new Circle(1.2)     // create another Circle instance
bigger  = Circle.max(c1,c2)   // use the max() class method
```

Private static variables

In order to create private static variables shared between objects we can use a self-executing anonymous function


```
var Person = (function () {
  var population = 0      // private static `class' variable

  return function (value) {     // constructor
    population++
    var name     = value        // private instance variable
    this.setName = function (value) { name = value }
    this.getName = function () { return name }
    this.getPop  = function () { return population }
  }
}())

person1  = new Person('Peter')
person2  = new Person('James')
```



```
person1.getName()                              --> 'Peter'
person2.getName()                              --> 'James'
person1.name                                   --> undefined
Person.population || person1.population        --> undefined
person1.getPop()                               --> 2
```

```
person1.setName('David')
```

```
person1.getName()                              --> 'David'
```

Pre-defined objects: String

- JavaScript has a collection of pre-defined objects, including Array, String, Date
- A String object encapsulates values of the primitive datatype `string`
- Properties of a String object include
  - `length` the number of characters in the string
- Methods of a String object include
  - `charAt(index)`
    the character at position *index* (counting from 0)
  - `substring(start,end)`
    returns the part of a string between positions *start* (inclusive) and *end* (exclusive)
  - `toUpperCase()`
    returns a copy of a string with all letters in uppercase
  - `toLowerCase()`
    returns a copy of a string with all letters in lowercase

Pre-defined objects: String and RegExp

- JavaScript supports (Perl-like) regular expressions and the String objects have methods that use regular expressions:

  - `search(regexp)`
    matches *regexp* with a string and returns the start position of the first match if found, -1 if not
  - `match(regexp)`
    - without *g* modifier returns the matching groups for the first match or if no match is found returns null
    - with *g* modifier returns an array containing all the matches for the whole expression
  - `replace(regexp,replacement)`
    replaces matches for *regexp* with *replacement*,
    and returns the resulting string


```
name1  = 'Dave Shield'.replace(/(\w+)\s(\w+)/, "$2, $1")
regexp =  new RegExp("(\\w+)\\s(\\w+)")
name2  = 'Ken Chan'.replace(regexp, "$2, $1")
```

Pre-defined objects: Date

- The Date object can be used to access the (local) date and time
- The Date object supports various constructors

  - `new Date()` current date and time
  - `new Date(milliseconds)` set date to milliseconds since 1 Januar 1970
  - `new Date(dateString)` set date according to *dateString*

  - `new Date(year, month, day, hours, min, sec, msec)`
- Methods provided by Date include
  - `toString()`
    returns a string representation of the Date object

  - `getFullYear()`
    returns a four digit string representation of the (current) year
  - `parse()`
    parses a date string and returns the number of milliseconds since midnight of 1 January 1970

Revision

Read

- Chapter 16: JavaScript Functions, Objects, and Arrays
- Chapter 17: JavaScript and PHP Validation and Error Handling (Regular Expressions)

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

- http://coffeeonthekeyboard.com/private-variables-in-javascript-177/
- http://coffeeonthekeyboard.com/javascript-private-static-members-part-1-208/
- http://coffeeonthekeyboard.com/javascript-private-static-members-part-2-218/

COMP284 Scripting Languages

Lecture 17: JavaScript (Part 4)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Window and Document objects

JavaScript provides two objects that are essential to the creation of dynamic web pages and interactive web applications:

window object

- a JavaScript object that represents a browser window or tab
- automatically created whith every instance of a `<body>` or `<frameset>` tag
- allows properties of a window to be accessed and manipulated
  ⤳ JavaScript provides methods that allow window objects to be created and manipulated
  Example: window.open('http://www.csc.liv.ac.uk','Home')
- whenever an object method or property is referenced in a script without an object name and dot prefix it is assumed by JavaScript to be a member of the window object

  Example: We can write `alert()` instead of `window.alert()`

Window object

- A window object represents an open window in a browser.
- If a document contain frames, then there is
  - one window object, `window`, for the HTML document
  - and one additional window object for each frame, accessible via an array `window.frames`
- A window object has properties including

`document`	document object for the window
`history`	history object for the window
`location`	location object (current URL) for the window
`navigator`	navigator (web browser) object for the window
`opener`	reference to the window that created the window
`innerHeight`	inner height of a window's content area
`innerWidth`	inner width of a window's content area
`closed`	boolean value indicating whether the window is (still) open

Navigator object

Properties of a navigator object include

`navigator.appName`	the web brower's name
`navigator.appVersion`	the web brower's version

Example: Load different style sheets depending on browser


```
<html><head><title>Navigator example</title>
<script type="text/javascript">
if (navigator.appName == 'Netscape') {
  document.writeln('<link rel=stylesheet type="text/css" '+
                          href="Netscape.css">')
} else if (navigator.appName == 'Opera') {
  document.writeln('<link rel=stylesheet type="text/css" '+
                           href="Opera.css">')
} else {
  document.writeln('<link rel=stylesheet type="text/css" '+
                           href="Others.css">')
}
</script></head>
```

Window object

Methods provided by a window object include

- `open(url, name [, features])`
  - opens a new browser window/tab
  - returns a reference to a window object
  - *url* is the URL to access in the new window; can be the empty string
  - *name* is a name given to the window for later reference
  - *features* is a string that determines various window features

The standard sequence for the creation of a new windows is not:

```
// new instance of 'Window' class
var newWin = new Window(...)
newWin.document.write('<html>...</html>')
```

instead it is

```
// new window created by using 'open' with an existing one
var newWin = window.open(...)
newWin.document.write('<html>...</html>')
```

Window object

Methods provided by a window object include

- `close()`
  - closes a browser window/tab
- `focus()`
  - give focus to a window (bring the window to the front)
- `blur()`
  - removes focus from a window (moves the window behind others)
- `print()`
  - prints (sends to a printer) the contents of the current window

Window object: Example


```
<html><head><title>Window handling</title>
<script type="text/javascript">
function Help() {
  var OutputWindow = window.open('','Help','resizable=1')
  with (OutputWindow.document) {
    open()
    writeln("<!DOCTYPE html><html><head><title>Help</title>\
    </head><body>This might be a context-sensitive help\
    message, depending on the application and state of the
    page.</body></html>");
    close()
  }
}
</script></head><body>
<form name="ButtonForm" id="ButtonForm" action="">
<p>
  <input type="button" value="Click for Help"
         onclick="Help();">
</p>
</form></body></html>
```

Window object: Dialog boxes

- Often we only want to open a new window in order to
  - display a message
  - ask for confirmation of an action
  - request an input

- For these purposes, the window object in JavaScript provides pre-defined methods for the handling of dialog boxes (windows for simple dialogs):
  - `null alert(message_string)`
  - `bool confirm(message_string)`
  - `string prompt(message_string, default)`

Window object: Dialog boxes

- `null alert(message_string)`
  - creates a message box displaying *message_string*
  - the box contains an 'OK' button that the user will have to click (alternatively, the message box can be closed) for the execution of the remaining code to proceed

Example:

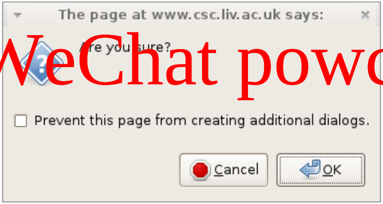
```
alert("Local time: " + (new Date).toString())
```

Window object: Dialog boxes

- `bool confirm(message_string)`
  - creates a message box displaying *message_string*
  - the box contains two buttons 'Cancel' and 'OK'
  - the function returns `true` if the user selects 'OK', `false` otherwise

Example:

```
var answer = confirm("Are you sure?")
```



Window object: Dialog boxes

- `string prompt(message_string, default)`
  - creates a dialog box displaying *message_string* and an input field
  - if a second argument *default* is given, *default* will be shown in the input field
  - the box contains two buttons 'Cancel' and 'OK'
  - if the user selects 'OK' then the current value entered in the input field is returned as a string, otherwise `null` is returned

Example:

```
var userName =
  prompt("What is your name?",
         "")
```

Window object: Dialog boxes

- `prompt()` always returns a string, even if the user enters a number
- To convert a string to number the following functions can be used:
  - `number parseInt(string [,base])`
    - converts *string* to an integer number wrt numeral system *base*
    - only converts up to the first invalid character in *string*
    - if the first non-whitespace character in *string* is not a digit, returns `NaN`
  - `number parseFloat(string)`
    - converts *string* to a floating-point number
    - only converts up to the first invalid character in *string*
    - if the first non-whitespace character in *string* is not a digit, returns `NaN`
  - `number Number(string)`
    - returns `NaN` if *string* contains an invalid character

Dialog boxes: Example


```
<html>
 <head><title>Interaction example</title></head>
<body>
<script type='text/javascript'>
do {
  string   = prompt("How many items do you want to buy?")
  quantity = parseInt(string)
} while (isNaN(quantity) || quantity <= 0)
do {
  string = prompt("How much does an item cost?")
  price  = parseFloat(string)
} while (isNaN(price) || price <= 0)
buy = confirm("You will have to pay "+
              (price*quantity).toFixed(2)+
              "\nDo you want to proceed?")
if (buy) alert("Purchase made")
</script>
</body></html>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/jsPrompt.html

User input validation

- A common use of JavaScript is the validation of user input in a HTML form before it is processed:
  - check that required fields have not been left empty
  - check that fields only contain allowed characters or comply to a certain grammar
  - check that values are within allowed bounds


```
<form method="post" action="process.php"
      onSubmit="return␣validate(this)">
  <label>User name:     <input type="text" name="user"></label>
  <label>Email address: <input type="text" name="email"></label>
  <input type="submit" name="submit">
</form>
<script>
function validate(form) {
  fail  = validateUser(form.user.value)
  fail += validateEmail(form.email.value)
  if (fail == "") return true
  else { alert(fail); return false } }
</script>
```

User input validation


```
function validateUser(field) {
  if (field == "") return "No username entered\n"
  else if (field.length < 5)
    return "Username too short\n"
  else if (/[^a-zA-Z0-9_-]/.test(field))
    return "Invalid character in username\n"
  else return ""
}

function validateEmail(field) {
  if (field == "") return "No email entered\n"
  else if (!((field.indexOf(".") > 0) &&
             (field.indexOf("@") > 0)) ||
           /[^a-zA-Z0-9.@_-]/.test(field))
    return "Invalid character in email\n"
  else return ""
}
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/jsValidate.html

Window and Document objects

JavaScript provides two objects that are essential to the creation of dynamic web pages and interactive web applications:

document object

- an object-oriented representation of a web page (HTML document) that is displayed in a window
- allows interaction with the Document Object Model (DOM) of a page

  Example: `document.writeln()` adds content to a web page

Document Object Model

A platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of HTML, XHTML and XML documents

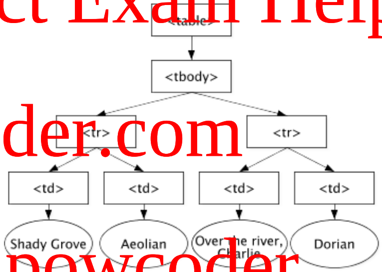
Document Object Model

Example:

The HTML table below

```
<table>
  <tbody>
    <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
    </tr>
    <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
    </tr>
  </tbody>
</table>
```

is parsed into the following DOM



Arnaud Le Hors, et al, editors: Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation 07 April 2004. World Wide Web Consortium, 2004.
https://www.w3.org/TR/DOM-Level-3-Core/ [accessed 9 January 2017]

Accessing HTML elements: Object methods

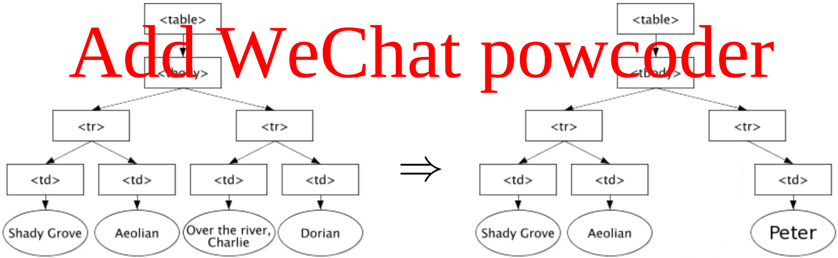
Example:

```
// access the tbody element from the table element
var myTbodyElement = myTableElement.firstChild;

// access its second tr element; the list of children starts at 0 (not 1).
var mySecondTrElement = myTbodyElement.childNodes[1];

// remove its first td element
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

// change the text content of the remaining td element
mySecondTrElement.firstChild.firstChild.data = "Peter";
```

Accessing HTML elements: Names (1)

Instead of using methods such as `firstChild` and `childNodes[n]`, it is possible to assign names to denote the children of a HTML element

Example:

```
<form name="form1" action="">
<label>Temperature in Fahrenheit:</label>
<input type="text" name="fahrenheit" size="10" value="0"><br>
<label>Temperature in Celsius:</label>
<input type="text" name="celsius" size="10" value="">
</form>
```


Then – `document.form1`
  Refers to the whole form

– `document.form1.celsius`
  Refers to the text field named `celsius` in `document.form1`

– `document.form1.celsius.value`
  Refers to the attribute `value` in the text field named `celsius` in `document.form1`

Accessing HTML elements: Names (2)

Accessing HTML elements by giving them names and using paths within the Document Object Model tree structure is still problematic

⇝ If that tree structure changes, then those paths no longer work

Example:

Changing the previous form to

```
<form name="form1" action="">
<div class="field" name="fdiv">
<label>Temperature in Fahrenheit:</label>
<input type="text" name ="fahrenheit" size=10 value="0" />
</div>
<div class="field" name="cdiv">
<label>Temperature in Celsius:</label>
<input type="text" name="celsius"     size="10" value="" />
</div>
</form>
```


means that `document.form1.celsius` no longer works as there is now a `div` element between form and text field, we would now need to use `document.form1.cdiv.celsius`

Accessing HTML elements: IDs

A more reliable way is to give each HTML element an ID (using the `id` attribute) and to use `getElementById` to retrieve a HTML element by its ID

Example:

```
<form id="form1" action="">
<label>Temperature in Fahrenheit:</label>
<input type="text" id="fahrenheit" size="10" value="0"><br>
<label>Temperature in Celsius:</label>
<input type="text" id="celsius"    size="10" value="">
</form>
```


Then

– `document.getElementById('celsius')`
  Refers to the HTML element with ID `celsius` `document`
– `document.getElementById('celsius').value`
  Refers to the attribute `value` in the HTML element with ID `celsius` in `document`

Manipulating HTML elements

It is not only possible to access HTML elements, but also possible to change them on-the-fly


```
<html><head><title>Manipulating HTML elements</title>
<style>
  td.RedBG { background: #f00; }
</style>
<script>
function changeBackground1(id) {
  document.getElementById(id).style.background = "#00f";
  document.getElementById(id).innerHTML = "blue";
}
function changeBackground2(id) {
  document.getElementById(id).cell.className = "RedBG";
  document.getElementById(id).cell.innerHTML = "red";
}
</script></head><body>
<table border="1"><tr>
 <td id="0" onclick="changeBackground1('0');">white</td>
 <td id="1" onclick="changeBackground2('1');">white</td>
</tr></table></body></html>
```

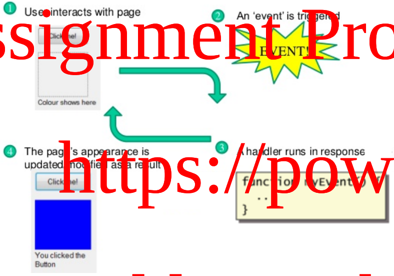
http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/jsBG.html

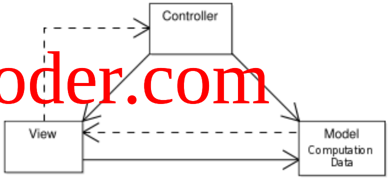
Event-driven JavaScript Programs

- The JavaScript programs we have seen so far were all executed sequentially
  - programs have a particular starting point
  - programs are executed step-by-step, involving control structures and function execution
  - programs reach a point at which their execution stops

Event-Driven JavaScript Programs

- Web applications are event-driven
  ⇝ they react to events such as mouse clicks and key strokes





nickywalters: What is Event Driven Programming?
SlideShare, 7 September 2014.
`https://tinyurl.com/ya58xbs9` [accessed 5/11/2017]

- With JavaScript,
  - we can define event handler functions for a wide variety of events
  - event handler functions can manipulate the `document` object (changing the web page in situ)

Event Handlers and HTML Elements

- HTML events are things, mostly user actions, that happen to HTML elements
- Event handlers are JavaScript functions that process events
- Event handlers must be associated with HTML elements for specific events
- This can be done via attributes


```
<input type="button" value="Help"  onclick="Help()">
```

- Alternatively, a JavaScript function can be used to add a handler to an HTML element


```
// All good browsers
window.addEventListener("load", Hello)
// MS IE browser
window.attachEvent("onload", Hello)
```

More than one event handler can be added this way to the same element for the same event

Event Handlers and HTML Elements

- As our scripts should work with as many browsers as possible, we need to detect which method works:

```
if (window.addEventListener) {
   window.addEventListener("load", Hello)
} else {
   window.attachEvent("onload", Hello)
}
```

- Event handlers can also be removed

```
if (window.removeEventListener) {
   window.removeEventListener("load", Hello)
} else {
   window.detachEvent("onload", Hello)
}
```

Events: Load

- An (on)load event occurs when an object has been loaded
- Typically, event handlers for onload events are associated with the window object or the body element of an HTML document


```
<html>
  <head>
    <title>Onload Example</title>
    <script type="text/javascript">
      function Hello()   { alert("Welcome to my page!")  }
    </script>
  </head>
  <body onload="Hello()">
    <p>Content of the web page</p>
  </body>
</html>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsOnload.html

Events: Focus / Change

- A focus event occurs when a form field receives input focus by tabbing with the keyboard or clicking with the mouse

  ⤳ onFocus attribute

- A change event occurs when a select, text, or textarea field loses focus and its value has been modified

  ⤳ onChange attribute

Example:

```
<form name="form1" method="post" action="process.php">
  <select name="select" required
          onChange="document.form1.submit();">
    <option value="">Select a name</option>
    <option value="200812345">Tom Beck</option>
    <option value="200867890">Jim Kent</option>
  </select>
</form>
```

Events: Focus / Change

- A focus event occurs when a form field receives input focus by tabbing with the keyboard or clicking with the mouse
  ⇝ onFocus attribute
- A change event occurs when a select, text, or textarea field loses focus and its value has been modified
  ⇝ onChange attribute


```
<form>
<label>Temperature in Fahrenheit:</label>
<input type="text" id="fahrenheit" size="10" value="0"
 onchange="document.getElementById('celsius').value =
 FahrenheitToCelsius(parseFloat(
  document.getElementById('fahrenheit').value)).toFixed(1);"
 ><br>
<label>Temperature in Celsius:</label>
<input type="text" id="celsius"
 size="10" value="" onfocus="blur();"></form>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsOnchange.html

Events: Blur / Click

- A blur event occurs when an HTML element loses focus
  ⤳ onBlur attribute
- A click event occurs when an object on a form is clicked
  ⤳ onClick attribute

Example:

```
<html><head><title>Onclick Example</title></head><body>
<form name="form1" action="">
  Enter a number here:
  <input type="text" size="12" id="number" value="3.1">
  <br><br>
  <input type="button" value="Double"
    onclick="document.getElementById('number').value =
    parseFloat(document.getElementById('number').value)
    * 2;">
</form></body></html>
```


http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/jsOnclick.html

Events: MouseOver / Select / Submit

- A keydown event occurs when the user presses a key
  ⇝ `onkeydown` attribute
- A mouseOver event occurs once each time the mouse pointer moves over an HTML element from outside that element
  ⇝ `onMouseOver` attribute
- A select event occurs when a user selects some of the text within a text or textarea field
  ⇝ `onSelect` attribute
- A submit event occurs when a user submits a form
  ⇝ `onSubmit` attribute

Events and DOM

- When an event occurs, an event object is created
  ⤳ an event object has attributes and methods
  ⤳ event objects can be created by your code independent of an event occurring
- In most browsers, the event object is passed to event handler functions as an argument
- In most versions of Microsoft Internet Explorer, the most recent event can only be accessed via `window.event`

```
<html><body onKeyDown="processKey(event)">
  <script>
    function processKey(e) {
      e = e || window.event
      document.getElementById("key").innerHTML =
       String.fromCharCode(e.keyCode)+' has been pressed'}
  </script>
  <!-- key code will appear in the paragraph below -->
  <p id="key"></p>
</body></html>
```

Revision

Read

- Chapter 17: JavaScript and PHP Validation and Error Handling
- Chapter 18: Using Ajax

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

- Mozilla Developer Network and individual contributors: Document Object Model (DOM), 18 March 2014. `https://developer.mozilla.org/en/docs/DOM` [accessed 18 March 2014].
- W3Schools: JavaScript and HTML DOM Reference, 18 March 2014. `http://www.w3schools.com/jsref/` [accessed 18 March 2014].