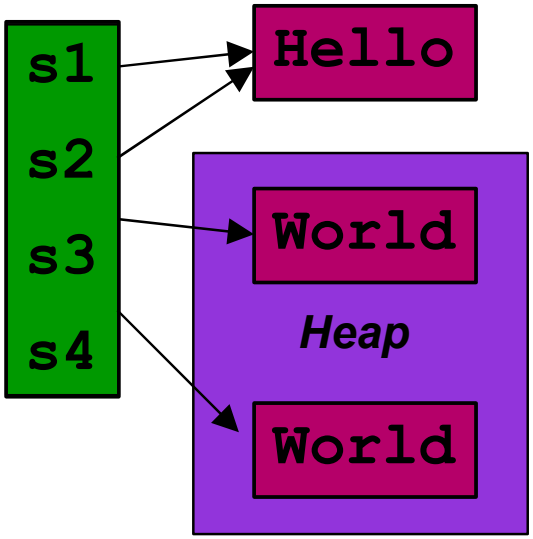
Performances

String vs StringBuffer

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("World");
String s4 = new String("World");
boolean b1 = (s1 == s2);
boolean b2 = (s3 == s4);
boolean b3 = s3.equals(s4);
```



Les Strings dans le tas sont moins efficaces !

String vs StringBuffer

Les objets **`String`** sont immuables

- la concaténation crée de multiples représentations intermédiaires

Il vaut mieux utiliser **`StringBuilder`** si on n'a pas besoin de synchronisation

Et si besoin de synchronisation, utiliser **`StringBuffer`**

```
String badStr = new String();
StringBuffer goodBuf = new StringBuilder(1000);
for (int i = 0; i < 1000; i++) {
   badStr += myArray[i];              // creates new strings
   goodBuf.append(myArray[i]);        // same buffer
}
String goodStr = new String(goodBuf);
```

Collections

`List`: *ArrayList* est la plus connue

- redimensionnable, taille inconnue à priori

`Set`: *HashSet* est la plus connue

- unicité, hash-based

`Queue`: *LinkedList* est la plus connue

- accès aux éléments dans un ordre spécifique

Map: *HashMap* est la plus connue

- stockage et récupération des données basés sur des clés distinctes

Collections (cont.)

Comment choisir ?

- type des données
- ordre important
- duplication des éléments
- performances pour les opérations importantes pour vous

Collections (cont.)

Lists

Lists Comparison Table	Add/remove element in the beginning	Add/remove element in the middle	Add/remove element in the end	Get i-th element (random access)	Find element	Traversal order
ArrayList	*O(n)*	*O(n)*	*O(1)*	*O(1)*	*O(n)*, *O(log(n))* if sorted	as inserted
LinkedList	*O(1)*	*O(1)*	*O(1)*	*O(n)*	*O(n)*	as inserted

Sets

Sets Comparison Table	Add element	Remove element	Find element	Traversal order
HashSet	amortized *O(1)*	amortized *O(1)*	*O(1)*	random, scattered by the hash function
LinkedHashSet	amortized *O(1)*	amortized *O(1)*	*O(1)*	as inserted
TreeSet	*O(log(n))*	*O(log(n))*	*O(log(n))*	sorted, according to elements comparison criterion
EnumSet	*O(1)*	*O(1)*	*O(1)*	according to the definition order of the enum values

Collections (cont.)

Maps

Maps Comparison Table	Add element	Remove element	Find element	Traversal order
HashMap	amortized *O(1)*	amortized *O(1)*	*O(1)*	random, scattered by the hash function
LinkedHashMap	amortized *O(1)*	amortized *O(1)*	*O(1)*	as inserted
TreeMap	*O(log(n))*	*O(log(n))*	*O(log(n))*	sorted, according to elements comparison criterion
EnumMap	*O(1)*	*O(1)*	*O(1)*	according to the definition order of the enum values

Queues

- ***LinkedList***, ***ArrayDeque*** - implantations de l'interface Queue pour des structures de données de types pile, file d'attente et mise en file d'attente (ArrayDeque est plus rapide que LinkedList)
- ***PriorityQueue*** - implantation de l'interface Queue utilisée pour la récupération rapide (O(1)) des éléments, qui ont la priorité la plus élevée. L'ajout et la suppression s'effectuent en O(log(n))

Collections (cont.)

`Vector` et **`Hashtable`** sont **synchronisés** sur toutes les méthodes

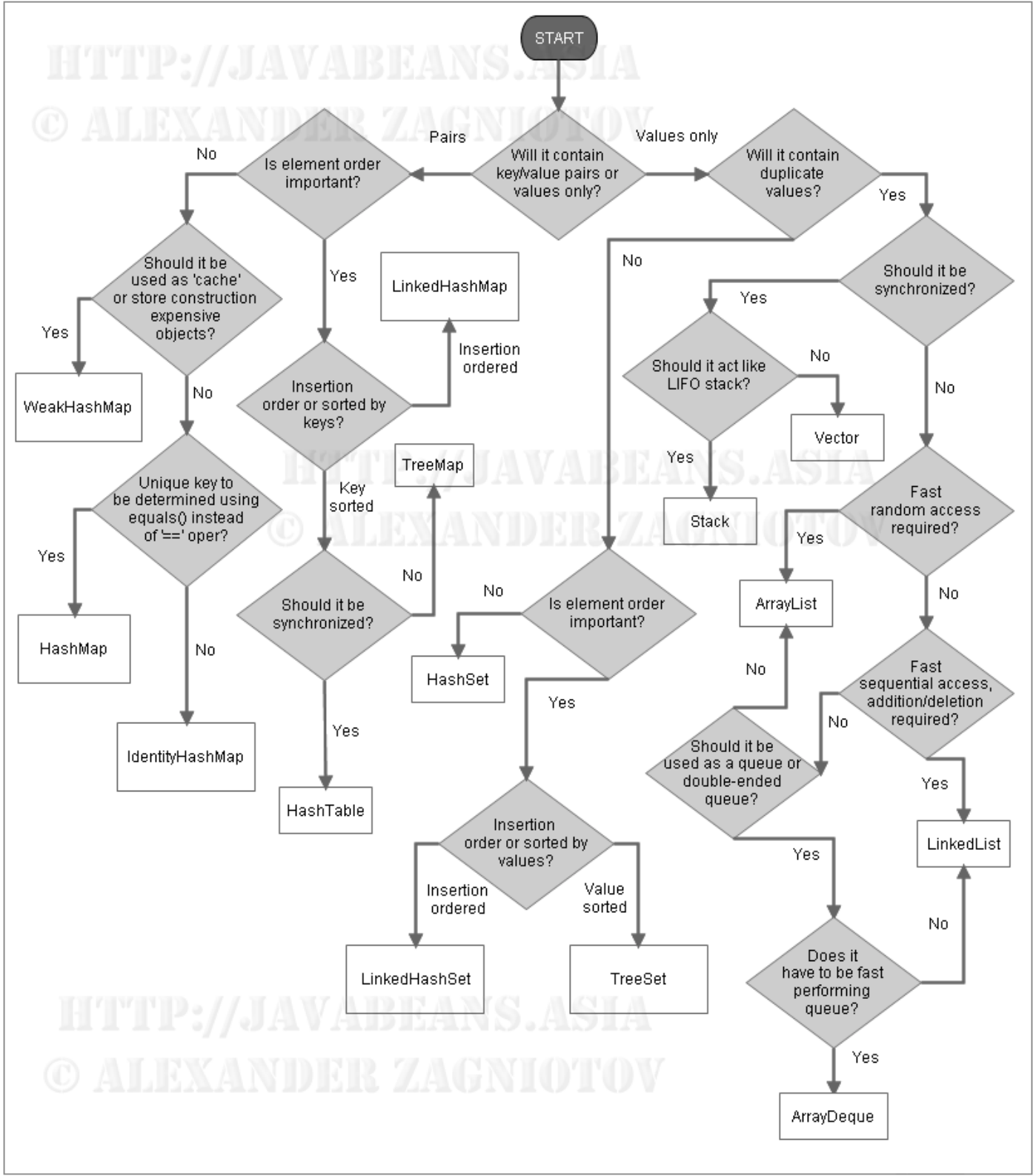
- coût de la sécurité
- inutile si un seul thread accède à la collection

Les autres collections ne sont pas synchronisées par défaut (elles peuvent l'être)

- **`ArrayList`**, **`LinkedList`** peuvent remplacer **`Vector`**
- **`HashSet`**, **`HashMap`** peuvent remplacer **`Hashtable`**

Efficaces !

- synchronisation: utilisation du pattern *wrapper*
  - *Collections.synchronizedList(new ArrayList())*



arraycopy()

```
public void useArraycopy() {
   double[] copyD = new double[kNum];
   String[] copyS = new String[kNum];
   System.arraycopy(fValues, 0, copyD, kNum);
   System.arraycopy(fLabels, 0, copyS, kNum);
}
```

Permet de copier un tableau très rapidement

- avec boucle: 0,05s
- avec **`System.arraycopy()`**: 0,01s

Exception

thro

try r

N'uti

Utilis

-
-
-

```
private int[] fArray = new int[100000];

protected void testOne(int n) {
    fSum = 0;
    for (int i = 0; i < n; i++) {
        try {
            fSum += fArray[i];
        } catch (IndexOutOfBoundsException e) {}
    }
}
protected void testTwo(int n) {
    fSum = 0;
    for (int i = 0; i < n; i++) {
        if (i < fArray.length)
            fSum += fArray[i];
    }
}
```

Synchronisation de threads

Utiliser **`synchronized`**

- portions critiques uniquement (conserver le verrou le moins de temps possible)

```
synchronized double getBalance() {
  Account acct = verify(name, password);
  return acct.balance;
}
```

```
double getBalance() {
    synchronized (this) {
        Account acct = verify(name, password);
        return acct.balance;
    }
}
```

```
double getBalance() {
  Account acct = verify(name, password);
  synchronized (acct) { return acct.balance };
}
```

Conseils sur l'optimisation

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.

—William A. Wulf [Wulf72]

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

—Donald E. Knuth [Knuth74]

We follow two rules in the matter of optimization:
***Rule 1**. Don't do it.*
***Rule 2** (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution.*

—M. A. Jackson [Jackson75]

Conseils sur l’optimisation (cont.)

Efforcez-vous d’écrire d'abord des applications **correctes** plutôt des applications rapides

Considérez les conséquences de vos **décisions de conception** (architecture, API…) sur les performances

C’est une assez mauvaise idée de vouloir une API bizarre juste pour des raisons de performance

Mesurez la performance avant et après chaque tentative d’optimisation

Sécurité

Créer des copies «défensives» [J. Bloch]

```
public class Attaque {
      private Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

      private Attaque() {
         ht.put(1, "12-34-56-789");
      }

      public Hashtable<Integer,String> getValues(){
         return (Hashtable<Integer, String>) ht.clone();
      }
   }
```

```
      public static void main(String[] args) {
         Attaque mutable = new Attaque();
         Hashtable<Integer, String> ht1 = mutable.getValues();
         System.out.println(ht1);
         Hashtable<Integer, String> ht2 = mutable.getValues();
         System.out.println(ht2);
         ht1.remove(1);
         System.out.println(ht2);
      }
```

Retourner des tableaux vides [J. Bloch]

```
priva
 /**
  * @
  *
  */
 publ
    if
        return null;
    ...
}
```

```
// The right way to return a copy of a collection
  public List<Cheese> getCheeseList() {
    if (cheesesInStock.isEmpty())
      return Collections.emptyList(); // Always returns same list
    else
      return new ArrayList<Cheese>(cheesesInStock);
  }
```

```
Cheese[] cheeses = shop.getCheeses();
```

```
// The right way to return an array from a collection
  private final List<Cheese> cheesesInStock = ...;
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
  /**
   * @return an array containing all of the cheeses in the shop.
   */
  public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

For-each vs. for

```
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());
List<Card> deck = new ArrayList<Card>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

NoSuchElementException

```
// Méthode préférable pour itérer sur des collections et des arrays
  for (Suit suit : suits)
      for (Rank rank : ranks)
          deck.add(new Card(suit, rank));
```

Code smells

```
this.state = {
  isLoading: true,
  isError: false,
  isSuccess: false
};
```

Conseils sur la sécurité

Écrivez du code Java simple et propre

- limitez les interactions
- minimiser la taille des classes et des méthodes
- utilisez des bibliothèques et des frameworks connus et testés
- refactorisez souvent votre code

Protégez les informations sensibles

- stockez les hashs des mots de passe
- filtrez les informations sensibles
- n'enregistrez jamais d'informations confidentielles

Conseils sur la sécurité

Évitez la sérialisation des données

- réfléchissez avant de l'utiliser
- utilisez une autre forme de transfert de données
- utilisez la sérialisation avec précaution si vous en avez réellement besoin

Gérez correctement les erreurs

- gardez les messages d'erreur aussi génériques que possible
- enregistrez judicieusement les messages d'erreur
- surveillez toujours vos systèmes

Conseils sur la sécurité

Prévenez les attaques par injection

- paramétrez vos requêtes
- configurez vos parsers (XML, JSON...)
- désinfectez (*sanitize*) et validez vos données

- Security is everyone's job.
- Don't nobody do nothing stupid and nobody gets hurt.
- Deterrence is not effective.
- The design of a system should not require secrecy; and compromise of the system should not inconvenience the correspondents.
- There is no security in obscurity.
- Cryptography is not security.
- Security must be factored into every decision.
- You can't add security, just as you can't add reliability.
- The Impossible is not Possible.
- False security is worse than no security.
- Any unit of software should be given just the *capabilities* it needs to do its work, and no more.
- Confusion aids the enemy.
- Never trust a machine that is not under your absolute control.
- Inconvenience is not secuirty.
- Identity is not security.
- Taint ain't security.
- Intrusion detection is not security.
- Security is everyone's job.