Aplikationssicherheit

Jan Fässler & Ege Kaba

5. Semester (HS 2013)

If you use this documentation for a exam, you should offer a beer to the authors!

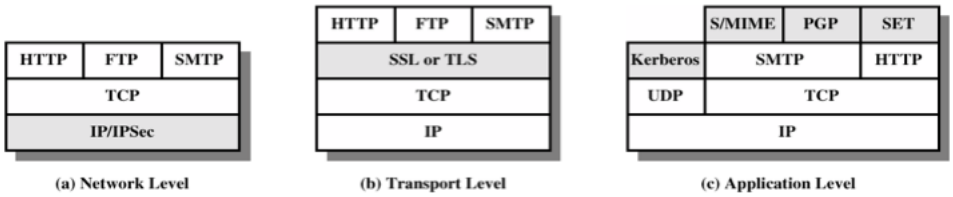
Inhaltsverzeichnis

1 SSL

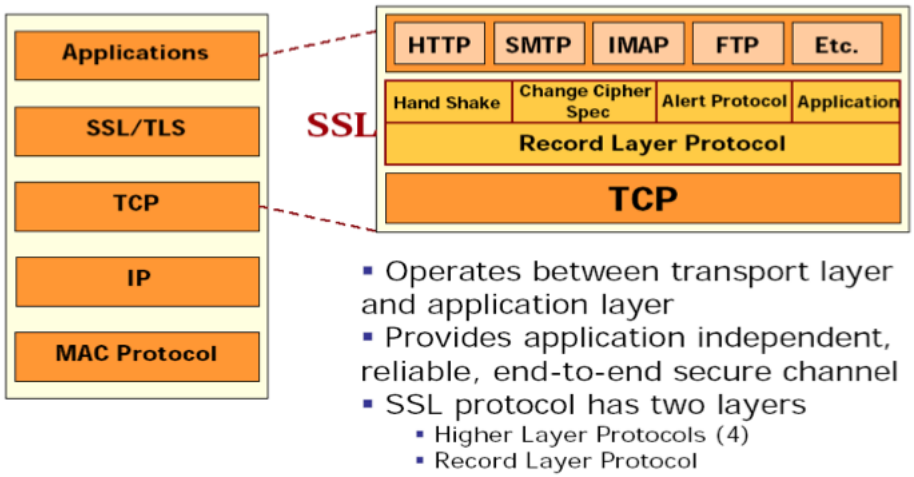
1.1 The secure channel

- A, B key-exchange protocol that establishes an **Authenticated, Secret Session Key** between A and B
- This **session key** is used together with a **symetric key** cryptographic function to protect the **integrity** and/or **secrecy** of the transmitted data.

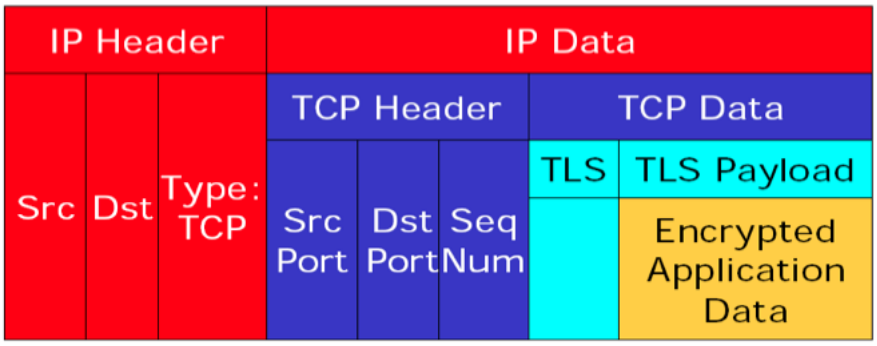
1.2 security in the TCP/IP stack



1.3 diffrent parts of the SSL protocoll



1.4 SSL(TLS) over TCP/IP



1.5 SSL characteristics

- Netscape Secure Socket Layer

- Layered between the application and TCP
- Server authenticated with public keys (X509)
- Can authenticate client (rare)
- Privacy enforced by encryption
- Integrity enforced by MACs
- Works with any TCP application

1.6 SSL protocols

SSL provides end to end security, based on a communication protocol. The different protocols implemented by SSL:

- Handshake
- Record (specifies how the data are encrypted)
- Change cipher specifications
- Alert services

1.7 Handshake protocol

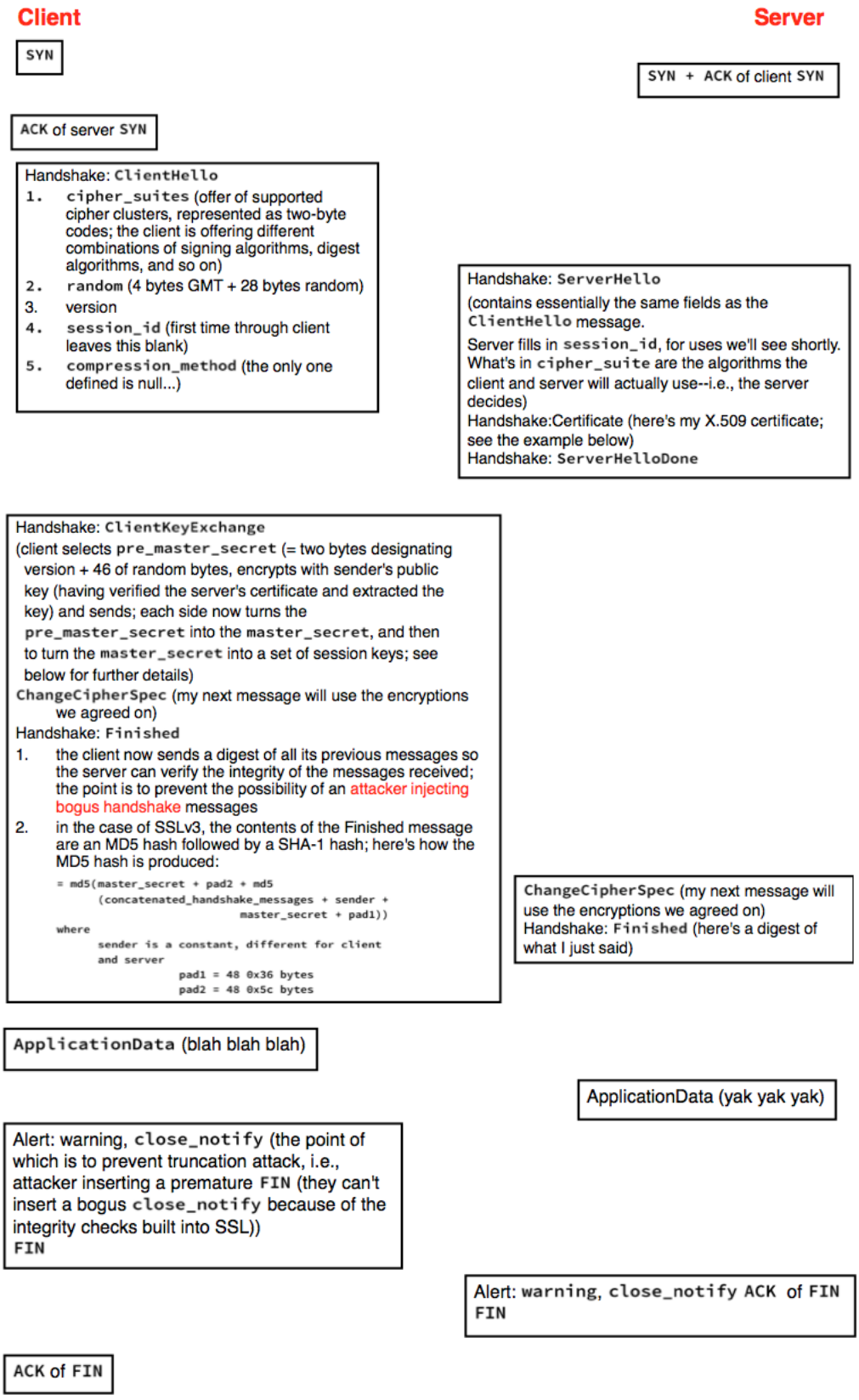
The ost complex and unsecure part of SSL are:

- Allows the server and client (**optional**) to athenticate each other
- Negotiate encryption, message authentication code algorithm and cryptographic keys
- Used before any application data are transmitted

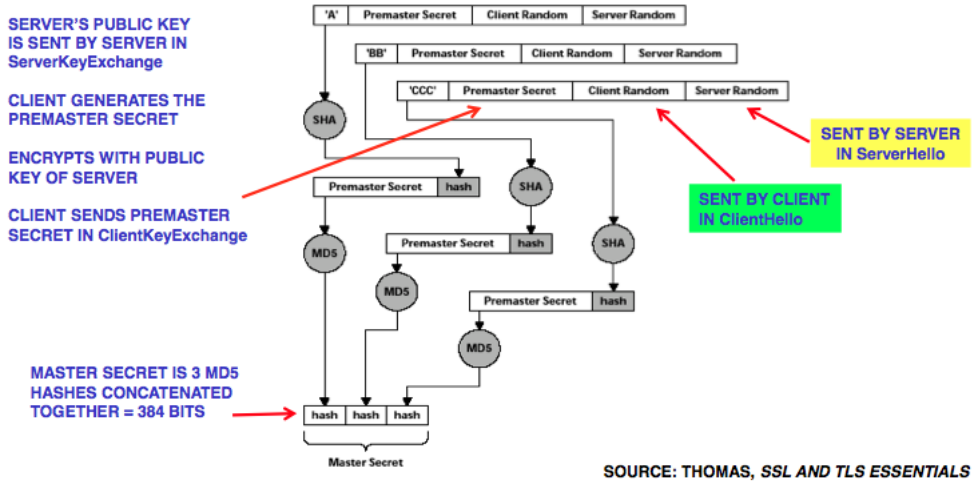
1.7.1 SSL protocol basic handshake features

1. The client initiates the connection to the server and tells the server which SSL cipher suites the client supports.
2. The server responds with the cipher suites that it supports.
3. The server sends the client a certificate that should authenticate it.
4. The server initiate a session key exchange algorithm, based in part on the information contained in the certificate it has just sent, and sends the necessary key exchange information to the client.
5. The client completes the key exchange algorithm and sends the necessary key exchange information to the server. Along the way, it verifies the certificate.
6. Based on the type of key exchange algorithm (that in turn is based on the type of key in the server's certificate), the client selects an appropriate cipher suite and tells the server which suite it wishes to use.
7. The server makes a final decision as to which cipher suite to use.

1.7.2 protocol gritty details

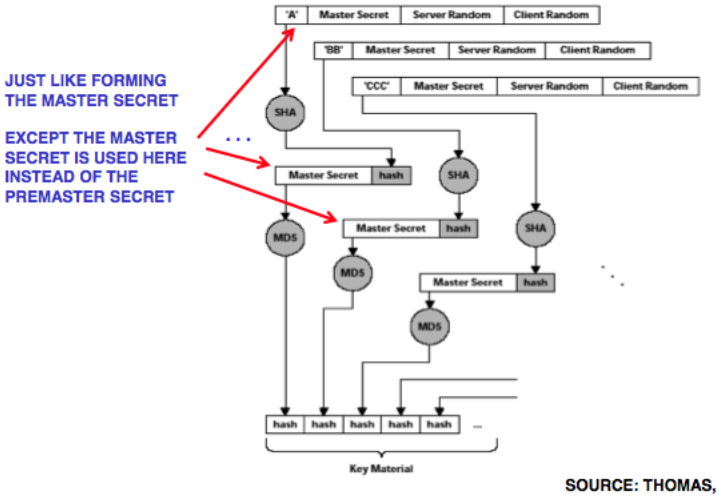


1.7.3 generating Master Secret



SOURCE: THOMAS, *SSL AND TLS ESSENTIALS*

1.7.4 generating key material



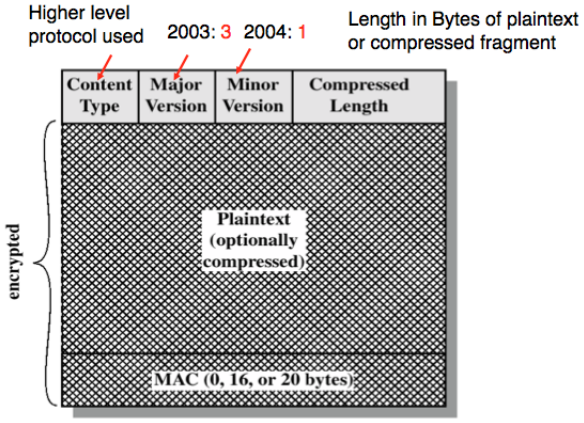
SOURCE: THOMAS, *SSL AND TLS ESSENTIALS*

1.8 Record protocol

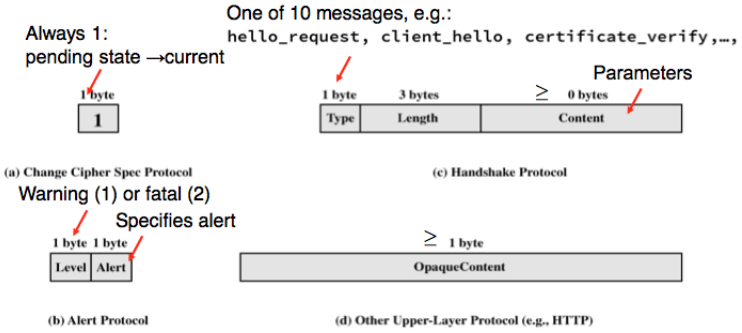
1.8.1 Ablauf

1. Provides confidentiality and integrity.
2. Fragments message (214 bytes).
3. Optional compression (default: none).
4. Calculates MAC (Message Authentication Code = integrity). MAC traditionally is based on symmetric block cipher, i.e. MAC = CK(M) where M is the message, MAC a fixed length authenticator, K a secret symmetric key shared between sender and receiver and C(.) a function.
5. Modified H(Hash)MAC to include sequence number (prevent replay attacks).
6. Encrypts both message and MAC (confidentiality+integrity).
7. Propend header.

1.8.2 Format



1.9 SSL remaining protocols



1.10 TLS enhancements

Differences in the:

- version number
- message authentication code
- pseudorandomfunction(PRF)
- alert codes
- cipher suites
- client certificate types
- certificate_verify and finished message
- cryptographic computations
- padding

Otherwise similar to SSL v3.1. Important: the same record format as the SSL record.

1.11 Words to the wise about applied cryptography

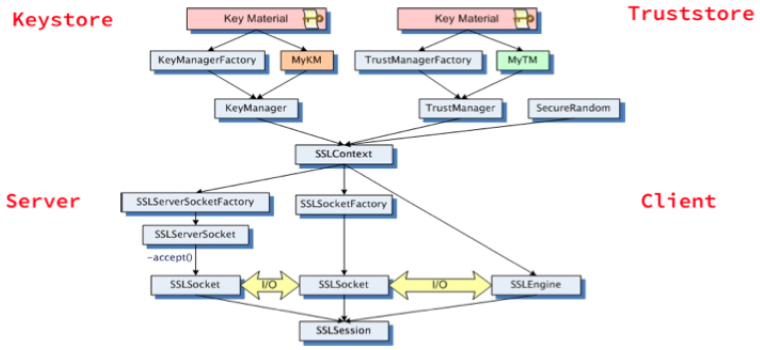
1. Hashing and symmetric key ciphers are fast: use them for session encryption.
2. RSA public key is slow, but it is useful for key exchange and user/server authentication

3. Informal web of trust of PGP (you are in charge) versus CA based (you do not know, who controls it: in fact the NSA controls it (note added in 2013))
4. A good protocol should:
   (a) negotiate **cryptographic** parameters
   (b) establish **shared** secret
   (c) authenticate **endpoints**
5. Use ssh, pgp liberally as the crypto part of an application.
6. SSL: a transport interface, but you still needs to modify the application
7. IPsec places cryptography where it belongs: at the network layer.

1.12 SSL and Java applications

1.12.1 Java SSL software (JSSE)

Set of API to construct SSL-Client and SSL-Server sockets.
Two important new concepts: **trust store** and **key store**. Both are databases that hold certificates. Key store are used to provide credentials; trust store to verify them.



1.12.2 X.509 certificate

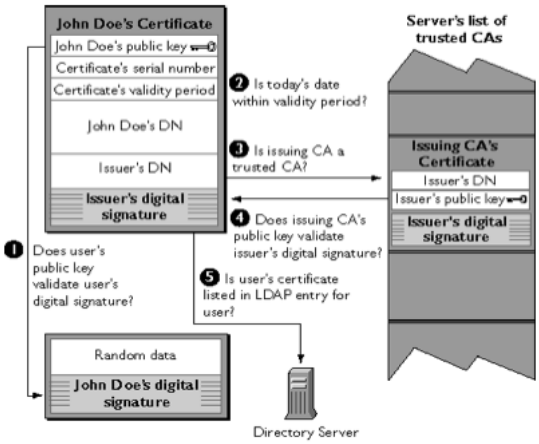
Every X.509 certificate consists of two sections: (i) data and (ii) signature.
The data section includes the following information:

1. The **version number** of the X.509 standard supported by the certificate.
2. The certificate's **serial number**. Every certificate issued by a CA has a serial number that is unique to the certificates issued by that CA.
3. Information about the user's **public key**, including the algorithm used and a representation of the key itself.
4. The **DN** of the CA that issued the certificate.
5. The period during which the certificate is **valid** (for example, between 1:00 p.m. on January 1, 2000 and 1:00 p.m. December 31, 2000).
6. The DN of the certificate **subject** (for example, in a client SSL certificate this would be the user's DN), also called the subject name.
7. Optional **certificate extensions**, which may provide additional data used by the client or server. For example, the certificate type extension indicates the type of certificate - that is, whether it is a client SSL certificate, a server SSL certificate, a certificate for signing email, and so on. Certificate extensions can also be used for a variety of other purposes.

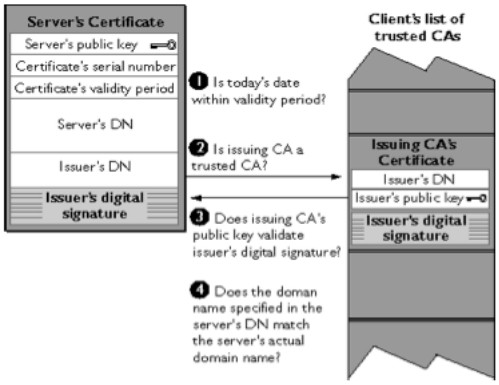
The signature section includes:

1. The **cryptographic algorithm**, or cipher, used by the issuing **certificate authority (CA)** to create its own digital signature.
2. The CA's **digital signature**, obtained by hashing all of the data in the certificate together and encrypting it with the CA's private key.

1.12.3 Webserver verification of client's certificate



1.12.4 Client verification of server's certificate



1.12.5 Certification path

The chain, or path, begins with the certificate of that entity, and each certificate in the chain is signed by the entity identified by the next certificate in the chain. The chain terminates with a root CA certificate. The root CA certificate is always signed by the CA itself.

1.12.6 A secure Webserver in Java

Listing 1: secure webserver

```
1 public class HTTPSSErver {
    public static void main(String[] args) {
      SSLServerSocketFactory ssf = (SSLServerSocketFactory)SSLServerSocketFactory.getDefault
          ();
      SSLServerSocket ss = ssf.createServerSocket(8443);
```

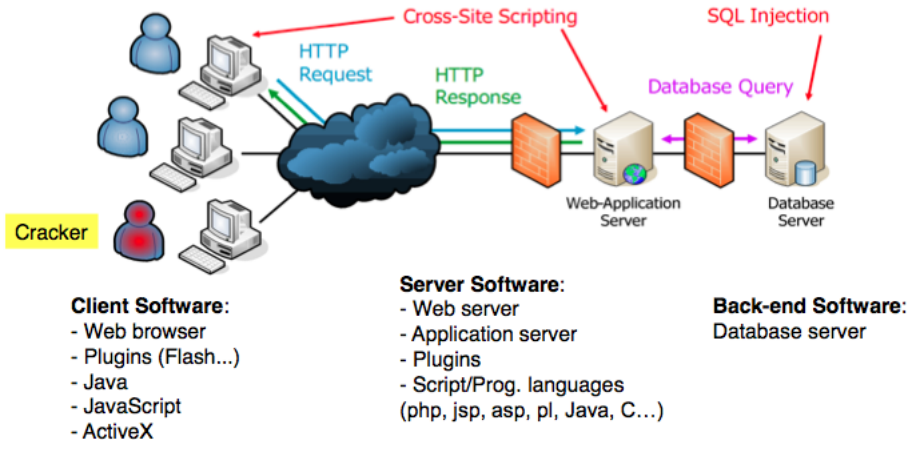
```
6       System.out.println("SSLSocketServer is listening...");

        while (true) {
          try {
            Socket socket = ss.accept();
11          OutputStream out = socket.getOutputStream();
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream
                ()));
            String line = null;
            while(((line = in.readLine()) != null) && (!("".equals(line)))) {
              System.out.println(line);
16          }

            StringBuffer buffer = new StringBuffer();
            buffer.append("<html>\n");
            buffer.append("<head><title>HTTPS Server</title></head>\n");
21          buffer.append("<body>");
            // more noise here ...
            buffer.append("</body>\n</html>");

            byte[] data = buffer.toString().getBytes();
26          out.write("HTTP/1.0 200 OK\n".getBytes());
            out.write(new String("Content-Length: "+data.length+"\n").getBytes());
            out.write(data);
            out.flush();
            out.close();
31          in.close();
            s.close();
          } catch(Exception e) { e.printStackTrace(); }
        }
      }
36  }
```

2 Validation

2.1 Fundamental principles

1. **Least privilege**: Both users and applications should have only the minimum set of privileges necessary to run.
2. **Economy of mechanisms/Simplicity**: "Techniques such as line by line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential."
3. **Open design**: The protection mechanism must not depend on the attacker's ignorance. Security by obfuscation is not only stupid, but dangerous too.
4. **Complete mediation**: Every access attempt must be checked.
5. **Fail-safe defaults**: The default should always be denial of access. The protection scheme should clearly identify under which conditions access is permitted.
6. **Separation of privilege**: Access to objects should depend on more than one condition, so that defeating one protection's layer doesn't grant complete access. (Compartmentalisation.)
7. **Least common set of mechanism**: Minimize the amount and use of shared mechanisms.
8. **Easy to use**: If the protection is not intuitively simple to learn, the users will ignore it.

2.2 Points of attack



2.3 SQL injection

2.3.1 login attack

The cracker (always aptly named Oscar) logs in:

- He tries to manipulate the SQL query in such a way that it always returns at least one row.
- With logins, this often works with 'or' '='
- GET/path/login.pl?user='or"='&pwd='or"='HTTP/1.0
- Resulting SQL query: SELECT * FROM Users WHERE users=" or "=" AND pwd=" or "="
- Since the WHERE clause is always true, the query returns all rows
- The attacker is allowed to "enter the system" and gets the identity of the first entry in the table Users

2.3.2 some more tricks

Suppose users can select car models via a drop-list:

- Selected entry is sent as a number to the web server
- GET/path/showcars.pl?brand_id=17; DROP TABLE Users HTTP/1.0
- Resulting SQL query: SELECT * FROM Cars WHERE brandID=17; DROP TABLE Users

SQL injection is powerful, but it is not always easy to carry out:

- One must first find out a vulnerable web application
- The internal structure of the database is normally unknown to the attacker
- The type of database the application uses, is mostly unknown
- Requires a good knowledge of SQL and of different database products
- Resulting SQL queries must be syntactically correct

Good strategy:
Insert a '(single quote) in web form fields: This usually breaks the SQL statements and (if we are lucky) gives information about the SQL via an error message;

2.3.3 Defenses against SQL injection

- On the server, **validate**, **constrain** and **sanitize** all data provided by the client (length restriction, disallow critical characters (single quote etc.))
- Do not use client parameters directly in SQL queries, but use prepared statements (type-checking, parameters are escaped by the DBMS, only one query is executed)
- A similar effect can be achieved by parameterized stored procedures
- Avoid disclosing detailed database error information to the client
- Access the database with **minimal** privileges

2.3.4 SQL statements' sanitation

Listing 2: PreparedStatement Pseudocode

```
updateSales = con.prepareStatement("UDATE Cars SET Name = ? WHERE id = ?");
updateSales.setString(1, e.getKey());
updateSales.setInt(2, e.getValue().intValue());
```

2.4 Cross-Site Scripting

2.4.1 XSS

Dynamically generated web pages send often data entered by the user in web forms back to the user. With XSS, an attacker exploits formulars:

- The attacker submits a JavaScript in a web form
- The dynamically generated web page contains the script
- When displayed in a browser, the script is executed
- Unlike SQL injection, the attack targets another user, not the server!
- Effective attack because (i) most browsers have JavaScript enabled and (ii) it is platform independent

If an attacker manages to execute JavaScript code in the browser of a victim, the following is possible:

- He can fetch arbitrary additional JavaScript code from a server
- He can retrieve the currently used session-id, which allows to take over the session by session hijacking (e-Commerce, e-Banking...)
- He can dynamically adapt the web page during the loading/rendering process in the victim's browser, e.g. to replace the login dialogue with an own version and to steal the victim's credentials

Difficulties for the attacker: To execute a JavaScript in the browser of another user, that user must send the JavaScript to the vulnerable web server

2.4.2 Defenses against XSS

Secure communication protocols, packet-filtering firewalls are useless. But: using HTTPS helps against Session Hijacking.
Server side: do the following:

- Validate all data provided by client
- Sanitize all client-provided data before sending them back to the browser
- Especially watch for <script>, </script>, javascript:
- Replace < and > around script of client-provided data with and browser interprets this as string literals and not as scripts

Client side: disable JavaScript

2.5 Regular Expressions

2.5.1 Cheet Sheet

Literal Characters	
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\a	Alarm (beep)
\e	Escape
\xHH	The ASCII character specified by the two digit hexadecimal code. For octal use \OOO except JS
\x{HHHH}	PHP: ASCII character represented by a four digit hexadecimal code. Javascript uses \uHHHH
\cX	The control character ^X. For example, \cI is equivalent to \t and \cJ is equivalent to \n

Character Classes							
[...]	Any one character between the brackets.						
[^...]	Any one character not between the brackets.						
.	Any character except newline. Equivalent to [^\n]						
\w	Any word character. Equivalent to [a-zA-Z0-9_] and [[:alnum:]_]						
\W	Any non-word character. Equivalent to [^a-zA-Z0-9_] and [^[:alnum:]_]						
\s	Any whitespace character. Equivalent to [\t\n\r\f\v] and [[:space:]]						
\S	Any non-whitespace. Equivalent to [^ \t\n\r\f\v] and [^[:space:]] Note: \w != \S						
\d	Any digit. Equivalent to [0-9] and [[:digit:]]						
\D	Any character other than a digit. Equivalent to [^0-9] and [^[:digit:]]						
[\b]	A literal backspace (special case)						
[[:class:]]	alnum	alpha	ascii	blank	cntrl	digit	graph
	lower	print	punct	space	upper	xdigit	

Replacement	
\	Turn off the special meaning of the following character.
\n	Restore the text matched by the nth pattern previously saved by \(and \). n is a number from 1 to 9, with 1 starting on the left.
&	Reuse the text matched by the search pattern as part of the replacement pattern.
~	Reuse the previous replacement pattern in the current replacement pattern. Must be the only character in the replacement pattern. (ex and vi).
%	Reuse the previous replacement pattern in the current replacement pattern. Must be the only character in the replacement pattern. (ed).
\u	Convert first character of replacement pattern to uppercase.
\U	Convert entire replacement pattern to uppercase.
\l	Convert first character of replacement pattern to lowercase.
\L	Convert entire replacement pattern to lowercase.

Repetition	
`{n,m}`	Match the previous item at least n times but no more than m times.
`{n,}`	Match the previous item n or more times.
`{n}`	Match exactly n occurrences of the previous item.
`?`	Match zero or one occurrences of the previous item. Equivalent to {0,1}
`+`	Match one or more occurrences of the previous item. Equivalent to {1,}
`*`	Match zero or more occurrences of the previous item. Equivalent to {0,}
`{}?`	Non-greedy match - will not include the following group/match characters.
`??`	Non-greedy match - will not include the following group/match characters.
`+?`	Non-greedy match - will not include the following group/match characters.
`*?`	Non-greedy match. E.g. `^(.*?)\s*$` the grouped expression will not include trailing spaces.

Options	
`g`	Perform a global match. That is, find all matches rather than stopping after the first match.
`i`	Do case-insensitive pattern matching.
`m`	Treat string as multiple lines: ^ and $ match internal \n
`s`	Treat string as single line: ^ and $ ignore \n, but . matches \n
`x`	Extend your pattern's legibility with whitespace and comments.

Extended Regular Expression	
`(?#...)`	Comment, "..." is ignored.
`(?:...)`	Matches but doesn't return "..."
`(?=...)`	Matches if expression would match "..." next
`(?!...)`	Matches if expression wouldn't match "..." next
`(?imsx)`	Change matching rules (see options) midway through an expression.

Grouping	
`(...)`	Grouping. Group several items into a single unit that can be used with *, +, ?, |, and so on, and remember the characters that match this group for use with later references.
`|`	Alternation. Match either the subexpressions to the left or the subexpression to the right.
`\n`	Match the same characters that were matched when group number n was first matched. Groups are subexpressions within (possibly nested) parentheses.

Anchors	
`^`	Match the beginning of the string, and, in multiline searches (/m), the beginning of a line. PHP: Use \A to match beginning of string in all line matching modes.
`$`	Match the end of the string, and, in multiline searches (/m), the end of a line. PHP: Use \z and \Z to match the end of a string or end of text respectively.
`\b`	Match a word boundary. That is, match the position between a \w character and a \W character. (Note, however, that [\b] matches backspace.)
`\B`	Match a position that is not a word boundary.

2.5.2 Example

Listing 3: email checker

```
  public class Email AddressChecker {
2   public static void main(String[] args) {
```

```
      // Checks for email addresses starting with
      // inappropriate symbols like dots or @ signs.
      Pattern p = Pattern.compile("^\\.|^\\@");
      Matcher m = p.matcher(args[0]);
      if (m.find()) System.err.println("Email addresses don't start with dorts or @ signs.");

      // Cekcs for email addresses that start with
      // www. and prints a message if it does.
      p = Pattern.compile("^www\\.");
      m = p.matcher(args[0]);
      if (m.find()) System.err.println("Email addresses don't start with www, only web pages"
          );

      p = Pattern.compile("[\\@] [\\.]+");
      m = p.matcher(args[0]);
      if (!m.find()) System.err.println("Emails must contain at most a @ and at least a .");

      p = Pattern.compile("^([A-Za-z0-9]+[-._+&])*[0-9a-zA-Z]+@([-0-9a-zA-Z]+[.])+[a-zA-Z
          ]{2,6}$");
      m = p.matcher(args[0]);
      Stringbuffer sb = new StringBuffer();
      boolean result = m.find();
      boolean deletedIllegalChars = false;
      while (result) {
        deletedIllegalChars = true;
        m.appendReplacement(sb, "");
        results = m.find();
      }

      // Add the last segment of input to the new String
      m.appendTail(sb);
      String input = sb.toString();
      System.out.println(input);
      if (deletedIllegalChars) {
        System.out.println("It contained incorrect characters, such as spaces or commas.");
      }
    }
  }
```

3 Robust Programming

3.1 Einleitung

Writing programs is not a trivial task. Most (large) programs that are written contain errors: that means that the program doesn't do what it's meant to do.

There are many sources of error:

- **Syntactic errors** (good compilers take care of that);
- **Semantic errors** (our brains are not perfect programming machines ...)
- Programs are "correct"but **input/output fail** and resource are not ready ! exception handling.
- **Languages' ambiguous specifications** (or their implementations) break the program (example: Java memory model and concurrency).

A good definition of robustness is therefore: **Graceful behaviour in presence of errors**
This means that if the program fails, it falls into a well known state and at least logs why it has failed.

3.2 Specific tools enhance robustness

The most useful techniques are:

- Judicious use of types and visibility attributes;
- Annotations (compile time);
- Correct use of the exception handling mechnisms (run time);
- Assertions (run time).

3.3 Types and visibility in Java

Rule 1 All fields are **private**

Rule 2 Fields, once initialized by the class constructor, are **never** updated.

Rule 1 + Rule 2 $\rightarrow$ the class is **immutable**!

3.4 Code inheritance breaks encapsulation

Against code inheritance:

1. Strong relationship between base and derived class
2. The extension of a class with sub classing, requires an in-depth knowledge about the implementation of the base class
3. Specialization interface means:
    - Protected methods: Disrupts the private/public only model.
    - Specialization semantics

Inheritance breaks encapsulation: Support for inheritance implies that (some) implementation details have to be published!

3.4.1 Instead inheritance use composition

The advantages of composition:

- Only the client interface is used
- No call-backs (in contrast to the inheritance based solutions)
- No dependency on the implementation

3.4.2 Instead inheritance use Template/Hooks

Template methods:

- Template method pattern provides a robust method to allow for extensions
- Base class provides extension points

3.5 Annotations

3.5.1 General remarks

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.
Main usages:

- **Information for the compiler**: Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compiler-time and deployment-time processing**: Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing**: Some annotations are available to be examined at runtime.

3.5.2 Why are annotations so important?

1. Annotations act mostly at compile-time: and this is a very good thing. We want to catch all of our subtle and not so subtle errors at compile time and not at run- time in front of an important customer ...
2. Annotations can be extracted by external tools. Instead of looking for methods with a particular name or signature, retrieve all methods with a specific annotation.
3. Annotations are like classes. They have a specific type. They can contain fields to store details.
4. Meta-specifications for annotations include:
   (a) Where they can appear on (e.g. only on classes, or only on methods)
   (b) A retention policy: when and where they are made available:

3.5.3 Overview

Meta
Im Paket java.lang.annotation - diese werden nur für die Definition von Annotationen gebraucht.

Annotation	Beschreibung
@Target	Dieser Annotationstyp wird bei der Programmierung einer Annotation angewandt. Damit wird festgelegt, auf welche Elemente eines Programms sie angewendet werden darf. Die möglichen Werte für eine Annotation dieses Typs sind in der Enumeration ElementType aufgeführt: TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE.
@Retention	Enumeration in RetentionPolicy. Specifies wheter the annotation is only available for: SOURCE: at compile time, discarded by compiler CLASS: stored in class file, discarded by VM RUNTIME: retained in class file and loaded by VM, accessible at runtime using reflection
@Inherited	The annotation get applied to subclasses as well (only CLASS annotation)
@Documented	Annotation should appear in the JavaDoc

Vordefinierte

Annotation	Beschreibung	Beispiel
@Deprecated	[RUNTIME] Generates compiler warnings	
@SupressWarnings	[SOURCE] Indicates that the named compiler warnings should be supressed	
@Override	[SOURCE] Indicates that a method is intended to override a method in a super class.	
@Resource	[RUNTIME] Any class or component that provides some useful functionality to an application can be thought of as a Resource and the same can be marked with @Resource annotation. The programmer specifies, how this resource must be accessed.	`@Resource(name = "MyQueue", type = javax.jms.Queue, shareable = false, authenticationType = Resource.AuthenticationType.CONTAINER, description = "A Test Queue" )` `private javax.jms.Queue myQueue;`
@PostConstruct	[RUNTIME] Used on a method that needs to be executed after dependency injection (Spring FW!) is done to perform any initialization. This method must be invoked before the class is put into service.	`@PostConstruct` `public void initIt() throws Exception { System.out.println("Init method after properties are set : " + message); }`
@PreDestroy	[RUNTIME] Used to release resources that it has been holding.	`@PreDestroy` `public void cleanUp() throws Exception { System.out.println("Spring Container is destroyed! Customer clean up"); } }`
@CheckReturnValue	Telling the compiler that it should issue a warning if the return value isn't used.	`@CheckReturnValue` `public String slimDownString() { return str.trim(); }`

@Nonnull	The annotated element must not be null. Annotated fields must not be null after construction has completed. Annotated methods must have non-null return values.	`public String myToUppercase(@NonNull String parameter) { return parameter.toUpperCase(Locale.getDefault()); }`
		`@NonNull`
		`public String getValue() { return this.value; }`
		`public void justUseGetValue() { int length = getValue().length(); System.out.println("Value length: " + length); }`

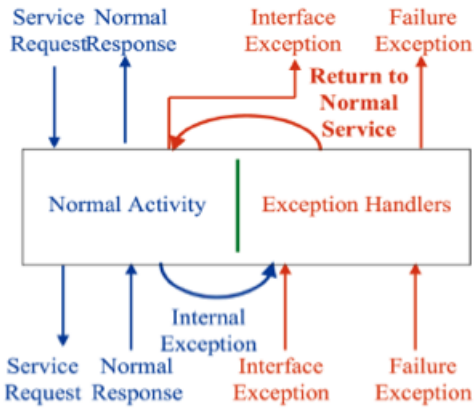
@CheckForNull	The annotated element might be null, and uses of the element should check for null. When this annotation is applied to a method it applies to the method return value.	`@CheckForNull`
		`public String value;`
		`public int getValueLength() { if (this.value == null) return 0; return this.value.length(); }`

Custom

Annotation	Beispiel
public @interface ToDo{public String value();}	@ToDo("improve quality")
public @interface Authors{public String[] names;}	@Authors(names={"Grz","Nc"})
public @interface RequestForEnhancements{ public int id(); public String assigned() default ""; public String date();}	@RequestForEnhancements(id=5, date="11.12.13")

3.6 exception handling

3.6.1 Ideal fault-tolerant software components



3.6.2 Requirements for exception handling

1. **Simplicity**: Should be simple to understand and use
2. **Unobtrusiveness**: Exception handling code should not obscure the understanding of the software: (1) and (2) are crucial for designing reliable systems!
3. **Efficiency**: Run-time overheads should be incurred only when handling an exception. This may be relaxed, e.g. if speed of recovery is not critical
4. **Uniformity**: Uniform treatment of exceptions detected both by the environment and by the program
5. **Recovery**: It should allow recovery actions

3.6.3 Classes of exceptions

Who detects the error?

- Environmental error detection
- Application error detection

When it is detected?

- **Synchronously**: as an immediate result of a process attempting an inappropriate operation
- **Asynchronously**: some time after the operation causing the error. May be raised either in the process which executed the operation or in another process. The latter is also called asynchronous notification or signal. Mostly an issue with concurrent programming.

3.7 Assertions

These three roles collectively support what is called the design-by-contract model of programming, a model that is well supported for example by the Eiffel programming language. Java doesn't have built-in support for the design-by-contract model of programming. But you can use the java.assertion package to enforce these rules.

3.7.1 Decleration

An assertion statement in Java takes one of the following two forms:

- assert condition:

- assert condition : error message;

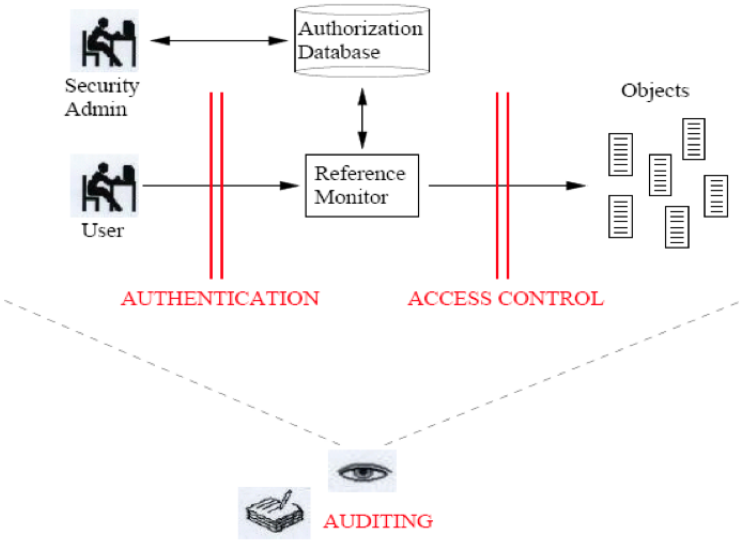
3.7.2 enable/disable

To enable assertions at various granularities, use the -enableassertions To disable assertions at various granularities, use the -disableassertions

4 Action Control

4.1 Goal

Specify the limits within which a legitimate user can work with the system.



4.2 AC policies

Subjects detain privileges on objects according to AC policies (models).
Policy: specifies how accesses are controlled and access decisions determined.

Mechanism (structure): implements or enforces a policy.

- Access matrix.
- AC list (ACL).
- Capability list.

4.2.1 Discretionary AC

This policy restricts access to system objects based on the identity of the users and/or the groups to which they belong.
Two types of DAC policies are used in practice:

- **Closed DAC**: authorization must be explicitly specified, since the default decision is always denial.
- **Open DAC**: denials must be explicitly specified since the default decision is always access.

Problems:

- Danger of users' mistakes or negligence;
- The dissemination of information is not controlled: for example a user who is able to read data can pass it to other users not authorized to read it without the actual owner knowing it.

4.2.2 Mandatory AC

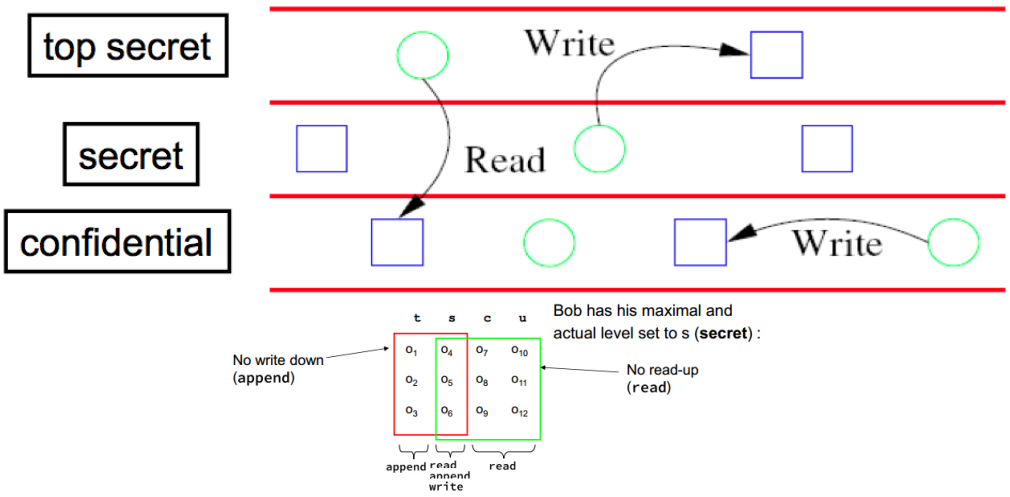
Idea: each subject and each object is assigned a security level.

1. Subject's label (clearance) specifies the level of trust associated with that subject.
2. Object's label specifies the level of trust that a subject must have to be able to access that file.

Pro: Reduction of possible damage
Contra: Loss of flexibility.

4.3 Bell-LaPadula (BLP) policy model

- The Simple Security Policy: no process may read data at a higher level. **No Read Up (NRU)** or: a subject can only read an object of less or equal security level.
- The *-property: no process may write data to a lower level. **No Write Down (NWD)** or: a subject can only write into an object of greater or equal security level.



4.4 Other policy concepts

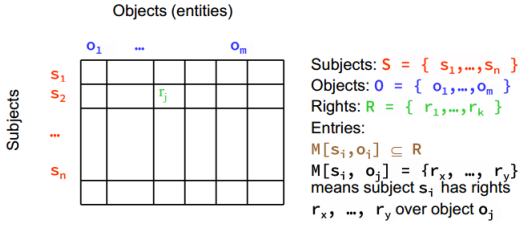
- Seperation of Duty: Check is only valid if signed y two authorized people
- Chinese Wall Policy

4.5 Role-based access control (RBAC)

Permissions are associated with:

- roles (= set of actions and responsibilities, associated with particular working activities)
- users/subjects

4.6 ACM/ACL



Right to copy: Two rights in stead of one Read (r / rc). Attenuation of privilege: You can't give away rights you do not possess. This principle is usually ignored for the owner of an object. **Summary**

- Access control matrix is the simplest abstraction mechanism for representing a protection state within a system.
- Transitions alter the protection state.
- Six primitive operations alter the AC matrix: Transitions can be expressed as commands composed of these operations and, possibly , conditions.

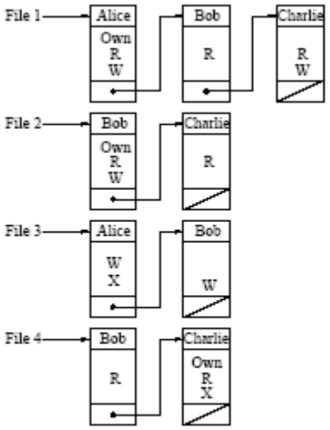
4.7 AC list (ACL)

In large systems the matrix will be big and sparse. Therefore in practice, the matrix is implemented as a list.

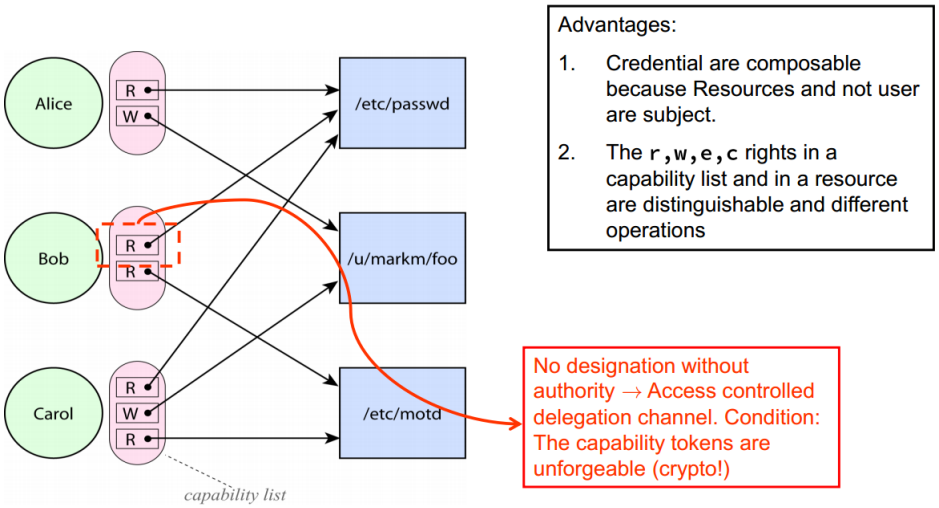
AC matrix

	File 1	File 2	File 3	File 4	Account 1	Account 2
Alice	Own R W		W X		Inquiry Credit	
Bob	R	Own R W	W	R	Inquiry Debit	Inquiry Credit
Charlie	R W	R		Own R X		Inquiry Debit

AC list (ACL)



4.8 Capability list



Delegation: Process can pass capability at run time **Revocation:**

- Only if OS knows which data is associated with a capability. If capability is used for multiple resources, one has to revoke all or none.
- Indirection: capability points to pointer to resource (C=¿P=¿R, set P=0 to revoke)

4.9 Android

4.9.1 Basic Components of an Application

- Activity: User Interface
- Service: Java daemon
- Content provider: Store and share data
- Broadcast receiver: For messages from other applications
- Intents: asynchronous mesaging system

4.9.2 Security steps

- Signing with certificates
- Application sandbox: runs with its UID and own Dalvik virtual machine
- Linux and ACL
  - Each application executes with its own user identity as Linux process
  - Android middleware has a reference monitor that mediates the establishment of inter-component communication (ICC)

4.9.3 Android manifest file

Focus on Inter Component Communication (ICC) whose security policy is defined in the Android manifest file. Allows developers to specify a high-level ACL to access the components:

- Each component can be assigned an access permission label
- Each application requests a list of permission labels

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      ...
<application android:icon="@drawable/icon“android:label="@string/app_name">
  <activity android:name=".BatteryCheck"android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
      <action android:name="ch.fhnw.android.sms.receiver.BATTERY_SMS" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
  </activity> </application>
<uses-permission android:name="android.permission.BATTERY_STATS" />
<uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_SMS"></uses-permission>
</manifest>
```

- Components can be public or private (exported attribute). It specifies whether the activity can be launched by components of other applications. Default value is not reliable, set it always explicitly.
- If the manifest does not specify an access permission, any component in any application can access it. Components without access permissions should be exceptional cases, and possible inputs must be carefully analysed (consider splitting components).
- If no permission label is set on a broadcast, any unprivileged application can read it. Always specify an access permission on Intent broadcasts (unless you specify the destination explicitly).

Listing 4: Intent broadcast permissions

```
Intent resultIntent = new Intent();
String action = "ch.fhnw.android.battery.BATTERY_SMS";
resultIntent.setAction(action);
resultIntent.putExtra("Sending Activity", "Battery Check");
resultIntent.putExtra("Battery Info", batteryRcv.batteryInfo);
sendBroadcast(resultIntent);
```

Android content provider permissions

- seperate ”read and ”write”
- URI permissions to specify which data subsets of the parent content provider permission can be granted for
- Always define separate read and write permissions. Use URI permissions to delegate right to other components.

```
<provider android:authorities="friends"
        android:name="FriendProvider"
        android:readPermission="“ch.fhnw.apsi.permission.READ_FRIENDS"
        android:writePermission="ch.fhnw.apsi.permission.WRITE_FRIENDS">
</provider>
```

- The application developer can add reference monitor hooks
- Use checkCallingPermission() to mediate ädministrativeöperations. Alternatively, create separate Services.

```
private final IFriendTracker.Stub mBinder = new IFriendTracker.Stub() {
  public boolean isTracking() {return mTracking;}
  public boolean addNickname(String nick, int contactId) {
   if (FriendTracker.this.checkCallingPermission(PERMISSION_FRIEND_SERVICE_ADD)
                                    != PackageManager.PERMISSION_GRANTED) {
    throw new SecurityException("Requires " + PERMISSION_FRIEND_SERVICE_ADD);
  } ... }
};
```

Permission categories: normal, dangerous, signature, signatureOrSystem

- Use signature permissions for dangerous permissions. Otherwise and always include informative descriptions

```
<permission android:name="ch.fhnw.apsi.permission.FRIEND_NEAR“
        android:label="@string/permlab_friendNear"
        android:description="@string/permdesc_friendNear“
        android:protectionLevel="dangerous">
</permission>
```

Defined in string.xml

5 JavaIdiocies

5.1 Visibility modifiers

(default, private, protected, public)

5.1.1 Purpose

- Information hiding
- Compartmentalization of program's components (things you should know (public) and things you can safely ignore(private))

5.1.2 Attention

- Java allows increasing the visibility of a member in subclasses
- protected is unsafe: subclassing and a class can claim to have the same package.

5.2 Protecting packages

5.2.1 Sealed JAR archives

Add header in the manifest. All classes in a package must be found in the same JAR file.

5.2.2 Signing JAR archives

Sending point: jarsigner -keystore x -signedJar sCount.jar Count.jar signFiles
keytool -export -keystore susanstore -alias signFiles -file SusanJones.cer
Receiving end:

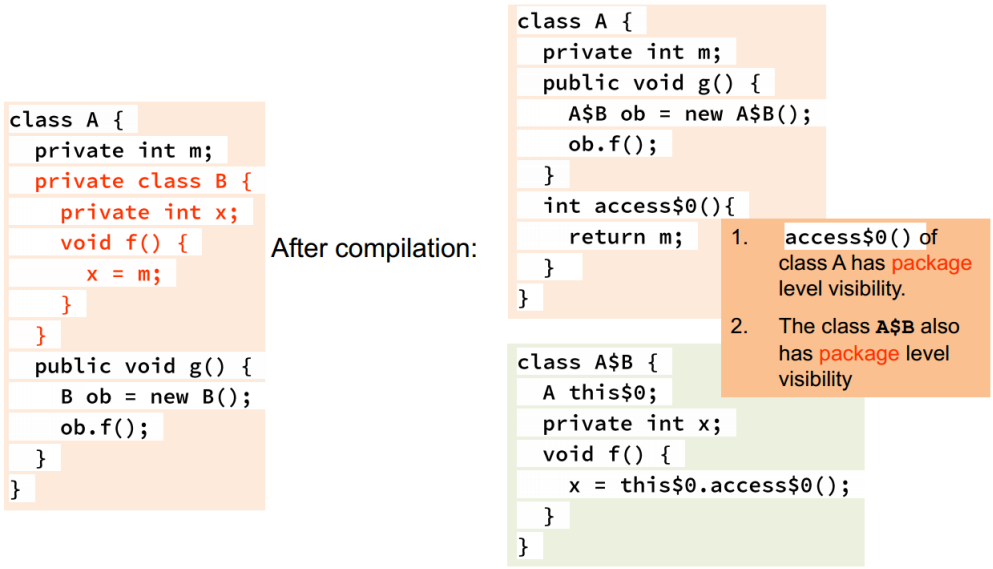
```
grant CodeBase "file:./..." SignedBy "Susan" {
  permission java.io.FilePermission "C:\\TestData\\*", "read";
}
VM-Argumente:
java -Djava.security.manager -Djava.security.policy=raypolicy -cp sCount.jar Count C:\
    TestData\data
```

5.2.3 Join packages via security policy PAP

Can insert in the java.security file:
package.definition=com.ibneco.securepackage
Attention: no class loaders from Sun support this at present

5.3 Inner classes

Inner classes are not understood by JVM:



Other classes belonging to the same package can call the access function and tamper the private data member.

5.4 Common Java Antipatterns

5.4.1 Assuming objects are immutable

Object[] signers;
public Object[] getSigners()return signers;
Problem: We quickly forget that mutable input and output objects can be modified by the caller application.
Solution: Make a copy of mutable I/O parameters.

5.5 Basing security checks on untrusted sources

Signature (final java.io.File f) ... f.getPath() ...
Problem: Do not forget that security checks can be always fooled if they are based on information that attackers can control!

- You can not trust libraries
- non-final classes/methods can be sub classed
- mutable types can be modified

Solution: Make copys of inputs or make a copy of the source of the input. Never invoke doPrivileged() with caller-provided inputs.

5.6 False inheritance relationship

Problem: class Stack¡E¿ extends Vector¡E¿. Stack is not a Vector but all Methods that can be applied to a Vector can now applied to the Stack.

5.7 Ignoring changes to super classes

Problem: Changes to a super class may affect its sub classes which leads to unwanted behaviour or misuse of inherited methods.

Solution: Subclass only when the inheritance model is well-specified and well-understood. Monitor changes to super classes.

5.8 Neglecting to validate inputs

Problems: Invalidated inputs can be out-of-bounds values, escape characters, can affect processing of requests or delegating them to sub-components (SQL). Additional issues when calling native methods.
Solution: Validate all inputs

- Check for escape characters
- Check for out-of-bounds values
- Check for malformed requests
- Regex API can help validate String inputs
- Wrap native methods in Java language wrapper to validate inputs
- Make all natives method private

5.9 Misusing public static variables

Problem: Static variables can be modified or misused as a communication channel globally across a Java runtime environment.
Solution: Reduce the scope of static fields and define acces methods (with securityManagerCheck();). Treat public static fields primarly as constants (final).

Listing 5: Possible implementation of securityManagerCheck()

```
private void securityManagerCheck() {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkPermission(...);
  }
}
```

5.10 Believing a constructor exception destroys the object

Problem: If the constructor throws an exception, a reference on it still exists on the uninitialized object in the heap. Java 1.7 destroys automatically objects whose constructor has thrown an exception.
Solution: Make the class final if possible. If the finalize method can be overridden, ensure that partially initialized instances are unusable (initialized-flag).

5.11 TOC2TOU - Time Of Check To Time Of Use

Example: Check for a file-permission is done at the opening but never ever checked again, even if privileged actions are done. That is typical for Unix systems.
Solution: Always check the user's permission while executing a privileged operation.

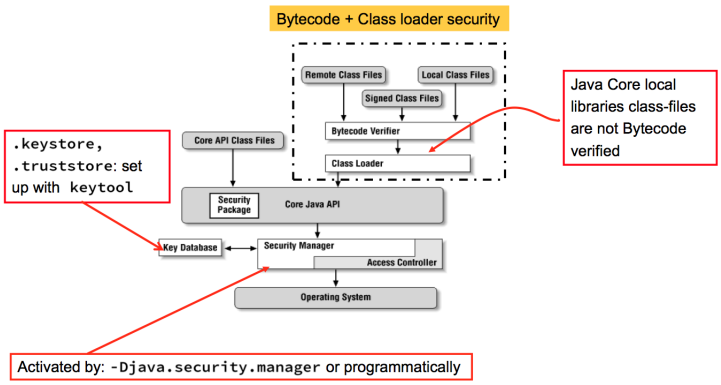
5.12 Twelve (very conservative) guidelines for writing safer Java

- Do not depend on implicit initialization
- Limit access to entities
- Make everything final

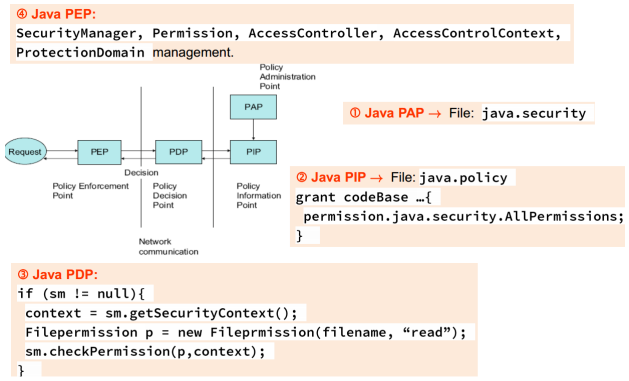
- Do not depend on package scope
- Do not use inner classes
- Avoid signing your code. Code that isn't signed will run without special privileges i.e. less likely to do damage!
- If you must sign, put all signed code in one jar archive
- Make classes uncloneable. Java's object cloning mechanism allows an attacker to build new instances of the classes you define, without using any of your constructors.
- Make classes unserializeable
- Make classes undeserializeable
- Do not compare classes by name
- Do not store secrets

6 Java security overview

6.1 Java security in a nutshell



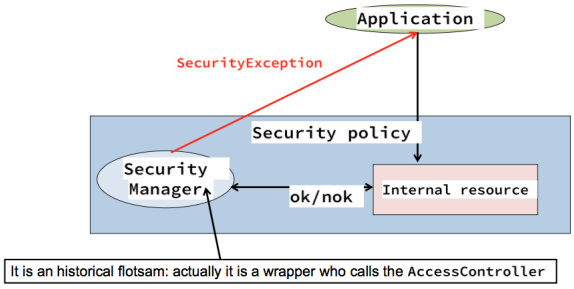
6.2 Java security model



6.2.1 SecurityManager

The SecurityManager can

- control access to files
- control access to network resources
- protect the JVM
- protect system resources
- protect security



6.2.2 Java policy files

The security manager operates according to the policy (defined in `java.policy`) which consists of a set of rules, e.g. the default is:

```
// Standard extensions get all permissions by default
grant CodeBase "file:${java.home}/lib/ext/*" {
  permission java.security.AllPermission;
};
```

Or a more fine grained one:

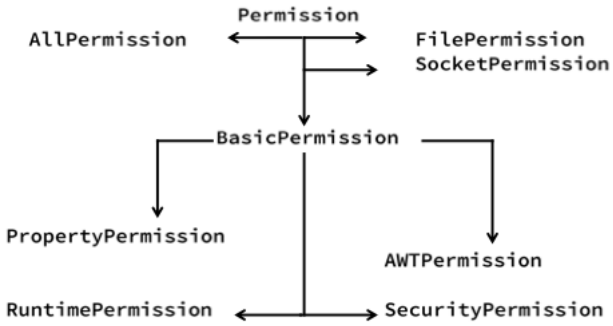
```
grant {
  // allows anyone to listen on un-privileged ports
  permission java.net.SocketPermission "localhost:1024-", "listen";

  // "standard" properies that can be read by anyone
  permission java.util.PropertyPermission "java.version", "read";
};
grant CodeBase "http://java.sun.com/foo.jar" SignedBy "Susan" {
      permission java.io.FilePermission "C:\\TestData\\*", "read";
}
```

At run time these policies are enforced by the class AccessController that is automatically called when the SecurityManager is activated.

All domains which the actual thread crosses must own the desired priviege

6.2.3 Permissions in Java

The permissions are positive, they grant acces rather than deny access. By default, nothing extra is allowed.



Permission Types:

Permission

File
Granting write access to the entiere file system is effectively the same as granting AllPermission

Socket
Permissions to access a networkt via sockets

- The target host is specified with DNS name or IP address along witha portnumber
- the actions: accept, connect, listen, resolve
- Resolve is implied by any of other actions.

Properties
Represents the permission to access various Java properties:

- Target examples: java.home, os.name, user.name
- Actions: read (getPRoperty), write (setProperty)

BasicPermission
Base class for named permissions: i.e. a permission that contains a name instead of a pair (target, action-set)

Runtime
Examples: stopThread, modifyThread, setSecurityManager, writeFileDescriptor

AWT
Examples: accessClipboard, accessEventQueue, readDiosplayPixels

Net
Examples: requestPasswordAuthentication, specifyStreamHandler

Security
Examples: getPolicay, setPolicy, insertProvider, addIdentityCertificate, getSignerPrivateKey, setSignerKeyPair

creating new permissions:

Listing 6: creating own permissions

```
public class WordCheckPErmission extends Permission {
  public boolean implies(Permission other) {
    if (!(other instanceof WordCheckPermission)) return false;
    WordCheckPermission b = (WordCheckPermission) other;
    if (action.equals("insert")) {
      return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0;
    } else if (action.equals("avoid")) {
      if (b.action.equals("avoid"))
        return b.badWordSet().cointainsAll(badWordSet());
      else if (b.action.equals("insert")) {
        for (String badWord : badWordSet())
          if (b.getName().indexOf(badWord) >= 0) return false;
        return true;
      else return false;
    } else return false;
  }
  public void insertWords(String words) {
    try {
      textArea.append(words + "\n");
    } catch (SecurityException e) {
      JOptionPane.showMessageDialog(this, "I am sorry, but I cannot do that.");
    }
  }
  class WordCheckTextArea extends JTextArea {
    public void append(String text) {
      WordCheckPermission p = new WordCheckPermission(text, "insert");
      SecurityManager m = System.getSecurityManager();
      if (manager != null) m. checkPErmission(p);
      super.append(text);
    }
  }
```

The file CustomPERmission.policy is defined as follows:

Listing 7: creating own permissions

```
grant {
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission ch.fhnw.WordCheckPermission "fuck,wixxer,nazi", "avoid";
```

6.2.4 ProtectionDomain

- Created from a CodeSource and a PermissionCollection
- Defines the set of permissions granted to classes; changes the PermissionCollection to change permissions
- Each class belongs to ONE ProtectionDomain instance, set at the class creation time and never ganged again
- Access to these objects is restricted; getting its reference requires a Runtime Permission getPRotectionDomain
- One ClassLoader can hava more than one protection domain

6.2.5 CodeSource

This class extens tha concept of a codeBase to encapsulate not only the location but also the certificate that were used to verify the signed code originating from that location. This classis immutable.

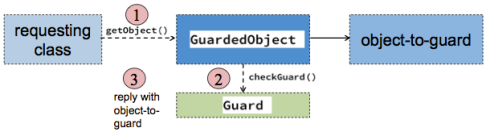
6.3 Byte code verifier

The Goal is to prevent access to underlying physical machine via forged pointers, crashes and undefined states. Checks a class file for validity:

- Code has only valid instructions and register use
- Code does not overflow/underflow the stack
- Code does not convert data types illegally or forge pointers
- Code accesses objects as correct type
- Method calls use correct number and types of arguments
- Refers to other classes use legal names

6.4 Java security at the method level

- To protect one method in all instances, use the SecurityManager & AccessController directly as we have shown so far
- To protect a reference to an individual instance, consider using the class GuardedObject



6.5 Java class loaders

The class loader is the fountain head of Java Security:

- The loading and **verification** steps expel bogus class files before they even get into the JVM
- **Protection domains** drive all later security decisions
- **Namespaces** keep separate classes and packages coming from different places

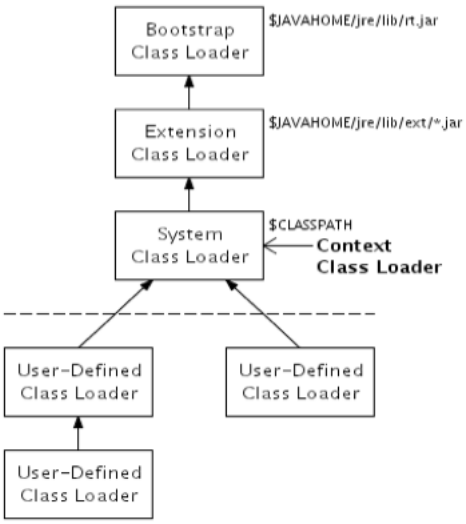
6.5.1 Java class loaders hierarchy

User-defined class loaders created by the program:
The system class loader is the parent class loader by default.
Another parent class loader can be specified explicitly.

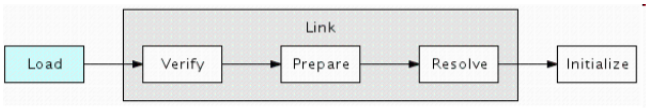
Each thread has an associated context class loader:
Default context class loader is the system class loader.
A different context class loader can be specified by calling
`Thread.setContextClassLoader()`
Obtain a thread's context class loader by calling
`Thread.getContextClassLoader()`

Delegation model:
When asked to load a class, a class loader first asks its parent to load the class. If the parent succeeds, the class loader returns the class from the parent. If the parent fails, the class loader attempts to load the class itself . The class loader hierarchy is thus searched from the top back down to the starting point.

Which class loader is used?
You can explicitly load a class into a class loader by calling the class loader's `loadClass()` method. If a class needs to be loaded and a class loader is not explicitly specified , the class is loaded into the same class loader that loaded the class that contains the code that is executing.



6.5.2 The class loader security phases



load:

→ **Given the type's fully-qualified name, obtain its class file (byte sequence)**
From a file in a directory, from a file in a JAR, from a URL, . . .

→ **Parse the class file (byte sequence) into implementation-dependent internal data structures in the method area**
Parsing problems cause various errors to be thrown (example: the first four bytes are not `0xCAFEBABE`

→ **Resolve the symbolic reference to the super class**
Resolution problems cause various errors to be thrown (example: the super class is not accessible to this class).

→ **Resolve the symbolic references to the interfaces if any**
Resolution problems cause various errors to be thrown (example: if an interface is not accessible to this class).

→ **Create an instance of class `Class` to represent the type**

verify:

Checks a `.class` file for validity (Bytecode Verifier):

→ Final classes are not sub classed
→ Final methods are not overridden
→ Every class has a super class
→ Constant pool entries obey all specified constraints
→ All field and method references have valid classes, names, and type descriptors

1st part

→ Code has only valid instructions and register use;
→ Code does not overflow/underflow the stack;
→ Code does not convert data types illegally or forge pointers;
→ Code accesses objects as correct type;
→ Method calls use the correct number and types of arguments;
→ References to other classes use legal names.
→ Goal is to prevent access to the underlying machine:
For example via forged pointers, crashes, or undefined states.

2nd part

prepare:

- → **Allocate storage for the class's static fields** (this is a shared memory region between all classes)
- → **Set the class's static fields to their default initial values**
  Setting static fields to explicit initial values is performed later, in the Initialization phase.

resolve:

- → **Validate each symbolic class, field, and method reference**
  Search up the inheritance hierarchy to find where the class, field, or method is declared
  Verify that the class, field, or method are accessible to the class which uses them.
- → **Replace each symbolic reference with an (implementation-dependent) direct reference**
- → **Each symbolic reference can be resolved when a class is initially linked** (greedy resolution)
- → **Alternatively, each symbolic reference can be resolved later during execution, when the reference is actually used** (lazy resolution)

initialize:

- → **Execute the class's static `clinit` method**
- → **The `clinit` method is compiler-generated and contains:**
  - → Code for explicit initialization of static fields
  - → Code from all static blocks: a typical covert channel a cracker can use to do mischief (see Java Anti-pattern lecture)

Regex Metacharacters, Modes, and Constructs

The metacharacters and metasequences shown here represent most available types of regular expression constructs and their most common syntax. However, syntax and availability vary by implementation.

Character representations

Many implementations provide shortcuts to represent characters that may be difficult to input. (See MRE 115–118.)

Character shorthands
: Most implementations have specific shorthands for the `alert`, `backspace`, `escape character`, `form feed`, `newline`, `carriage return`, `horizontal tab`, and `vertical tab` characters. For example, `\n` is often a shorthand for the newline character, which is usually LF (012 octal), but can sometimes be CR (015 octal), depending on the operating system. Confusingly, many implementations use `\b` to mean both `backspace` and word boundary (position between a "word" character and a nonword character). For these implementations, `\b` means `backspace` in a character class (a set of possible characters to match in the string), and word boundary elsewhere.

Octal escape: `\num`
: Represents a character corresponding to a two- or three-digit octal number. For example, `\015\012` matches an ASCII CR/LF sequence.

Hex and Unicode escapes: `\xnum`, `\x{num}`, `\unum`, `\Unum`
: Represent characters corresponding to hexadecimal numbers. Four-digit and larger hex numbers can represent the range of Unicode characters. For example, `\x0D\x0A` matches an ASCII CR/LF sequence.

Control characters: `\cchar`
: Corresponds to ASCII control characters encoded with values less than 32. To be safe, always use an uppercase *char*—some implementations do not handle lowercase representations. For example, `\cH` matches Control-H, an ASCII backspace character.

Character classes and class-like constructs

Character classes are used to specify a set of characters. A character class matches a single character in the input string that is within the defined set of characters. (See MRE 118–128.)

Normal classes: `[...]` *and* `[^...]`
: Character classes, `[...]`, and negated character classes, `[^...]`, allow you to list the characters that you do or do not want to match. A character class always matches one character. The - (dash) indicates a range of characters. For example, `[a-z]` matches any lowercase ASCII letter. To include the dash in the list of characters, either list it first, or escape it.

Almost any character: dot (.)
: Usually matches any character except a newline. However, the match mode usually can be changed so that dot also matches newlines. Inside a character class, dot matches just a dot.

Class shorthands: `\w`, `\d`, `\s`, `\W`, `\D`, `\S`
: Commonly provided shorthands for word character, digit, and space character classes. A word character is often all ASCII alphanumeric characters plus the underscore. However, the list of alphanumerics can include additional locale or Unicode alphanumerics, depending on the implementation. A lowercase shorthand (e.g., `\s`) matches a character from the class; uppercase (e.g., `\S`) matches a character not from the class. For example, `\d` matches a single digit character, and is usually equivalent to `[0-9]`.

POSIX character class: `[:alnum:]`
: POSIX defines several character classes that can be used only within regular expression character classes (see Table 1). Take, for example, `[:lower:]`. When written as `[[:lower:]]`, it is equivalent to `[a-z]` in the ASCII locale.

Table 1. POSIX character classes

Class	Meaning
`Alnum`	Letters and digits.
`Alpha`	Letters.
`Blank`	Space or tab only.
`Cntrl`	Control characters.
`Digit`	Decimal digits.
`Graph`	Printing characters, excluding space.
`Lower`	Lowercase letters.
`Print`	Printing characters, including space.
`Punct`	Printing characters, excluding letters and digits.
`Space`	Whitespace.
`Upper`	Uppercase letters.
`Xdigit`	Hexadecimal digits.

Unicode properties, scripts, and blocks: `\p{`*prop*`}`, `\P{`*prop*`}`
: The Unicode standard defines classes of characters that have a particular property, belong to a script, or exist within a block. *Properties* are the character's defining characteristics, such as being a letter or a number (see Table 2). *Scripts* are systems of writing, such as Hebrew, Latin, or Han. *Blocks* are ranges of characters on the Unicode character map. Some implementations require that Unicode properties be prefixed with `Is` or `In`. For example, `\p{Ll}` matches lowercase letters in any Unicode-supported language, such as a or α.

Unicode combining character sequence: `\X`
: Matches a Unicode base character followed by any number of Unicode-combining characters. This is a shorthand for `\P{M}\p{M}`. For example, `\X` matches è; as well as the two characters e'.

Table 2. Standard Unicode properties

Property	Meaning
`\p{L}`	Letters.
`\p{Ll}`	Lowercase letters.
`\p{Lm}`	Modifier letters.
`\p{Lo}`	Letters, other. These have no case, and are not considered modifiers.
`\p{Lt}`	Titlecase letters.
`\p{Lu}`	Uppercase letters.
`\p{C}`	Control codes and characters not in other categories.
`\p{Cc}`	ASCII and Latin-1 control characters.
`\p{Cf}`	Nonvisible formatting characters.
`\p{Cn}`	Unassigned code points.
`\p{Co}`	Private use, such as company logos.
`\p{Cs}`	Surrogates.
`\p{M}`	Marks meant to combine with base characters, such as accent marks.
`\p{Mc}`	Modification characters that take up their own space. Examples include "vowel signs."
`\p{Me}`	Marks that enclose other characters, such as circles, squares, and diamonds.
`\p{Mn}`	Characters that modify other characters, such as accents and umlauts.
`\p{N}`	Numeric characters.
`\p{Nd}`	Decimal digits in various scripts.
`\p{Nl}`	Letters that represent numbers, such as Roman numerals.
`\p{No}`	Superscripts, symbols, or nondigit characters representing numbers.
`\p{P}`	Punctuation.
`\p{Pc}`	Connecting punctuation, such as an underscore.
`\p{Pd}`	Dashes and hyphens.
`\p{Pe}`	Closing punctuation complementing `\p{Ps}`.
`\p{Pi}`	Initial punctuation, such as opening quotes.

Table 2. Standard Unicode properties (continued)

Property	Meaning
`\p{Pf}`	Final punctuation, such as closing quotes.
`\p{Po}`	Other punctuation marks.
`\p{Ps}`	Opening punctuation, such as opening parentheses.
`\p{S}`	Symbols.
`\p{Sc}`	Currency.
`\p{Sk}`	Combining characters represented as individual characters.
`\p{Sm}`	Math symbols.
`\p{So}`	Other symbols.
`\p{Z}`	Separating characters with no visual representation.
`\p{Zl}`	Line separators.
`\p{Zp}`	Paragraph separators.
`\p{Zs}`	Space characters.

Anchors and zero-width assertions

Anchors and "zero-width assertions" match positions in the input string. (See MRE 128–134.)

Start of line/string: `^`, `\A`
: Matches at the beginning of the text being searched. In multiline mode, `^` matches after any newline. Some implementations support `\A`, which matches only at the beginning of the text.

End of line/string: `$`, `\Z`, `\z`
: `$` matches at the end of a string. In multiline mode, `$` matches before any newline. When supported, `\Z` matches the end of string or the point before a string-ending newline, regardless of match mode. Some implementations also provide `\z`, which matches only the end of the string, regardless of newlines.

Start of match: `\G`
: In iterative matching, `\G` matches the position where the previous match ended. Often, this spot is reset to the beginning of a string on a failed match.

Word boundary: `\b`, `\B`, `\<`, `\>`
: Word boundary metacharacters match a location where a word character is next to a nonword character. `\b` often specifies a word boundary location, and `\B` often specifies a not-word-boundary location. Some implementations provide separate metasequences for start- and end-of-word boundaries, often `\<` and `\>`.

Lookahead: `(?=...)`, `(?!...)`
Lookbehind: `(?<=...)`, `(?<!...)`
: *Lookaround constructs* match a location in the text where the subpattern would match (lookahead), would not match (negative lookahead), would have finished matching (lookbehind), or would not have finished matching (negative lookbehind). For example, `foo(?=bar)` matches `foo` in `foobar`, but not `food`. Implementations often limit lookbehind constructs to subpatterns with a predetermined length.

Comments and mode modifiers

Mode modifiers change how the regular expression engine interprets a regular expression. (See MRE 110–113, 135–136.)

Multiline mode: `m`
: Changes the behavior of `^` and `$` to match next to newlines within the input string.

Single-line mode: `s`
: Changes the behavior of `.` (dot) to match all characters, including newlines, within the input string.

Case-insensitive mode: `i`
: Treat letters that differ only in case as identical.

Free-spacing mode: `x`
: Allows for whitespace and comments within a regular expression. The whitespace and comments (starting with `#` and extending to the end of the line) are ignored by the regular expression engine.

Mode modifiers: `(?i)`, `(?-i)`, `(?mod:...)`
: Usually, mode modifiers may be set within a regular expression with `(?mod)` to turn modes on for the rest of the current subexpression; `(?-mod)` to turn modes off for the rest of the current subexpression; and `(?mod:...)` to turn modes on or off between the colon and the closing parentheses. For example, `use (?i:perl)` matches `use perl`, `use Perl`, `use PeRl`, etc.

Comments: `(?#...)` *and* `#`
: In free-spacing mode, `#` indicates that the rest of the line is a comment. When supported, the comment span `(?#...)` can be embedded anywhere in a regular expression, regardless of mode. For example, `.{0,80}(?#Field limit is 80 chars)` allows you to make notes about why you wrote `.{0,80}`.

Literal-text span: `\Q...\E`
: Escapes metacharacters between `\Q` and `\E`. For example, `\Q(.*)\E` is the same as `\(\.\*\)`.

Grouping, capturing, conditionals, and control

This section covers syntax for grouping subpatterns, capturing submatches, conditional submatches, and quantifying the number of times a subpattern matches. (See MRE 137–142.)

Capturing and grouping parentheses: `(...)` *and* `\1`, `\2`, *etc.*
: Parentheses perform two functions: grouping and capturing. Text matched by the subpattern within parentheses is captured for later use. Capturing parentheses are numbered by counting their opening parentheses from the left. If backreferences are available, the submatch can be referred to later in the same match with `\1`, `\2`, etc. The captured text is made available after a match by implementation-specific methods. For example, `\b(\w+)\b \s+\1\b` matches duplicate words, such as `the the`.

Grouping-only parentheses: `(?:...)`
: Groups a subexpression, possibly for alternation or quantifiers, but does not capture the submatch. This is useful for efficiency and reusability. For example, `(?:foobar)` matches `foobar`, but does not save the match to a capture group.

Named capture: `(?<name>...)`
: Performs capturing and grouping, with captured text later referenced by *name*. For example, `Subject:(?<subject>.*)` captures the text following `Subject:` to a capture group that can be referenced by the name `subject`.

Atomic grouping: `(?>...)`
: Text matched within the group is never backtracked into, even if this leads to a match failure. For example, `(?>[ab]*)\w\w` matches `aabbcc`, but not `aabbaa`.

Alternation: `...|...`
: Allows several subexpressions to be tested. Alternation's low precedence sometimes causes subexpressions to be longer than intended, so use parentheses to specifically group what you want alternated. Thus, `\b(foo|bar)\b` matches the words `foo` or `bar`.

Conditional: `(?(if)then|else)`
: The *if* is implementation-dependent, but generally is a reference to a captured subexpression or a lookaround. The *then* and *else* parts are both regular expression patterns. If the *if* part is true, the *then* is applied. Otherwise, *else* is applied. For example, `(<)?foo(?(1)>|bar)` matches `<foo>` as well as `foobar`.

Greedy quantifiers: `*`, `+`, `?`, `{num,num }`
: The greedy quantifiers determine how many times a construct may be applied. They attempt to match as many times as possible, but will backtrack and give up matches if necessary for the success of the overall match. For example, `(ab)+` matches all of `abababab`.

Lazy quantifiers: `*?`, `+?`, `??`, `{num,num }?`
: Lazy quantifiers control how many times a construct may be applied. However, unlike greedy quantifiers, they attempt to match as few times as possible. For example, `(an)+?` matches only `an` of `banana`.

Possessive quantifiers: `*+`, `++`, `?+`, `{num,num }+`
: Possessive quantifiers are like greedy quantifiers, except that they "lock in" their match, disallowing later backtracking to break up the submatch. For example, `(ab)++ab` will not match `abababab`.

Unicode Support

The *Unicode* character set gives unique numbers to the characters in all the world's languages. Because of the large number of possible characters, Unicode requires more than one byte to represent a character. Some regular expression implementations will not understand Unicode characters because they expect 1 byte ASCII characters. Basic support for Unicode characters starts with the ability to match a literal string of Unicode characters. Advanced support includes character classes and other constructs that incorporate characters from all Unicode-supported languages. For example, `\w` might match `è;` as well as `e`.

Regular Expression Cookbook

This section contains simple versions of common regular expression patterns. You may need to adjust them to meet your needs.

Each expression is presented here with target strings that it matches, and target strings that it does not match, so you can get a sense of what adjustments you may need to make for your own use cases.

They are written in the Perl style:

```
/pattern/mode
s/pattern/replacement/mode
```


Recipes

Removing leading and trailing whitespace

```
s/^\s+//
s/\s+$//
```

Matches: `" foo bar "`, `"foo "`

Nonmatches: `"foo bar"`

Numbers from 0 to 999999

```
/^\d{1,6}$/
```

Matches: `42`, `678234`

Nonmatches: `10,000`

Valid HTML Hex code

```
/^#([a-fA-F0-9]){3}(([a-fA-F0-9]){3})?$/
```

Matches: `#fff`, `#1a1`, `#996633`

Nonmatches: `#ff`, `FFFFFF`

U.S. Social Security number

```
/^\d{3}-\d{2}-\d{4}$/
```

Matches: `078-05-1120`

Nonmatches: `078051120`, `1234-12-12`

U.S. zip code

```
/^\d{5}(-\d{4})?$/
```

Matches: `94941-3232`, `10024`

Nonmatches: `949413232`

U.S. currency

```
/^\$\(d{1,3}(\,\d{3})*|\d+)(\.\d{2})?$/
```

Matches: `$20`, `$15,000.01`

Nonmatches: `$1.001`, `$.99`

Match date: MM/DD/YYYY HH:MM:SS

```
/^\d\d\/\d\d\/\d\d\d\d \d\d:\d\d:\d\d$/
```

Matches: `04/30/1978 20:45:38`

Nonmatches: `4/30/1978 20:45:38`, `4/30/78`

Leading pathname

```
/^.*\//
```

Matches: `/usr/local/bin/apachectl`

Nonmatches: `C:\\System\foo.exe`

(See MRE 190–192.)

Dotted Quad IP address

```
/^(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]
\d|25[0-5])\.
(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]
\d|25[0-5])$/
```

Matches: `127.0.0.1`, `224.22.5.110`

Nonmatches: `127.1`

(See MRE 187–189.)

MAC address

```
/^([0-9a-fA-F]{2}:){5}[0-9a-fA-F]{2}$/
```

Matches: `01:23:45:67:89:ab`

Nonmatches: `01:23:45`, `0123456789ab`

Email

```
/^[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z_+])*@([0-9a-zA-Z][-\w]*
[0-9a-zA-Z]\.)+[a-zA-Z]{2,9}$/
```

Matches: *tony@example.com*, *tony@i-e.com*, `tony@mail.example.museum`

Nonmatches: `.@example.com`, *tony@i-.com*, *tony@example.a*

(See MRE 70.)

Java (java.util.regex)

Java 1.4 introduced regular expressions with Sun's `java.util.regex` package. Although there are competing packages available for previous versions of Java, Sun's is now the standard. Sun's package uses a Traditional NFA match engine. For an explanation of the rules behind a Traditional NFA engine, see "Introduction to Regexes and Pattern Matching." This section covers regular expressions in Java 1.5 and 1.6.

Supported Metacharacters

`java.util.regex` supports the metacharacters and metasequences listed in Table 11 through Table 15. For expanded definitions of each metacharacter, see "Regex Metacharacters, Modes, and Constructs."

Table 11. Java character representations

Sequence	Meaning
`\a`	Alert (bell).
`\b`	Backspace, `\x08`, supported only in character class.
`\e`	Esc character, `\x1B`.
`\n`	Newline, `\x0A`.
`\r`	Carriage return, `\x0D`.
`\f`	Form feed, `\x0C`.
`\t`	Horizontal tab, `\x09`.
`\0`*octal*	Character specified by a one-, two-, or three-digit octal code.
`\x`*hex*	Character specified by a two-digit hexadecimal code.
`\u`*hex*	Unicode character specified by a four-digit hexadecimal code.
`\c`*char*	Named control character.

Table 12. Java character classes and class-like constructs

Class	Meaning
`[...]`	A single character listed or contained in a listed range.
`[^...]`	A single character not liste and not contained within a listed range.
`.`	Any character, except a line terminator (unless `DOTALL` mode).
`\w`	Word character, `[a-zA-Z0-9_]`.
`\W`	Nonword character, `[^a-zA-Z0-9_]`.
`\d`	Digit, `[0-9]`.
`\D`	Nondigit, `[^0-9]`.
`\s`	Whitespace character, `[ \t\n\f\r\x0B]`.
`\S`	Nonwhitespace character, `[^ \t\n\f\r\x0B]`.
`\p{`*prop*`}`	Character contained by given POSIX character class, Unicode property, or Unicode block.
`\P{`*prop*`}`	Character not contained by given POSIX character class, Unicode property, or Unicode block.

Table 13. Java anchors and other zero-width tests

Sequence	Meaning
`^`	Start of string, or the point after any newline if in `MULTILINE` mode.
`\A`	Beginning of string, in any match mode.
`$`	End of string, or the point before any newline if in `MULTILINE` mode.
`\Z`	End of string, but before any final line terminator, in any match mode.
`\z`	End of string, in any match mode.
`\b`	Word boundary.
`\B`	Not-word-boundary.
`\G`	Beginning of current search.

Table 13. Java anchors and other zero-width tests (continued)

Sequence	Meaning
`(?=...)`	Positive lookahead.
`(?!...)`	Negative lookahead.
`(?<=...)`	Positive lookbehind.
`(?<!...)`	Negative lookbehind.

Table 14. Java comments and mode modifiers

Modifier/sequence	Mode character	Meaning
`Pattern.UNIX_LINES`	`d`	Treat `\n` as the only line terminator.
`Pattern.DOTALL`	`s`	Dot (`.`) matches any character, including a line terminator.
`Pattern.MULTILINE`	`m`	`^` and `$` match next to embedded line terminators.
`Pattern.COMMENTS`	`x`	Ignore whitespace, and allow embedded comments starting with #.
`Pattern.CASE_INSENSITIVE`	`i`	Case-insensitive match for ASCII characters.
`Pattern.UNICODE_CASE`	`u`	Case-insensitive match for Unicode characters.
`Pattern.CANON_EQ`		Unicode "canonical equivalence" mode, where characters, or sequences of a base character and combining characters with identical visual representations, are treated as equals.
`(?`*mode*`)`		Turn listed modes (one or more of `idmsux`) on for the rest of the subexpression.
`(?-`*mode*`)`		Turn listed modes (one or more of `idmsux`) off for the rest of the subexpression.
`(?`*mode*`:...)`		Turn listed modes (one or more of `idmsux`) on within parentheses.

Table 14. Java comments and mode modifiers (continued)

Modifier/sequence	Mode character	Meaning
`(?-mode:...)`		Turn listed modes (one or more of `idmsux`) off within parentheses.
`#...`		Treat rest of line as a comment in `/x` mode.

Table 15. Java grouping, capturing, conditional, and control

Sequence	Meaning
`(...)`	Group subpattern and capture submatch into `\1`,`\2`,… and `$1`, `$2`,….
`\n`	Contains text matched by the *n*th capture group.
`$n`	In a replacement string, contains text matched by the *n*th capture group.
`(?:...)`	Groups subpattern, but does not capture submatch.
`(?>...)`	Atomic grouping.
`...|...`	Try subpatterns in alternation.
`*`	Match 0 or more times.
`+`	Match 1 or more times.
`?`	Match 1 or 0 times.
`{n}`	Match exactly *n* times.
`{n,}`	Match at least *n* times.
`{x,y}`	Match at least *x* times, but no more than *y* times.
`*?`	Match 0 or more times, but as few times as possible.
`+?`	Match 1 or more times, but as few times as possible.
`??`	Match 0 or 1 times, but as few times as possible.
`{n,}?`	Match at least *n* times, but as few times as possible.
`{x,y}?`	Match at least *x* times, no more than *y* times, and as few times as possible.
`*+`	Match 0 or more times, and never backtrack.
`++`	Match 1 or more times, and never backtrack.

Table 15. Java grouping, capturing, conditional, and control (continued)

Sequence	Meaning
`?+`	Match 0 or 1 times, and never backtrack.
`{n}+`	Match at least *n* times, and never backtrack.
`{n,}+`	Match at least *n* times, and never backtrack.
`{x,y}+`	Match at least *x* times, no more than *y* times, and never backtrack.

Regular Expression Classes and Interfaces

Regular expression functions are contained in two main classes, `java.util.regex.Pattern` and `java.util.regex.Matcher`; an exception, `java.util.regex.PatternSyntaxException`; and an interface, `CharSequence`. Additionally, the `String` class implements the `CharSequence` interface to provide basic pattern-matching methods. `Pattern` objects are compiled regular expressions that can be applied to any `CharSequence`. A `Matcher` is a stateful object that scans for one or more occurrences of a `Pattern` applied in a string (or any object implementing `CharSequence`).

Backslashes in regular expression `String` literals need to be escaped. So, `\n` (newline) becomes `\\n` when used in a Java `String` literal that is to be used as a regular expression.

java.lang.String

Description

Methods for pattern matching.

Methods

`boolean matches(String regex)`
: Return true if *regex* matches the entire `String`.

`String[ ] split(String regex)`
: Return an array of the substrings surrounding matches of *regex*.

`String [ ] split(String regex, int limit)`
: Return an array of the substrings surrounding the first *limit*-1 matches of *regex*.

`String replaceFirst(String regex, String replacement)`
: Replace the substring matched by *regex* with *replacement*.

`String replaceAll(String regex, String replacement)`
: Replace all substrings matched by *regex* with *replacement*.

java.util.regex.Pattern

Description

Models a regular expression pattern.

Methods

`static Pattern compile(String regex)`
: Construct a `Pattern` object from *regex*.

`static Pattern compile(String regex, int flags)`
: Construct a new `Pattern` object out of *regex*, and the OR'd mode-modifier constants *flags*.

`int flags( )`
: Return the `Pattern`'s mode modifiers.

`Matcher matcher(CharSequence input)`
: Construct a `Matcher` object that will match this `Pattern` against *input*.

`static boolean matches(String regex, CharSequence input)`
: Return true if *regex* matches the entire string *input*.

`String pattern( )`
: Return the regular expression used to create this `Pattern`.

`static String quote(String text)`
: Escapes the text so that regular expression operators will be matched literally.

`String[ ] split(CharSequence input)`
: Return an array of the substrings surrounding matches of this `Pattern` in *input*.

`String[ ] split(CharSequence input, int limit)`
: Return an array of the substrings surrounding the first *limit* matches of this pattern in *regex*.

java.util.regex.Matcher

Description

Models a stateful regular expression pattern matcher and pattern matching results.

Methods

`Matcher appendReplacement(StringBuffer sb, String replacement)`
: Append substring preceding match and *replacement* to *sb*.

`StringBuffer appendTail(StringBuffer sb)`
: Append substring following end of match to *sb*.

`int end( )`
: Index of the first character after the end of the match.

`int end(int group)`
: Index of the first character after the text captured by *group*.

`boolean find( )`
: Find the next match in the input string.

`boolean find(int start)`
: Find the next match after character position *start*.

`String group( )`
: Text matched by this `Pattern`.

`String group(int group)`
: Text captured by capture group *group*.

`int groupCount( )`
: Number of capturing groups in `Pattern`.

`boolean hasAnchoringBounds( )`
: Return true if this `Matcher` uses anchoring bounds so that anchor operators match at the region boundaries, not just at the start and end of the target string.

`boolean hasTransparentBounds( )`
: True if this `Matcher` uses transparent bounds so that lookaround operators can see outside the current search bounds. Defaults to false.

`boolean hitEnd( )`
: True if the last match attempts to inspect beyond the end of the input. In scanners, this is an indication that more input may have resulted in a longer match.

`boolean lookingAt( )`
: True if the pattern matches at the beginning of the input.

`boolean matches( )`
: Return true if `Pattern` matches entire input string.

`Pattern pattern( )`
: Return `Pattern` object used by this `Matcher`.

`static String quoteReplacement(String string)`
: Escape special characters evaluated during replacements.

`Matcher region(int start, int end)`
: Return this matcher and run future matches in the region between *`start`* characters and *`end`* characters from the beginning of the string.

`int regionStart( )`
: Return the starting offset of the search region. Defaults to zero.

`int regionEnd( )`
: Return the ending offset of the search region. Defaults to the length of the target string.

`String replaceAll(String replacement)`
: Replace every match with *`replacement`*.

`String replaceFirst(String replacement)`
: Replace first match with *`replacement`*.

`boolean requireEnd( )`
: Return true if the success of the last match relied on the end of the input. In scanners, this is an indication that more input may have caused a failed match.

`Matcher reset( )`
: Reset this matcher so that the next match starts at the beginning of the input string.

`Matcher reset(CharSequence input)`
: Reset this matcher with new *`input`*.

`int start( )`
: Index of first character matched.

`int start(int group)`
: Index of first character matched in captured substring *`group`*.

`MatchResult toMatchResult( )`
: Return a `MatchResult` object for the most recent match.

`String toString( )`
: Return a string representation of the matcher for debugging.

`Matcher useAnchorBounds(boolean b)`
: If true, set the `Matcher` to use anchor bounds so that anchor operators match at the beginning and end of the current search bounds, rather than the beginning and end of the search string. Defaults to true.

`Matcher usePattern(Pattern p)`
: Replace the `Matcher`'s pattern, but keep the rest of the match state.

`Matcher useTransparentBounds(boolean b)`
: If true, set the `Matcher` to use transparent bounds so that lookaround operators can see outside of the current search bounds. Defaults to false.

java.util.regex.PatternSyntaxException

Description

Thrown to indicate a syntax error in a regular expression pattern.

Methods

`PatternSyntaxException(String desc, String regex, int index)`
: Construct an instance of this class.

`String getDescription( )`
: Return error description.

`int getIndex( )`
: Return error index.

`String getMessage( )`
: Return a multiline error message containing error description, index, regular expression pattern, and indication of the position of the error within the pattern.

`String getPattern( )`
: Return the regular expression pattern that threw the exception.

java.lang.CharSequence

Description

Defines an interface for read-only access so that regular expression patterns may be applied to a sequence of characters.

Methods

`char charAt(int index)`
: Return the character at the zero-based position *index*.

`int length( )`
: Return the number of characters in the sequence.

`CharSequence subSequence(int start, int end)`
: Return a subsequence, including the *start* index, and excluding the *end* index.

`String toString( )`
: Return a `String` representation of the sequence.

Unicode Support

This package supports Unicode 4.0, although `\w`, `\W`, `\d`, `\D`, `\s`, and `\S` support only ASCII. You can use the equivalent Unicode properties `\p{L}`, `\P{L}`, `\p{Nd}`, `\P{Nd}`, `\p{Z}`, and `\P{Z}`. The word boundary sequences—`\b` and `\B`—do understand Unicode.

For supported Unicode properties and blocks, see Table 2. This package supports only the short property names, such as `\p{Lu}`, and not `\p{Lowercase_Letter}`. Block names require the `In` prefix, and support only the name form without spaces or underscores, for example, `\p{InGreekExtended}`, not `\p{In_Greek_Extended}` or `\p{In Greek Extended}`.

Examples

Example 5. Simple match

```
import java.util.regex.*;

// Find Spider-Man, Spiderman, SPIDER-MAN, etc.
public class StringRegexTest {
  public static void main(String[] args) throws Exception {
    String dailyBugle = "Spider-Man Menaces City!";

    //regex must match entire string
```


Example 5. Simple match (continued)

```
    String regex = "(?i).*spider[- ]?man.*";

    if (dailyBugle.matches(regex)) {
      System.out.println("Matched: " + dailyBugle);
    }
  }
}
```

Example 6. Match and capture group

```
// Match dates formatted like MM/DD/YYYY, MM-DD-YY,...
import java.util.regex.*;

public class MatchTest {
  public static void main(String[] args) throws Exception {
    String date = "12/30/1969";
    Pattern p =
      Pattern.compile("^(\\d\\d)[-/](\\d\\d)[-/](\\d\\d(?:\\d\
\d)?)$");

    Matcher m = p.matcher(date);

    if (m.find( )) {
      String month = m.group(1);
      String day   = m.group(2);
      String year  = m.group(3);
      System.out.printf("Found %s-%s-%s\n", year, month, day);
    }
  }
}
```

Example 7. Simple substitution

```
// Example -. Simple substitution
// Convert <br> to <br /> for XHTML compliance
import java.util.regex.*;

public class SimpleSubstitutionTest {
  public static void main(String[] args) {
    String text = "Hello world. <br>";
```

Example 7. Simple substitution (continued)

```
    Pattern p = Pattern.compile("<br>", Pattern.CASE_
INSENSITIVE);
    Matcher m = p.matcher(text);

    String result = m.replaceAll("<br />");
    System.out.println(result);
  }
}
```

Example 8. Harder substitution

```
// urlify - turn URLs into HTML links
import java.util.regex.*;

public class Urlify {
  public static void main (String[  ] args) throws Exception {
   String text = "Check the web site, http://www.oreilly.com/
catalog/regexppr.";
   String regex =
        "\\b                        # start at word\n"
     +  "                           # boundary\n"
     +  "(                          # capture to $1\n"
     +  "(https?|telnet|gopher|file|wais|ftp) : \n"
     +  "                           # resource and colon\n"
     +  "[\\w/\\#~:.?+=&%@!\\-] +?   # one or more valid\n"
     +  "                           # characters\n"
     +  "                           # but take as little\n"
     +  "                           # as possible\n"
     +  ")\n"
     +  "(?=                        # lookahead\n"
     +  "[.:?\\-] *                 # for possible punc\n"
     +  "(?: [^\\w/\\#~:.?+=&%@!\\-] # invalid character\n"
     +  "| $ )                      # or end of string\n"
     +  ")";

    Pattern p = Pattern.compile(regex,
        Pattern.CASE_INSENSITIVE + Pattern.COMMENTS);
    Matcher m = p.matcher(text);
    String result = m.replaceAll("<a href=\"$1\">$1</a>");
    System.out.println(result);
  }
}
```