UV 105 Examen – Mai 2009

Notes personnelles, documents distribués en cours, calculatrice autorisés.

Nom :
Prénom :
Filière :

test

On s'intéresse au système d'écriture prédictive, appelé *T9*, utilisé notamment sur les téléphones portables. Wikipedia introduit le sujet ainsi :

La **saisie** ou **frappe intuitive** ou **prédictive** (de l'anglais ***predictive text***) est une technologie conçue afin de simplifier la saisie de texte sur les claviers téléphoniques.

Initialement brevetée aux États-Unis en 1985 en tant que méthode de communication téléphonique avec les sourds[1], la saisie intuitive n'a connu que des applications limitées avant l'avènement de la téléphonie mobile et de son service de messages textuels, où elle trouve aujourd'hui son utilité principale. Mais elle pourrait connaître à l'avenir de nouveaux champs d'application avec l'évolution des téléphones mobiles en assistants personnels et le développement de l'informatique embarquée à bord d'équipements ne disposant pas de clavier adapté (dans le domaine automobile par exemple). Le plus répandu d'entre eux a été mis au point par Tegic Communications : il s'agit du T9, sigle de *« text on 9 keys »* (littéralement « texto sur 9 touches »).

Principe de fonctionnement avec et sans la saisie intuitive

Les touches d'un clavier téléphonique sont généralement disposées comme sur l'image ci-contre : à chaque touche correspond un groupe de plusieurs lettres (3 ou 4). À l'origine, leur présence n'avait pas pour vocation de permettre la saisie de texte : c'est là encore la popularisation du SMS qui a conduit à les utiliser à cet effet. Avec les premières générations de terminaux mobiles et leur système dénommé ***MultiTap*** (ou parfois plus simplement **ABC**), la saisie de chaque lettre nécessite de presser *p* fois la touche sur laquelle elle est inscrite, où *p* désigne sa position dans le groupe.

Par exemple la lettre `Y` figure en position *p* = 3 du groupe `WXYZ` inscrit sur la touche `9` : il faut donc presser cette dernière à trois reprises pour afficher un `Y`.

Supposons qu'un utilisateur souhaite taper le mot `SAGE`. Il devra alors presser :

4 fois la touche `7` **(PQRS)** pour obtenir la lettre **S**,

1 fois la touche `2` **(ABC)** pour obtenir la lettre **A**,

\+ **1** fois la touche `4` **(GHI)** pour obtenir la lettre **G**,

\+ **2** fois la touche `3` **(DEF)** pour obtenir la lettre **E**,

c'est-à-dire `77772433`, soit en tout **8 pressions**.

Le système de saisie intuitive, ou *T9*, permet quant à lui de s'affranchir de ces multiples frappes : une seule pression par lettre suffit. Ainsi pour le même mot, un téléphone qui en est équipé permettra de ne presser que :

1 fois la touche `7` pour sélectionner le groupe **PQRS**,

\+ **1** fois la touche `2` pour sélectionner le groupe **ABC**,

\+ **1** fois la touche `4` pour sélectionner le groupe **GHI**,

\+ **1** fois la touche `3` pour sélectionner le groupe **DEF**,

c'est-à-dire `7243`, soit seulement **4 pressions** au total.

La mise en place de ce système prédictif nécessite de procéder par conjecture . Mais, là où une combinaison de touches correspond, dans le système traditionnel, à un mot et un seul (bijectivité), le système de saisie intuitive doit faire face à un problème : la pluralité des combinaisons de lettres correspondant à une même saisie.

- **Combien y-a-t'il de combinaisons possibles correspondant à la saisie de la séquence de touches "7243"** ?

108

- **Donner un minorant et un majorant du nombre de mots (un mot est une séquence de lettres quelconques) qu'on peut associer à une séquence de *n* touches.**

Minorant	Majorant
3^n	4^n

- **Quelle séquence de touches correspond en T9 à chacun des mots suivants ?**

Mot	*Séquence sans T9 (ex : 22234455662)*	*Séquence avec T9 (ex: 7334725)*
`couvait`	*2226668888824448*	*2688248*
`bottait`	*226668824448*	*2688248*
`coutait`	*22266688824448*	*2688248*

- **Expliquer pourquoi il est nécessaire que le système T9 soit couplé avec un dictionnaire pour fonctionner**

Chaque séquence de n touches T9 correspond à un ensemble de mots (dont le nombre est compris entre 3^n et 4^n). Parmi tous ces mots seuls un petit nombre (éventuellement nul) est un mot de la langue chosie par l'utilisateur.

Dans la suite on souhaite maquetter en Java une application d'écriture prédictive par *T9*. On s'intéresse à la création d'une classe *T9Engine* concentrant les différents services nécessaires pour la mise en place de ce système. On ne tient pas compte, dans la suite, de la casse, des ponctuations, des accents etc...

L'**interface** *IT9Engine,* destinée à être implémentée par *T9Engine,* expose deux services :

```
public interface IT9Engine {

abstract public String getCode(String word);

abstract public LinkedList<String> getWords(String code);

}
```

Le service *getCode* retourne la séquence de codes associée à un mot donné (pour le mot "chat" par exemple il retourne "2428").

Le service *getWords* sera étudié plus loin.

- Pour chacune des déclarations suivantes indiquer (par une croix) si elle est correcte ou non (on suppose dans cette question que les classes concernées possèdent bien un constructeur sans paramètres):

Déclaration(s)	***correcte***	***incorrecte***
`IT9Engine t9;`	*X*	
`T9Engine t9;`	*X*	
`IT9Engine t9=new IT9Engine();`		*X*
`IT9Engine t9=new T9Engine();`	*X*	
`T9Engine t9=new IT9Engine();`		*X*
`T9Engine t9=new T9Engine();`	*X*	

`IT9Engine it9;T9Engine t9;` `it9=t9;`	X	
`IT9Engine it9;T9Engine t9;` `t9=it9;`		X

getCode

On s'intéresse à la méthode *getCode*. Une première implémentation de cette méthode basée sur l'énumération explicite des 26 lettres de l'alphabet pourrait être la suivante (+= correspond bien sûr à une concaténation dans ce qui suit):

```
public String getCode(String word) {
 String code="";
 for(int i=0;i<word.length();i++) {
 if (word.getAt(i)=='a') code+="2";
 else if (word.getAt(i)=='b') code+="2";
 else if (word.getAt(i)=='c') code+="2";
 else if (word.getAt(i)=='d') code+="3";
 else if (word.getAt(i)=='e') code+="3";
 ...
 else if (word.getAt(i)=='z') code+="9";
 else break;
 }
 return code;
}
```

- En supposant l'équiprobabilité de chaque lettre de l'alphabet dans un mot, quel serait le nombre moyen, avec cet algorithme, de comparaisons engendrées par le codage d'un mot de longueur *n* ?

*13*n*

À tort ou à raison le développeur de la classe *T9Engine* décide que ce coût est trop élevé et décide d'associer une valeur de code à chaque lettre par le biais d'une structure de *TreeMap* où la clé est une lettre de l'alphabet et où la valeur associée est le code de la touche correspondante. Un résumé de la documentation concernant *TreeMap* est en annexe. Le constructeur de *T9Engine* construit une seule fois cette structure comme indiquée ci-dessous et la méthode *getCode* est transformée en conséquence :

```
public class T9Engine implements IT9Engine {
 private TreeMap<Character,Integer> codeMap=new TreeMap<Character, Integer>();
 public T9Engine() {
      codeMap.put('a',2);codeMap.put('b',2);codeMap.put('c',2);
      codeMap.put('d',3);codeMap.put('e',3);codeMap.put('f',3);
      codeMap.put('g',4);codeMap.put('h',4);codeMap.put('i',4);
      codeMap.put('j',5);codeMap.put('k',5);codeMap.put('k',5);
      codeMap.put('m',6);codeMap.put('n',6);codeMap.put('o',6);
      codeMap.put('p',7);codeMap.put('q',7);codeMap.put('r',7);codeMap.put('s',7);
      codeMap.put('t',8);codeMap.put('u',8);codeMap.put('v',8);
      codeMap.put('w',9);codeMap.put('x',9);codeMap.put('y',9);codeMap.put('z',9);
 }
     public String getCode(String word) {
 String code="";
      for(int i=0;i<word.length();i++) code+=codeMap.get(word.charAt(i));
      return code;
     }
     public LinkedList<String> getWords(String code); {
      return null; // pas implémentée à ce stade
     }
}
```

- A quels endroits dans ce code l'*autoboxing* apporte-t-il une commodité appréciable d'écriture du code ?

Chaque invocation de put met en oeuvre l'autoboxing, par exemple :

```
codeMap.put('a',2)
```

équivaut à :

```
codeMap.put(new Character('a'),new Integer(2))
```

De même :

```
code+=codeMap.get(word.charAt(i));
```

équivaut à :

```
code+=(codeMap.get(new Character(word.charAt(i)))).charValue();
```

- A combien peut-on évaluer cette fois le nombre moyen de comparaisons engendrées par le codage d'un mot de *n* caractères ?

*ln(26)*n, soit environ 3*n*

- Quelle est la propriété fondamentale d'un *TreeMap* permettant d'obtenir ce gain ?

TreeMap est, sur la clef, implémenté sous la forme d'un ABR équilibré. Comme la clef prend les valeurs des 26 lettres de l'alphabet, codeMap est en interne une arbre binaire équilibré à 26 noeuds.

- On s'intéresse maintenant à la cosntruction de la structure TreeMap telle qu'elle se fait dans le constructeur T9Engine. Dessiner les 6 premières étapes d'insertion des lettres (de 'a' à "f' compris) dans l'ABR en mettant en oeuvre les opérations de rééquilibrage nécessaires au fur et à mesure (vous ne représenterez que les clés, c'est à dire les lettres, et non les valeurs (les chiffres)). La ligne 1 correspond à l'état après l'insertion du 'a', la ligne 2 correspond à l'état après l'insertion du 'b' etc...

	Avant équilibrage	Equilibrage
1	a	a
2	a b	a b
	a b c	b a c
4	b a c d	b a c d
5	b a c d e	b a d c e
6	b a d c e f	c b e a d f

L'association d'un code à chaque mot partitionne de fait l'ensemble des mots (de la langue française par exemple) en un ensemble de classes d'équivalence. Deux mot sont dits *texto-équivalents*, ou *textonymes*, s'ils ont le même code numérique par T9. Les mots *acte, cave* et *baud* par exemple sont textonymes (code 2283)

La portion de programme qui suit saisit deux mots au clavier et affiche *oui* si les deux mots sont textonymes ou *non* sinon.

- Compléter le programme selon les lignes pointillés en réutilisant correctement la classe *T9Engine* telle qu'elle est définie ci-dessus, de façon à ce que programme se comporte comme annoncé.

```
public class Launcher {

public static void main(String[] args) {

T9Engine t9Engine=new T9Engine(...);
String s1,s2;
Scanner scan=new Scanner(System.in);
System.out.println("Entrer un premier mot");
s1=scan.next();
System.out.println("Entrer un second mot");
s2=scan.next();

if (t9Engine.getCode(s1) equalsTo t9Engine.getCode(s2))
System.out.println("Oui");
else
System.out.println("Non");

}
}
```

getWords

On s'intéresse maintenant à la méthode *getWords* de la classe *T9Engine*. Cette méthode retourne pour un code numérique donné une liste des mots correspondant. Ainsi pour le code "*26883*" la liste est constituée des mots *aoute, coute, cotte, botte, couve et boute*. Le logiciel du téléphone portable proposera donc à l'utilisateur avec une interface appropriée cette liste de mots. Néanmoins, pour améliorer l'ergonomie du dispositif le logiciel présentera en première position celui de ces mots dont l'occurence est la plus probable dans la langue choisie par l'utilisateur. Cela suppose que le logiciel dispose non seulement d'un dictionnaire de cette langue, que ce dictionnaire possède toutes formes possibles de chacun des mots de la langue (formes singulières, plurielles, masculines, féminines, conjuguées etc...) mais aussi que ce dictionnaire soit en mesure d'associer à chaque mot une probabilité d'occurrence (exprimée dans la suite sous la forme d'un réel positif entre 0 et 1) en fonction de la fréquence d'apparition du mot telle qu'on peut l'analyser par des études linguistiques et statistiques.

Pour un tel dictionnaire concernant la langue française cela correspond à une liste de 400 000 mots environ, associés chacun à un nombre réel. On ignore dans la suite les accents, majuscules etc... Voici un très court extrait :

...

chant 0,05321

chants 0,04321

chantage 0,25321

chantages 0,20321

chantai 0,01321

chanta 0,005321

chantasse 0,000021

chantas 0,004321

chantames 0,005321

chantasses 0,000011

...

La classe *T9Engine* doit donc incorporer un tel dictionnaire sous une forme appropriée pour en permettre une exploitation efficace dans le but de décider rapidement (c'est à dire au fur et à mesure de la frappe de l'utilisateur) si une combinaison de lettres donnée correspond bien à un ou plusieurs mots de la langue française. Voici à titre d'exemple la succession d'états présentée par la saisie du mot *admirable* (code `236472253`) et les suggestions de l'interface graphique qui accompagnent cette saisie.

2	**23**	**236**	**2364**	**23647**	**236472**	**2364722**	**23647225**	**236472253**
a	ce cf	ben	beni	admis benir benis	admira benira	*<vide>*	*<vide>*	admirable

- Lors de la saisie, une étape donnée peut-elle réutiliser les résultats de l'étape précédente ?

non

Pour cette algorithme on choisit dans un premier temps une stratégie où l'ensemble des mots correspondant à un code donné est soumis au dictionnaire à chaque saisie.

- Dans l'exemple ci-dessus combien (une approximation suffit) de mots seront soumis au dictionnaire après que l'utilisateur ait tapé le dernier 3 de ce code (pour ne retenir finalement que le mot *admirable*)?

Entre 3^9 et 4^9 c'est à dire en gros entre 20000 et 26000 mots

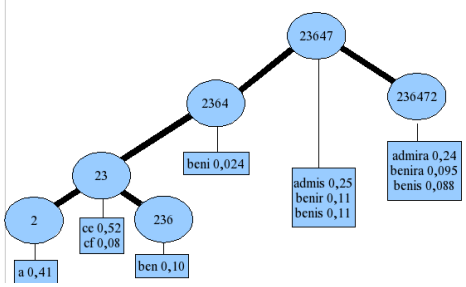
- Même question avec le mot le plus long de la langue française, c'est à dire *anticonstitutionnellement*

Entre 3^{25} et 4^{25} c'est à dire en gros entre 900 milliard et 1 million de milliard de mots

- Pensez-vous qu'un algorithme basé sur l'idée qu'on soumet au dictionnaire tous les mots issus d'un code donné, pour qu'il ne retienne que ceux qui y figurent, soit praticable et pourquoi?

Les résultats précédents montrent qu'on est confronté à une explosion combinatoire de possibilités à soummettre au dictionnaire qui devient rapidement redhibitoire avec les mots d'une certaine longueur. De plus on ne peut pas s'aider dans l'analyse d'une combinaison de n touches de l'analyse déjà effectuée pour les n-1 précédentes touches.

Pour résoudre les problèmes précédents on décide d'opter pour une structure de données qui évite l'explosion combinatoire. Il s'agit d'un *TreeMap* dont la clé est un code numérique T9 et donc la valeur est une liste chaînée (*LinkedList*) des mots correspondant à cette clé associé chacun avec leur valeur de probabilité . Voici une représentation graphique possible d'une (très) petite portion de cette structure, les clés sont dans une ellipse, les valeurs correspondantes dans un rectangle:



- Tous les codes apparaissant en tant que clé possède en tant que valeur une liste contenant au moins un mot du dictionnaire (et sa probabilité associée). Autrement dit il n'y a pas de noeud associé à une liste vide. Pourquoi cette remarque est-elle extrêmement importante ? (rappel : un dictionnaire contient 400 000 mots environ, le mot le plus long contient 25 caractères)

En se limitant aux noeuds correspondant à un code associé à au moins un mot du dictionnaire on limite le nombre de noeuds à une valeur inférieure ou égale aux nombre de mots du dictionnaire (soit 400 000 environ). Si on avait choisi de faire un arbre dont les noeuds sont toutes les combinaisons possibles de touches (jusqu'à 25 lettres) celui-ci aurait une taille gigantesque impropre à sa représentation dans la mémoire vive d'un ordinateur, soit au minimum : somme pour i=1 à 25 de 3^i (environ égal à 3^{26}), soit un arbre de 1000 milliard de noeuds environ!

Le code de la classe *T9Engine* implémentant la création de cette structure est le suivant :

```
public class T9Engine implements IT9Engine {
      TreeMap<String,LinkedList<Pair>> treeMapList=new TreeMap<String, LinkedList<Pair>>(); public T9Engine(String pathDict) {
           try {
           Scanner scanner = new Scanner(new File(pathDict));
           while (scanner.hasNext()) {
                String word=scanner.next().toLowerCase();
                double proba=scanner.nextDouble();
                insert(word,proba);
           }
           } catch (FileNotFoundException e) { e.printStackTrace(); }
      }

      public void insert(String word, double proba) {
           LinkedList<Pair> listWords;
           String code=getCode(word);
           if (!treeMapList.containsKey(code)) {
                // premiere insertion de mot pour ce code
                listWords=new LinkedList<Pair>();
                listWords.add(new Pair(word,proba));
          }
           else {
                // le code est deja associe a un ou plusieurs mots
                listWords=treeMapList.get(code);
                for(Pair pair:listWords)
                if (word.equalsIgnoreCase(pair.getWord()))
                     return; // doublon, on ignore le reste du code
                listWords.add(new Pair(word,proba));
                Collections.sort(listWords);
           }
           treeMapList.put(code,listWords);
           }
      public LinkedList<Pair> getWords(String code) {
           return treeMapList.get(code);
      }
      … autres méthodes et données de la classe...
}
```

On s'intéresse à la classe *Pair* qui est utilisée mais pas explicitée ci-dessus. Cette classe implémente l'association entre un mot et sa probabilité. Sa documentation au format *JavaDoc* est la suivante :

Class Pair

`java.lang.Object` **`Pair`**

All Implemented Interfaces:
java.lang.Comparable<Pair>

`public class` **`Pair`** `extends java.lang.Object`

Constructor Summary

`Pair``(java.lang.String word, double proba)`	

Method Summary

`double`	**`getProba`**`()`
`java.lang.String`	**`getWord`**`()`
`void`	**`setProba`**`(double proba)`
`void`	**`setWord`**`(java.lang.String word)`
`java.lang.String`	**`toString`**`()`

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, wait, wait, wait`

- Écrire complètement cette classe. La méthode *toString* notamment doit permettre un affichage d'une paire sous la forme *[chat, 0.489]* par exemple

```
public class Pair {
      private double proba;
      private String word;
      public Pair(String word, double proba) { this.proba=proba;this.word=word; }
      public void setProba(double proba) { this.proba=proba; }
      public void setWord (String word) { this.word=word; }
      public String getWord() { return word; }
      public double getProba() { return proba; }
      public String toString() { return "[ "+word+","+proba+" ]"; }
}
```

En réalité lorqu'on implémente la classe *Pair* à partir de la javadoc ci-dessus, et que l'on teste la classe *T9Engine*, la compilation du programme échoue avec le message suivant : *Bound mismatch: The generic method sort(List<T>) of type Collections is not applicable for the arguments (LinkedList<Pair>). The inferred type Pair is not a valid substitute for the bounded parameter <T extends Comparable< T>>*

- Quel est précisément l'instruction qui déclenche cette erreur dans la classe *T9Engine* ?

```
Collections.sort(listWords);
```

Pour corriger cette erreur on ajoute ceci à la déclaration de la classe *Pair*

```
public class Pair implements Comparable<Pair> {
. . . <reste de la classe>
}
```

Cette fois le message précédent disparaît, mais un nouveau message d'erreur apparaît localisé sur la déclaration ci-dessus : *The type Pair must implement the inherited abstract method Comparable<Pair>.compareTo(Pair)*

- Expliquer ce message

Comparable est une interface. Implémenter une interface est un engagement à implémenter les méthodes déclarées dans l'interface, ici la méthode *compareTo*. Le compilateur nous rappelle ici que ce travail n'a pas été effectué.

On souhaite maintenant implémenter la méthode *public int compareTo(Pair o)* dans la classe *Pair* de façon à définir pour les éléments de la liste des mots et probabilités une relation d'ordre basée sur la probabilité (pour être en mesure d'offrir à l'interface graphique une liste de mots classés par probabilités décroissantes). Rappel : la méthode *compareTo* retourne -1, 0 ou +1 selon les trois relations *supérieur*, *égal*, *inférieur*.

- Ecrire cette méthode *compareTo*

```
public int compareTo(Pair o) {
    if (o.getProba()>proba) return 1;
    if (o.getProba()<proba) return -1;
    return 0;
}
```

On suppose maintenant que les classes *IT9Engine*, *T9Engine* et *Pair* sont totalement construites. La *javadoc* de *IT9Engine* est la suivante :

Interface IT9Engine

All Known Implementing Classes:
T9Engine
public interface **IT9Engine**

Method Summary

java.lang.String	**getCode**(java.lang.String word)
java.util.LinkedList<Pair>	**getWords**(java.lang.String code)

Class T9Engine

java.lang.Object **T9Engine**

All Implemented Interfaces:
IT9Engine

public class **T9Engine**extends java.lang.Objectimplements IT9Engine

Constructor Summary

T9Engine(java.lang.String pathDict)	

Method Summary

java.lang.String	**getCode**(java.lang.String word)
java.util.TreeMap<java.lang.String,java.util.LinkedList<Pair>>	**getTreeMapList**()
java.util.LinkedList<Pair>	**getWords**(java.lang.String code)
void	**insert**(java.lang.String word, double proba)

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

La suite est constituée de courts exercices de codage : ils s'appuient sur la documentation concernant *IT9Engine*, *T9Engine*, *Pair*, *TreeMap* fournie dans ce sujet. Le code demandé prend sa place dans la section *main* d'une classe Launcher qu'il est donc inutile de réécrire à chaque fois. Le dictionnaire qu'on utilise est **/usr/dict/dico.txt**

```
public class Launcher {
public static void main(String[] args) {
. . . votre code ...
}
}
```

- Exercice 1 : Le programme demande à l'utilisateur de saisir un mot au clavier et affiche les *textonymes* de ce mot. Si le mot n'en possède pas il affiche *aucun textonyme*. si le mot n'existe pas dans le dictionnaire il affiche *mot inconnu*.

```
T9Engine t9Engine=new T9Engine();
Scanner scan=new Scanner(System.in);
String word=scan.next();
LinkedList<Pair> list=t9Engine.getWords(t9Engine.getCode(word));
if (list==null) System.out.println("mot inconnu");
else if (list.size()==0) System.out.println("aucun textonyme");
else for(Pair pair:list) System.out.println(pair.getWord());
```

- Exercice 2 : Ecrire un programme qui affiche le résultat suivant (donné à titre d'exemple, il ne s'agit pas d'utiliser un simple *println...*!)

Il y a 384267 mots dans le dictionnaire, dont 361045 sans textonyme.

```
T9Engine t9Engine = new T9Engine();
TreeMap<String, LinkedList<Pair>> tm=t9Engine.getTreeMapList();
int countTotal=0,countMono=0;
for(String key:tm.descendingKeySet()) {
if (tm.get(key).size()==1) countMono++;
countTotal+=tm.get(key).size();
}
System.out.print("Il y a "+countTotal+" mots dans le dictionnaire");
System.out.println(", dont "+countMono+" sans textonymes");
```

- Exercice 3 : écrire un programme qui détermine le mot du dictionnaire comportant le plus de textonymes

```
int sizeMax=0; String keyMax=null;
for(String key:tm.descendingKeySet()) {
if (tm.get(key).size()>sizeMax) { keyMax=key; sizeMax=tm.get(key).size(); }
}
System.out.println("Plus longue liste : "+tm.get(keyMax).size()+ " mots");
for(Pair pair:tm.get(keyMax)) System.out.println(pair.getWord());
```

```
Plus longue liste : 11 mots
couter
coutes
bouter
aouter
aoutes
bottes
cottes
couver
botter
boutes
couves
```

- Etendre (hors *Launcher*) la classe *T9Engine* par héritage en créant la classe *T9EngineEx* conforme à la documentation suivante

Class T9EngineEx

java.lang.Object T9Engine **T9EngineEx**

All Implemented Interfaces:
IT9Engine

public class **T9EngineEx**extends T9Engine

Constructor Summary

T9EngineEx(java.lang.String path)

Method Summary

java.util.LinkedList<java.lang.String>	**getTextonyms**(java.lang.String word)

Methods inherited from class T9Engine

getCode, getTreeMapList, getWords, insert

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

T9EngineEx

public **T9EngineEx**(java.lang.String path)

Method Detail

getTextonyms

public java.util.LinkedList<java.lang.String> **getTextonyms**(java.lang.String word)

Parameters:
word - : le mot dont on recherche les textonymes
Returns:

- **null** si **word** est absent du dictionnaire
- liste vide si **word** est present mais sans textonyme
- liste des textonymes de **word** sinon (sans **word** lui-même)

```
public class T9EngineEx extends T9Engine {
public T9EngineEx(String path) {
super(path);
}
public LinkedList<String> getTextonyms(String word) {
String code = getCode(word);
if (!treeMapList.containsKey(code))
return null;
if (treeMapList.get(code).size() == 1)
return new LinkedList<String>();
LinkedList<String> list = new LinkedList<String>();
for (Pair pair : treeMapList.get(code)) {
if (!pair.getWord().equalsIgnoreCase(word))
list.add(pair.getWord());
}
return list;
}
}
```

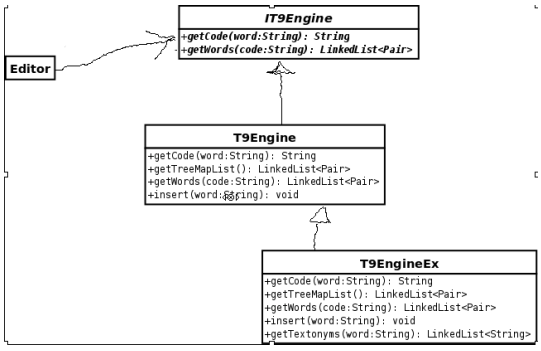
Les variantes de *T9Engine* sont bien sûr utilisées par l'éditeur incorporé au logiciel du téléphone. Les frappes successives sur les touches sont gérées par l'éditeur (*Editor*) et celui-ci fait appel au T9Engine au fur et à mesure de la frappe pour proposer à l'utilisateur des mots pertinents. Le concepteur de l'application tient toutefois dans un souci de découplage de ses classes à préserver les évolutions possibles de T9Engine sans compromettre le reste du code.

- Pour quelle raison le concepteur souhaite-t-il préserver autant que possible le découplage des classes ?

Pour assurer un certain niveau de qualité interne de code : fiabilité, évolutivité, réusabilité

- Moins les classes sont couplées, moins les erreurs se propagent
- Moins les classes sont couplées, moins les évolutions impactent en cascade l'ensemble des classes
- Moins les classes sont couplées, plus le potentiel de réutilisation de chaque classe est élevé.

- Dessiner les relations UML entre les 4 entités ci-dessous qui expriment les relations entre ces entités et qui correspond aux souhait du concepteur tel que décrit ci-dessus.



java.util

Class TreeMap<K,V>

java.lang.Object └ java.util.AbstractMap<K,V> └ **java.util.TreeMap<K,V>**

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values
All Implemented Interfaces:
Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>

public class **TreeMap<K,V>**extends AbstractMap<K,V>implements NavigableMap<K,V>, Cloneable, Serializable

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be *consistent with equals* if this sorted map is to correctly implement the `Map` interface. (See `Comparable` or `Comparator` for a precise definition of *consistent with equals*.) This is so because the `Map` interface is defined in terms of the equals operation, but a map performs all key comparisons using its `compareTo` (or `compare`) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Map` interface.

This class is a member of the Java Collections Framework.

Since:
1.2

See Also:
`Map`, `HashMap`, `Hashtable`, `Comparable`, `Comparator`, `Collection`, Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class java.util.AbstractMap

`AbstractMap.SimpleEntry<K,V>`, `AbstractMap.SimpleImmutableEntry<K,V>`

Constructor Summary

`TreeMap``()` Constructs a new, empty tree map, using the natural ordering of its keys.	
`TreeMap``(Comparator<? super K> comparator)` Constructs a new, empty tree map, ordered according to the given comparator.	
`TreeMap``(Map<? extends K,? extends V> m)` Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.	
`TreeMap``(SortedMap<K,? extends V> m)` Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.	

Method Summary

`Map.Entry<K,V>`	**`ceilingEntry`**`(K key)` Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
`K`	**`ceilingKey`**`(K key)` Returns the least key greater than or equal to the given key, or `null` if there is no such key.
`void`	**`clear`**`()` Removes all of the mappings from this map.
`Object`	**`clone`**`()` Returns a shallow copy of this `TreeMap` instance.
`Comparator<? super K>`	**`comparator`**`()` Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
`boolean`	**`containsKey`**`(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue`**`(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`NavigableSet<K>`	**`descendingKeySet`**`()` Returns a reverse order `NavigableSet` view of the keys contained in this map.
`NavigableMap<K,V>`	**`descendingMap`**`()` Returns a reverse order view of the mappings contained in this map.
`Set<Map.Entry<K,V>>`	**`entrySet`**`()` Returns a `Set` view of the mappings contained in this map.
`Map.Entry<K,V>`	**`firstEntry`**`()` Returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`K`	**`firstKey`**`()` Returns the first (lowest) key currently in this map.
`Map.Entry<K,V>`	**`floorEntry`**`(K key)` Returns a key-value mapping associated with the greatest key less than or equal to the given key, or `null` if there is no such key.
`K`	**`floorKey`**`(K key)` Returns the greatest key less than or equal to the given key, or `null` if there is no such key.
`V`	**`get`**`(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`SortedMap<K,V>`	**`headMap`**`(K toKey)` Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`NavigableMap<K,V>`	**`headMap`**`(K toKey, boolean inclusive)` Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
`Map.Entry<K,V>`	**`higherEntry`**`(K key)` Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.
`K`	**`higherKey`**`(K key)` Returns the least key strictly greater than the given key, or `null` if there is no such key.
`Set<K>`	**`keySet`**`()` Returns a `Set` view of the keys contained in this map.
`Map.Entry<K,V>`	**`lastEntry`**`()` Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`K`	**`lastKey`**`()` Returns the last (highest) key currently in this map.
`Map.Entry<K,V>`	**`lowerEntry`**`(K key)` Returns a key-value mapping associated with the greatest key strictly less than the given key, or `null` if there is no such key.
`K`	**`lowerKey`**`(K key)` Returns the greatest key strictly less than the given key, or `null` if there is no such key.
`NavigableSet<K>`	**`navigableKeySet`**`()` Returns a `NavigableSet` view of the keys contained in this map.
`Map.Entry<K,V>`	**`pollFirstEntry`**`()` Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`Map.Entry<K,V>`	**`pollLastEntry`**`()` Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`V`	**`put`**`(K key, V value)` Associates the specified value with the specified key in this map.

void	**putAll**(Map<? extends K,? extends V> map) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key) Removes the mapping for this key from this TreeMap if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

Methods inherited from class java.util.AbstractMap
equals, hashCode, isEmpty, toString

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Map
equals, hashCode, isEmpty