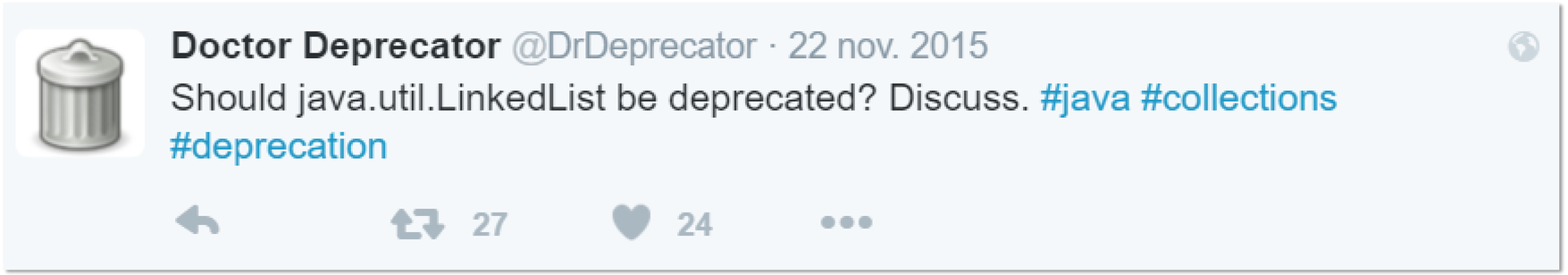
ArrayList et LinkedList sont dans un bateau

Qu’est-ce qu’il reste ?

José Paumard

Tout commença…

- Le 22 novembre 2015 par un tweet

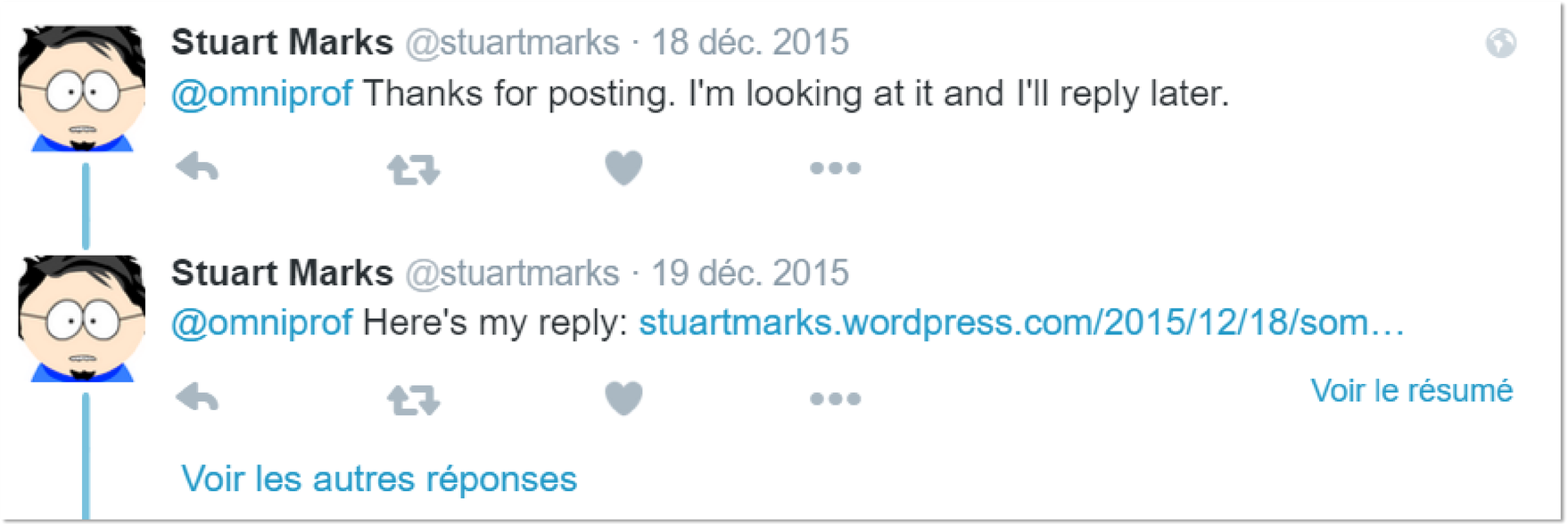


Tout commença…

- Puis par une discussion

Tout commença…

- Une trentaine d’échanges plus tard :



José PAUMARD

MCF Um. Paris 13

PhD Appl C.S.

Open source dev.

Independant

José PAUMARD

Open source dev.

Independant

José PAUMARD

Parin JUG

José PAUMARD

pluralsight
hardcore dev and IT training

Parleys

Microsoft Virtual Academy

José PAUMARD

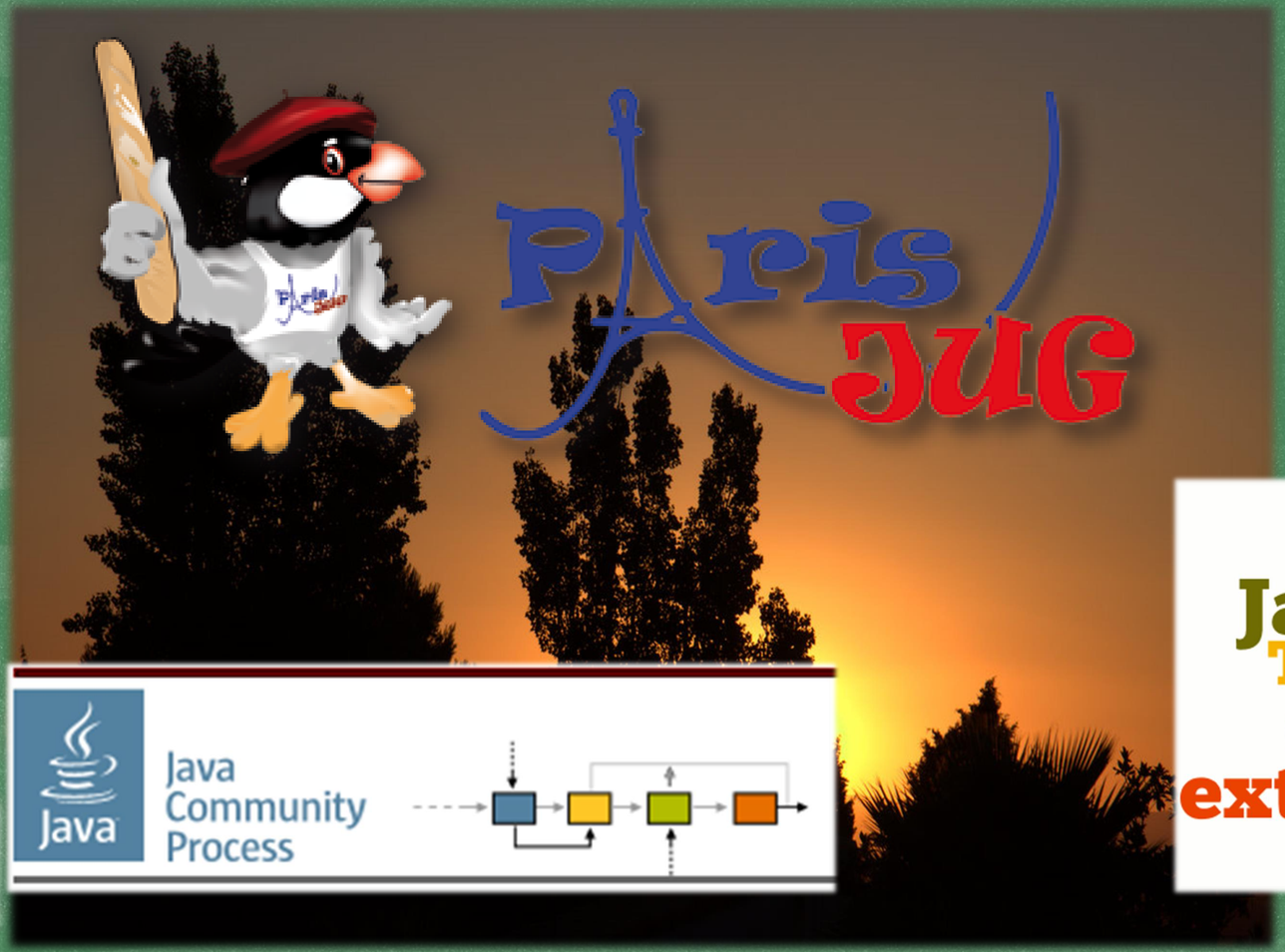
JavaOne™

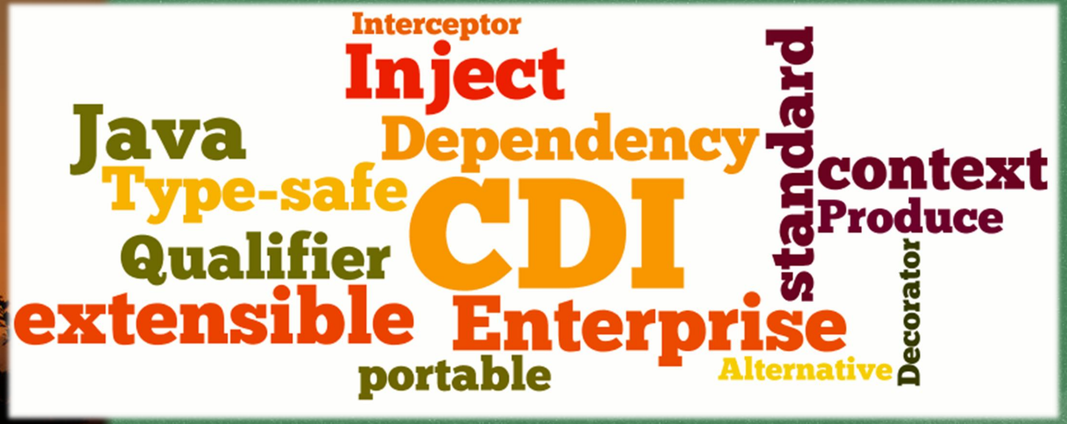
DEVOXX™

Java™ magazine

By and for the Java community

José PAUMARD





DEVOXX France

Questions ?
#ListJ9

Conclusion de cette discussion

- Le débat ArrayList vs LinkedList n'est pas qu'un troll…
  - Même si cela ressemble à de la provoc !

- Y a-t-il une réponse à la question : une des deux implémentations est-elle meilleure que l'autre ?
  - Oui !

Conclusion de cette discussion

- Mais pour arriver à cette réponse, on a besoin de comprendre beaucoup de choses

Conclusion de cette discussion

- Mais pour arriver à cette réponse, on a besoin de comprendre beaucoup de choses

- Et une fois que l'on y a répondu (à peu près) complètement, on a répondu à de nombreuses autres questions !

DEVOXX France

1ère partie

- Algorithmique
- Complexité
- Implémentation
- Structure des CPU
- « cache friendly »

- Des slides, des bullet points…

2ème partie

- Implémentation efficace de `List` et `Map` (surprise inside)
- Live coding !
- Quasiment pas de slides
  - Mais toujours des bullet points !

Quel point de départ ?

- En fait on peut en trouver plusieurs…

Quel point de départ ?

- En fait on peut en trouver plusieurs…
- L'algorithmique !

Quel point de départ ?

- En fait on peut en trouver plusieurs…

- L'algorithmique !
- L'implémentation sur le CPU

Quel point de départ ?

- En fait on peut en trouver plusieurs…

- L'algorithmique !
- L'implémentation sur le CPU
- L'utilisation…

Que sait-on ?

1) ArrayList est construite sur un tableau
2) LinkedList est construite sur une liste chaînée

On a des résultats là-dessus

Approche

- Examiner les opérations élémentaires sur les listes

- Création / modification :
  - Ajout d'un élément à la fin / au milieu de la liste
  - Effacement d'un élément par son index
  - Effacement d'un élément
  - Appel à contains()
  - Méthode removeIf() et sort()

Approche

- Examiner les opérations élémentaires sur les listes

- Création / modification :

```
List<String> list = ...

list.add("one");      // add
list.add(12, "one"); // insert
```

Approche

- Examiner les opérations élémentaires sur les listes

- Création / modification :

```
List<String> list = ...

list.remove("one"); // remove
list.remove(12);    // remove by index
```

Approche

- Examiner les opérations élémentaires sur les listes

- Création / modification :

```
List<String> list = ...

list.sort(comparingBy(String::length)); // sort
list.removeIf(s -> s.length() > 10);    // remove if
```

Approche

- Examiner les opérations élémentaires sur les listes

- Lecture :
  - Itération
  - Accès à un élément par son index
  - Construction d’un stream

Approche

- Examiner les opérations élémentaires sur les listes

- Lecture :

```
List<String> list = ...

list.forEach(System.out::println); // list traversal
list.get(12);                      // access by index
list.stream();                     // stream creation
```

ArrayList

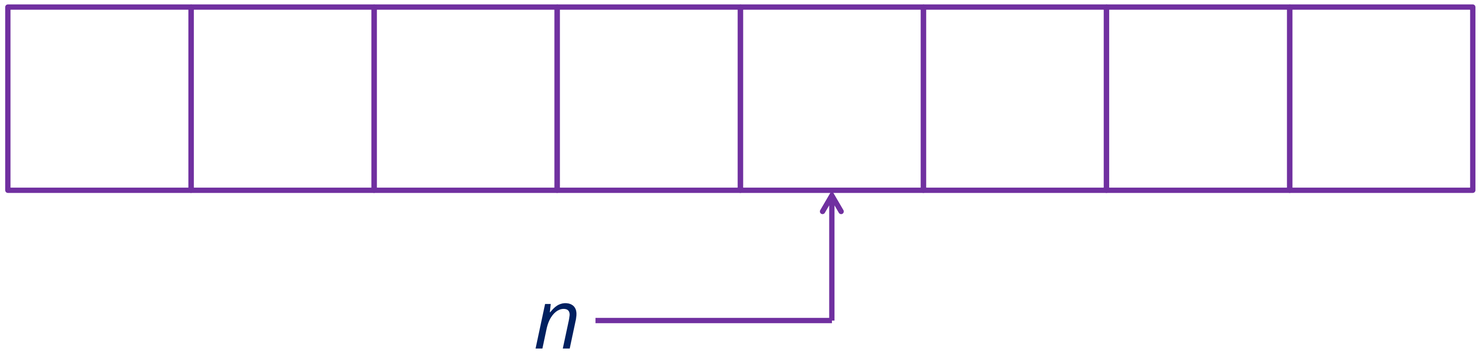
- Construit sur un tableau
- Accès au $n^{ème}$ élément : instantané

- Ajout : à peu près gratuit sauf si…

- Insertion : décalage des éléments vers la droite
- Suppression : décalage des éléments vers la gauche

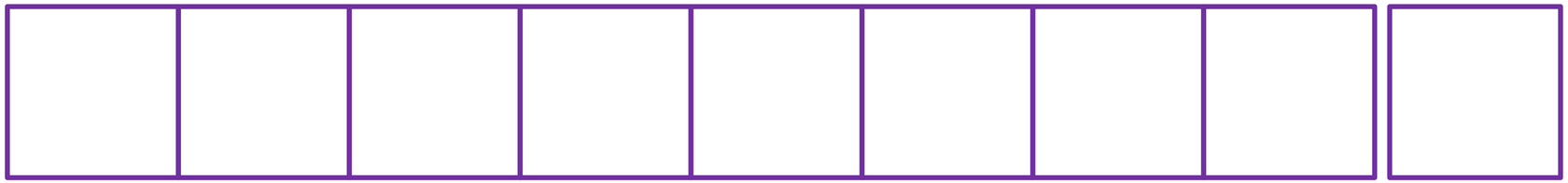
ArrayList

- Construit sur un tableau
- Accès au $n^{ème}$ élément : instantané



ArrayList

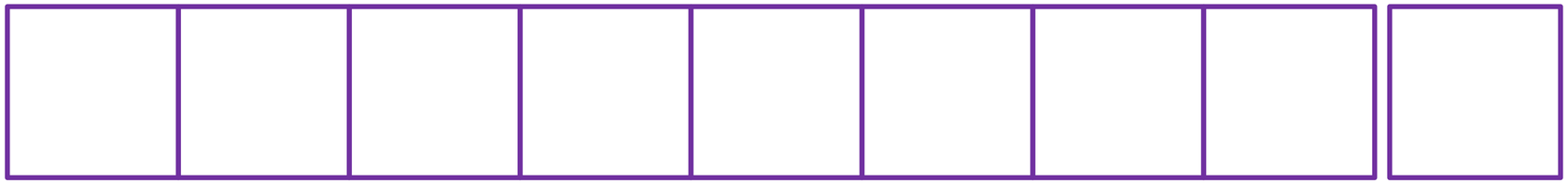
- Construit sur un tableau
- Ajout : à peu près gratuit sauf si…



```
private void grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

ArrayList

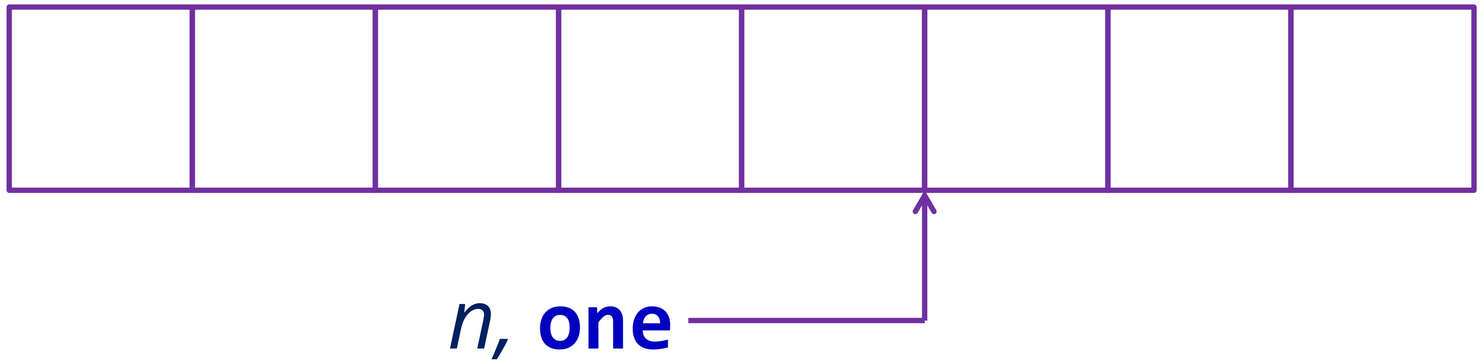
- Construit sur un tableau
- Ajout : à peu près gratuit sauf si…



- De temps en temps le tableau doit grandir
- On peut fixer à l'avance la taille du tableau

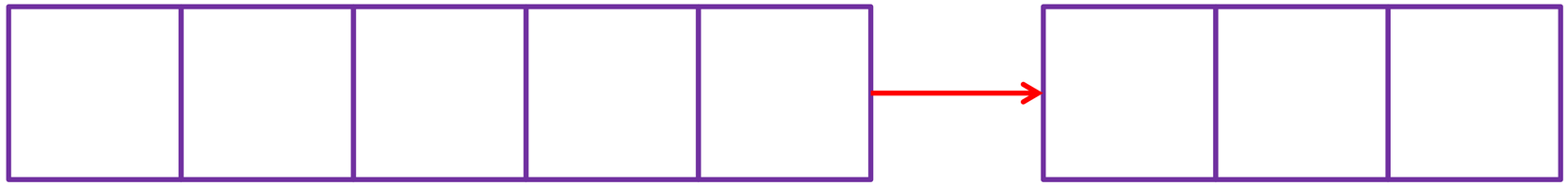
ArrayList

- Construit sur un tableau
- Insertion



ArrayList

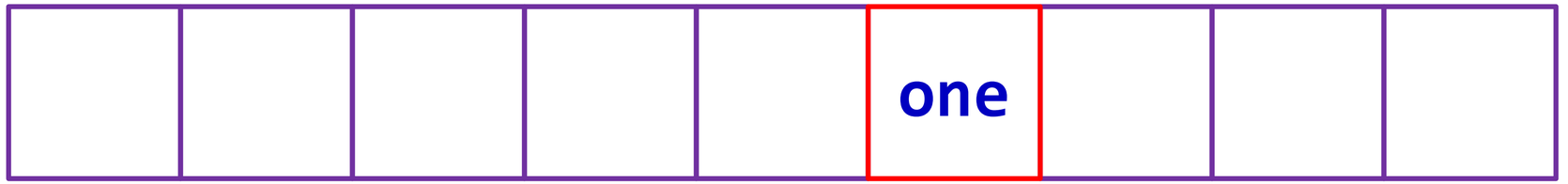
- Construit sur un tableau
- Insertion



```
public void add(int index, E element) {
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

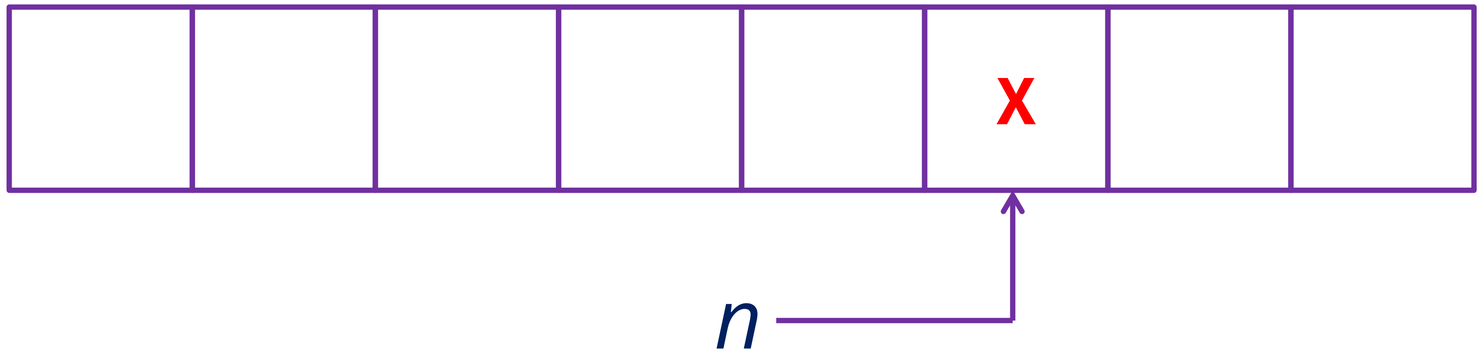
ArrayList

- Construit sur un tableau
- Insertion



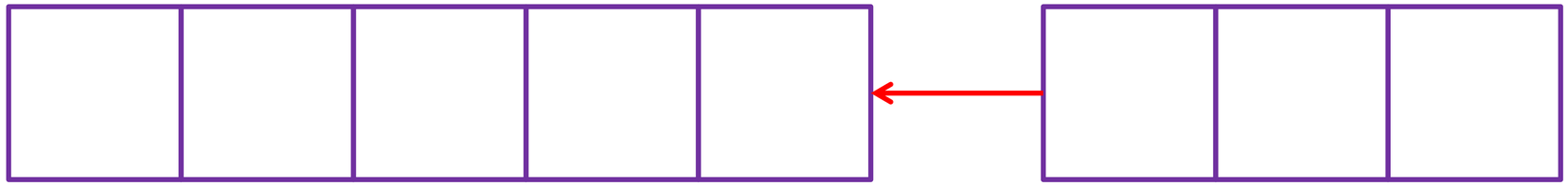
ArrayList

- Construit sur un tableau
- Suppression



ArrayList

- Construit sur un tableau
- Suppression



```
public E remove(int index) {
    E oldValue = elementData(index);
    System.arraycopy(elementData, index+1, elementData, index,
                                     numMoved);
    return oldValue;
}
```

ArrayList

- Conclusion (très) partielle

- Accès rapide
- Ajout rapide sauf en cas de dépassement de capacité
- Insertion / suppression aléatoires surcoût du fait de la « gestion des trous »

LinkedList

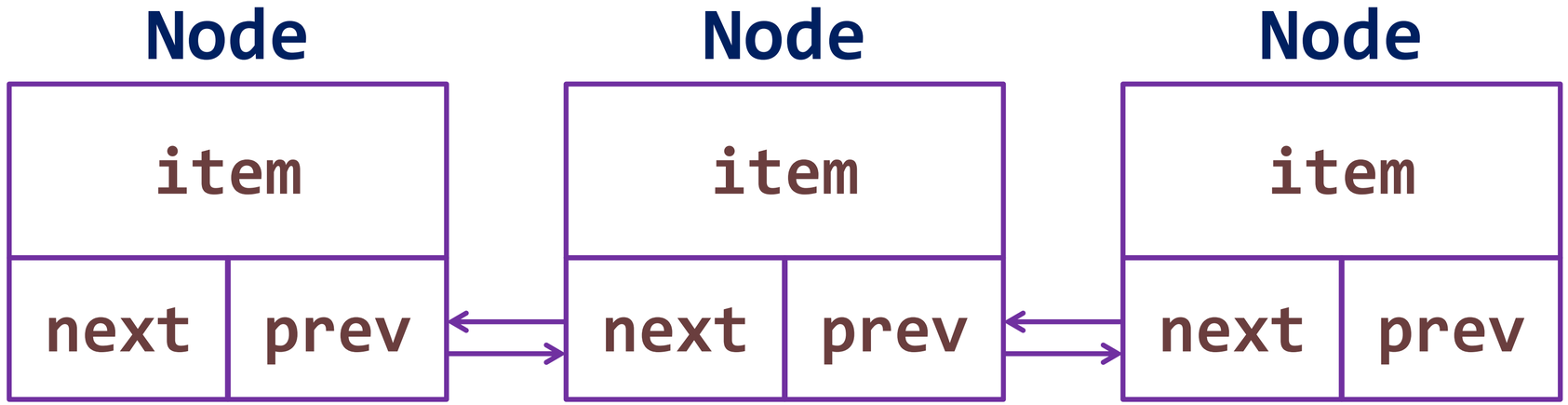
- Construit sur liste doublement chaînée
- Accès au $n^{ème}$ élément

- Ajout d'un élément en fin de liste

- Insertion
- Suppression

LinkedList

- Structure de base



LinkedList

- Construit sur liste doublement chaînée
- Accès au $n^{ème}$ élément : parcourt la liste en comptant

- Ajout : gratuit, jeu de pointeurs

- Insertion : gratuit, jeu de pointeurs
- Suppression : gratuit, jeu de pointeurs

LinkedList

- Accès au n^{ème} élément

```
Node<E> node(int index) {
    if (index < (size >> 1)) {
        Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
        return x;
    }
    // same for last
}
```

LinkedList

- Accès au n^{ème} élément : parcourt la liste en comptant

- On dit que cette opération est « en *N* »
- Ou en O(*n*)

Notation O(*N*)

- Cela signifie que si l'on multiplie par 2 le nombre d'éléments, le nombre d'opérations à effectuer sera aussi multiplié par 2

Notation O(*N*)

- On note la complexité d'un algorithme par O(*f(n)*)
- Exemple : $f(n) = n^2$

Notation O(*N*)

- On note la complexité d'un algorithme par O(*f(n)*)
- Exemple : $f(n) = n^2$
- En fait, cela signifie que le nombre d'opérations vaut
  $g(n) = \alpha n^2 + \beta n + \gamma$
- Et à partir d'une *certaine* valeur de n
  $g(n) \sim n^2$

Notation O(*N*)

- Sauf que l'on ne fait pas des maths, on traite des données
- Et pour nos applications, *n* a une valeur, de même que α, β et γ…

- Il se peut que cela modifie la validité de l'approximation théorique !

Complexités comparées

	$\log_2(n)$			
1	0			
10	3			
100	7			
1 000	10			
1 000 000	20			

Complexités comparées

	$\log_2(n)$	n		
1	0	1		
10	3	10		
100	7	100		
1 000	10	1 000		
1 000 000	20	1 000 000		

Complexités comparées

	$\log_2(n)$	n	n x $\log_2(n)$	
1	0	1	0	
10	3	10	30	
100	7	100	700	
1 000	10	1 000	10 000	
1 000 000	20	1 000 000	20 000 000	

Complexités comparées

	$\log_2(n)$	n	n x $\log_2(n)$	n^2
1	0	1	0	1
10	3	10	30	100
100	7	100	700	10 000
1 000	10	1 000	10 000	1 000 000
1 000 000	20	1 000 000	20 000 000	beaucoup

Complexités comparées

« beaucoup » veut dire que sur un CPU, le traitement de mille milliards d'éléments prend au minimum 20mn

- Besoin de distribuer le calcul sur un grand nombre de processeurs (au minimum quelques milliers)
  - On change d'algorithme
  - On sort d'un traitement dans une JVM unique

Complexité des listes

- Sur les opérations de base

	ArrayList	LinkedList
`add(e)`	1	1
`add(index, e)`	1	*N*
`set(index, e)`	1	*N*
`remove(index)`	1	*N*
`iterator()`	1	1

Complexité des listes

- Quelques précautions…
- Prendre en compte les `System.arrayCopy()` de ArrayList qui ne sont pas présents dans LinkedList

Complexité des listes

- Quelques précautions…
- Exemple : sur le `add(index, e)`, le `System.arrayCopy()` est-il plus coûteux que le parcours de la moitié de la liste ?
- Y aurait-il d'autres coûts cachés non identifiés ?

Un peu de bench

- Pour évaluer la performance d'un traitement on utilise JMH (outil standard Java 9)

- Un bench est une classe annotée
- On génère un JAR « transformé » (Maven)
- On exécute ce JAR, et on a le résultat du bench

Un peu de bench

- Classe « bench »

```
@Warmup(iterations=5, time=200, timeUnit=TimeUnit.MILLISECONDS)
@Measurement(iterations=10, time=100, timeUnit=TimeUnit.MILLISECONDS)
@Fork(value=1,
      jvmArgsAppend = {"-XX:+UseParallelGC", "-Xms3g", "-Xmx3g"})
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class Bench {

   // bench
}
```

Un peu de bench

- Dans le POM, JMH version 1.12

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>${jmh.version}</version> <!-- 1.12 -->
</dependency>
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>${jmh.version}</version>
    <scope>provided</scope>
</dependency>
```

Un peu de bench

- Dans la classe

```
@Param({"10", "100", "1000"})
private int size;

@Param({LINKED_LIST, ARRAY_LIST})
private String type;
```

Un peu de bench

- Dans la classe

```
@Setup
public void setup() {
   switch(type) {
      case ARRAY_LIST  :
         list = IntStream.range(0, size)
                         .mapToObj(Integer::toString)
                         .collect(Collectors.toCollection(
                            () -> new ArrayList<String>(size)));
      break;
      // other cases
}
```

Un peu de bench

- Simple add :
  - Peut déclencher un `System.arrayCopy()` sur `ArrayList`
  - Rapide sur `LinkedList` puisque l'on a un pointeur vers la fin de la liste

```
@Benchmark
public boolean simpleAdd() {
    return list.add("one more");
}
```

Un peu de bench

- Simple add indexé :
  - Déclenche un `System.arrayCopy()` sur `ArrayList`
  - Lent sur `LinkedList` qui doit parcourir la liste

```
@Benchmark
public boolean simpleAdd() {
   list.add(size / 2, "one more");
   return true;
}
```

Un peu de bench

- Résultats Simple Add

	LinkedList	ArrayList	Big ArrayList
10	8,7 ns	5,9 ns	6,2 ns
100	8,2 ns	6,0 ns	5,9 ns
1 000	8,1 ns	6,0 ns	6,0 ns
1 000 000	8,0 ns	5,9 ns	6,8 ns

Un peu de bench

- Résultats Simple Add Indexé

	LinkedList	ArrayList	Big ArrayList
10		30 us	30 us
100	56 ns	30 us	30 us
1 000	831 ns	30 us	30 us
1 000 000	3,78 ms	92 us	92 us

Un peu de bench

- Index Read :
  - Simple lecture en milieu de liste
  - On s'attend à ce que `LinkedList` soit désavantagée

```
@Benchmark
public boolean indexRead() {
   return list.get(size / 2);
}
```

Un peu de bench

- Index Set :
  - Écriture en milieu de liste
  - On s'attend encore à ce que `LinkedList` soit désavantagée

```
@Benchmark
public boolean indexSet() {
   return list.set(size / 2, "one more");
}
```

Un peu de bench

- Résultats Index Read

	LinkedList	ArrayList
10	28 ns	16 ns
100	196 ns	16 ns
1 000	3 us	16 ns
1 000 000	9,3 ms	16 ns

Un peu de bench

- Résultats Index Set

	LinkedList	ArrayList
10	30 ns	20 ns
100	173 ns	19 ns
1 000	3 us	18 ns
1 000 000	8,9 ms	19 ns

Un peu de bench

- Itération :
  - Itération sur la totalité de la liste
  - Pattern iterator / pattern stream

```
@Benchmark
public long iterate() {
    long sum = 0;
    for (String s : list) {
        sum += s.length();
    }
    return sum;
}
```

Un peu de bench

- Itération :
  - Itération sur la totalité de la liste
  - Pattern iterator / pattern stream

```
@Benchmark
public long streamSum() {
    return list.stream()
               .mapToLong(String::length)
               .sum();
}
```

Un peu de bench

- Résultats Iterate

	LinkedList	ArrayList
10	84 ns	65 ns
100	647 ns	429 ns
1 000	10,4 us	4,86 us
1 000 000	18,8 ms	9,1 ms

Un peu de bench

- Résultats Stream

	LinkedList	ArrayList
10	56 ns	53 ns
100	228 ns	154 ns
1 000	2,87 us	1,46 us
1 000 000	6,16 ms	3,77 ms

Le cas de removeIf() et sort()

- Direction le code

Conclusion (partielle)

- Le surcoût de `System.arrayCopy()` n'est pas très handicapant
  - Et en organisant son code, on peut le maîtriser
- Cela dit, on a toujours une différence de performance non négligeable qu'il faut expliquer

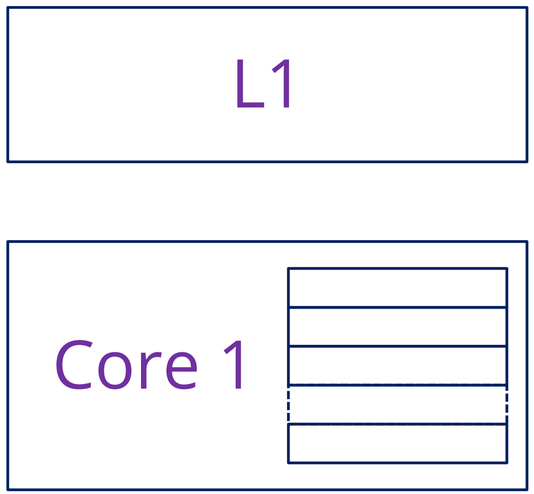
Structure des CPU

- Les CPU multicœurs ne fonctionnent pas n'importe comment…
- Entre la mémoire centrale et l'unité de calcul se trouvent 3 niveaux de cache, et des registres

Structure des CPU

Temps d’accès aux registres < 1ns

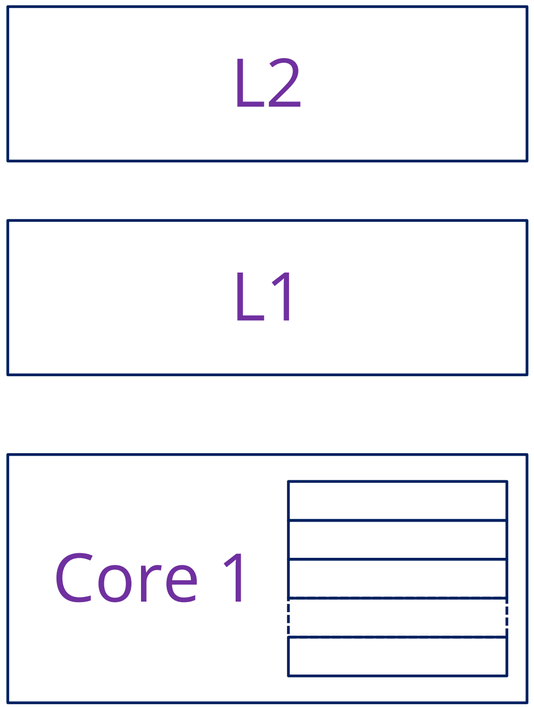
Structure des CPU



Temps d’accès ~1ns
32kO data / code

Temps d’accès aux registres < 1ns

Structure des CPU



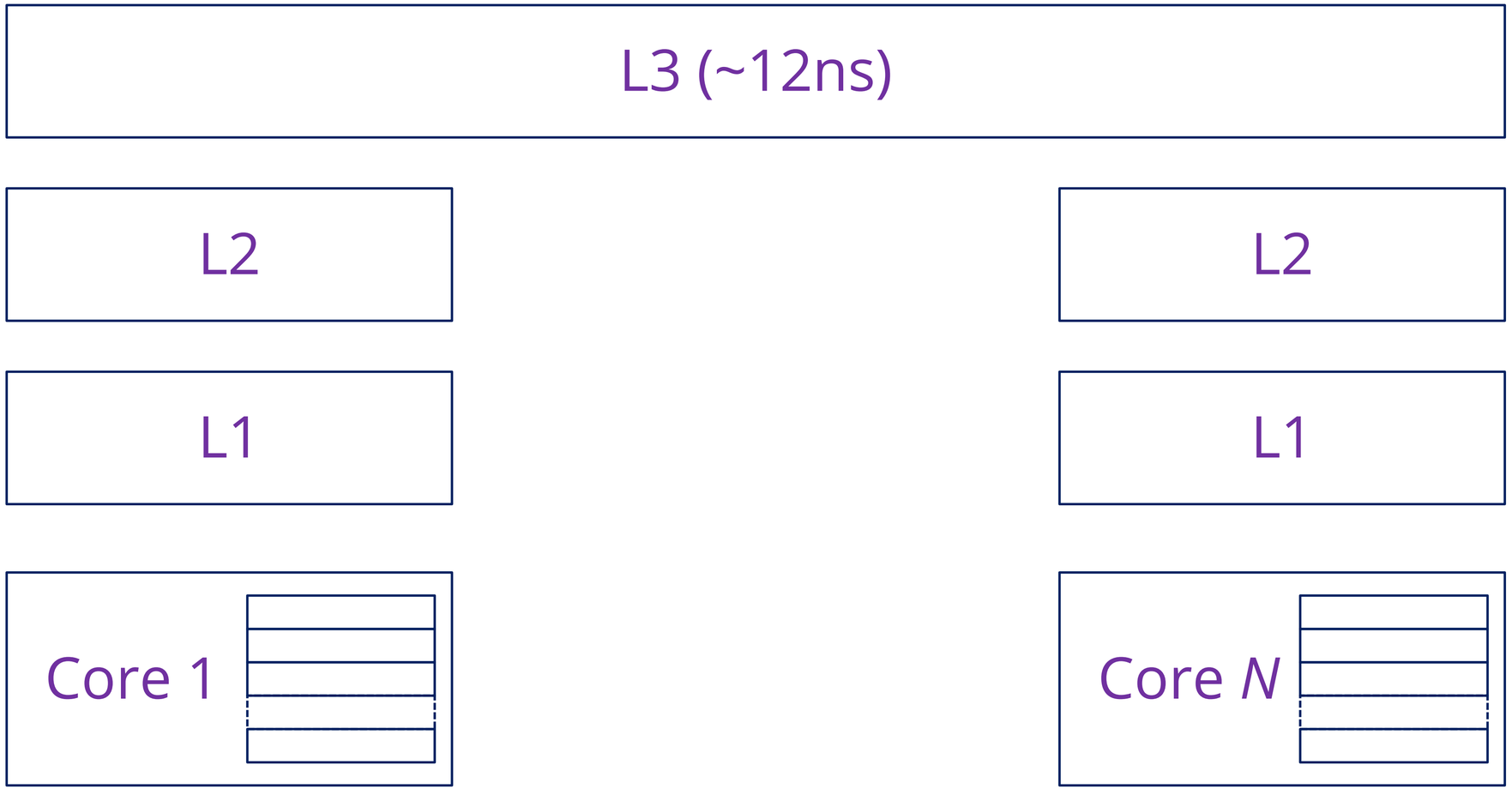
Temps d’accès ~3ns
256 kO

Temps d’accès ~1ns
32kO data / code

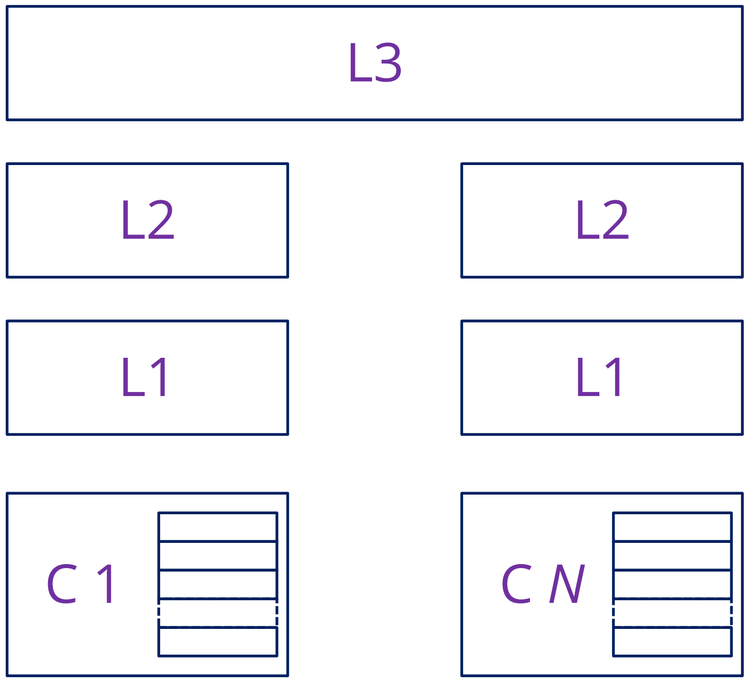
Temps d’accès aux registres < 1ns

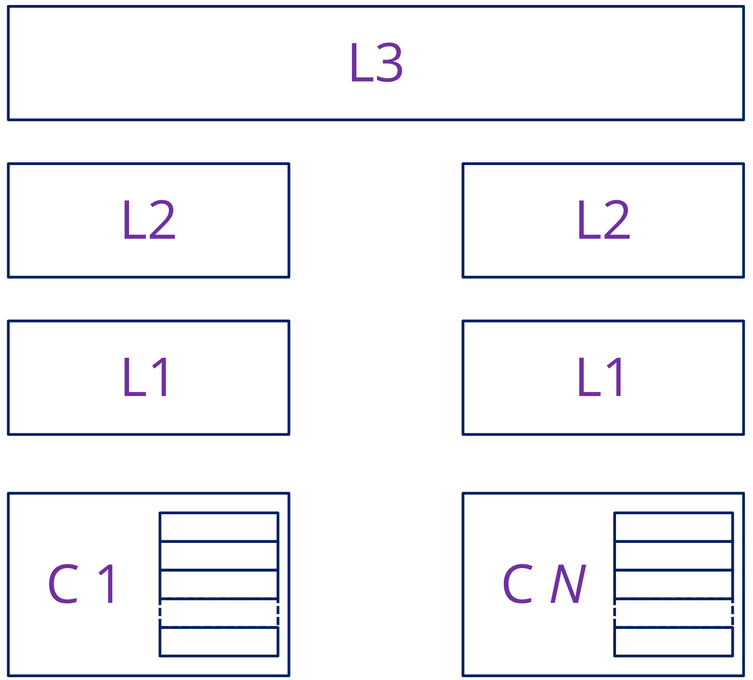
Structure des CPU

Structure des CPU

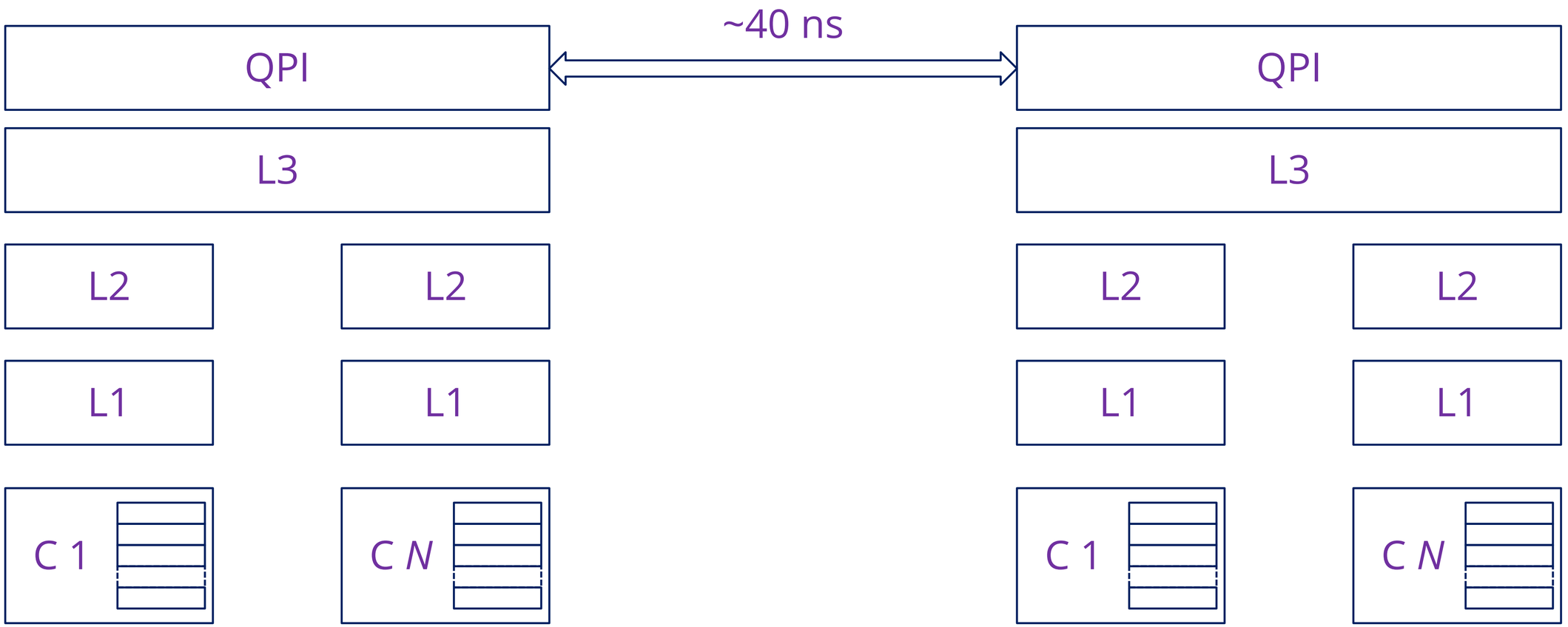


Structure des CPU

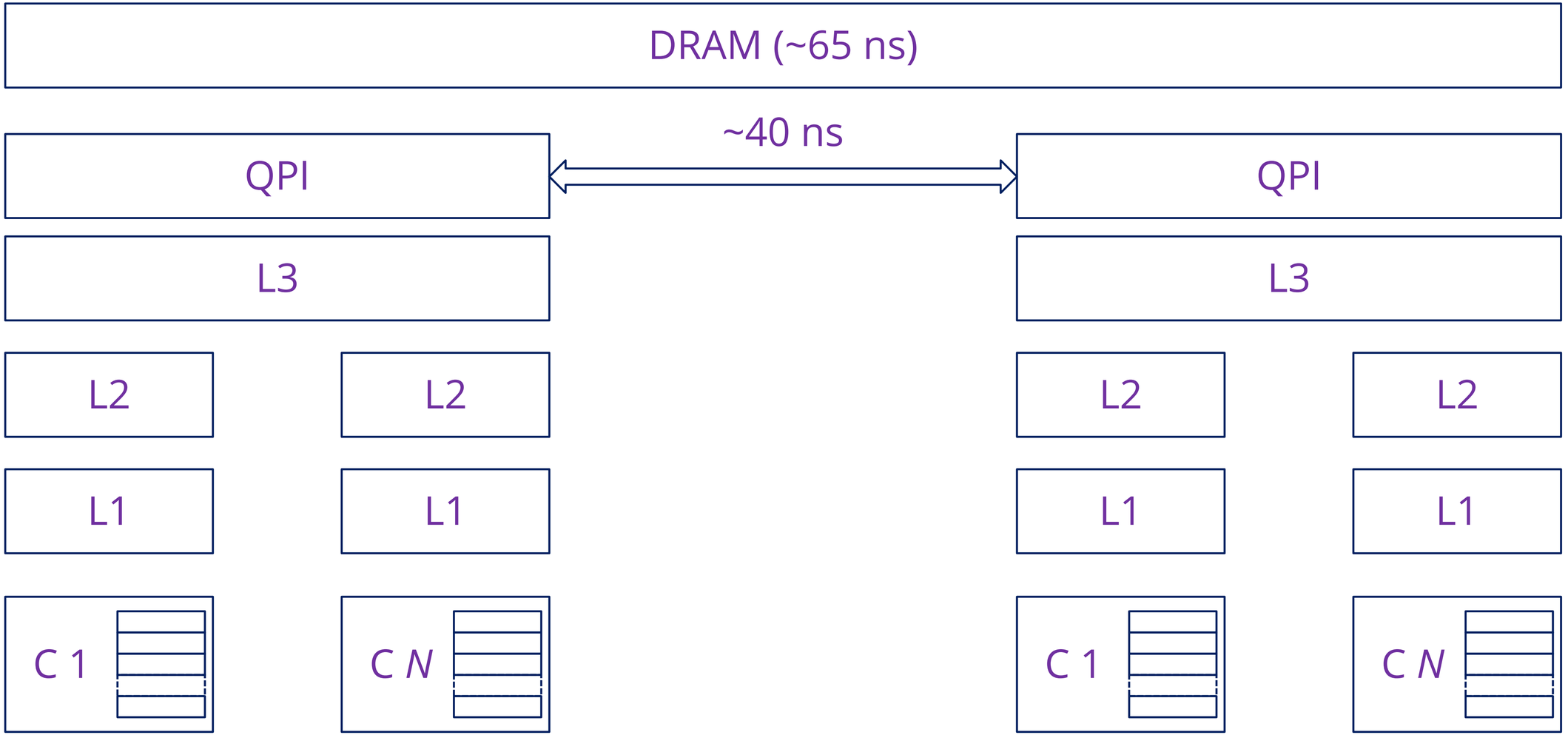




Structure des CPU



Structure des CPU



Donc…

- Il ne suffit pas de programmer des algorithmes performants…
- Encore faut-il qu'ils soient adaptés à cette structure !

Donc…

- Il ne suffit pas de programmer des traitements performants…
- Encore faut-il qu'ils soient adaptés à cette structure !

- Ce qui fait la rapidité d'un traitement, c'est sa capacité à transférer ses données dans le cache L1 le plus vite possible

L’ennemi c’est le cache miss !

- Cache miss = le CPU a besoin d’une donnée qui ne se trouve pas dans le cache L1, il faut donc aller la chercher

- Un cache miss peut représenter un retard d’environ 500 instructions

Structure du cache L1

- Le cache L1 est organisé en lignes de 8 long
- Les transferts entre caches et avec la mémoire se font ligne par ligne

- Et c'est là que se fait la différence entre `ArrayList` et `LinkedList`…

Transfert d'une liste dans L1

- Le transfert d'une `ArrayList` dans le cache L1 peut se faire 8 fois plus vite que pour une `LinkedList`

- Car les éléments d'une `ArrayList` sont rangés dans une zone contiguë de la mémoire
- Alors que les nœuds de `LinkedList` sont distribués aléatoirement…

Pointer chasing

- Le pointer chasing (chasse aux pointeurs) peut tuer les performances d’un traitement

Pointer chasing

- Exemple

```
int[] tab     = {1, 2, 3, 4, 5};
Integer[] tab = {1, 2, 3, 4, 5};
```

Pointer chasing

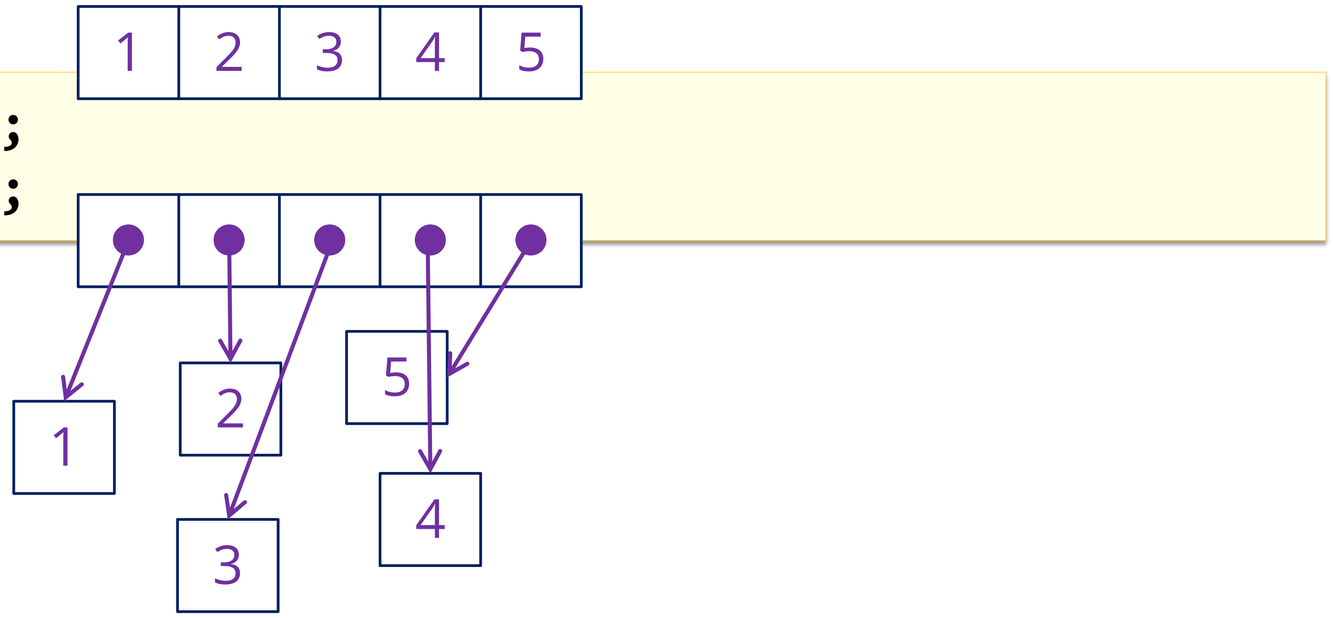
- Exemple


```
int[] tab     = {1, 2, 3, 4, 5};
Integer[] tab = {1, 2, 3, 4, 5};
```

Pointer chasing

- Exemple

```
int[] tab     = {1, 2, 3, 4, 5};
Integer[] tab = {1, 2, 3, 4, 5};
```



Pointer chasing

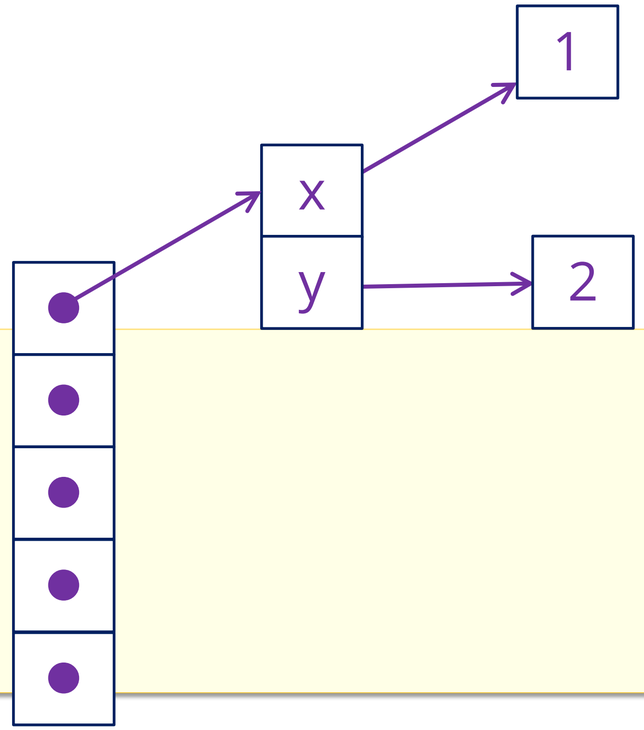
- Exemple

```
public class Point {
    Integer x, y;
}

List<Point> points = new ArrayList<>();
```

Pointer chasing

- Exemple

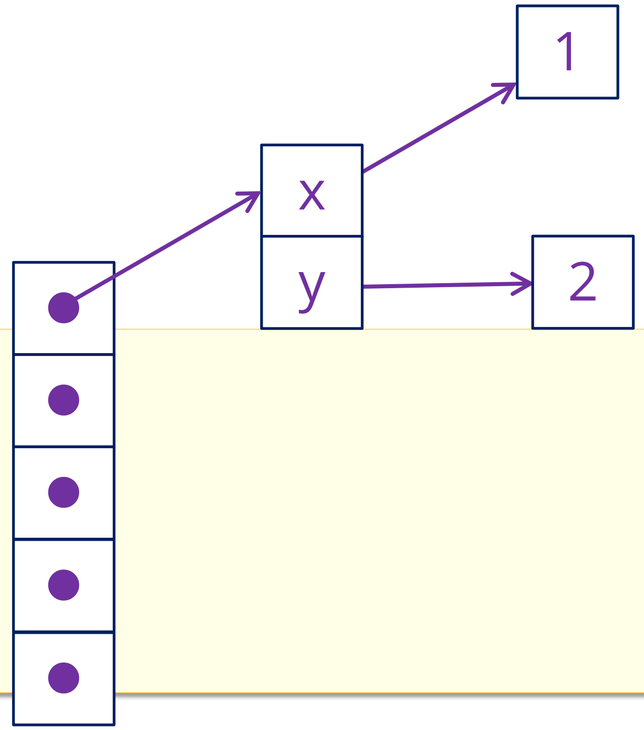
```
public class Point {
    Integer x, y;
}

List<Point> points = new ArrayList<>();
```



Pointer chasing

- Exemple

```
public class Point {
    Integer x, y;
}

List<Point> points = new ArrayList<>();
```



- Un traitement sur une telle structure va passer son temps à suivre des pointeurs…

Pointer chasing

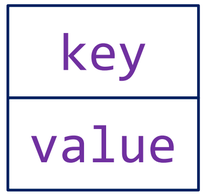
- Clairement, `LinkedList` n'est pas adaptée à la structure des CPU
- Structure « cache friendly »

Pointer chasing

- Autre exemple : `HashMap`
  - Qu'est-ce qu'une `HashMap` ? Un tableau de `Map.Entry`

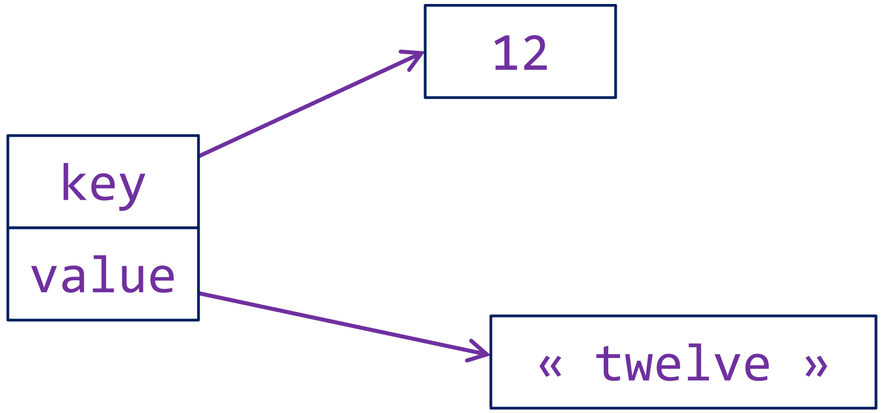
Pointer chasing

- Autre exemple : `HashMap`
  - Qu'est-ce qu'une `HashMap` ? Un tableau de `Map.Entry`



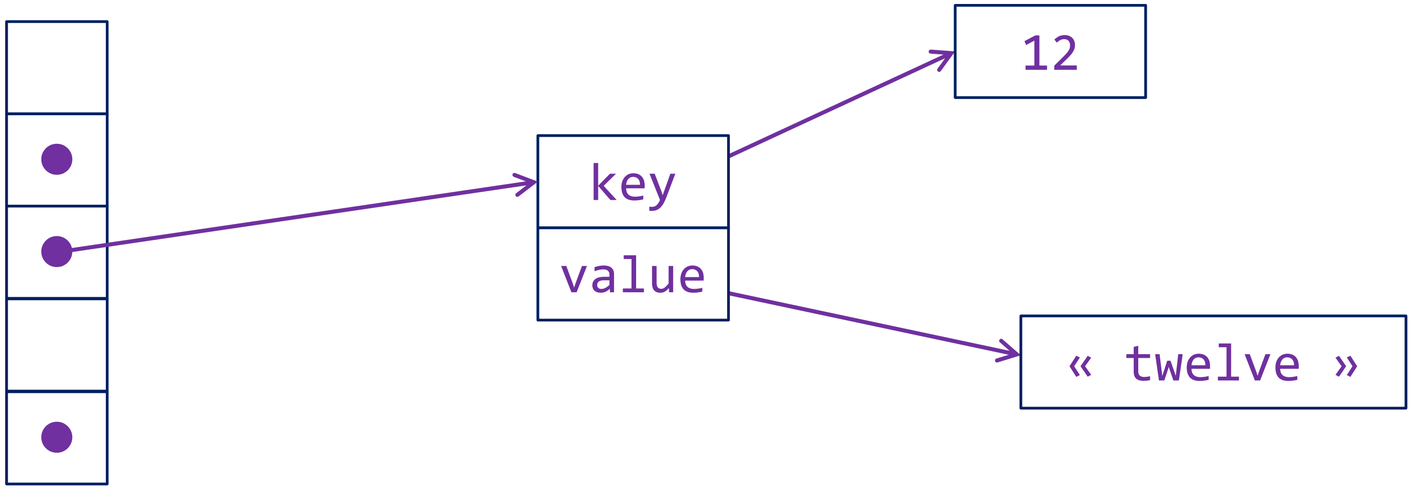
Pointer chasing

- Autre exemple : `HashMap`
  - Qu'est-ce qu'une `HashMap` ? Un tableau de `Map.Entry`



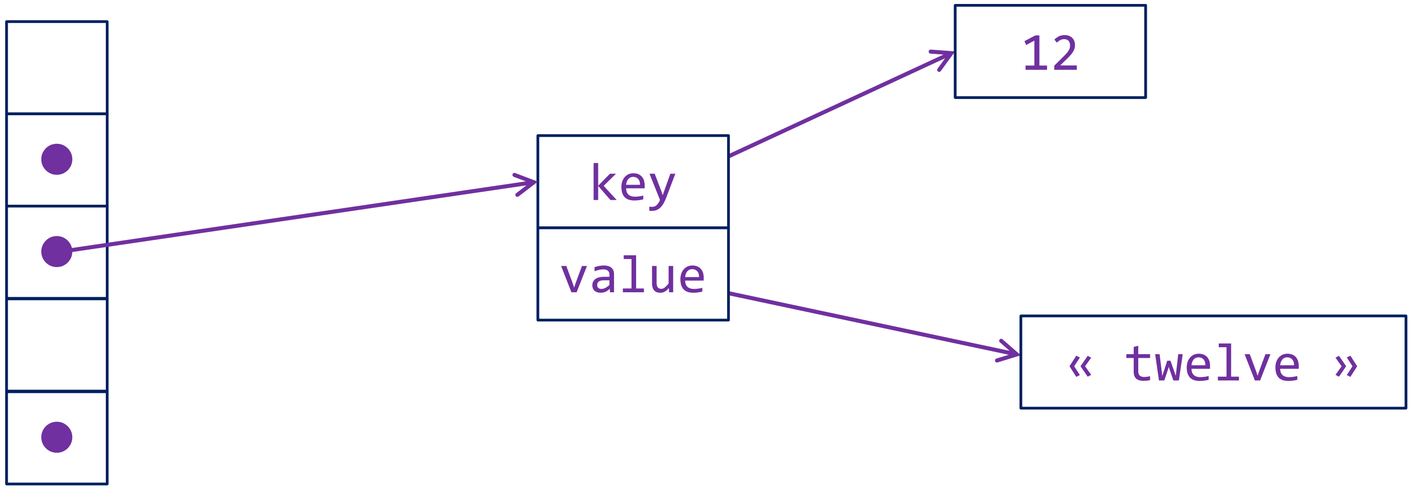
Pointer chasing

- Autre exemple : `HashMap`
  - Qu'est-ce qu'une `HashMap` ? Un tableau de `Map.Entry`



Pointer chasing

- Autre exemple : `HashMap`
  - Qu'est-ce qu'une `HashMap` ? Un tableau de `Map.Entry`



- Idem pour HashSet…

Un peu de bench

- Résultats Iterate

	HashSet	ArrayList
10	142 ns	65 ns
100	1,27 us	429 ns
1 000	19,9 us	4,86 us
1 000 000	41,0 ms	9,1 ms

Des alternatives ?

- Dans le futur : `List<int>` !
  - Projet Valhalla : http://mail.openjdk.java.net/pipermail/valhalla-spec-experts/

Des alternatives ?

- Dans le présent :
  - Eclipse Collections (prev. GS Collections)
  - HPPC
  - Koloboke
  - Trove

http://java-performance.info/hashmap-overview-jdk-fastutil-goldman-sachs-hppc-koloboke-trove-january-2015/

List sans objet ?

- Construire des implémentations de `Set` / `List` / `Map` sans pointer chasing ?

List sans objet ?

- Construire des implémentations de `Set` / `List` / `Map` sans pointer chasing ?

- Comme le pointer chasing vient du fait que l'on utilise des objets, peut-on les éliminer de ces implémentations ?

Questions et Réponses
(peut-être…)

José Paumard

ArrayList et LinkedList sont dans un bateau

… et sont tombées à l'eau !

José Paumard

Que veut-on faire ?

- Point de départ : Java 9 !

- Question : comment créer une liste préremplie en Java ?
- Réponse : ce n’est pas si simple !

Un petit coup d'œil à Java 9

- Jusqu'en Java 8 (syntaxe à moustaches) :

```
Map<Integer, String> map = new HashMap<Integer, String>() {{
    put(1, "one") ;
    put(2, "two") ;
    put(3, "three") ;
}} ;
```

- Et si l'on veut avoir une table immutable (unmodifiable), il faut le faire en deux temps…

Un petit coup d'œil à Java 9

- Nouveaux patterns pour les listes et sets :

```
List<Integer> list = List.of(1, 2, 3) ;
Set<Integer> set = Set.of(1, 2, 3) ;
```

  - Sont immutables
  - Éléments nuls non autorisés
  - Les Set jettent une IllegalArgumentException en cas de doublon
  - Serializable…

Un petit coup d'œil à Java 9

- Nouveaux patterns pour les listes et sets :

```
List<Integer> list = List.of(1, 2, 3) ;
Set<Integer> set = Set.of(1, 2, 3) ;
```

- Mais elles sont aussi…
  - À itérations aléatoires
  - Avec des implémentations optimisées !

Un petit coup d'œil à Java 9

- Implémentations optimisées :

```
public static List<E> of(E e1);
public static List<E> of(E e1, E e2);
public static List<E> of(E e1, E e2, E e3);
public static List<E> of(E e1, E e2, E e3, E e4);
...
public static List<E> of(E... e);
```

Un petit coup d'œil à Java 9

- Cas de Map

```
public static Map<K, V> of(K key1, V value1);
public static Map<K, V> of(K key1, V value1, K key2, V value2);
...
public static Map<K, V> of(K... K, V... v);
```

Un petit coup d'œil à Java 9

- Cas de Map

```
public static Map<K, V> of(K key1, V value1);
public static Map<K, V> of(K key1, V value1, K key2, V value2);
...
public static Map<K, V> of(K... K, V... v);
```

```
public static Entry<K, V> of(K key, V value);

public static Map<K, V> ofEntries(Entry... e);
```

  - Jette une exception en cas de clé dupliquée, pourquoi ?

Un petit coup d'œil à Java 9

```
Map<String, TokenType> tokens =
   Map.ofEntries(
      entry("for",   KEYWORD),
      entry("while", KEYWORD),
      entry("do",    KEYWORD),
      entry("break", KEYWORD),
      entry(":",     COLON),
      entry("+",     PLUS),
      entry("---",   MINUS),
      entry(">",     GREATER),
      entry("<",     LESS),
      entry(":",     PAAMAYIM_NEKUDOTAYIM),
      entry("(",     LPAREN),
      entry(")",     RPAREN)
   );
```

@ Stuart Mark

Un petit coup d'œil à Java 9

```
Map<String, TokenType> tokens =
   Map.ofEntries(
      entry("for",   KEYWORD),
      entry("while", KEYWORD),
      entry("do",    KEYWORD),
      entry("break", KEYWORD),
      entry(":",     COLON),
      entry("+",     PLUS),
      entry("---",   MINUS),
      entry(">",     GREATER),
      entry("<",     LESS),
      entry(":",     PAAMAYIM_NEKUDOTAYIM),
      entry("(",     LPAREN),
      entry(")",     RPAREN)
   );
```

@ Stuart Mark

Que veut-on faire ?

- Question : comment construire une implémentation optimisée d'une liste à 1 ou 2 éléments ?
- `Set`, `List` et `Map` (et `Map.Entry`)
- Si possible optimale, en minimisant le pointer chasing

Un peu de bench

- Résultats Index Read

	ArrayList	SingletonList	TwoElementList
1	3,5 ns	2,7 ns	
2	2,9 ns		2,6 ns

Un peu de bench

- Résultats Iterate

	ArrayList	SingletonList	TwoElementList
1	4,7 ns	2,3 ns	
2	6,5 ns		3,4 ns

Un peu de bench

- Résultats Iterate

	HashSet	SingletonSet	TwoElementSet
1	7,3 ns	2,3 ns	
2	9,5 ns		3,5 ns

Conclusion

- Les lambdas peuvent être utilisées dans des contextes un peu inattendus…
- Repérer les classes qui dépendent d’une unique méthode
- Idem en conception de nouvelles applications
- Cf la présentation de Rémi Forax :
  Implémenter le GoF avec des lambda

Conclusion

- On n’a pas fini de digérer les lambdas en Java !

José Paumard

Questions et Réponses

(si on a le temps…)

José Paumard

ArrayList et LinkedList sont dans un bateau

… et c’est l’heure d’aller déjeuner !

José Paumard