Typage des ressources Internet

- Types MIME (*Multi-purpose Internet Mail Extension*)
  - RFC 2046. Composé par un type et un sous-type
  - Les types principaux sont les suivants
    - text
    - image
    - audio
    - video
    - message
    - multipart
    - application
  - Ex: text/html, image/gif, application/octet-stream, etc.

Types MIME (2)

- Possibilité de spécifier un paramètre du sous-type
  - Après le sous-type et un point-virgule
  - Par exemple, codage des caractères pour le type texte
    - text/html; charset=iso-8859-1
- Pour les documents composites contenant différents types MIME
  - **multipart** avec **boundary** pour délimiter les parties
    - Sous-types:
      - mixed (par défaut),
      - alternative (la même information dans différents formats)
- La syntaxe des types MIME est ouverte

Les URI

- ***Uniform Resource Identifier*** (RFC 2396)
  - URL (Locator): localisation sur le réseau
  - URN (Name): indépendant de la localisation (site mirroir)
- URI identifie une ressource à partir d'un **schéma**
  - http, ftp, mailto, etc.
  - Attention: tous les URI ne sont pas nécessairement associés à des protocoles de communication
    - Exemple: URN:ISBN:2-7117-8689-7
  - Syntaxe générale:
    - **[<schéma>:]<partie dépendante du schéma>[#fragment]**

Syntaxe dans les URI

- Le schéma
  - doit commencer par une lettre minuscule, puis être composé de lettres minuscules, de chiffres et des signes "+", "." et "-"
- La partie dépendante du schéma
  - Composée de caractères ASCII ou de caractères codés en hexadécimal de la forme "**%xx**"
  - Certains caractères peuvent avoir une interprétation spécifique (ils sont en général traduits automatiquement, si besoin, par les navigateurs). Il s'agit de l'espace et de
    " # < > ` | % { } \ ^ [ ] + ; / ? : @ = &

URI hierarchiques

- Certains URI sont dits «**hiérarchiques**» (par opposition à «**opaques**»): leur partie dépendante du schéma peut être décomposée comme suit:

  [<schéma>:][//<authority>][<path>][?<query>][#<fragment>]

  - **<authority>**: autorité responsable du nommage de la ressource
  - **<path>**: chemin d'accès à la ressource: segments séparés par des "**/**" et suivis d'un ou plusieurs paramètres séparés par des "**;**"
  - **<query>**: chaîne de caractères interprétée par les deux champs précédents
  - Si l'autorité est liée à un serveur, elle a la forme suivante: **[<user-info>@]<machine>[:<port>]**
  - Si port n'est pas spécifié, port par défaut associé au schéma

Schémas http et https

- Schéma **http**
  - http://<machine>:<port>/<path>?<query>
  - Port par défaut **80**
  - <path> peut être interprété comme un chemin d'accès dans un système de fichiers, mais pas forcément
  - <query> informations supplémentaires (paramètres pour l'exécution d'une commande, par exemple)
- Schéma **https**
  - Protocole HTTP au dessus du protocole SSL
  - Même format, mais port par défaut **443**

Schéma ftp

- Transférer un fichier ou lister un répertoire
  - **ftp://<user>:<passwd>@<machine>:<port>/<r1>/<r2>/...**
    .../name;type=<typecode>
  - Si ni **<user>** ni **<passwd>** ne sont spécifiés, équivaut à ftp anonyme avec l'adresse e-mail en passwd
  - Port par défaut **21**
  - Suite **<r1>/<r2>/.../name** correspond au chemin d'accès relatif au répertoire de **<user>**.
  - **;type=<typecode>** est optionnel (typecode vaut "d" pour lister un répertoire, "i" pour binaire, "a" pour ASCII)
  - Exemples : **ftp://ftp.inria.fr/rfc**
    ftp://duris:monmo2pas@igm.univ-mlv.fr/W3/img.gif;type=i

Schéma mailto

- Désigne les boîtes aux lettres, courrier électronique
  - RFC 822
  - Historiquement URL, mais qqfois considéré comme URN
  - **mailto:<adr1>,<adr2>?<header1>=<value1>&<header2>=<value2>**
  - **<adr i>** sont les adresses destinataires du courrier
  - **<header i>=<value i>** sont des en-têtes à placer dans le courrier. Si **<header>** vaut **body**, c'est le corps du message
  - Au moins les en-têtes **body** et **subject** doivent être reconnus
  - Exemples:
    mailto:roussel@univ-mlv.fr
    mailto:roussel@univ-mlv.fr,bedon@univ-mlv.fr?subject=test&body=test

Schéma telnet

- Terminal interactif
- **`telnet://<user>:<passwd>@<machine>:<port>/`**
  - Port par défaut **23**
  - Ouvre un terminal sur la machine spécifiée, avec comme nom d'utilisateur **`<user>`**, à condition que le mot de passe soit valide

Schéma file

- Accès à un système de fichier (en général local)
- **`file://<machine>/<path>`**
  - On n'utilise pas, en général, la partie machine, mais le "**`/`**" du path doit, en principe, être présent
- **`file:///home/duris/COURS/JAVA/http/progJavaHTTP.sxi`**
  - Même pour des fichiers Windows, les séparateurs sont des "**`/`**"
- **`file:///C:/Program%20Files/`**

Les URI relatifs

- Concerne les URI hiérarchiques
  - Parler d'une ressource « relativement » à un URI de base. Forme la + complète d'un URI relatif:
  - **`//<net_loc>/<path1>/.../<pathn>;<params>?<query>`**
  - Chacun des préfixes peut être omis et sera remplacé par l'URI de base (s'il est moins précis)
  - **`http://igm.univ-mlv.fr/~roussel/test/index.html`**
  - **`toto.html`** signifie alors **`http://igm.univ-mlv.fr/~roussel/test/toto.html`**
  - **`/~duris/index.html`** signifie alors **`http://igm.univ-mlv.fr/~duris/index.html`**
  - **`//etudiant.univ-mlv.fr`** signifie alors **`http://etudiant.univ-mlv.fr/`**

Le protocole HTTP

- (*Hyper Text Transfert Protocol*) RFC 2616 (version 1.1)
  - Protocole **sans état**, **rapide**, **léger**, de niveau application permettant le **transfert de données** au dessus d'une **connexion** de type TCP (sure)
    - Utilise les URLs
  - Port par défaut **80**
  - Basé sur le paradigme de **requête / réponse**
    - Chaque requête et chaque réponse est un « **message HTTP** »
  - Largement utilisé sur le World Wide Web

Principe Requête/Réponse

- Connexion TCP établie entre un client et le port 80 (en général) du serveur HTTP
- Envoi par le client d'un message HTTP, dit *requête*
- Traitement de cette requête et envoi par le serveur d'un message, dit *réponse*, au client.
- En HTTP 0.9 et 1.0, la connexion est fermée par le serveur après chaque réponse
- En HTTP 1.1, il est possible de conserver la même connexion TCP pour plusieurs échanges de requêtes/réponses.

Format des messages HTTP

- **`Ligne de début`**
  `Champ d'en-tête_1: valeur_1`
  `...`
  `Champ d'en-tête_n: valeur_n`
  `¶` *// ligne vide « \r\n »*
  `[ corps éventuel ]`
  - La présence de zéro, un ou plusieurs champs d'en-tête permet de spécifier des paramètres
  - La présence d'un corps de message dépend de la ligne de début de ce message

Message de requête

- Constitution de la première ligne de requête
  - ***`Méthode Ressource Version`***
    Exemple:
    ```
    GET /index.html HTTP/1.1
    Host: www.univ-mlv.fr
    ```
  - ***Méthode***: `GET, HEAD, POST, PUT, TRACE,...`
    - Représente le type d'opération à effectuer
  - ***Ressource***: identifie la ressource recherchée
    - HTTP/1.1 exige que le champ « *Host* » soit spécifié
  - ***Version***: identifie la version du protocole utilisée
    - Si rien n'est spécifié, la version HTTP 0.9 est considérée

Message de réponse

- Constitution de la ligne dite « **de statut** »
  - ***`Version Code Message`*** Exemple:
    ```
    HTTP/1.1 200 OK
    Server: Netscape-Enterprise/4.1
    Date: Wed, 18 Sep 2002 09:30:24 GMT
    ...
    Content-type: text/html
    Content-length: 13298
    ¶
    <!DOCTYPE HTML PUBLIC ....
    <html><head>
    ...
    </html>
    ```

Les codes de statut

- Organisés en 5 catégories
  - **1xx « informatifs »**: acceptés uniquement dans les versions 1.1. Réponse « intermédiaire » du serveur.
    Ex: `100 Continue`
  - **2xx « de succès »**: la réception, la compréhension ou l'acceptation de la requête est réussie. Ex: `200 OK`
  - **3xx « de redirection »**: d'autres actions sont requises pour traiter la requête. Ex: `301 Moved Permanently`
  - **4xx « erreurs du client »**: erreur de syntaxe ou de spécification.
    Ex: `404 Not Found`
  - **5xx « erreurs du serveur »**: serveur incapable de répondre.
    Ex: `501 Method Not Implemented`

Quelques champs d'en-tête

- Spécifiques (requêtes/réponses) ou bien généraux:
- **Server** (réponse): informations sur le serveur HTTP
- **Date** (général): estampille les messages HTTP
- **Last-modified** (serveur): date de dernière modification
- **ETag** (): identifiant de la ressource géré par le serveur
- **Content-type** (général): type le corps du message
- **Content-length** (général): nombre d'octets du corps
- **Accept: text/plain** (requête): ce qu'accepte le client
- **Accept-charset**, **Accept-language**, **Accept-encoding**
  Content-encoding, **Transfert-encoding...**

Quelques requêtes

- `GET`: récupérer une ressource
- `HEAD`: identique à `GET`, mais sans corps de msg
  - Peut servir à savoir si une ressource existe et si elle a changé, sans nécessairement la récupérer
- `TRACE`: écho de la requête
  - le serveur renvoit, dans le corps du message, le message qu'il a reçu du client (diagnostique)
- `OPTIONS`: demande d'information sur les méthodes
  - Le serveur renvoit toutes les méthodes qu'il implante dans le champ d'en-tête « `Allow` »

Connexion persistante ou "keep alive"

- Avant 1.1, champ **`Content-length`** pas obligatoire:
  - Comme le serveur fermait la connexion à la fin de la réponse, ce champ servait simplement à vérifier que OK
- Avec la **connexion persistante** (***keep alive***)
  - Serveur peut garder la connexion après la fin de réponse
  - Plus efficace (délais d'ouverture et fermeture de connexion)
  - Possibilité d'exploiter le full-duplex de la connexion TCP (plusieurs transactions peuvent se recouvrir)
- Toutes les réponses doivent contenir **`Content-length`** (sauf réponses sans corps de message ou `chunked`)
  - Indispensable pour déterminer la fin d'un message

```
% telnet www.w3.org 80
Trying 193.51.208.67...
Connected to www.w3.org.
Escape character is '^]'

GET /Overview HTTP/1.1
Host: www.w3.org

HTTP/1.1 200 OK
Date: Sun, 30 Jun 2002 15:33:43 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.9
Last-Modified: Sat, 29 Jun 2002 16:08:59 GMT
ETag: "2d2c96-2c69-3778ef9b"
Accept-Ranges: bytes
Content-Length: 11369
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC ...
<html lang="en"> ... le code du document html ...
</html>
```

La connexion n'est pas fermée...


```
HEAD /Overview HTTP/1.1
Host: www.w3.org

HTTP/1.1 200 OK
Date: Sun, 30 Jun 2002 15:33:58 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.9
Last-Modified: Sat, 29 Jun 2002 16:08:59 GMT
ETag: "2d2c96-2c69-3778ef9b"
Accept-Ranges: bytes
Content-Length: 11369
Content-Type: text/html; charset=iso-8859-1

OPTIONS / HTTP/1.1
Host: www.w3.org
Connection: close

HTTP/1.1 200 OK
Date: Sun, 30 Jun 2002 15:34:17 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.9
Content-Length: 0
Allow: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, PATCH,
  PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK, TRACE
Connection: close

Connection closed by foreign host.
%
```

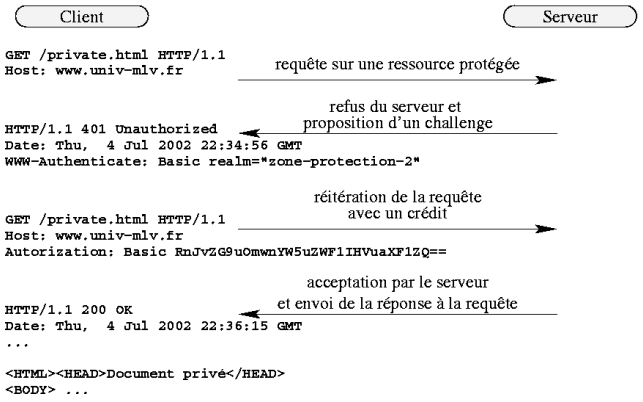

L'authentification

- Pour limiter l'accès aux ressources à qq clients
  - Mécanisme de *challenge/crédit* pour l'authentification des requêtes
  - Utilise les champs **`WWW-Authenticate`** et **`Autorization`**
- Lors d'une requête sur une ressource protégée
  - Serveur répond **`401 Unauthorized`** avec, un champ **`WWW-Authenticate`** qui contient au moins un *challenge* applicable à la ressource concernée (une liste)
    - Un challenge est constitué d'un schéma (mot, ex: Basic), suivi de paires attribut=valeur. Exemple:
      `Basic realm= "zone-protection-2"`

L'authentification (suite)

- Le client doit alors réémettre sa requête avec un *crédit* dans le champ **`Autorization`**. Exemple:
  - **`Autorization: Basic blablabla`**
    où **`blablabla`** représente le codage en base 64 de la chaîne « login:passwd »
  - Attention, base 64 est réversible, et le crédit passe en clair
- Autre schéma: ***Digest***
  - Requiert une valeur d'autorisation obtenue par hachage cryptographique utilisant, en plus de user et passwd de client, des informations liées à la requête et une valeur aléatoire tirée par le serveur

Authentification (exemple)



Envoyer des données

- Une requête peut servir à envoyer des informations vers le serveur HTTP
  - Méthode POST, dans le corps du message, mais aussi
  - Méthode GET, dans l'URL sous la forme de "*query string*": permet de transmettre quelques arguments
- Ces données sont principalement utilisées pour "générer" une ressource dynamiquement
  - Utilise des "programmes" satellites au serveur
  - CGI, Servlettes/JSP, PHP, ASP...

Les formulaires HTML (GET)

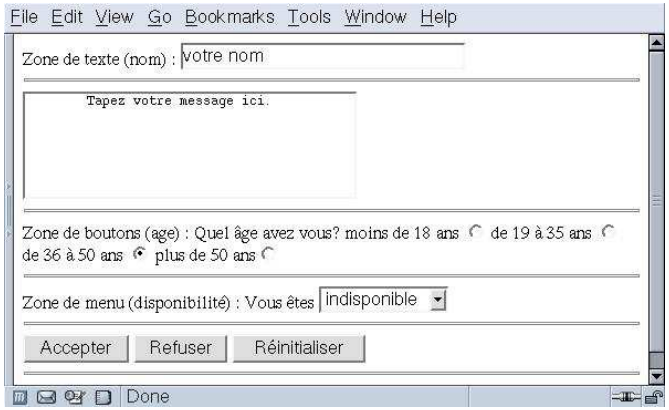
- Permet de définir des zones de saisie pour le client
  - <form enctype="application/x-www-form-urlencoded"
    action="http://www.serv.fr/path"
    method="GET">
    <input type="text" name="user" value="votre nom">
    <input type="submit" name="ok" value="Envoyer">
    </form>

  - Un clic sur Envoyer crée une requête GET à l'URL suivant: http://www.serv.fr/path?user=Etienne+Duris&ok=Envoyer

Les formulaires HTML (POST)

- <form action="http://www.serv.fr/path" method="**POST**">
  <input type="text" name="user" value="votre nom">
  <input type="submit" name="ok" value="Envoyer">
  </form>
- Même apparence dans un navigateur
- Un clic sur Envoyer crée une requête POST
  - à l'URL http://www.serv.fr/path,
  - Avec comme champs d'en-tête
    Host: www.serv.fr
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 25
  - dont le **corps du message** contient
    user=votre+nom&ok=Envoyer

Formulaire plus riche (illustration)




```
<form enctype="application/x-www-form-urlencoded"
  action="http://www.serv.fr/path" method="GET">
  Zone de texte (nom) :
  <input type="text" size="30" name="user" value="votre nom"><hr>
  <textarea name="srctext" rows="6" cols="40">
   Tapez votre message ici.
  </textarea><hr>
  Zone de boutons (age) : Quel âge avez vous?
  moins de 18 ans <input type="radio" name="age" value="-18">
  de 19 à 35 ans <input type="radio" name="age" value="19-35">
  de 36 à 50 ans <input type="radio" name="age" value="36-50" checked>
  plus de 50 ans<input type="radio" name="age" value="50-"><hr>
  Zone de menu (disponibilité) : Vous êtes
  <select name="dispo">
   <option value="ok">disponible
   <option selected value="ko">indisponible
  </select><hr>
  <input type="hidden" name="client" value="a1234-567-8901">
  <input type="submit" name="choix1" value="Accepter">
  <input type="submit" name="choix2" value="Refuser">
  <input type="reset" value="Réinitialiser">
  <hr>
</form>
```


Corps de message « par morceaux »

- Lorsqu'une requête génère une réponse longue
  - On devrait attendre toute la génération de la réponse afin de connaître sa taille AVANT de créer l'en-tête de la réponse et son champ **`Content-Length`**
  - On peut alors « découper » la réponse en morceaux (***chunked***) et commencer à envoyer les premiers avant d'avoir fini de générer la totalité du document
  - Utilise le champ **`Transfert-Encoding: chunked`**
  - Il n'y a plus de champ d'en-tête **`Content-length`**
  - Chaque morceau est préfixé par sa taille
  - Le premier morceau de taille nulle indique la fin du corps

Chunked (exemple)

```
GET /search?q=test HTTP/1.1
Host: www.google.fr

HTTP/1.1 200 OK
Server: GWS/2.0
Date: Mon, 23 Sep 2002 11:58:56 GMT
Transfer-Encoding: chunked
Content-Type: text/html

b3f
<html><head>
..... Premier morceau ...
3ce2
..... Deuxième morceau ...
</body></html>
0
```

Envoyer des fichiers

- Si le formulaire spécifie un nom de fichier, c'est le contenu du fichier qu'on désire envoyer
  - le type (enctype) ne doit alors plus être "**application/x-www-form-urlencoded**", mais "**multipart/form-data**", introduit en HTML 4 pour cela

Envoyer des fichiers (code)

```
<form enctype="multipart/form-data"
   action="/sendfile" method="POST">
   Fichier 1 : <input type="file"
               name="fichier1" size="20"><br><br>
   Fichier 2 : <input type="file"
               name="fichier2" size="20"><br><br>
   <input type="hidden" name="ID" value="roussel">
   <input type="submit" name="submit" value="Envoyer">
</form>
```

- Supposons que le fichier contienne le texte suivant:
  - Test de fichier
- Alors la requête générée par un click sur Envoyer sera la suivante...


```
POST /sendfile HTTP/1.1
Host: server
Content-Type: multipart/form-data; boundary=ABCDEF
Content-Length: 372

--ABCDEF
Content-Disposition: form-data; name="fichier1"; filename="test.txt"
Content-Type: text/plain

Test de fichier

--ABCDEF
Content-Disposition: form-data; name="fichier2"; filename=""
Content-Type: application/octet-stream


--ABCDEF
Content-Disposition: form-data; name="ID"

roussel
--ABCDEF
Content-Disposition: form-data; name="submit"

Envoyer
--ABCDEF--
```


Requête GET ou POST

- Les deux méthodes sont utilisables dans un formulaire
- Différences principales
  - POST transmet les données dans le corps
  - GET transmet les données dans l'URL
  - POST + Ésthétique (arguments n'apparaissent pas)
  - POST non limité en taille (les serveurs limitent une taille maximale pour les URLs). Dans ce cas, code de statut `414 Requested-URI Too Large`
  - Un formulaire GET peut être « rejoué » en conservant l'URL (bookmarks du navigateur)

D'autres requêtes

- PUT permet d'envoyer au serveur HTTP des données (fichiers) qu'il devra stocker
  - Différent de POST, dont les données sont destinées à une ressource identifiée par l'URL (qui doit normalement interpréter les données)
  - Dans le cas de PUT, les données **sont** la ressource à placer à l'endroit spécifié par l'URL
  - Mise à jour (`200 OK`) ou création (`201 Created`)
- DELETE permet de supprimer des données
- Requièrent le droit d'accès (authentification)

Les relais

- Modèle client/serveur trop simpliste
  - Beaucoup de machines intermédiaires qui jouent le rôle de serveur pour le client et de client pour le serveur
  - Peuvent également stocker temporairement des réponses pour les « rejouer » à la prochaine requête identique: ***proxy cache***
  - Un client peut spécifier explicitement (statiquement) qu'il veut utiliser un proxy
  - Nombreux champs d'en-tête spécifiques
  - `Proxy-Authenticate, Proxy-Authorization, Via, Max-Forwards, Cache-Control (Pragma: no-cache), Vary...`

Session Tracking

- HTTP protocole sans état. Comment savoir quelle était la précédente requête du même client?
- Idée: faire « sauvegarder un état » par le client, qu'il devra renvoyer dans la prochaine requête
  - Champs « `hidden` » des formulaires
  - **Réécriture des URL** des liens hypertextes contenus dans les documents retournés. Exemple:
    `<a href="http://serveur/request_for_client_03728612">`
    Ne requiert aucune participation du client
  - **Cookies** (plus simple pour le serveur)
    introduit par Netscape, puis RFC 2109
    Requiert la participation du client

Cookies

- C'est une paire <clé>=<valeur>
  - Initialement fournie par le serveur dans une réponse
  - Stockée par le client (navigateur)
  - Puis retransmise par le client lorsqu'il émet des requêtes
- Spécifié par un champ d'en-tête dans la réponse
  - `Set-Cookie: <clé>=<valeur> [; expires=<DATE>] [; path=<PATH>] [; domain=<DOM>] [; secure]`
- Rejoué par les clients:
  - `Cookie: <clé1>=<valeur1>; <clé2>=<valeur2> ...`

Cookies (suite)

- Attributs optionnels
  - **expires**: pour la date de péremption. Au delà, le client ne doit pas conserver le cookie
  - **domain** et **path** servent à déterminer les URL pour lesquels ce cookie doit être utilisé
    (par défaut, ceux de l'URL de la requête sont utilisés)
    - Les cookies dont la valeur de domain est différente du domaine du serveur ayant généré la réponse doivent être ignorés
    - Les cookies trop larges (moins de deux « . ») ignorés
  - **secure**: cookie à n'utiliser que si connexion sécurisée
  - Si même path et même domain, le nouveau écrase l'ancien. Si path diffère, on stocke et renvoit les deux.

java.net.URI

- Depuis j2sdk1.4: manipulation syntaxique des URI
  - Neuf composants répertoriés sur un URI
    - **scheme** String
    - **scheme-specific-part** String
    - **authority** String
    - **user-info** String
    - **host** String
    - **port** int
    - **path** String
    - **query** String
    - **fragment** String

Classe URI

- Constructeur acceptant la chaîne URI en argument
  - Vérifie la syntaxe des URI (RFC 2396):
  - URI u1 = new URI("http://igm.univ-mlv.fr");
  - URI u2 = new URI("nimporte:quoi");
  - URI u3 = new URI("relatif");
  - URI u4 = new URI("\\\interdit");
    // lève une **URISyntaxException**
- Méthode statique create(String uri) identique
  - Mais lève une **IllegalArgumentException** qu'on a pas besoin de récupérer ni de propager (RuntimeException).

Classe URI (suite)

- Quatre autres constructeurs spécifiant des parties
  - **URI (String schema, String user, String machine, int port, String path, String query, String fragment)**
    Ex:URI("ftp", "user", "www.test.fr", 21, "/d/file.txt", "param=val", "ancre") qui correspond à ftp://user@www.test.fr:21/d/file.txt?param=val#ancre
  - **URI (String schema, String autorité, String path, String query, String fragment)**
    Ex:URI("ftp", "user@www.test.fr:21", "/d/file.txt", "param=val", "ancre")
  - **URI(String schema, String partieDépendante, String fragment)**
    Ex:URI("ftp", "user@www.test.fr:21/d/file.txt?param=val", "ancre")
  - **URI(String schema, String machine, String path, String fragment)**
    équivalent à un appel au premier constructeur avec URI(schéma, null, machine, -1, path, fragment)
- Plus généralement, null ou -1 pour un champ absent

Classe URI (spécificités)

- Principal avantage: permet d'utiliser des caractères Unicode en plus des caractères ASCII
  - **URI ipv6 = new URI("sch","1080::02C:417A","/mon fichier",null);**
    System.out.println(ipv6);
    // Affiche: sch:[1080::02C:417A]/mon%20fichier
  - "Analyse" des parties de l'URL, w.r.t le constructeur
    URI bad1 = new URI ("sch://500.1.2.3/");
    System.out.println(bad1); // Affiche: sch://500.1.2.3/
    l'authorité de nommage pourrait être considérée comme non liée à un serveur
  - **URI bad2 = new URI ("sch", "500.1.2.3", "/", null);**
    System.out.println(bad2);
    // lève: URISyntaxException: Malformed IPv4 address...
    ici, le 2nd argument est (syntaxiquement) une adresse IP

Classe URI (accesseurs)

- Découpage d'un objet URI en « morceaux »
  - getScheme(), getUserInfo(), getHost(), getAuthority(), getPort(), getPath(), getQuery(), getFragment()
  - isOpaque(), getSchemeSpecificPart()
  - Méthode parseServerAuthority() vérifie que l'autorité correspond à des informations relatives à un serveur
    - Sinon lève **URISyntaxException**
  - get**Raw**UserInfo(), get**Raw**Authority(), get**Raw**Path(), etc... donnent les champs avant décodage Unicode
  - getASCIIString() donne cet URI en US-ASCII

Classe URI (relatifs et absolus)

- Dans le cas d'un URI représentant un URI relatif, toutes les méthodes correspondant à une partie manquante retournent null
- Méthode **resolve(URI relatif)**
  - Retourne l'URI absolu correspondant à l'URI realtif passé en argument, résolu par rapport à l'URI de base sur lequel la méthode est appelée
  - Existe avec un argument de type **String**
- Méthode **relativize(URI uri)**
  - Retourne l'URI relatif qui représente **uri** par rapport à l'URI de base sur lequel est appelé la méthode
- Méthode **normalize()** élimine les parties inutiles
  - http://machine/dir/./subdir/../file.html => http://machine/dir/file.html

Classes URI et URL

- Instance de la classe URI
  - Référence à la RFC 2396, au sens syntaxique du terme.
  - Peut être absolu ou relatif
  - Analysé selon la syntaxe générique, indépendamment du schéma.
  - Pas de résolution du nom de l'hôte, ni gestionnaire de schéma.
  - Égalité, hachage et comparaison sur les caractères décrivant l'URI.
- Instance de la classe URL
  - Composants syntaxiques + informations pour l'accès à la ressource.
  - Doit être absolu, c'est-à dire spécifier un schéma.
  - Analysé en fonction du schéma.
  - Donne lieu à l'établissement d'un gestionnaire de schéma associé.
  - Ne peut pas créer d'URL pour un schéma dont on a pas le gestionnaire.
  - Égalité, hachage et comparaison dépendent du schéma et de l'hôte.

Classe URL

- Historiquement antérieure à URI
  - Ne respecte pas rigoureusement la terminologie des URI (RFC 2396)
  - Interprétation restricitve (ex: schéma = protocole, path = file)
  - **Aucun « codage »** sur les morceaux de l'URL
- Comment contruire un URL (**MalformedURLException**)
  - **URL url = new URI(...).toURL();**
- Six constructeurs
  - **URL(String spec)**
  - **URL(String scheme, String host, int port, String path)**
  - **URL(String scheme, String host, String path)**
  - **URL(URL context, String spec)**
  - **URL(URL context, String spec, URLStreamHandler)**
  - **URL(String sch, String host, int port, String path,URLStreamHandler)**

URL: accès aux caractéristiques

- Méthodes d'accès aux différentes parties de l'URL
  - getProtocol(), getAuthority(), getHost(), getPort(), getDefaultPort(), getFile() (avec <query>), getPath() (sans <query>), getQuery(), getRef()
  - toExternalForm() retourne la forme canonique de l'URL
- Deux méthodes de comparaison
  - equals() retourne true si les 2 URL sont différents de null, et référencent exactement la même ancre de la même ressource sur la même machine
  - sameFile() identique mais retourne true même si les ancres diffèrent

Comparaison entre URIs / entre URLs

- La classe URI ne s'occupe « que» des caractères

```
URI uri1 = new URI("http://igm.univ-mlv.fr/~duris/index.html");
URI uri2 = new URI("http://monge.univ-mlv.fr/~duris/index.html");
System.out.println(uri1.equals(uri2));          // Affiche: false
```

- La classe URL « analyse » la localisation du contenu

...plus particulièrement la machine et le port

```
URL url1 = new URL("http://igm.univ-mlv.fr/~duris/index.html");
URL url2 = new URL("http://monge.univ-mlv.fr/~duris/index.html");
URL url3 = new URL("http://193.55.63.80:80/~duris/index.html");
System.out.println(url1.equals(url2));          // Affiche: true
System.out.println(url1.equals(url3));          // Affiche: true
```


URL: accès au contenu

- La méthode getContent() retourne un objet correspondant à la ressource
  - URL u = new URL("http://igm.univ-mlv.fr/~duris/logo.gif");
    System.out.println(u.getContent());
    // Affiche: sun.awt.image.URLImageSource@a0dcd9
- Récupérer un flot de lecture sur cette ressource
  - InputStream openStream()
    - Pas toujours très pertinent
- Ces deux méthodes sont des raccourcis de
  - openConnection().getContent()
  - openConnection().getInputStream()

Classe URLConnection

- **Gestionnaire de connection**
- Classe abstraite commune à tous les objets qui gèrent les connexions vers des ressources référencées par des URL
  - Méthodes « indépendantes » du schéma
  - Récupération d'attributs de la connexion
  - Passage d'attributs à la connexion
- En général, une instance de **URLConnection** est créée par un appel à **openConnection()** sur un objet **URL**

Cycle de vie de URLConnection

- 1. L'objet représentant la connexion est créé par invocation de la méthode openConnection() sur un URL donné
- 2. Les paramètres de cette (future) connexion peuvent être manipulés
- 3. La connexion est réellement établie par un appel à la méthode **connect()**
- 4. L'objet (la ressource) distant(e) peut être accédé(e) ainsi que les champs d'en-tête associés à cette connexion

Manipuler les paramètres de connexion

- Les paramètres de la connexion doivent être fixés avant qu'elle soit effectivement établie
  - setRequestProperty(String key, String value)
  - getRequestProperty(String key)
  - Méthodes dédiées à certains paramètres: doInput, doOutput, allowUserInteraction, ifModifiedSince, useCaches...
  - Les méthodes setDefaultAllowUserInteraction() et setDefaultUseCaches() donnent des valeurs par défaut au paramètre correspondant pour toutes les futures connexions

Paramètres de connexion: exemple

-
```
// Création de l'url
URL url = new URL("http://www.univ-mlv.fr/index.html");
// Création de l'objet connexion
URLConnection uc = url.openConnection();
// spécification de paramètres pour la connexion
uc.setRequestProperty("Connection","close");
SimpleDateFormat f = new SimpleDateFormat("dd/MM/yy zzzz");
Date d = f.parse("01/01/2002 GMT", new ParsePosition(0));
uc.setIfModifiedSince(d.getTime());
```
- Génère les en-têtes suivants dans la requête
  Connection: close
  If-Modified-Since: Tue, 01, Jan 2002 00:00:00 GMT
- Et d'autres, par défaut, dont
  User-Agent: Java/1.5.0

Utiliser la connexion

- Commencer par appeler connect()
  - La plupart des méthodes suivantes font implicitement un appel à connect() s'il n'a pas déjà été fait
- Exploitation de la connexion
  - getContent() donne la ressource sous la forme d'objet
  - getHeaderField() donne la valeur d'un champ d'en-tête
  - getInputStream() flot de lecture sur le contenu
  - getOutputStream() flot d'écriture sur la connexion

Récupération des champs d'en-tête

- getHeaderField() est décliné pour certains champs d'en-tête spécifiques (valeurs de retour de type String ou int):
  - getContentType(), getContentEncoding(), getContentLength(), getDate(), getExpiration(), getLastModified()
- Méthodes utilitaires
  - int getHeaderFieldInt(String key, int defaut)
    long getHeaderFieldDate(String key, long defaut)
    String getHeaderFieldKey(int n)
    String getHeaderField(int n)

Sous-classes de URLConnection

- **HttpURLConnection** (classe abstraite)
  - Méthodes d'accès spécifiques au protocole HTTP, ex:
  - setRequestMethod() spécifie la méthode de requête
  - getResponseCode() et getResponseMessage() donnent le code et le message de réponse (constantes définies)
  - setFollowRedirects() pour suivre les redirections (3xx) pour les instances de cette classe de gestionnaire, et setInstanceFollowRedirects() pour une instance particulière
  - disconnect() demande la déconnexion si *keep alive*
  - usingProxy() permet d'indiquer le passage par un relais

Sous-classes de URLConnection (2)

- **JarURLConnection** (classe abstraite)
  - Accès à une archive Java (Jar) locale ou distante
  - URL de la forme **jar:<url>!/<entry>** par exemple: jar:http://www.server.org/archive.jar!/dir/TheClass.class
  - Méthodes d'accès aux différentes parties et aux différents attributs des entrées de l'archive
  - Permet de récupérer des objets de la classe java.util.jar.JarEntry
  - Permet d'obtenir un flot de lecture sur l'entrée spécifiée par l'URL

Authentification

- Concerne les champs d'en-tête Authorization ou Proxy-Authorization lors de réponses
  - 401 Unauthorized ou 407 Proxy Authentication Required.
- Classe abstraite Authenticator
  - Gestionnaire d'authentification, installé par la méthode statique setDefault(Authenticator a)
  - Une fois installé, chaque requête reçevant une réponse 401 ou 407 fait appel, via l'objet URL, à la méthode statique requestPasswordAuthentication() qui appelle la méthode protégée getPasswordAuthentication() qui retourne un objet de classe PasswordAuthentication

Authentification (2)

- Classe PasswordAuthentication représente
  - Un identificateur (String)
  - Un mot de passe (char [])
- Par défaut getPasswordAuthentication() retourne null
  - Redéfinir cette méthode dans une sous classe concrète de Authenticator d'une méthode
  - Utiliser les méthodes getRequestingSite() (machine), getRequestingProtocol(), getRequestingScheme(), getRequestingPrompt() (realm du challenge), getRequestingPort() de la classe Authenticator
  - L'objet URL place ces valeurs dans les champs d'en-tête

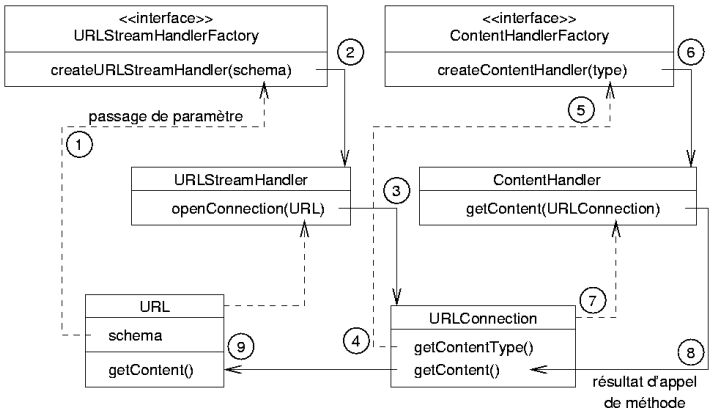
Gestionnaires

- Architecture ouverte autour des URL pour ajouter:
  - De nouveaux schémas d'URL
  - De nouveaux types de ressources reconnus
  - Ses propres implantations des gestionnaires existants
- Les classes d'objets concernés sont:
  - Gestionnaires de schémas (URLStreamHandler)
    - Analyse de l'URL et création du gestionnaire de connexion
  - Gestionnaires de connexion (URLConnection)
    - Gérer l'accès à la ressource et à ses attributs *via* un protocole
  - Gestionnaires de contenus (ContentHandler)
    - Transforme le flot sur la ressource en un objet Java

Gestionnaires par défaut

- Gestionnaire de schéma créé dynamiquement
  - à la première création d'URL ayant ce schéma <s>
  - sun.net.www.protocol.<s>.Handler
  - MalformedURLException si aucune classe ne correspond
- Gestionnaire de schéma détermine la classe des gestionnaires de connexion
  - openConnection() retourne un gestionnaire de connexion
- Gestionnaire de contenu créé par gestion de connexion
  - en fonction du type MIME <type>/<soustype> de la ressource
    - à sa première rencontre, lorsque getContent() est appelée
  - sun.net.www.content.<type>.<soustype>
    - UnknownServiceException si aucune classe trouvée de ce nom

Mécanisme gestionnaires



Mise en place de ses propres gestionnaires

- Pour utiliser son propore gestionnaire de schéma
  - Le passer en argument du constructeur de la classe URL
  - Sinon, installer une fabrique, implantant l'interface URLStreamHandlerFactory, par la méthde setURLStreamHandlerFactory() de la classe URL.
  - Elle retourne l'objet URLStreamHandler correspondant au protocole qu'elle sait traiter, ou sinon doit retourner null pour que le schéma par défaut s'applique
  - Sa méthode openConnection() retourne un gestionnaire de connexion spécifique, sous-classe de URLConnection
- Même principe existe pour les gestionnaires de contenu
  - URLConnection uc = ...;
    uc.setContentHandlerFactory(ContentHandlerFactory chf);

Applettes

- Délocaliser l'exécution de programmes sur le client
- Spécifié dans une page HTML (balises <applet>)
- C'est le navigateur du client qui exécute le bytecode
- Participe à l'*équilibrage de charge (load balancing)*
- Utilise les mécanismes de chargement de classe à distance
  - URLClassLoader
  - Permet de trouver et charger une classe parmi un ensemble d'URL de base, locaux ou distants

Chargement de classe

- Pour utiliser une classe (ou interface), la machine virtuelle doit avoir *chargé* son bytecode
- C'est un chargeur de classe qui s'en occupe
  - Classe abstraite java.lang.ClassLoader
- A toute classe A est associé un chargeur
  - Celui-ci est responsable du chargement des classes qui sont utilisées par A, et transitivement...
  - Possibilité de déléguer le chargement à un autre chargeur
  - Possibilité de demander si la classe n'a pas déjà été chargée

Chargement de classe (2)

- Pour la machine virtuelle, chaque classe chargée est identifiée par:
  - Son nom complet (avec package, ex: java.lang.String)
  - Son chargeur (celui qui a effectué le chargement du bytecode)
  - IMPORTANT: deux classes différentes de même nom complet peuvent cohabiter dans la machine virtuelle si elles n'ont pas été chargées par le même chargeur
- En plus, une politique de sécurité est associée à chacune de ces classes chargées
  - Idée: même degré de sécurité pour des classes provenant du même endroit

Chargeur de bootstrap

- Les objets chargeurs de classes sont des instances d'une sous-classe de ClassLoader
  - Le bytecode de cette sous-classe doit avoir été chargé...
  - Ce mécanisme récursif se termine par un chargeur de classe particulier dit ***bootstrap classloader***
- La méthode Class getClassLoader() appliquée sur un objet chargé par le bootstrap classloader retourne null
  - Sinon, elle retourne l'instance de ClassLoader qui a chargé la classe ayant créé cet objet

Trois groupes de classes

- Les classes de bootstrap
  - Nécessaires à l'environnement d'exécution standard
  - Distribuées par Sun dans la plate-forme standard et supportées dans le futur. Ex: les classes de **rt.jar**
  - Fichiers d'archives spécifiés par sun.boot.class.path
- Les classes de bibliothèque d'extensions standard
  - Fichiers d'archives spécifiés par java.ext.dirs
- Les classes définies par l'utilisateur
  - Par défaut, seul le répertoire courrant, mais on peut spécifier avec l'option -classpath ou variable d'environnement CLASSPATH (java.class.path)

La classe ClassLoader

- Par défaut, une classe est d'abord recherchée parmi les classes de bootstrap, puis parmi les classes d'extension et enfin seulement parmi les classes utilisateur
- Les chargeurs de classes sont organisés hiérarchiquement => notion de délégation
  - A chaque demande de chargement, un chargeur de classe délègue le chargement de la classe à son chargeur de classe parent, avant de tenter de le faire réellement lui-même en cas d'échec du chargeur parent

Les méthodes de ClassLoader

- Class loadClass(String name)
  - Demande au chargeur d'initier le chargement d'une classe (via des délégations éventuelles)
- Class findLoadedClass(String name)
  - Demande au chargeur s'il a déjà réellement chargé une classe
- Class findClass(String name)
  - Demande au chargeur de charger réellement une classe (sans aucune délégation à un autre chargeur)
- ClassLoader getParent()
- void resolveClass(Class c)
  - Prépare l'instance c pour son utilisation prochaine, fait des vérifications de bytecode, prépare et alloue des structures, réalise l'édition de liens

Schéma de délégation

- Le chargeur de classe, via sa méthode loadClass(), commence par rechercher, grâce à la méthode findLoadedClass(), s'il n'a pas déjà chargé la classe.
- S'il ne l'a pas déjà réellement chargée, il tente de déléguer le chargement au chargeur de classe parent grâce à getParent()
- Si le chargeur de classe parent n'a pas trouvé la classe, alors le chargeur courant appelle la méthode findClass()
- En cas d'échec au final, une exception ClassNotFoundException est levée

Pseudo-code de chargement de classe

```
public Class loadClass(String name, boolean resolve)
  throws ClassNotFoundException {
  // Recherche de la classe parmi les classes déjà chargées
  Class c = findLoadedClass(name);
  if (c != null) return c;
  try {
    // Récupère le chargeur parent (référence supposée non nulle)
    ClassLoader parent = getParent();
    c = parent.loadClass(name); //Recherche avec le chargeur parent
    if (c != null) return c;
  } catch (ClassNotFoundException e) {} // Le parent n'a pas trouvé
    /* Si le chargeur parent n'a pas chargé la classe, c'est au
       chargeur courant de le faire. Si une RuntimeException est
       survenue pendant que le parent chargeait la classe, ou
       pendant le getParent(), ou bien encore pendant l'appel
       suivant, il faut la propager. */
  c = findClass(name);
  if (resolve) resolveClass(c);
  return c;
}
```


Modifier le comportement par défaut

- Pour modifier le comportement par défaut du chargement de classe
  - mais sans changer le schéma de délégation, il faut redéfinir dans son propre chargeur de classe la méthode findClass(), et ne pas modifier la méthode loadClass()
  - La méthode findClass() se charge de construire un objet de la classe Class grâce à la méthode Class defineClass(String name, byte[] b, int off, int len, ProtectionDomain domain) throws ClassFormatError
  - La méthode resolveClass() finira de préparer la classe chargée en vue de son utilisation

Création d'objet et exécution de code

- Une fois la classe obtenue et chargée par *le chargeur de classe* (celui qui a appelé la méthode defineClass())
  - On peut en créer des instances (newInstance(), par exemple)
  - Toutes les classes requises pour exécuter du code sur l'objet ainsi obtenu seront chargées par le même chargeur de classe
  - Il est ainsi possible que la même classe soit chargée par deux chargeurs différents: les deux classes chargées sont incompatibles

Sous-classes de ClassLoader

- SecureClassLoader (hérite de ClassLoader)
  - Utilise la politique de sécurité courante pour définir le domaine de sécurité en fonction d'un objet de la classe CodeSource (encapsule les notions d'URL de base et de certificats des signataires du code)
- URLClassLoader (hérite de SecureClassLoader)
  - Définit en plus une stratégie de recherche des classes parmi une liste d'URL correspondant
    - à des répertoires si les URL se terminent par des "/"
    - ou à des archives JAR dans tous les autres cas

URLClassLoader

- Constructeur le plus général
  - URLClassLoader(URL[] urls, ClassLoader parent, URLStreamHandlerFactory fabrique)
- En plus des méthodes loadClass(), findClass(), etc... plusieurs méthodes permettent de manipuler l'objet chargeur de classe relativement aux ressources qu'il peut désigner
  - URL findResource(String nom)
  - Enumeration findResources(String nom)
  - URL[] getURLs()

java.applet.Applet

- Recherche, chargement, exécution et contrôle de l'applette sont à la charge du navigateur
- Le programmeur dispose de quatre méthodes
  - **init()** appelée après le chargement et l'initialisation (conf)
  - **start()** appelée après **init()**, puis chaque fois que la fenêtre redevient visible
  - **stop()** appelée chaque fois que la fenêtre est cachée, pour suspendre l'animation par exemple
  - **destroy()** appelée pour terminer le cycle de vie de l'applette et libérer les ressources
  - Ces méthodes ne font rien par défaut, sauf isActive()

Graphique et navigateur

- Graphique
  - java.applet.Applet hérite de java.awt.Panel
  - Pour swing javax.swing.JApplet hérite de Applet
- Dans un document HTML,
  - entre les balises <applet> et </applet>, par exemple:
  - <applet code="NomDeClasse" width=150 height=80>
    <param name="NomParamètre" value="Valeur">
    </applet>
  - Éventuellement, spécifier grâce à codebase l'URL où trouver le bytecode si localisation différente
  - getCodeBase() *versus* getDocumentBase()

Souche et Contexte

- Une applette peut interagir avec le navigateur
  - setStub() est appelée par le navigateur à la création de l'applette, pour positionner une « souche » (AppletStub)
    Cette souche sert d'interface entre le navigateur et l'applette
    - isActive(), getAppletContext(), getCodeBase(), getDocumentBase(), getParameter(), etc...
  - getAppletContext() retourne un objet représentant le contexte d'exécution de l'applette (AppletContext).
    C'est le même objet contexte pour toutes les applettes d'un même document HTML
    - Possibilité de les récupérer via getApplets()

Droits des applettes

- Par défaut, les navigateurs interdisent
  - L'accès au système de fichier local (lecture et écriture)
  - Le lancement de processus au moyen de System.exec()
  - Le chargement de bibliothèque ou la définition de méthodes natives
  - L'accès ou la modification de propriétés systèmes donnant des informations sur l'utilisateur ou la machine locale (ex: user.name)
  - L'accès à un autre groupe de processus légers
  - L'installation de ClassLoader ou Factory
  - La connexion vers une autre adresse Internet que celle d'où provient l'applette
  - L'acceptation de nouvelles connexions
- Sinon, il faut signer l'applet et utiliser la politique de sécurité