Programmation événementielle et réseau

IUT Béthune
DUT2 Réseaux & Télécommunications

Daniel Porumbel

1. Sockets : Communication entre machines
2. Le Flux du Programme : Exceptions, Événements, et Fils d'Exécution (Threads)
3. Interface Graphique `Swing` : Les Événements

Notation : Sec. 1 : 40%, Sec. 2 : 20%, Sec. 3 : 40%
Évaluation : 30% DS + 20% TD (colles) + 50% TP (contrôle) + 10% Bonus

1. Sockets : Communication entre machines
   - Définition socket et rappels pile TCP/IP
   - Fonctionnement des sockets
   - Les sockets en pratique

2. Le Flux du Programme : Exceptions, Événements, et Fils d'Exécution (Threads)

3. Interface Graphique `Swing` : Les Événements

A quoi servent les sockets ?

Applications client/serveur

- Serveurs et clients web
- Transfert de fichiers
- Jeux en réseau
- Console à distance
- Groupes de discussions

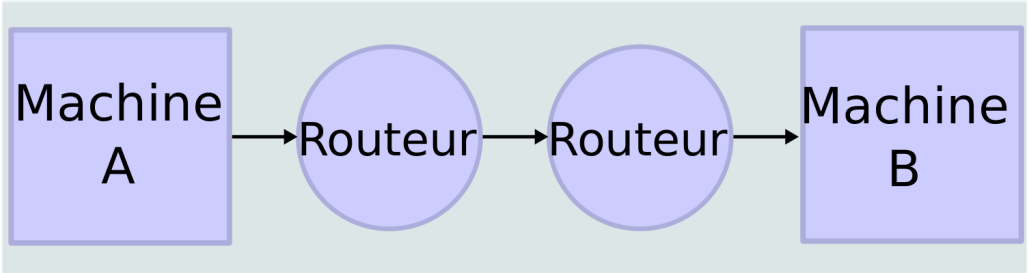
La solution pour implémenter ces applications : les sockets

Rappels : pile TCP/IP et couche transport

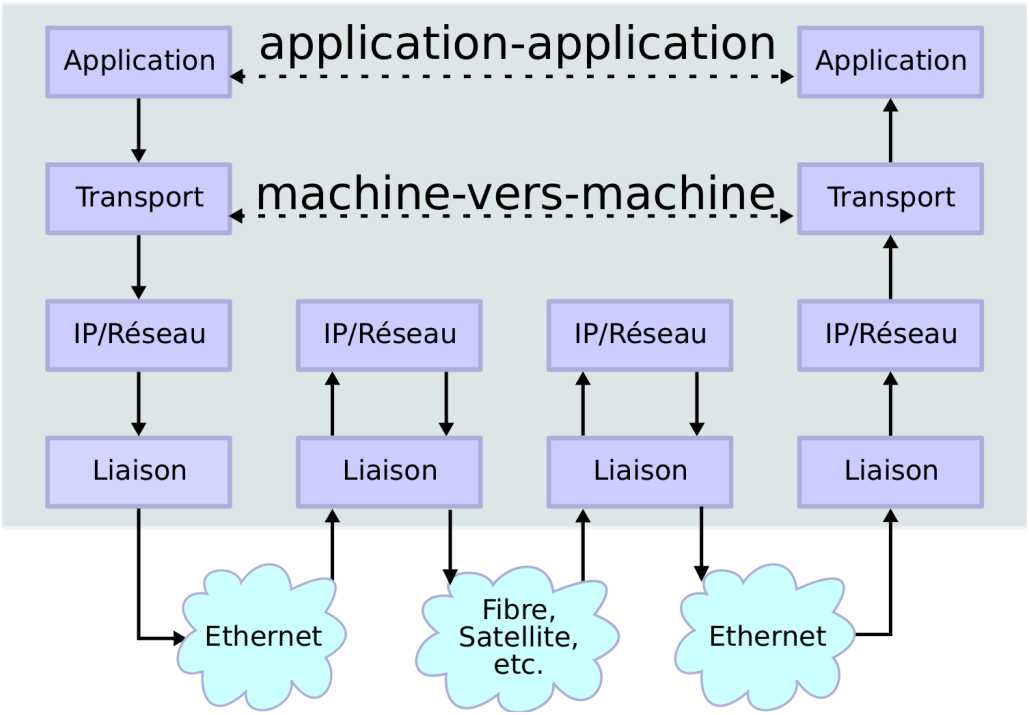
Quatre couches de protocoles qui s'appuient sur la couche physique :

1. Couche Application (Firefox, MSN, FTP)
2. Couche Transport (TCP/UDP)
3. Couche Réseaux (routage)
4. Couche Liaison

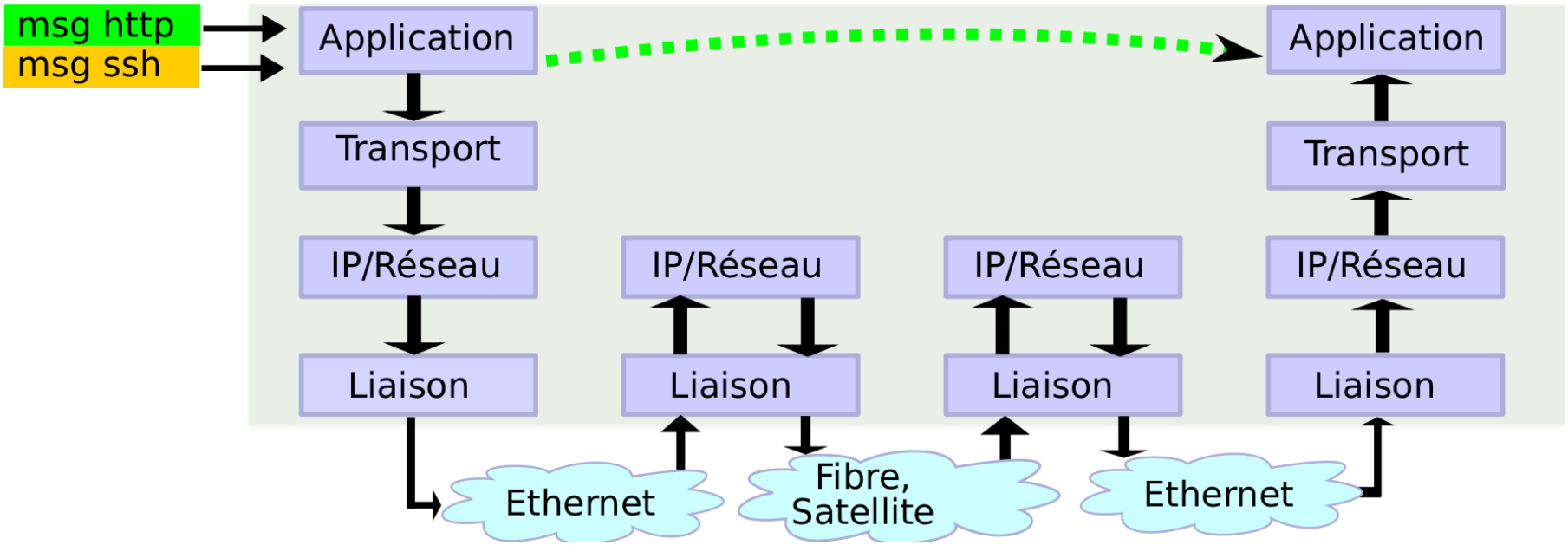
Topologie Réseau



Flux de données

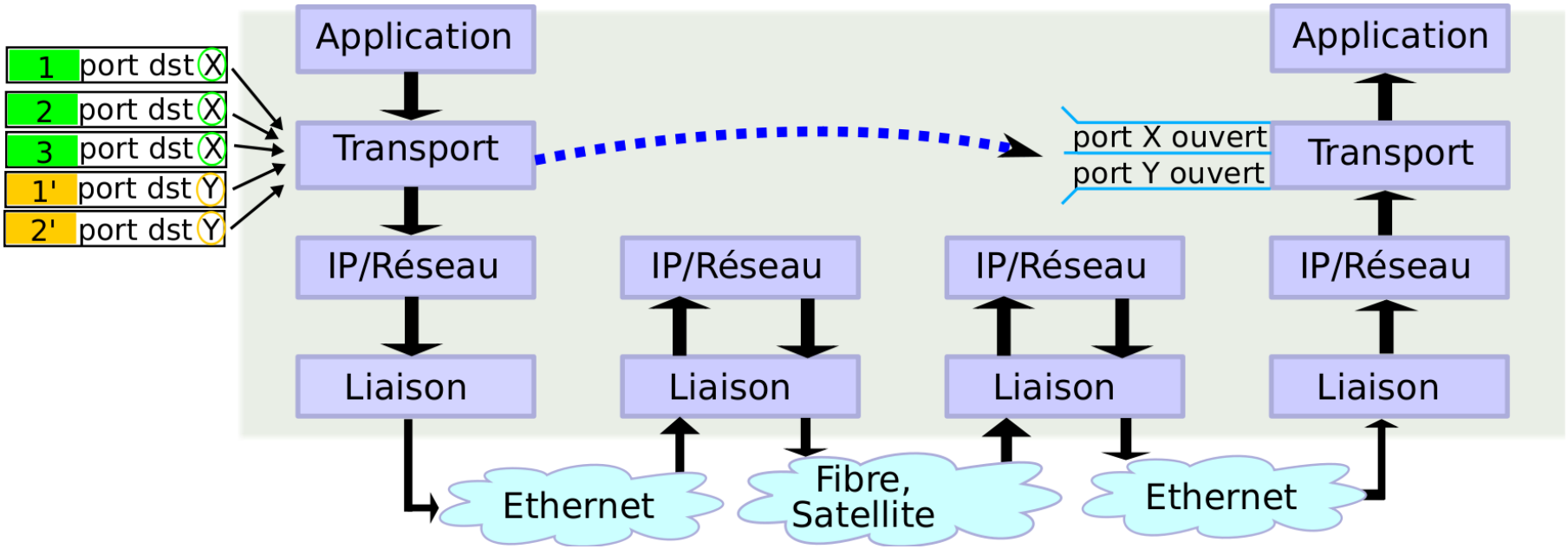


La Couche Application



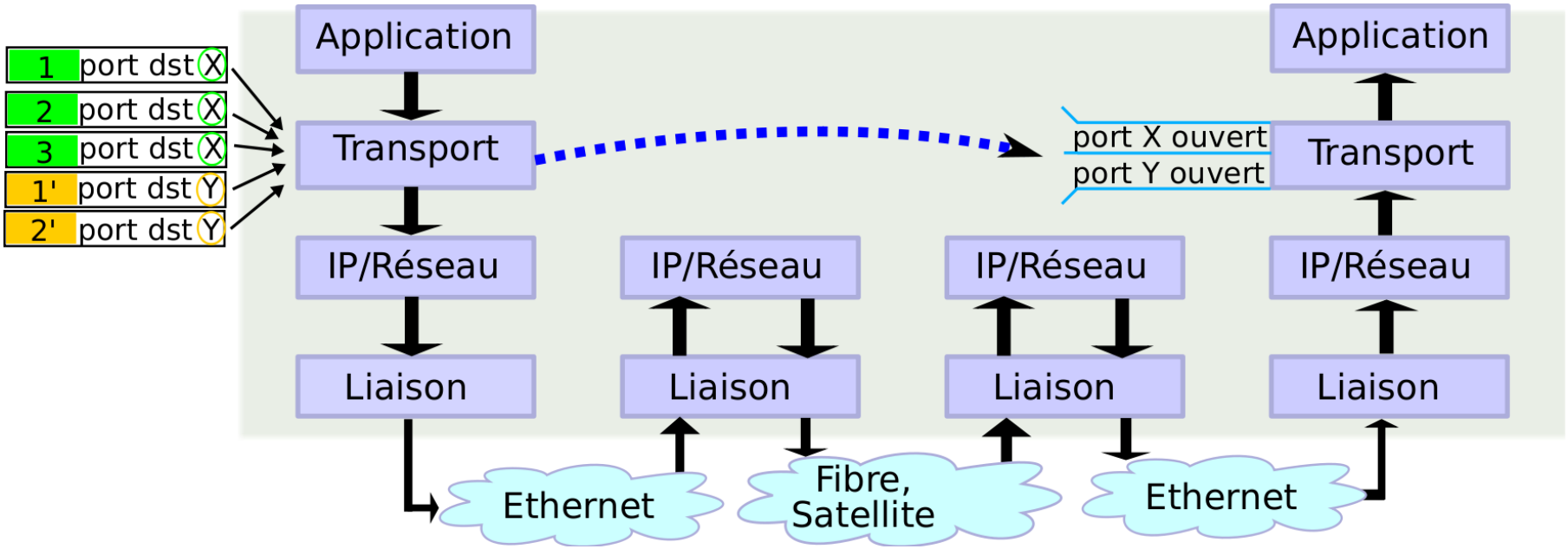
- Cette couche analyse des clients et serveurs (ex. `http`)
  - tout programme client/serveur est associé à un processus
  - plusieurs processus s'exécutent en parallèle sur une seule machine
- Exemple communication entre processus :
  - processus client (firefox) → processus serveur (apache) : `10.0.0.1` vers `10.0.0.2` : "Donne moi le `index.html`"
  - Question : Comment sait `10.0.0.2` à quel processus transférer cette requête ?

Rappel Couche Transport



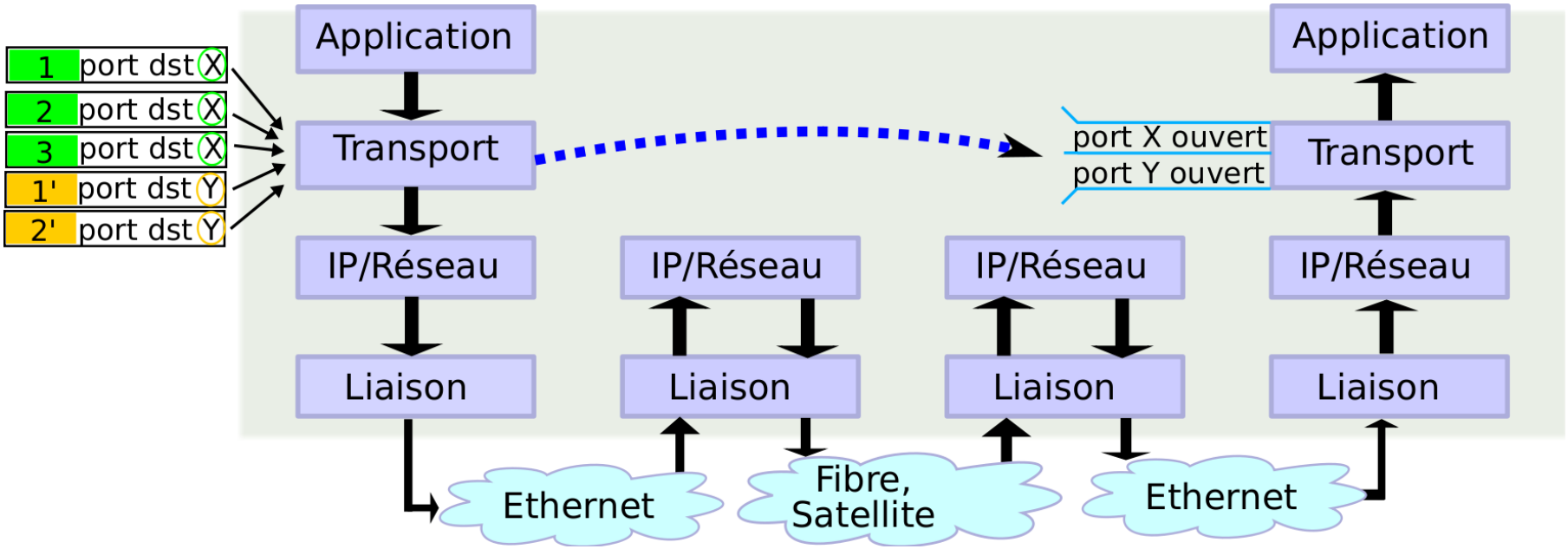
- Un *message* d'un processus est découpé en *segments*
- Chaque segment reçoit une adresse socket source + une adresse socket destination
  - Une socket = une adresse IP + un port, ex. `10.0.0.2:80` ou `10.0.0.2:22`
  - Ces ports permettent le multiplexage de plusieurs processus sur une seule machine (web et ssh ci-dessus)
- On fait abstraction des machines intermédiaires

Rappel Couche Transport



Cette couche régule le flux de données et assure soit :

Rappel Couche Transport



Cette couche régule le flux de données et assure soit :

- un transport fiable (données transmises sans erreur et reçues dans l'ordre de leur émission) dans le cas de TCP (*Transmission Control Protocol*)

ou

- un transport non fiable dans le cas de UDP (*User Datagram Protocol*). Il n'est pas garanti qu'un segment arrive à destination, c'est à la couche application de s'en assurer.

Couche Transport et Couche Application

- Adresse socket = combinaison d'une adresse IP (couche Réseau) et d'un port (couche Transport/Application)
- Chaque port ouvert est mappé à un processus applicatif
  - Exemple : toute information reçue sur le port 80 de la machine locale est transférée à un serveur web (ex. Apache)
  - `20/21 (FTP), 22 (SSH), 23 (Telnet), 53 (DNS), 67/68 (DHCP), 80/8080 (HTTP), 443 (HTTPS), etc`

Couche Transport et Couche Application

- Adresse socket = combinaison d'une adresse IP (couche Réseau) et d'un port (couche Transport/Application)
- Chaque port ouvert est mappé à un processus applicatif
  - Exemple : toute information reçue sur le port 80 de la machine locale est transférée à un serveur web (ex. Apache)
  - `20/21 (FTP), 22 (SSH), 23 (Telnet), 53 (DNS), 67/68 (DHCP), 80/8080 (HTTP), 443 (HTTPS), etc`
- L'affectation des ports
  - Les ports < 1024 : uniquement en mode administrateur, i.e., ports affectés à des services de système
  - Les ports > 1024 : à utiliser par le programmeur

Exemple de communication entre sockets

1. Le serveur ouvre son port serveur et attend des requêtes (ex. serveur Web sur le port 80)
2. Le client ouvre son port (ex. 12000) et envoie la requête sur le port du serveur
   - Cette requête contient souvent une demande d'exécution d'un traitement à distance (ex. donner moi la page web)
3. Le serveur renvoie le résultat de la demande sur le socket client (ex. la page web demandée, language `http`)

Pour gérer plusieurs clients au même temps, le serveur utilise des fils d'exécution ou threads (présentés plus tard)

Plusieurs Types de Sockets

- Les sockets sont souvent associées à la couche Transport
  - Il faut utiliser les services de la couche Transport offerts par la machine
  1. Sockets “Stream” (TCP)
  2. Sockets “Datagram” (UDP)

Plusieurs Types de Sockets

- Les sockets sont souvent associées à la couche Transport
  - Il faut utiliser les services de la couche Transport offerts par la machine
  1. Sockets “Stream” (TCP)
  2. Sockets “Datagram” (UDP)
- Il existe aussi des sockets “Raw” :
  - Les applications forgent et analysent directement leurs paquets IP, sans utiliser les services Transport
  - Exemples : ICMP (ping), OSPF (le protocole de routage “Open Shortest Path First”)

Sockets en ligne de commande

- L'utilitaire `netcat` permet de manipuler tout type de sockets
  - Attention : installer le paquet `netcat-traditional` et pas `openbsd`
- En raison de sa polyvalence, `netcat` est aussi appelé le "couteau suisse du TCP/IP"

Sockets en ligne de commande

- L'utilitaire `netcat` permet de manipuler tout type de sockets
  - Attention : installer le paquet `netcat-traditional` et pas `openbsd`
- En raison de sa polyvalence, `netcat` est aussi appelé le "couteau suisse du TCP/IP"
- mode client : `netcat IP PORT`
- mode serveur : `netcat -l -p PORT`
- Par défaut, `netcat` utilise des sockets STREAM (TCP) : l'option "-u" permet de passer en mode DGRAM (UDP).

```
1 netcat iut-rt 22          #connexion ssh
2 netcat -l -p 8080         #ecouter le port 8080
3 netcat -u -l -p 8080      #ecouter le port 8080
    UDP
4 netcat -u -b 172.31.25.255 3128 #envoie
    message UDP en diffusion
```

Sockets en Ruby : client TCP

1. Nous allons utiliser la bibliothèque `'socket'`
2. `TCPSocket.open` renvoie un objet `client` qui est ensuite utilisé pour gérer la connexion
3. Si la connexion est bien établie, l'objet `client` permet de communiquer via des méthodes comme `gets` ou `puts`
4. La connexion doit être fermée

```
1 require 'socket'
2 client = TCPSocket.open("127.0.0.1", 1234)
3 client.puts("Salut"); client.puts("Au␣revoir
    ")
4 client.close()
```

Sockets en Ruby : serveur TCP

- La classe `TCPServer` permet de construire un serveur (via `TCPServer.open`)
- Ce serveur doit “accepter” un client qui se connecte
  - Résultat : un objet `session` de connexion socket serveur $\leftrightarrow$ socket client
- L'objet résultant permet de recevoir des données (via `gets`) ou bien d'envoyer (`puts`)

```
1 #!/usr/bin/ruby
2 require 'socket'
3 server = TCPServer.open(1234)
4 session = server.accept()
5 puts(session.gets())
6 session.puts 'Bien␣recu'
7 session.close()
```

Sockets en Ruby : client UDP

- Le client UDP est différent du client TCP. Voici quelques opérations
  - Création socket local via `UDPSocket.new`
  - “Connexion” à une socket UDP distant
    - pas une vraie connexion comme pour TCP
    - cette opération fait le nécessaire pour pouvoir envoyer/recevoir des données
  - Le port local est disponible via l’attribut `addr`
  - Envoi et réception de données via `send()/recvfrom()`

```
1 #!/usr/bin/ruby
2 require 'socket'
3 maSocketUdp = UDPSocket.new
4 maSocketUdp.connect("127.0.0.1", 1234)
5 puts maSocketUdp.addr[1]
6 maSocketUdp.send("aaa",0)
7 puts maSocketUdp.recvfrom(100,0)
8 maSocketUdp.close
```

Sockets en Ruby : serveur UDP

- On utilise toujours un objet `UDPSocket` : la méthode `bind()` associe la socket UDP à un port local
  - `IP` local : `0.0.0.0`, port à votre choix
- Réception de messages : `recvfrom` (pas de `gets()` comme pour TCP) → un tableau à deux positions :

pos 0 Le message : une chaîne de caractères

pos 1 Informations sur l'émetteur : tableau à 4 positions : `[0] type adresse, [1] port, [2] nom, [3] ip émetteur`

```
1 require 'socket'
2 sockUdp = UDPSocket.open()
3 sockUdp.bind("0.0.0.0", 1234)
4 for i in 1..5 do
5  msg = sockUDP.recvfrom(100,0)
6  puts "Recu msg de "+msg[1][3]+" : "+msg[0]
7 end
8 sockUdp.close()
```

Sockets en Ruby : UDP en diffusion

- Pour pouvoir envoyer des paquets en diffusion, l'objet `UDPSocket` doit activer l'option `Socket::SO_BROADCAST`
  - Méthode utilisée : `setsockopt(`*`niveau`*`,`*`option`*`,`*`valeur`*`)`
  - *`niveau`*`=Socket::SOL_SOCKET` – modifications au niveau de la socket (et non pas `SOL_TCP, SOL_UDP`)
  - *`option`*`=Socket::SO_BROADCAST` et *`valeur`*`=true`
- L'adresse de diffusion est disponible via `ifconfig`

```
1 require 'socket'
2 sockUdpBCast = UDPSocket.new
3 sockUdpBCast.setsockopt(Socket::SOL_SOCKET,
      Socket::SO_BROADCAST, true)
4 sockUdpBCast.connect("172.31.25.255", 1234)
5 sockUdpBCast.send("Salut le monde",0)
6 sockUdpBCast.close()
```

Méthodes importantes (bibliothèque socket)

- `Socket.gethostname()` : le nom de la machine locale
- `Socket.getservbyname(`*`nomService`*`)` : renvoie le port associé au service *nomService*
- `Socket.getnameinfo(`*`domain,port,ip`*`)` : renvoie des informations sur la socket : `ip:port`
  - Exemple : `Socket.getnameinfo(["AF_INET", '23', '88.221.216.34'])`
  - `"AF_INET"` indique le fait qu'il s'agit d'une adresse IPv4 (Internet)
- `Socket.getaddrinfo(`*`nommachine, service`*`)` : renvoie des informations numériques (l'IP) par rapport au service *service* et la machine *nommachine*.
  - Exemple : `Socket.getaddrinfo('www.microsoft.com', 'http')`

Résumé des sockets

Sockets TCP

client Exemple : `session=TCPSocket.open(...)`

serveur Deux opérations : open et accept

- `serv=TCPServer.open(...)`
- `session=serv.accept()`

utilisation `session.puts(...)` et `session.gets(...)`

Sockets UDP

client Création objet et connexion à distance (pas une vraie connexion comme pour TCP)

- `sockUDP=UDPSocket.new`
- `sockUDP.connect(...)`

serveur Deux opérations : open et bind

`sockUDP=UDPSocket.open()`

`sockUDP.bind(...)`

utilisation `sockUdp.send(...)` ou `sockUdp.recvfrom`

1. Sockets : Communication entre machines

2. Le Flux du Programme : Exceptions, Événements, et Fils d'Exécution (Threads)
   - Exceptions et événements
   - Fils d'exécution (threads) et programmation concurrente

3. Interface Graphique `Swing` : Les Événements

Le flux du programme : paradigme classique

Paradigme séquentiel

Un programme = une séquence d'instructions en série :

1. `a=b`
2. `c=a*a-a`
3. `puts(Math.log(c))`
4. `.....`

Le flux du programme : paradigme classique

Paradigme séquentiel

Un programme = une séquence d'instructions en série :

1. `a=b`
2. `c=a*a-a`
3. `puts(Math.log(c))`
4. `.....`

1 toute erreur peut provoquer un crash du programme

Le flux du programme : paradigme classique

Paradigme séquentiel

Un programme = une séquence d'instructions en série :

1. `a=b`
2. `c=a*a-a`
3. `puts(Math.log(c))`
4. `.....`

1. toute erreur peut provoquer un crash du programme
2. pas possible de gérer plusieurs clients au même temps
3. l'interaction avec l'utilisateur est programmée en avance
   - l'utilisateur intervient uniquement lorsqu'on arrive à une instruction comme `gets()`

- Le programme ci-dessus cache une erreur

Le flux du programme : paradigme classique

Paradigme séquentiel

Un programme = une séquence d'instructions en série :

1. `a=b`
2. `c=a*a-a`
3. `puts(Math.log(c))`
4. `.....`

1. toute erreur peut provoquer un crash du programme
2. pas possible de gérer plusieurs clients au même temps
3. l'interaction avec l'utilisateur est programmée en avance
   - l'utilisateur intervient uniquement lorsqu'on arrive à une instruction comme `gets()`

- Le programme ci-dessus cache une erreur
- La valeur de `c` peut être négative : $a < 1 \Rightarrow a \times a - a < 0$

Le flux du programme : nouveau paradigme

Paradigme non-séquentiel

La programmation graphique/réseau change par rapport à la programmation "classique" séquentielle

Exceptions définir la réaction aux conditions exceptionnelles rencontrées pendant l'exécution du programme

Événements définir la réaction différents événements (un clic de souris)

Fils d'exécutions (threads) permettre l'exécution de plusieurs `routines` au même temps (ex. plusieurs clients)

2. Le Flux du Programme : Exceptions, Événements, et Fils d'Exécution (Threads)
   - Exceptions et événements
   - Fils d'exécution (threads) et programmation concurrente

Une exception simple en Ruby

- Le programme ci-dessous peut se planter et indiquer une erreur (exception) codifiée `Errno::ERANGE` ou `Errno::EDOM`

```
1 a=gets().to_f()
2 b=Math.log(a*a-a)
3 puts(b)
```

- `Errno::ERANGE` est une classe dérivée d'`Exception`

Une exception simple en Ruby

- Le programme ci-dessous peut se planter et indiquer une erreur (exception) codifiée `Errno::ERANGE` ou `Errno::EDOM`

```
1 a=gets().to_f()
2 b=Math.log(a*a-a)
3 puts(b)
```

- `Errno::ERANGE` est une classe dérivée d'`Exception`

$\Rightarrow$ La solution : utiliser des exceptions

```
1 begin
2     a=gets().to_f()
3     b=Math.log(a*a-a)
4     puts(b)
5 rescue  Errno::ERANGE
6     puts("La fonction logarithme ne s'
          applique pas")
7 end
```

Utilisation des exceptions en `Ruby`

- La directive `rescue` indique une “mission de sauvetage”
  - un bloc de code à exécuter si l'exception indiquée après le mot clé `rescue` est déclenchée
- Toute exception hérite la classe `Exception`. Exemples :

  Errno::EDOM “Domain error”, problème du domaine d'une fonction mathématique – $[0, \infty)$ pour *log* ou *sqrt*

  SocketError problème de création socket serveur

  Errno::ECONNREFUSED le client ne peut pas se connecter : connexion refusée

Utilisation des exceptions en `Ruby`

- La directive `rescue` indique une “mission de sauvetage”
  - un bloc de code à exécuter si l’exception indiquée après le mot clé `rescue` est déclenchée
- Toute exception hérite la classe `Exception`. Exemples :

  Errno::EDOM “Domain error”, problème du domaine d’une fonction mathématique – $[0, \infty)$ pour *log* ou *sqrt*

  SocketError problème de création socket serveur

  Errno::ECONNREFUSED le client ne peut pas se connecter : connexion refusée
- Le même bloc de code accepte plusieurs directives `rescue`

Exemple de programmation dangereuse

- Code souvent fonctionnel mais dangereux

```
1 client = TCPSocket.open("127.0.0.1", 1234)
2 client.puts("salut")
```

Exemple de programmation dangereuse

- Code souvent fonctionnel mais dangereux

```
1 client = TCPSocket.open("127.0.0.1", 1234)
2 client.puts("salut")
```

- Programmation plus correcte : Prendre en compte des exceptions possibles – voir une liste dans la documentation de la classe `socket`

  Errno::ECONNREFUSED connexion refusée par le serveur
  Errno::EPIPE connexion fermée `avant` l'opération `put`
  SocketError impossible de trouver l'IP destination

  Liste d'exceptions : `ruby-doc.org` → `Standard Library API` → `Socket` ou directement à `www.ruby-doc.org/stdlib-1.9.2/libdoc/socket/rdoc/Socket.html`

  - Voir la liste de la méthode `bind`,

Exemple de programmation plus correcte

- Il faut gérer les exceptions : Errno::ECONNREFUSED, Errno::EPIPE, SocketError

```
 1 require 'socket'
 2 begin
 3   session = TCPSocket.open("localhosttttt", 1234)
 4   ligne=session.gets()
 5   session.puts(ligne)
 6 rescue Errno::ECONNREFUSED
 7     puts("Refus de connexion")
 8 rescue Errno::EPIPE=>boom
 9     puts("Le client a ferme la connexion trop
         vite")
10     puts(boom.backtrace())
11 rescue SocketError=>boom
12     puts ("Description erreur : " + boom)
13 end
```

Nos propres exceptions

Une classe exception doit hériter la classe `Exception` ou une autre classe dérivée : `StandardError`

```
1 class UnProbleme < Exception
2 end
3 def ajout_salarie (nom, an_naissance,
      mois_naissance, jour_naissance)
4       if(an_naissance>2020)
5           raise(UnProbleme,"pas possible")
6       end
7 end
8 begin
9 ajout_salarie("aaa",2238,2,3)
10 rescue UnProbleme=>boom
11      puts(boom.message)
12      puts(boom.backtrace)
13 end
```

Hiérarchie des classes `Exception`

```
Exception
 NoMemoryError
 ScriptError
   LoadError, NotImplementedError, SyntaxError
 SignalException
   Interrupt
 SystemExit
 StandardError
   ArgumentError, IOError, IndexError
   LocalJumpError, NameError, RangeError,
   RegexpError,RuntimeError, SecurityError
   ThreadError, SystemStackError
   TypeError, ZeroDivisionError
   SystemCallError
```

Les erreurs de type `Errno::....` héritent `SystemCallError`

Evénements

Principe similaire à celui des exceptions : au lieu de capter une exception, l'objectif est de capter et traiter un événement

Evénements

Principe similaire à celui des exceptions : au lieu de capter une exception, l'objectif est de capter et traiter un événement

- La directive `rescue` est remplacée par un objet (`ActionListener`) : une écouteur qui "capte" des événements
- L'écouteur possède une action pour chaque type événement (clic, déplacement de la souris)
  - Cette action est appelée lorque l'événement associé est capté

Exemple d'événement

```
1 class MonListener
2 include ActionListener
3     def actionPerformed(e)
4         puts 'je capte un clic'
5     end
6 end
7
8 btn = JButton.new ('Click Me')
9 btn.addActionListener (MonListener.new)
```

- On observe la classe `MonListener` qui implémente la méthode `actionPerformed`
- Ligne 9 : l'objet `btn` indique `MonListener` comme écouteur

2. Le Flux du Programme : Exceptions, Événements, et Fils d'Exécution (Threads)
   - Exceptions et événements
   - Fils d'exécution (threads) et programmation concurrente

Fils d'exécution–blocs exécutées en parallèle

- Fil d'exécution : une séquence d'instructions qui s'exécute en parallèle avec le programme principal.
- Étudions le code suivant :

```
1 Thread.new do
2    puts("Salut")
3    sleep (2)
4    puts("Oui")
5 end
6 sleep(1)
7 puts("Tout va bien?")
8 sleep(2)
9 puts("Super")
```

Texte affiché :

Fils d'exécution–blocs exécutées en parallèle

- Fil d'exécution : une séquence d'instructions qui s'exécute en parallèle avec le programme principal.
- Étudions le code suivant :

```
1 Thread.new do
2    puts("Salut")
3    sleep (2)
4    puts("Oui")
5 end
6 sleep(1)
7 puts("Tout va bien?")
8 sleep(2)
9 puts("Super")
```

Texte affiché :

```
1 Salut
2 Tout va bien?
3 Oui
4 Super
```

Exécution en parallèle

Scénario d'exécution :

Temps	Fil principal :	Fil secondaire :
0s	`Sleep(1)`	`puts("Salut")`
1s	`puts("Tout va bien?")`	`Sleep(2)`
2s	`Sleep(2)`	`puts("Oui")`
3s	`Puts("Super"))`	`Sleep(1)`

Un exemple plus compliqué

```
1 x=0
2 fil_exec = Thread.new do
3       x=7
4 end
5 puts("x="+x.to_s())
```

- La valeur affichée est :

Un exemple plus compliqué

```
1 x=0
2 fil_exec = Thread.new do
3       x=7
4 end
5 puts("x="+x.to_s())
```

- La valeur affichée est : inconnue
  - thread plus rapide $\Rightarrow x = 7$; thread moins rapide $\Rightarrow x = 0$
- Solution : ajouter `fil_exec.join()` pour faire le programme principal attendre le fil d'exécution

```
1 x=0
2 fil_exec = Thread.new do
3       x=7
4 end
5 fil_exec.join()
6 puts("x="+x.to_s())
```

Mutex : exclusion mutuelle

Étudions le code suivant

```
1 x=0
2 Thread.new do
3       for i in (1..10000)
4           x=x+1
5       end
6 end
7 for i in (1..10000)
8           y=x
9           x=y+1
10 end
11 puts (x)
```

- La valeur affichée est

Mutex : exclusion mutuelle

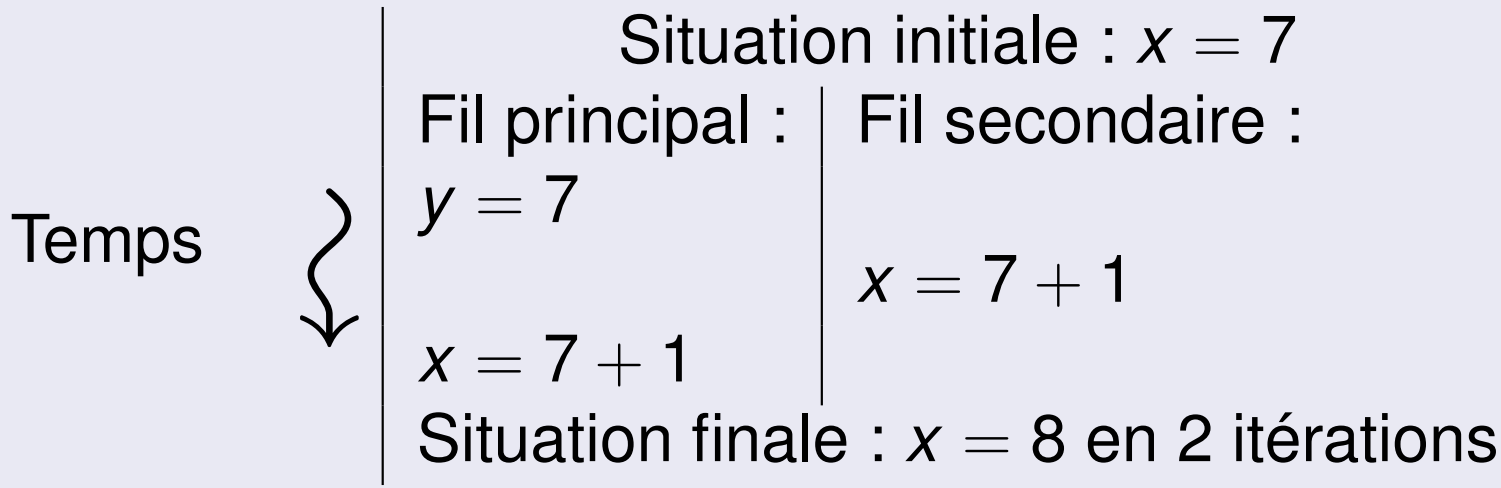
Étudions le code suivant

```
1 x=0
2 Thread.new do
3       for i in (1..10000)
4           x=x+1
5       end
6 end
7 for i in (1..10000)
8           y=x
9           x=y+1
10 end
11 puts (x)
```

- La valeur affichée est entre 14.000 et 20.000 et très rarement 20000
- Solution : une exclusion mutuelle en utilisant un mutex

Une solution Mutex pratique

Scénario d'exécution à éviter :



- Le bloc `x=x+1` ne doit pas s'intercaler dans le bloc `y=x` `x=y+1`
- Solution : mettre les deux blocs entre `mutex.synchronize do` et `end`
  - `mutex` est un objet de classe `Mutex`

Programme complet Mutex

```
 1 require "thread"
 2 mutex = Mutex.new
 3 x=0
 4 a = Thread.new do
 5     for i in (0..10000)
 6       mutex.synchronize do
 7         x=x+1
 8       end
 9 end end
10 for i in (0..10000)
11      mutex.synchronize do
12         y=x
13         x=y+1
14      end
15 end
16 a.join()
17 puts (x)
```

Messagerie instantanée

- Messagerie instantanée : flux entrant + flux sortant
  - flux entrant : fil principal
  - flux sortant : fil secondaire

```
 1 require 'socket'
 2 server = TCPServer.open(1234)
 3 session = server.accept()
 4     Thread.new do
 5         while (true) do
 6             puts(session.gets())
 7         end
 8     end
 9 while (true) do
10     session.puts(gets())
11 end
```

- Le client doit se connecter avec `TCPSocket.open(...)`

Un serveur Web à base de threads

Point de départ : serveur web sans fils d'exécution :

```
 1 require 'socket'
 2 server = TCPServer.open(8080)
 3 session = server.accept()
 4 addrtype,port,nom,ip = session.peeraddr
 5 session.puts("HTTP/1.1 200 OK")
 6 session.puts("")
 7 session.puts("<html><body>")
 8 session.puts("Salut machine "+nom+" d'IP "+
       ip.to_s+" et port "+port.to_s)
 9 session.puts("</body></html>")
10 session.close()
```

- Rappel : l'inconvénient est le fait que

Un serveur Web à base de threads

Point de départ : serveur web sans fils d'exécution :

```
require 'socket'
server = TCPServer.open(8080)
session = server.accept()
addrtype,port,nom,ip = session.peeraddr
session.puts("HTTP/1.1 200 OK")
session.puts("")
session.puts("<html><body>")
session.puts("Salut machine "+nom+" d'IP "+
    ip.to_s+" et port "+port.to_s)
session.puts("</body></html>")
session.close()
```

- Rappel : l'inconvénient est le fait que ce serveur accepte un seul client à se connecter
- Objectif : un fil d'exécution séparé dédié à chaque client

L'ajout des fils d'exécution

Après avoir initialisé un objet `session`, la manipulation de cet objet doit se faire dans un nouveau fil d'exécution

```
1 require 'socket'
2 server = TCPServer.open(8080)
3 for i in (1..10) do
4   $session = server.accept()
5   Thread.new do
6     session = $session
7     adrtype,port,nom,ip = session.peeraddr()
8     session.puts("HTTP/1.1 200 OK")
9     session.puts("")
10    session.puts("<html><body>")
11    session.puts("Salut machine "+nom+" d'IP
          "+ip.to_s()+" et port "+port.to_s())
12    session.puts("</body></html>")
13    session.close()
14 end end
```

1. Sockets : Communication entre machines
2. Le Flux du Programme : Exceptions, Événements, et Fils d'Exécution (Threads)
3. Interface Graphique `Swing` : Les Événements

Rappel Swing Jruby et Java

Jruby

```
1 require "java"
2 import javax.swing.JFrame
3 import javax.swing.JLabel
4 fenetre = JFrame.new("Bonjour");
5 fenetre.getContentPane().add((JLabel.new("Bonjour")));
6 fenetre.setLocationRelativeTo(NIL)
7 fenetre.setSize(100,500);
8 fenetre.setVisible(true);
9 fenetre.setDefaultCloseOperation(JFrame::DISPOSE_ON_CLOSE);
```

Java

```
1 import javax.swing.JFrame;
2 import javax.swing.JLabel;
3 public class Bonjour {
4  public static void main(String[] args) {
5   JFrame fenetre = new JFrame("Bonjour");
6   fenetre.getContentPane().add((new JLabel("Bonjour")));
7   fenetre.setSize(500,500);
8   fenetre.setVisible(true);
9   fenetre.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
10  }
11 }
```

Un bouton dans un panel dans une fenêtre

```
 1 require "java"
 2 import javax.swing.JFrame
 3 import javax.swing.JButton
 4 import javax.swing.JPanel
 5 monPanel = JPanel.new()
 6 monPanel.add(JButton.new("fermer"))
 7 .....
 8 fenetre = JFrame.new("Bonjour");
 9 fenetre.setSize(500,500);
10 fenetre.setVisible(true);
11 fenetre.setDefaultCloseOperation(JFrame::
       DISPOSE_ON_CLOSE);
12 fenetre.setContentPane(monPanel)
```

On va se focaliser sur les lignes 5–7 pour modifier le programme

Plusieurs boutons : les *layout*

On ajoute 20 boutons :

```
1 for i in 1..20 do
2     monPanel.add(JButton.new("ABCD"))
3 end
```

Comment seront placés ces boutons ?

Plusieurs boutons : les *layout*

On ajoute 20 boutons :

```
1 for i in 1..20 do
2       monPanel.add(JButton.new("ABCD"))
3 end
```

Comment seront placés ces boutons ? En fonction d'un *"Layout"* :

1. BoxLayout → une seule colonne de boutons
2. GridLayout → une grille de boutons
3. FlowLayout → *layout* par défaut

```
1 monPanel.setLayout(javax.swing.BoxLayout.new(
      monPanel, javax.swing.BoxLayout::X_AXIS))
2 monPanel.setLayout(java.awt.GridLayout.new(4,5))
```

Un label contrôlé par 2 boutons

- Code facile à suivre

```
1 monPanel = JPanel.new()
2 monLabel = JLabel.new("0")
3 bouton1 = JButton.new("augmenter")
4 bouton2 = JButton.new("baisser")
5 monPanel.add(monLabel)
6 monPanel.add(bouton1)
7 monPanel.add(bouton2)
```

- Objectif : utiliser les deux boutons pour augmenter ou baisser la valeur 0

Un label contrôlé par 2 boutons

- Code facile à suivre

```
1 monPanel = JPanel.new()
2 monLabel = JLabel.new("0")
3 bouton1 = JButton.new("augmenter")
4 bouton2 = JButton.new("baisser")
5 monPanel.add(monLabel)
6 monPanel.add(bouton1)
7 monPanel.add(bouton2)
```

- Objectif : utiliser les deux boutons pour augmenter ou baisser la valeur 0
- Solution : l'utilisation des événements

Les événements en `Ruby`

Exemple d'événements : un clic sur, les déplacements de la souris

- Les événements sont définis en deux étapes :
  1. Définir la classe utilisée pour capter les événements :
     - Cette classe (classe **listener**=**écouteur**) est dérivée de la classe `java.awt.event.ActionListener` par un processus simmilaire à l'héritage
  2. Associer un objet **écouteur** au composant où l'événement se produit (via `addActionListener`)
     - Exemple : le bouton sur lequel on clique
- L'objet **écouteur** est ainsi abonné aux événements du bouton

Les deux étapes en pratique

1. Définir la classe utilisée pour capter les événements

```
1 class MonAction
2  include java.awt.event.ActionListener
3    def actionPerformed(evenement)
4      source=evenement.getSource().getText()
5      ...
6    end
7 end
```

2. Associer cette classe à un ou plusieurs composants

```
1 bouton1.addActionListener(MonAction.new)
2 bouton2.addActionListener(MonAction.new)
```

à suivre : la définition complète de la classe `MonAction`

Implémentation détaillée de l'événement

```
1 class MonAction
2  include java.awt.event.ActionListener
3   def actionPerformed(evenement)
4     source=evenement.getSource().getText()
5     puts("Clic sur "+source)
6     valActuelle = $monLabel.getText().to_i()
7     if(source=="baisser") then
8         valNouvelle = valActuelle - 1
9     else
10        valNouvelle = valActuelle + 1
11    end
12    $monLabel.setText(valNouvelle.to_s())
13  end
14 end
```

- Pour quoi utiliser `$monLabel` et non pas `monLabel` ?

Les événements de clavier

étape 1 : définir les actions dans la classe `listener`

- `ActionListener` est remplacé par `KeyListener`
- trois actions à définir (et non pas seulement `actionPerformed`)
  - `keyPressed, keyReleased, keyTyped`

étape 2 : associer cette classe à des composants

Appel classique : `comp.addKeyListener()`

Attention : quel objet `comp` capte un événement ?

- clic de souris : l'objet sur lequel on clique
- un appui d'une touche : l'objet qui a le focus
  - indiqué par de petits traits faisant le tour d'un bouton

  - pas indiqué pour le panel du cadre `JFrame`, mais on peut le demander via `requestFocus`

Exemple d'événements de clavier

```
1 class MonKeyListener
2  include java.awt.event.KeyListener
3    def keyReleased(even)
4    end
5    def keyTyped(even)
6    end
7    def keyPressed(even)
8      puts("touche :␣"+even.getKeyChar().to_s())
9    end
0 end
1 .....
2 monPanel.addKeyListener(MonKeyListener.new)
3 monPanel.requestFocus()
```

Swing et Awt

- Nous avons utilisé plusieurs classes comme :
  - `java.awt.event.KeyListener`
  - `java.awt.event.ActionListener`
  - `java.awt.GridLayout`
- Ces classes font partie de la bibliothèque AWT : Abstract Windowing Toolkit
  - AWT est utilisée par Swing : de nombreuses classes Swing héritent des classes AWT
  - Classes AWT utilisées directement : des classes pour gérer les événements, ou bien `java.awt.Canvas`)