Programmation avancée – langage Java & environnement eclipse

14 février 2014

- Objectifs pédagogiques :
  Maîtriser les concepts avancés de la programmation objet en Java. Apprendre à identifier et utiliser les patrons (design pattern) adaptés à une situation de conception.
- Capacités et compétences visées :
  Maîtrise du langage Java et des concepts objets avancés cités dans le programme. Effectuer un choix de patron (pattern).

Utilisation de concepts avancés de programmation afin de résoudre des problématique complexes. En extraire un modèle (i.e.une conception) générique afin d'en élargir le domaine d'application.

Java	Technologie / Langage
Bytecode	Flux d'octets binaire codant des instructions devant être exécutées par une machine virtuelle.
JDK	Java Development Toolkit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
J2SE	Java (2) Standard Edition
J2EE	Java (2) Entreprise Edition
J2ME	Java (2) Micro Edition
SDK	Software Development Kit
API	Application Programming Interface

Langage de programmation : langage formel conçu pour communiquer [1] avec une machine. Selon le degrès d'abstraction, il permet d'exprimer des algorithmes et des concepts et de les traduire en code binaire (programme / bibliothèque) exécutable sur ordinateur [2]. Plusieurs paradigmes existes.

1. Emettre des instructions ou des commandes.

2. Étymologie du mot Ordinateur (source Wikipédia) : Le mot ordinateur fut introduit par IBM France en 1955 après que François Girard, alors responsable du service publicité de l'entreprise, eut l'idée de consulter son ancien professeur de lettres à Paris, Jacques Perret, lui demandant de proposer un « nom français pour sa nouvelle machine électronique destinées au traitement de l'information (IBM 650), en évitant d'utiliser la traduction littérale du mot anglais "computer" ("calculateur" ou "calculatrice"), qui était à cette époque plutôt réservé aux machines scientifiques »

Ce dernier proposa un mot composé centré autour d'« ordinateur », un mot tombé en désuétude, désignant anciennement un ordonnateur : celui qui met en ordre et qui avait aussi la notion d'ordre ecclésiastique dans l'église catholique (ordinant)...

Programmation impérative : C, Pascal, ...

Programmation orientée objet : Smalltalk, C++, Java, Objective-C, ...

Programmation fonctionnelle : Lisp, Scheme, ...

Programmation déclarative : Prolog, ...

...

- Code pour l'embarqué, SUN 1991.
- Bytecode pour exécution sur JVM.
- HotJava en 1995 :
  Navigateur écrit en Java et exécutant des applets.
- Java 2 en 1999.
- J2SE 1.4 en 2002.
- Rachat de SUN par Oracle et acquisition de Java en 2010.

- Encapsulation.
- Polymorphisme.
- Héritage ~~multiple~~ mais les interfaces.
- Les type primitifs.
- Les méthodes de classe.

Le langage Java vient avec un ensemble de règles liées au conventions de codage (nommage) [Con99].

- un commentaire : `/* ceci est un commentaire */`
- un nom de classe :
```
public class NomDeMaClasse {
...
}
```
- un nom de propriété :
```
private String ceciEstUneProprieteDeTypeString = "Hello World";
```
- un nom de méthode :
```
public void ceciEstUneMethodePublique {
...
}
```
- un nom de variable :
```
int ceciEstUneVariableDeTypeInt = 0;
```

```java
/*!\file HelloWorld
 * \brief mon premier code java.
 */
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World !");
    }
}
```

Figure: Mon premier code Java ...

```
#  Makefile
#  Auteur : Farès BELHADJ
#  Email  : amsi@ai.univ-paris8.fr
#  Date   : 10/10/2013

#définition des commandes utilisees
JAVA = java
JAVAC = javac
RM = rm -f
SOURCES = HelloWorld.java
OBJS = $(SOURCES:.java=.class)
PROGRAMME = exe

#dépendances
all: $(OBJS)

%.class: %.java
        $(JAVAC) $<

clean:
        @$(RM) $(OBJS) *~
```

Figure: ... et son Makefile.

Type	Occupation mémoire	Plage de valeurs
boolean	1 octet	true ou false
byte	1 octet	-128 $\rightarrow$ 127
char	2 octet	'\u0000' $\rightarrow$ '\uffff'
unsigned char	1 octet	0 $\rightarrow$ 255
int	4 octets	$2^{31} \rightarrow 2^{31} - 1$
unsigned int	4 octets	$0 \rightarrow 2^{32} - 1$
short	2 octets	-32768 $\rightarrow$ 32767
unsigned short	2 octets	0 $\rightarrow$ 65535
long	8 octets	$-2^{63} \rightarrow 2^{63} - 1$
unsigned long	8 octets	$0 \rightarrow 2^{64} - 1$
Type $\rightarrow$ virgule flottante		
float	4 octets	$3.4 \times 10^{-38} \rightarrow 3.4 \times 10^{38}$ (IEEE 754)
double	8 octets	$1.7 \times 10^{-308} \rightarrow 1.7 \times 10^{308}$ (IEEE 754)

Table: Types de données primitifs

Type primitif	Wrapper
boolean	Boolean
byte	Byte
int	Integer
short	Short
long	Long
float	Float
double	Double

Table: Wrappers de types primitifs

Priorité	Opérateur	Description	Exemple
0	( )	appel de fonction, associativité	foo() ; a = (b + c) * d ;
	[ ]	indexation	int tab[3] ; tab[0] = tab[1] = tab[2] = 0 ;
	.	nommage d'un champ	obj.cdr = null ;
1	!	négation	(!*a*) est vraie si *a* est fausse
	~	complément à 1	a & (~a) = 0
	-	opposé	
	++	incrémentation	i++ ; ++i ;
	- -	décrémentation	i- - ; - -i ;
	(type_de_donnée)	force le type (cast)	int i = (int)1.5 ;
	new	nouvel objet	Integer i = new Integer() ;
2	*	Multiplication	
	/	Division	
	%	Modulo	
3	+	Addition	
	-	Soustraction	
4	<<	Décalage à gauche	
	>>	Décalage à droite cons. signe	
	>>>	Décalage à droite	

Table: Table des priorités des opérateurs Java (1/2).

Priorité	Opérateur	Description	Exemple
5	<	Strictement inférieur	
	<=	Inférieur ou égal	
	>	Strictement supérieur	
	>=	Supérieur ou égal	
6	==	Egal	
	! =	Différent	
7	&	"et" binaire	
8	^	"ou" exclusif binaire	
9	|	"ou" binaire	
10	&&	"et" logique	
11	||	"ou" logique	
12	? :	conditionnelle	`c = (a < b)? a : b;`
13	= *= /= %= += -= ^= &= <<= >>= |=	Affectations	
14	,	Séquence	

Table: Table des priorités des opérateurs Java (2/2).

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
extends	false	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	null	package	private	protected
public	return	short	static	strictfp
super	switch	synchronized	this	throw
throws	transient	true	try	void
volatile	while			

Expression : en langage informatique, une expression est une combinaison de lexèmes (atome d'un point de vue lexical) pouvant être évaluée selon des règles de priorité et produisant un résultat.

En java, une expression suivie d'un " ;" est appelée *instruction*. Les instructions pouvent être regroupées au sein d'un même **bloc** commençant par une accolade ouvrante "{" et se terminant par une accolade fermante "}".

Branchement : le conditionnement de l'exécution d'une instruction ou d'un bloc par l'évaluation d'une expresssion qui a valeur de test vrai/faux (`true`/`false`).

Condition : expression dont la valeur est booléenne (`true`/`false`), elle même pouvant être composée de plusieurs conditions combinées via des opérateurs logiques (`||`, `&&`, `!`).

À faire : évaluation séquentielle des conditions (expressions) combinées via opérateur logique.

La structure conditionnelle `if`

```
if(condition) {
   instruction_cond_vraie_1;
   ...
   instruction_cond_vraie_m;
} else {
   instruction_cond_fausse_1;
   ...
   instruction_cond_fausse_n;
}
```

La structure conditionnelle `switch`

```
switch(expression) {
case valeur_A_pour_expression:
   instruction(s)
   break;
case valeur_B_pour_expression:
   instruction(s)
   break;
...
default:
/* si aucune des valeurs précédentes "matche" */
   instruction(s)
   break;
}
```

À faire : le `break` n'est pas obligatoire.

Opérateurs d'affectation conditionnelle "`? :`"

```
lvalue = (condition) ? expr_si_vraie : expr_sinon;
```

À faire : écrire la méthode de classe `min` renvoyant pour 2 entiers (`int`) passés en argument la plus petite des deux valeurs.

Boucle : Structure permettant d'itérer plusieurs fois sur une (série d') instruction(s). L'itération est parfois conditionnée dans la structure même de la boucle ; parfois l'arrêt est conditionné par une instruction de branchement suivie d'une sortie (`break`, `return`, `exit`). La condition est appelée **condition d'arrêt**. Il est aussi possible, dans le corp d'une boucle, de passer à l'itération suivante en sautant plusieurs instructions restantes ; ceci est possible à l'aide du saut inconditionnel `continue`.

la boucle `for`

```
for(init_compteur; condition_arret; incrément_compteur) {
   instruction(s)
}
/* Exemple : */
for(int i = 0; i < 7; i++)
   System.out.println(i);
```

À faire : Est-ce que le code ci-après est équivalent au précédent ?

```
int i;
for(i = 0; i < 7; i++);
   System.out.println(i);
```

Variantes de l'exemple précédent

```
System.out.println(‘‘Variante 1’’);
for(int i = 0; i < 7; System.out.println(i++));
System.out.println("Variante 2");
{
   int i = 0;
   for(; i < 7;)
      System.out.println(i++);
}
System.out.println("Variante 3");
{
   int i = 0;
   for(;;) {
      if(!(i < 7))
         break;
      System.out.println(i++);
   }
}
```

la boucle `while`

```
while(condition_arret) {
   instruction(s)
}
/* Exemple : */
int i = 0;
while(i < 7) {
   System.out.println(i);
   i++;
}
```

À faire : Écrire quelques variantes de cette forme de boucle.

la boucle `do while`

```
do {
   instruction(s)
} while(condition_arret);
/* Exemple : */
int i = 0;
do {
   System.out.println(i);
   i++;
} while(i < 7);
```

À faire : Est-ce équivalent à l'écriture par boucle `while` ?

Différence entre `while` et `do-while`

```
int i, n = ...;
.
...
.
i = 0;
while(i < n) {
   System.out.println(i);
   i++;
}
i = 0;
do {
   System.out.println(i);
   i++;
} while(i < n);
```

le `for` ensembliste

```
/* Pour tout élément dans l'ensemble */
for(type_ou_class variable: ensemble) {
   instruction(s)
}
/* Exemple : */
int t[7] = {0, 1, 2, 3, 4, 5, 6};
/* ou int[] t = {0, 1, 2, 3, 4, 5, 6}; */
for(int i: t)
   System.out.println(i);
```

À faire : `i` est-elle l'indice ou la valeur ?

Exercices :

- Écrire la méthode de classe `pair` (ou `impair`) imprimant tous les nombres pairs (ou impairs) inférieurs à `n`, `n` étant le paramètre de la méthode.
- Faire la même chose en récursif.
- Utiliser le `for` ensembliste.
- Utiliser le `continue`.

```
1   class SuperBreak {
        static public void main(String[] args) {
3           int i = 0, j = 0;
            Boucle1: for(i = 0; i < 100; i++) {
5               Boucle2: for(j = 0; j < 100; j++) {
                    if(i == 30 && j == 50)
7                       break Boucle1;
                }
9           }
            System.out.println(i + ", " + j);
11          return;
        }
13  }
```

Figure: Pas de goto, mais le break-label est là. Comment ça marche avec le continue ?

```
1   class SuperContinue {
        static public void main(String[] args) {
3           int i = 0, j = 0, k = 0;
            Boucle1: for(i = 0; i < 100; i++) {
5               Boucle2: for(j = 0; j < 100; j++) {
                    if(i == 30 && j == 17)
7                       continue Boucle1;
                    k++;
9               }
            }
11          System.out.println(i + ", " + j + ", " + k);
            return;
13      }
    }
```

Figure: Et on continue ...

```
class TestTry {
    static public void main(String[] args) {
        int i = 0, j = 0, k = 0;
        try {
            for(i = 0; i < 100; i++)
                for(j = 0; j < 100; j++)
                    for(k = 0; k < 100; k++)
                        if(i == 50 && j == 50 && k == 50)
                            throw new Exception();
        } catch(Exception e) {
            System.out.println(i + ", " + j + ", " + k);
            return;
        }
        System.out.println("FIN");
    }

}
```

Figure: Que fait ce programme ?

```
1   class Test {
        String[] _data;
3       Test(String args[]) {
            _data = args;
5       }
        private boolean isDigit(String str) {
7           char c;
            for(int i = 0; i < str.length(); i++)
9               if(!((c = str.charAt(i)) >= '0' && c <= '9'))
                    return false;
11          return true;
        }
13      private String encode(String str) {
            String ns = "";
15          for(int i = 0; i < str.length(); i++)
                ns += (char)('A' + (str.charAt(i) - '0'));
17          return ns;
        }
19      public void print() {
            for(String s: _data)
21              System.out.println(isDigit(s) ? "encode : <" + encode(s) + ">" : "non modifie : " + s);
        }
23      static public void main(String[] args) {
            Test t = new Test(args);
25          t.print();
        }
27  }
```

Figure: Que fait ce programme ? Compiler et exécuter avec : javac Test.java && java Test 12345 123AZ 123ER 878576846745765

Une classe est la description de la structure que va avoir l'objet qui est l'instance de cette même classe. Une classe est "un moule", "un plan", "une idée", "un concept", et son instance (l'objet né ou instancié de cette classe) et la concrétisation de ce concept. Une classe est définie de la manière suivante :

```
1  [static] [public,private,protected] class NomDeClasse [extends <ClasseMère>] [implements <Interface1,...>
       /* Description de la classe :
3       * Propriété & Méthodes d'instance ou de classe, interface ...
        */
5      private int UneDonnéeMembre = 0;
       public void uneMethodeDInstance() {
7          //Corps de la méthode
       }
9      static public void uneMethodeDeClasse() {
           //Corps de la méthode
11     }
   }
```

Une donnée, une fonction ou une classe peut être d'instance ou de classe, c'est à dire que l'élément en question peut appartenir à un objet ou à une classe (la structure) d'objets et que, dans ce dernier cas, il n'y aurait pas besoin d'instancier un objet de cette classe pour avoir accès à cette donnée ou fonctionnalité.
L'élement qui est de classe est précédé, lors sa déclaration, par le mot clé `static`.

L'accessibilité définie la portée, d'un point de vue droit d'accès, d'une donnée, d'une fonction et d'une manière générale d'un élément du langage telle qu'une classe ou une interface.
Les mots clés permettant de modifier l'acessibilité d'un élément sont :

public : **l'élément est accessible pour tous (du moment où, à partir d'une position hiérarchique, tous ces conteneurs successifs le sont) ;**

private : **l'élément n'est accessible que depuis l'intérieur de son conteneur (ie. au sein de sa classe originale) ;**

protected : **à mi-chemin entre public et private, l'élément est accessible au sein de sa classe et de ses héritières ;**

NON_SPEC : **quand rien n'est spécifié, l'accessibilité par défaut est une accessibilité aux "amis" ; par "amis" nous entendrons conteneurs (dans le sens classe et package) de même niveau (ex. éléments du même package) ou de niveau plus bas ;**

```
1
2
3
4
```

```java
public class Test {
   public static void main(String[] args) {
      new Lettre();
   }
}
```

Fichier Test.java

```
1
2
3
4
5
```

```java
class Lettre {
    private char moi = 'A';
    Lettre() {
        System.out.println(this.moi);
    }
}
```

Fichier Lettre.java

Compilation et exécution : `javac Test.java && java Test`

Quel est le résultat ?

```
class Lettre {
    private char moi = 'A';
    private void whoAmI() {
        System.out.println(this.moi);
    }
    Lettre() {
        this.whoAmI();
    }
}
```

Fichier Lettre.java

```
1  class Lettre {
       private char moi = 'A';
3      private void whoAmI() {
           System.out.println(this.moi);
5      }
       Lettre() {
7          this.whoAmI();
       }
9      Lettre(char moi) {
           this.moi = moi;
11         this.whoAmI();
       }
13 }
```

Fichier Lettre.java

Modifier la ligne 3 dans Test.java et mettre `new Lettre('B');` à la place. Quel est le résultat de l'exécution ?

```
1  class Lettre {
       private char moi = 'A';
3      private void whoAmI() {
           System.out.println(this.moi);
5      }
       Lettre() {
7          this.whoAmI();
       }
9  }
```

Fichier Lettre.java

```
1  class Lettre {
       private char moi = 'A';
3      Lettre() {}
       Lettre(char moi) { this.moi = moi; }
5      public void print() {
           System.out.println(this.moi);
7      }
   }
```

Fichier Lettre.java

```
   public class Test {
2      public static void main(String[] args) {
         Lettre l = new Lettre('C');
4        l.print();
       }
6  }
```

Fichier Test.java

```
public class Test {
2     public static void main(String[] args) {
        Lettre l = new Lettre();
4       l.moi = 'C';
        l.print();
6     }
}
```

Fichier Test.java

Quel est le résultat ?

```
1  class Lettre {
       private char moi = 'A';
3      Lettre() {}
       Lettre(char moi) { setMe(moi) }
5      public void print() {
           System.out.println(this.moi);
7      }
       public void setMe(char moi) { this.moi = moi; }
9  }
```

Fichier Lettre.java

```
1  public class Test {
       public static void main(String[] args) {
3        Lettre l = new Lettre();
         l.setMe('D');
5        l.print();
       }
7  }
```

Fichier Test.java

```
1  class Lettre {
       private char moi = 'A';
3      Lettre() {}
       Lettre(char moi) {
5        setMe(moi);
       }
7      public void print() {
         print(false);
9      }
       public void print(boolean withClassName) {
11       if(withClassName)
            System.out.println(this.getClass().getName() + " : " + this.moi);
13       else
            System.out.println(this.moi);
15     }
       public void setMe(char moi) {
17       this.moi = moi;
       }
19 }
```

Fichier Lettre.java

Changez la ligne 5 du Test.java et mettre `1.print(true);`. Quel est le résultat obtenu après exécution ?

Un peu d'héritage (mots clés : `extends`, `super`, `@Override`)

```
1  class LettrePlus extends Lettre {
   }
```

Fichier LettrePlus.java

```
   public class Test {
2      public static void main(String[] args) {
         LettrePlus l = new LettrePlus();
4        l.setMe('D');
         l.print(true);
6      }
   }
```

Fichier Test.java

Quel est le résultat obtenu après exécution ?

```
1  class LettrePlus extends Lettre {
       @Override
3      public void print() {
         print(true);
5      }
       public void print(boolean withClassName) {
7        System.out.println("C'est super !");
         super.print(withClassName);
9      }
   }
```

Fichier LettrePlus.java

```
   public class Test {
2      public static void main(String[] args) {
         LettrePlus l = new LettrePlus();
4        l.setMe('D');
         l.print();
6      }
   }
```

Fichier Test.java

Quel est le résultat obtenu après exécution ?

```
1   public class Test {
        static class A {
3           A() {
                System.out.println("A");
5           }
            public void foo() {
7               System.out.println("A.foo()");
            }
9       }
        static public class B extends A{
11          B() {
                System.out.println("B");
13          }
            @Override
15          public void foo() {
                System.out.println("B.foo()");
17          }
        }
19      public void test() {
            A i = new A();
21          B j = new B();
        }
23      public static void main(String[] args) {
            new Test().test();
25          new Test.B().foo();
        }
27  }
```

- Un package est équivalent à une notion de bibliothèque (dans d'autres langages) ; celà peut être vu comme une compilation de plusieurs classes ;
- Les packages sont hiérarchisables (comme des dossiers) ;
- L'accessibilité est aussi liée aux packages ;
- Le nommage des packages est souvent enrichie du domaine et du nom de la société ;
- Mots clés : package, import, CLASSPATH ...

- Une classe abstraite est une classe qui ne peut être instanciable : elle sert donc de base pour les classes qui en dérivent et qui peuvent, elles, être instanciables. C'est un ensemble commun non instanciable ;
- Une méthode abstraite est une méthode dont le prototype est déclaré mais dont le corps est non défini ; elle doit être définie dans une des classes dérivée pour être utilisée.

```
1   package fr.fbelhadj.cl;
    abstract class AlphaNum {
3       protected char moi;
        AlphaNum() {}
5       AlphaNum(char moi) {}
        public void print() { print(false); }
7       public void print(boolean withClassName) {
            if(withClassName)
9               System.out.println(this.getClass().getName()
                                   + " : " + this.moi);
11          else
                System.out.println(this.moi);
13      }
        public void setMe(char moi) { this.moi = moi; }
15      public abstract boolean isValide();
    }
```

Fichier MonProjetPackage/fr/fbelhadj/cl/AlphaNum.java

```
    package fr.fbelhadj.cl;
2   public class Lettre extends AlphaNum {
        private char moi = 'A';
4       public Lettre() {}
        public Lettre(char moi) { setMe(moi); }
6   }
```

Fichier MonProjetPackage/fr/fbelhadj/cl/Lettre.java

```
    package fr.fbelhadj.cl;
2   public class Chiffre extends AlphaNum {
        private char moi;
4       private Chiffre() {}
        public Chiffre(char moi) { setMe(moi); }
6       public boolean isValide() {
            return moi >= '0' && moi <= '9';
8       }
    }
```

Fichier MonProjetPackage/fr/fbelhadj/cl/Chiffre.java

```
1   import fr.fbelhadj.cl.*;
    public class Test {
3       public static void main(String[] args) {
            fr.fbelhadj.cl.Lettre l = new fr.fbelhadj.cl.Lettre('D');
5           l.print();
            Chiffre c = new Chiffre('0');
7           c.print(true);
            c.setMe('T');
9           System.out.println(c.isValide());
        }
11  }
```

Fichier MonProjetPackage/Test.java - Faire javac Test.java && java Test

- Une interface : une collection de prototypes de méthodes (abstraites) qui doivent être implémentées par la classe qui implémente cette interface ;
- Une interface peut être vue comme une classe 100% abstraite ;
- C'est un moyen détourné pour faire de l'héritage multiple.

```
1   public class Test {
        private interface IMonInterface {
3           public void imprime();
        }
5       static class A implements IMonInterface {
            private int a, b;
7           A(int a, int b) {
                this.a = a;
9               this.b = b;
            }
11          public void imprime() {
                System.out.println(a + "_" + b);
13          }
        }
15      static class B implements IMonInterface {
            String chaine;
17          B(String str) {
                chaine = new String(str);
19          }
            public void imprime() {
21              System.out.println(chaine);
            }
23      }
        public static void main(String[] args) {
25          new A(1, 2).imprime();
            new B("Hello World !").imprime();
27          Object o = new B("Re-Hello World !");
            ((IMonInterface)o).imprime();
29      }
    }
```

- **Exercice :** basée sur les interfaces, réaliser un programme faisant un appel en boucle d'une méthode qui doit faire un init une et une seule fois puis faire (seulement) ce pour quoi elle a été faite.

  Exemple pratique : écrire une méthode d'instance qui se nomme `trie` et qui réalise le tri à l'étape `i` d'un tri par sélection (une sélection à la fois). Cette méthode, lors du premier appel, allouera et initialisera le tableau de données à trier. Utiliser les interfaces pour éviter de faire un test "Est-ce que la donnée a été initialisée ? " à chaque étape.

- L'exemple ci-après effectue un tri par sélection au coup par coup. Remarquez qu'à chaque fois nous testons si le tableau est bien initialisé (ça ne sert réellement qu'une fois).

  Les assertions : moyen de définir des pré-requis (ou contraintes obligatoire) à l'exécution du code (utilisez `java -ea ...` pour les prendre en compte). La syntaxe est la suivante :

  ```
  assert condition [: object];}
  ```

  où `object` contient une information concernant les raisons de l'échec de l'assertion ; généralement `object` est une chaine de caractère expliquant l'échec.

```
public class Tri {
    static class A  {
        private int a[] = null;
        private int taille, debut;
        A(int taille) {
            /* Un exemple d'assertion : exécuter (java) avec l'option
             * -ea pour voir apparaître l'assertion en cas de non
             * satisfaction de la contrainte */
            assert taille > 0 : "La taille du tableau doit être > 0";
            this.taille = taille;
        }
        public boolean trie() {
            if(a == null) {
                a = new int[taille];
                for(int i = 0; i < taille; i++)
                    a[i] = (int)(taille * Math.random());
                debut = 0;
                return false;
            }
            int p = debut, v = a[p];
            for(int i = debut + 1; i < taille; i++)
                if(v > a[i])
                    v = a[p = i];
            a[p] = a[debut];
            a[debut] = v;
            return ++debut >= taille - 1;
        }
        public void imprime() {
            for(int i: a)
                System.out.println(i);
        }
    }
    public static void main(String[] args) {
        A a = new A(100);
        while(!a.trie());
        a.imprime();
    }
}
```

- L'exemple ci-après utilise les interfaces pour réaliser le tri sans refaire le test concernant la disponibilité des données à chaque fois. Au début, la référence `fct` pointe sur la première instance de l'interface `fct0` ce qui permet de réaliser l'initialisation des données et aussi faire passer la référence vers `fct1`. Cette référence reste ensuite attachée à `fct1` qui fait effectivement le tri des données sans se poser la question de la disponiblité des données.

```
public class Tri {
    static class A  {
        private interface ITrie {
            public boolean trie();
        }
        private int a[] = null;
        private int taille, debut;
        private ITrie fct = null;
        private ITrie fct0 = new ITrie() {
                @Override
                public boolean trie() {
                    a = new int[taille];
                    for(int i = 0; i < taille; i++)
                        a[i] = (int)(taille * Math.random());
                    debut = 0;
                    fct = fct1;
                    return false;
                }
            };
        private ITrie fct1 = new ITrie() {
                @Override
                public boolean trie() {
                    int p = debut, v = a[p];
                    for(int i = debut + 1; i < taille; i++)
                        if(v > a[i])
                            v = a[p = i];
                    a[p] = a[debut];
                    a[debut] = v;
                    return ++debut >= taille - 1;
                }
            };
        A(int taille) {
            assert taille > 0 : "taille doit être supérieure à 0";
            this.taille = taille;
            fct = fct0;
        }
        public boolean trie() {
            return fct.trie();
        }
        public void imprime() {
            for(int i: a)
                System.out.println(i);
        }
    }
    public static void main(String[] args) {
        A a = new A(100);
        while(!a.trie());
        a.imprime();
    }
}
```

- **Exercice :** basée sur les interfaces, et dans la même lignée que précédemment, réaliser un programme faisant un "ping-pong" entre méthodes.

  Exemple pratique : Réaliser une classe `Chante` dont la méthode `joue` imprime (println) tour à tour : "Ding", "Dang" et "Dong" sans jamais faire de test relatif au tour.

```
public class Chante {
    private interface IChante {
        public void chante();
    }
    IChante referenceVersChanteur = null;
    IChante ding = new IChante() {
            public void chante() {
                System.out.println("Ding");
                referenceVersChanteur = dang;
            }
        };
    IChante dang = new IChante() {
            public void chante() {
                System.out.println("Dang");
                referenceVersChanteur = dong;
            }
        };
    IChante dong = new IChante() {
            public void chante() {
                System.out.println("Dong");
                referenceVersChanteur = ding;
            }
        };
    public Chante() {
        referenceVersChanteur = ding;
    }
    public void joue() {
        referenceVersChanteur.chante();
    }
}
```

- Les exceptions font partie d'un système de gestion des erreurs produites lors de l'exécution d'un programme Java. Ainsi, une méthode devant réaliser un traitement peut, en cas d'échec, générer une exception ; cette dernière pouvant être typée en fonction de la nature de l'erreur ;
- La partie du code devant être surveillée, car elle serait susceptible de générer une exception, est ainsi placée dans le bloc `try` du branchement `try-catch` ;
- Si une exception est générée par la partie surveillée, l'erreur est "catchée" dans le bloc correspondant au type de l'exception (une analogie est possible avec les `switch`) ;
- Les `catch` d'exception doivent être appelés de la classe la plus spécialisée à la plus générique (car qui peut le plus peu le moins). Ainsi le `catch` la classe d'exception englobant le plus finement le type de l'exception retournée est choisi pour "catcher" l'erreur.

- Une méthode générant une exception contient, dans sa déclaration, le mot clé `throws` suivi du type d'exception ;
- Une exception est générée (généralement à l'endroit où il y a l'erreur) par le mot clé `throw` suivi d'une nouvelle instance d'exception ;
- Chacun peut créer son propre type d'excetion du moment où cette dernière dérive de la classe `Exception` ou de l'une de ses dérivées.

Déclanchement d'une exception lors d'un parse de float.
Remarquez les impressions sur la sortie standard et la sortie d'erreurs ainsi que le flush (force le rafraîchissement).

```
public class Exception01 {
    static public void main(String str[]) {
        try {
            /* mettre f à la place de p pour que la conversions se
             * fasse sans erreur */
            float t = Float.parseFloat("3.1415p");
            System.out.println(t);
        } catch(NumberFormatException nfe) {
            System.out.println("Je suis ici");
            System.out.flush();
            System.err.println("Exception :" + nfe.getClass().getName() +
                               " - " + nfe.getMessage() +
                               " - in " + Exception01.class.getName());
            System.err.flush();
        } catch(Exception e) {
            System.out.println("Je suis là");
            System.out.flush();
            System.err.println("Exception :" + e.getClass().getName() +
                               " - " + e.getMessage() +
                               " - in " + Exception01.class.getName());
        }
    }
}
```

Ici l'exception est déclanchée et est catchée par la cas général (`Exception`) étant donné que l'exception émise est une `ClassCastException` et que cette enttrée n'existe pas.

```
1  public class Exception02 {
       static public void main(String str[]) {
3            try {
                 /* mettre f à la place de p pour que la conversions se
5                 * fasse sans erreur */
                 Object t = (Object)(new Float(Float.parseFloat("3.1415f")));
7                System.out.println((Integer)t);
             } catch(NumberFormatException nfe) {
9                System.out.println("Je suis ici");
                 System.out.flush();
11               System.err.println("Exception :" + nfe.getClass().getName() +
                                    " - " + nfe.getMessage() +
13                                  " - in " + Exception02.class.getName());
                 System.err.flush();
15           } catch(Exception e) {
                 System.out.println("Je suis là");
17               System.out.flush();
                 System.err.println("Exception :" + e.getClass().getName() +
19                                  " - " + e.getMessage() +
                                    " - in " + Exception02.class.getName());
21           }
       }
23 }
```

Une classe d'Exception personnalisée et son utilisation

```
1  public class Exception03 {
       public static void main(String[] args) {
3          new Exception03();
       }
5      Exception03() {
           Parseur p = new Parseur();
7          try {
               float t = p.parse("OP");
9              System.out.println(t);
           } catch(Parseur.MyNumberFormatException nfe) {
11             System.out.println("Je suis ici");
               System.out.flush();
13             System.err.println("Exception :" + nfe.getClass().getName() + " - " + nfe.getMessage() + " - in " + Exception03.class.getName());
               System.err.flush();
15         } catch(Exception e) {
               System.out.println("Je suis là");
17             System.out.flush();
               System.err.println("Exception :" + e.getClass().getName() + " - " + e.getMessage() + " - in " + Exception03.class.getName());
19         }
       }
21     class Parseur {
           float parse(String str) throws MyNumberFormatException {
23             float f = 0;
               try {
25                 f = Float.parseFloat(str);
               } catch(Exception e) {
27                 System.out.println(e);
                   MyNumberFormatException me = new MyNumberFormatException();
29                 me.setMessage(e.getMessage());
                   throw me;
31             }
               return f;
33         }
           class MyNumberFormatException extends NumberFormatException {
35             private String message = null;
               @Override
37             public String getMessage() {
                   return "Je suis une exception personnalisée - " + message;
39             }
               public void setMessage(String message) {
41                 this.message = message;
               }
43         }
       }
45 }
```

- Les annotation ...

La généricité est l'indépendance par rapport au type de donnée (en Java on parlera particulièrement de classe). C'est la structuration de mécanismes de traitements de données externalisant la particularité de la donnée elle-même. Par exemple, un algorithme de tri peut être implémenté en dehors de tout type de donnée à partir du moment où l'algorithme connait la taille d'une donnée et peut en comparer deux.

- En Java, tout objet hérite implicitement de la classe `Object` (voir le package java.lang) ;
- Par conséquent tout objet est castable en `Object` ;
- Voir quelles sont le méthodes publiques héritées d'`Object` et ce qu'on peut en faire ;
- A l'aide du type `Object` combiné aux méthodes abstraites et/ou les interfaces, nous pouvons proposer un premier modèle de généricité.

Extrait de la javadoc de `java.lang.Object`

Method Summary

protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class< ? extends Object>	getClass() Returns the runtime class of an object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Tout objet peut-être référencé par un `Object` tout en gardant ses particularités (ie. les spécifications liées à sa classe).

```
1  public class JHeriteDObject {
       JHeriteDObject() {
3          super(); // appel explicite au constructeur du super (soit Object)
           System.out.print("Juste après le constructeur du super - ");
5          afficheMaClassName();
       }
7      public void afficheMaClassName() {
           System.out.print("Dans la méthode afficheMaClassName - ");
9          JHeriteDObject.afficheLaClassNameDe(this);
       }
11     static public void afficheLaClassNameDe(Object obj) {
           System.out.println("Je suis un : " + obj.getClass().getName());
13     }
       /* Tests de la classe */
15     static public void main(String str[]) {
           Object jeFaisSemblantDEtreUnObject = new JHeriteDObject();
17         System.out.println("Depuis la référence de type Object : " +
                              jeFaisSemblantDEtreUnObject.getClass().getName() + " - C'est pareil !");
19         ((JHeriteDObject)jeFaisSemblantDEtreUnObject).afficheMaClassName();
           JHeriteDObject jeNeFaisPasSemblant = new JHeriteDObject();
21         jeNeFaisPasSemblant.afficheMaClassName();
           System.out.println("Depuis un cast en Object : " +
23                            ((Object)jeNeFaisPasSemblant).getClass().getName() + " - Idem !");
           System.out.println("Nous pouvons tous nous faire passer pour des Objects !!!");
25     }
   }
```

Application de la généricité aux arbres binaires (1/4) : ci-après une classe permettant de gérer des arbres binaires d'entiers → pas de généricité.

```
public class Node {
    private int data;
    private Node fg = null /* fils gauche */, fd = null /* fils droit */;
    public Node(int data) { this.data = data; }
    public int getData() { return data; }
    public Node addNode(int data) {
        if(data < getData()) {
            if(this.fg != null)
                return this.fg.addNode(data);
            return this.fg = new Node(data);
        } else {
            if(this.fd != null)
                return this.fd.addNode(data);
            return this.fd = new Node(data);
        }
    }
    public void print() {
        if(this.fg != null)
            this.fg.print();
        System.out.print(this.data + " ");
        if(this.fd != null)
            this.fd.print();
    }
    static public void main(String str[]) {
        int t[] = { 5, 2, 8, 3, 4, 1, 6, 7, 12 };
        Node racine = new Node(t[0]);
        for(int i = 1; i < t.length; i++)
            racine.addNode(t[i]);
        racine.print();
    }
}
```

Application de la généricité aux arbres binaires (2/4) : remplacer le type `int` de `data` par des **objets** de la classe `Integer` ; modifier le getter et la comparaison utilisant l'opérateur inférieur < par une méthode de comparaison `compar`. Remarquez : le `println` s'en sort très bien en utilisant la méthode implicitement appelée `toString`.

```
1  public class Node {
       private Integer data; /* je suis un Objet ! */
3      private Node fg = null /* fils gauche */, fd = null /* fils droit */;
       public Node(Integer data) { this.data = data; }
5      public Integer getData() { return data; }
       public Node addNode(Integer data) {
7          if(compar(data, getData()) < 0) {
               if(this.fg != null)
9                  return this.fg.addNode(data);
               return this.fg = new Node(data);
11         } else {
               if(this.fd != null)
13                 return this.fd.addNode(data);
               return this.fd = new Node(data);
15         }
       }
17     public void print() {
           if(this.fg != null)
19             this.fg.print();
           System.out.print(this.data + " ");
21         if(this.fd != null)
               this.fd.print();
23     }
       private int compar(Integer data1, Integer data2) {
25         return data1.intValue() - data2.intValue();
       }
27     static public void main(String str[]) {
           int t[] = { 5, 2, 8, 3, 4, 1, 6, 7, 12 };
29         Node racine = new Node(new Integer(t[0]));
           for(int i = 1; i < t.length; i++)
31             racine.addNode(new Integer(t[i]));
           racine.print();
33     }
   }
```

Application de la généricité aux arbres binaires (3/4) : passer la donnée en `Object` et la méthode `compar` en méthode abstaite (ie. non implémentées) afin de laisser à l'utilisateur la possibilité (mais aussi l'obligation) d'implémenter les siennes (ie. celles qui sont propres à sa strucutre de données).

```
public abstract class Node { /* je vais plus loin dans l'abstraction */
    private Object data; /* moi aussi */
    private Node fg = null /* fils gauche */, fd = null /* fils droit */;
    public abstract int compar(Object data1, Object data2); /* A l'utilisateur de m'implementer */
    public Node(Object data) { this.data = data; }
    public Object getData() { return data; }
    public Node addNode(Object data) {
        final Node me = this;
        if(compar(data, getData()) < 0) {
            if(this.fg != null)
                return this.fg.addNode(data);
            return (this.fg = new Node(data) {
                    @Override
                    public int compar(Object data1, Object data2) {
                        /* j'utilise la comparaison du parent */
                        return me.compar(data1, data2);
                    }});
        } else {
            if(this.fd != null)
                return this.fd.addNode(data);
            return (this.fd = new Node(data) {
                    @Override
                    public int compar(Object data1, Object data2) {
                        /* j'utilise la comparaison du parent */
                        return me.compar(data1, data2);
                    }});
        }
    }
    /* tant qu'a faire, renommons print en toString */
    @Override
    public String toString() {
        return new String( ((this.fg != null) ? this.fg.toString() : "") +
                           this.data + " " +
                           ((this.fd != null) ? this.fd.toString() : "") );
    }
    static public void main(String str[]) {
        int t[] = { 5, 2, 8, 3, 4, 1, 6, 7, 12 };
        Node racine = new Node(new Integer(t[0])) {
                @Override
                public int compar(Object data1, Object data2) {
                    /* car ici je sais que ma data est composee d'Integer */
                    return ((Integer)data1).intValue() - ((Integer)data2).intValue();
                }
            };
        for(int i = 1; i < t.length; i++)
            racine.addNode(new Integer(t[i]));
        System.out.println(racine);
        String s[] = { "Hello", "je", "suis", "un", "arbre", "binaire", "je", "trie", "tout", "ce",
                       "qu'on", "me", "donne", "a", "trier" };
        racine = new Node(new String(s[0])) {
                @Override
                public int compar(Object data1, Object data2) {
                    /* Ca me donnes des idees, pas vous ??? */
                    return ((String)data1).compareTo((String)data2);
                }
            };
        for(int i = 1; i < s.length; i++)
            racine.addNode(new String(s[i]));
        System.out.println(racine);
    }
}
```

Application de la généricité aux arbres binaires (4/4) : Simplification utilisant l'interface `Comparable`.

```
1  public class Node {
       private Comparable data; /* voir l'interface Comparable */
3      private Node fg = null /* fils gauche */, fd = null /* fils droit */;
       public Node(Comparable data) { this.data = data; }
5      public Comparable getData() { return data; }
       @SuppressWarnings("unchecked") // + tard
7      public Node addNode(Comparable data) {
           if(data.compareTo(getData()) < 0) {
9              if(this.fg != null)
                   return this.fg.addNode(data);
11             return this.fg = new Node(data);
           } else {
13             if(this.fd != null)
                   return this.fd.addNode(data);
15             return this.fd = new Node(data);
           }
17     }
       @Override
19     public String toString() {
           return new String( ((this.fg != null) ? this.fg.toString() : "") +
21                            this.data + " " +
                              ((this.fd != null) ? this.fd.toString() : "") );
23     }
       static public void main(String str[]) {
25         int t[] = { 5, 2, 8, 3, 4, 1, 6, 7, 12 };
           Node racine = new Node(new Integer(t[0]));
27         for(int i = 1; i < t.length; i++)
               racine.addNode(new Integer(t[i]));
29         System.out.println(racine);
           String s[] = { "Hello", "je", "suis", "un", "arbre", "binaire", "je", "trie", "tout", "ce",
31                        "qu'on", "me", "donne", "a", "trier" };
           racine = new Node(new String(s[0]));
33         for(int i = 1; i < s.length; i++)
               racine.addNode(new String(s[i]));
35         System.out.println(racine);
       }
37 }
```

Les types/classes génériques

- Il s'agit de rendre paramétrable certains type de données utilisés dans une classe ;
- Nous avons vu que le type générique par excellence en Java est `Object`. Ainsi, à partir d'un `Object` nous pouvons, à l'aide de l'introspection, gérer les cas spécifiques à chaque type d'objet en déclarant des méthodes abstraites prévues à cet effet et devant être implémentée ;
- Q : Pouvons-nous passer à un méta-niveau permettant de symboliser cette généricité ? R : Oui
- Une classe générique est ainsi définie par :
  `class NomDeLaClasse<TP1, TP2, ..., TPn> { /* corps de classe */ }`
  où nous notons que cette classe prend des type de données comme paramètre, c'est les `TP1 à TPn` présentés entre `< et >`. Ces types (nous pourrions aussi utiliser le mot "symbols"), non définis, sont utilisables dans le corps de la classe.

Classe non générique utilisant une donnée non générique.

```
1  public class MaClasseUtilisantUneDonneeNonGenerique {
       Integer _data = null;
3      public Integer get() {
           return _data;
5      }
       public void set(Integer data) {
7          _data = data;
       }
9      /* Tests de la classe */
       static public void main(String str[]) {
11         MaClasseUtilisantUneDonneeNonGenerique o = new MaClasseUtilisantUneDonneeNonGenerique();
           o.set(new Integer(42));
13         System.out.println("La réponse est " + o.get());
       }
15 }
```

Classe non générique utilisant une donnée générique.

```
1  public class MaClasseUtilisantUneDonneeGenerique { /* déjà vu */
       Object _data = null;
3      public Object get() {
           return _data;
5      }
       public void set(Object data) {
7          _data = data;
       }
9      /* Tests de la classe */
       static public void main(String str[]) {
11         MaClasseUtilisantUneDonneeGenerique o1 = new MaClasseUtilisantUneDonneeGenerique();
           MaClasseUtilisantUneDonneeGenerique o2 = new MaClasseUtilisantUneDonneeGenerique();
13         o1.set(new String("La réponse est "));
           o2.set(new Integer(42));
15         System.out.println(o1.get() + "" + o2.get());
       }
17 }
```

Premier exemple de classe générique.

```
1  public class MaPremiereClasseGenerique <T> {
       T _data = null;
3      public T get() {
           return _data;
5      }
       public void set(T data) {
7          _data = data;
       }
9      /* Tests de la classe */
       static public void main(String str[]) {
11         MaPremiereClasseGenerique <String> o1 = new MaPremiereClasseGenerique <String>();
           MaPremiereClasseGenerique <Integer> o2 = new MaPremiereClasseGenerique <Integer>();
13         o1.set(new String("La réponse est "));
           o2.set(new Integer(42));
15         System.out.println(o1.get() + o2.get());
       }
17 }
```

Exercice 01

1. Ecrire une classe `LlNode` représentant les nœuds d'une liste chaînée d'`Integer` ;
2. Ecrire le code permettant de tester la classe précédente. Pour celà, initialiser aléatoirement un tableau d'`int` puis remplir la liste chaînée avec ;
3. Ecrire une classe `Stack` encapsulant `LlNode` et modifier le code de test. `Stack` stocke les éléments en LIFO (Last In First Out) c'est à dire que : chaque nouvel élément est ajouté en tête de liste, par la méthode `push` ; si la liste est non vide, un élément est extrait depuis la tête de liste avec la méthode `pop` et enfin la liste vide peut-être testée par la méthode `isEmpty` ;
4. Rendre la donnée de chaque nœud générique en utilisant la classe `Object` ;
5. Rendre le tout générique en utilisant les types/classes génériques ;
6. Implémenter un `Iterable` et l'utiliser (voir section suivante).

Solution - Exercice 01 - question 1 et 2

```
1  public class LlNode {
       private Integer _data = null;
3      private LlNode _next = null;
       public LlNode(Integer data) {
5          _data = data;
       }
7      public LlNode(Integer data, LlNode next) {
           _data = data;
9          _next = next;
       }
11     public Integer getData() {
           return _data;
13     }
       public LlNode getNext() {
15         return _next;
       }
17     public void setData(Integer data) {
           _data = data;
19     }
       public void setNext(LlNode next) {
21         _next = next;
       }
23     static public void main(String [] str) {
           int t[] = { 1, 2, 3, 4, 5, 6, 7};
25         LlNode tete = new LlNode(new Integer(t[0])), ref = tete;
           for(int i = 1; i < t.length; i++) {
27             ref.setNext(new LlNode(new Integer(t[i])));
               ref = ref.getNext();
29         }
           ref = tete;
31         while(ref != null) {
               System.out.println(ref.getData());
33             ref = ref.getNext();
           }
35     }
   }
```

Solution - Exercice 01 - question 3 - 1/2 (Fichier LlNode.java)

```
public class LlNode {
    private Integer _data = null;
    private LlNode _next = null;
    public LlNode(Integer data) {
        _data = data;
    }
    public LlNode(Integer data, LlNode next) {
        _data = data;
        _next = next;
    }
    public Integer getData() {
        return _data;
    }
    public LlNode getNext() {
        return _next;
    }
    public void setData(Integer data) {
        _data = data;
    }
    public void setNext(LlNode next) {
        _next = next;
    }
}
```

Solution - Exercice 01 - question 3 - 2/2 (Fichier Stack.java)

```
1  public class Stack {
       private LlNode _head = null;
3      public boolean isEmpty() {
           return _head == null;
5      }
       public void push(Integer data) {
7          _head = new LlNode(data, _head);
       }
9      public Integer pop() {
           LlNode ref = _head;
11         _head = _head.getNext();
           return ref.getData();
13     }
       static public void main(String [] str) {
15         int t[] = { 1, 2, 3, 4, 5, 6, 7};
           Stack ll = new Stack();
17         for(int i = 0; i < t.length; i++)
               ll.push(new Integer(t[i]));
19         while(!ll.isEmpty())
               System.out.println(ll.pop());
21     }
   }
```

Solution - Exercice 01 - question 4 - 1/2 (Fichier LlNode.java)

```
public class LlNode {
    private Object _data = null;
    private LlNode _next = null;
    public LlNode(Object data) {
        _data = data;
    }
    public LlNode(Object data, LlNode next) {
        _data = data;
        _next = next;
    }
    public Object getData() {
        return _data;
    }
    public LlNode getNext() {
        return _next;
    }
    public void setData(Object data) {
        _data = data;
    }
    public void setNext(LlNode next) {
        _next = next;
    }
}
```

Solution - Exercice 01 - question 4 - 2/2 (Fichier Stack.java)

```
1  public class Stack {
       private LlNode _head = null;
3      public boolean isEmpty() {
           return _head == null;
5      }
       public void push(Object data) {
7          _head = new LlNode(data, _head);
       }
9      public Object pop() {
           LlNode ref = _head;
11         _head = _head.getNext();
           return ref.getData();
13     }
       static public void main(String [] str) {
15         int t[] = { 1, 2, 3, 4, 5, 6, 7};
           Stack ll = new Stack();
17         for(int i = 0; i < t.length; i++)
               ll.push(new Integer(t[i]));
19         while(!ll.isEmpty())
               System.out.println(ll.pop());
21     }
   }
```

Solution - Exercice 01 - question 5 - 1/2 (Fichier LlNode.java)

```
public class LlNode<T> {
    private T _data = null;
    private LlNode<T> _next = null;
    public LlNode(T data) {
        _data = data;
    }
    public LlNode(T data, LlNode<T> next) {
        _data = data;
        _next = next;
    }
    public T getData() {
        return _data;
    }
    public LlNode<T> getNext() {
        return _next;
    }
    public void setData(T data) {
        _data = data;
    }
    public void setNext(LlNode<T> next) {
        _next = next;
    }
}
```

Solution - Exercice 01 - question 5 - 2/2 (Fichier Stack.java)

```
1  public class Stack<T> {
       private LlNode<T> _head = null;
3      public boolean isEmpty() {
           return _head == null;
5      }
       public void push(T data) {
7          _head = new LlNode<T>(data, _head);
       }
9      public T pop() {
           LlNode<T> ref = _head;
11         _head = _head.getNext();
           return ref.getData();
13     }
       static public void main(String [] str) {
15         int t[] = { 1, 2, 3, 4, 5, 6, 7};
           Stack<Integer> ll = new Stack<Integer>();
17         for(int i = 0; i < t.length; i++)
               ll.push(new Integer(t[i]));
19         while(!ll.isEmpty())
               System.out.println(ll.pop());
21     }
   }
```

Solution - Exercice 01 - question 6 - 1/1 (Fichier MyIterator.java)

Pourquoi surcharger Iterator (et par extension Iterable) ?

Afin de rajouter une nouvelle fonctionnalité - get - utilisée dans Stack.java (voir plus loin à 4/4).

```
import java.util.Iterator;
public interface MyIterator<E> extends Iterator<E> {
        public E get();
}
```

Solution - Exercice 01 - question 6 - 2/4 (Fichier MyIterable.java)

```
public interface MyIterable<E> extends Iterable<E> {
    @Override
    public MyIterator<E> iterator();
}
```

Solution - Exercice 01 - question 6 - 3/4 (Fichier LlNode.java)

```
class LlNode<T> { /* on le laisse friendly */
    private T _data = null;
    private LlNode<T> _next = null;
    public LlNode(T data) {
        _data = data;
    }
    public LlNode(T data, LlNode<T> next) {
        _data = data;
        _next = next;
    }
    public T getData () {
        return _data;
    }
    public LlNode<T> getNext() {
        return _next;
    }
    public void setData(T data) {
        _data = data;
    }
    public void setNext(LlNode<T> next) {
        _next = next;
    }
}
```

Solution - Exercice 01 - question 6 - 4/4 (Fichier Stack.java)

```
1  import java.util.Iterator;
   public class Stack<E> implements MyIterable<E> {
3      private LlNode<E> _head = null;
       public boolean isEmpty() { return _head == null; }
5      public void push(E data) {
           LlNode<E> n = new LlNode<E>(data, _head);
7          _head = n;
       }
9      public E pop() {
           E c = _head.getData();
11         _head = _head.getNext();
           return c;
13     }
       @Override
15     public MyIterator<E> iterator() {
           return new MyIterator<E>() {
17             LlNode<E> _iteratorHead = _head;
               LlNode<E> _oldIteratorHead = null;
19             @Override
               public boolean hasNext() {
21                 return _iteratorHead != null;
               }
23             @Override
               public E get() {
25                 return _iteratorHead.getData();
               }
27             @Override
               public E next() {
29                 _oldIteratorHead = _iteratorHead;
                   _iteratorHead = _iteratorHead.getNext();
31                 return _oldIteratorHead.getData();
               }
33             @Override
               public void remove() {
35                 if(_oldIteratorHead == null)
                       _head = _iteratorHead = _iteratorHead.getNext();
37                 else
                       _oldIteratorHead.setNext(_iteratorHead = _iteratorHead.getNext());
39             }
           };
41     }
       static public void main(String [] str) {
43         int t[] = {0,1,2,3,4,5,6,7,8,9,10,11};
           Stack <Integer > ll = new Stack <Integer >();
45         for(int i = 0; i < t.length; i++)
               ll.push(new Integer(t[i]));
47         /* exemples d'utilisation de l'itérateur */
           for (MyIterator<Integer> it = ll.iterator(); it.hasNext(); ) {
49             Integer i = it.next();
               System.out.print(i + " : ");
51             /* suppresion du suivant d'un impair */
               if((i.intValue()%2 == 1) && it.hasNext())
53                 it.remove();
               /* exemple d'utilisation de notre nouvelle fonctionnalité (get) */
55             if(it.hasNext() && it.get().intValue() == 5)
                   it.remove();
57             /* iterateur dans iterateur */
               for (Integer i2: ll)
59               System.out.print(i2 + " ");
               System.out.println("");
61         }
           /* on dépile ce qui reste */
63         while(!ll.isEmpty())
             System.out.println(ll.pop());
65         /* pourquoi le seul pair restant est 4 ? */
       }
67 }
```

Exercice 02

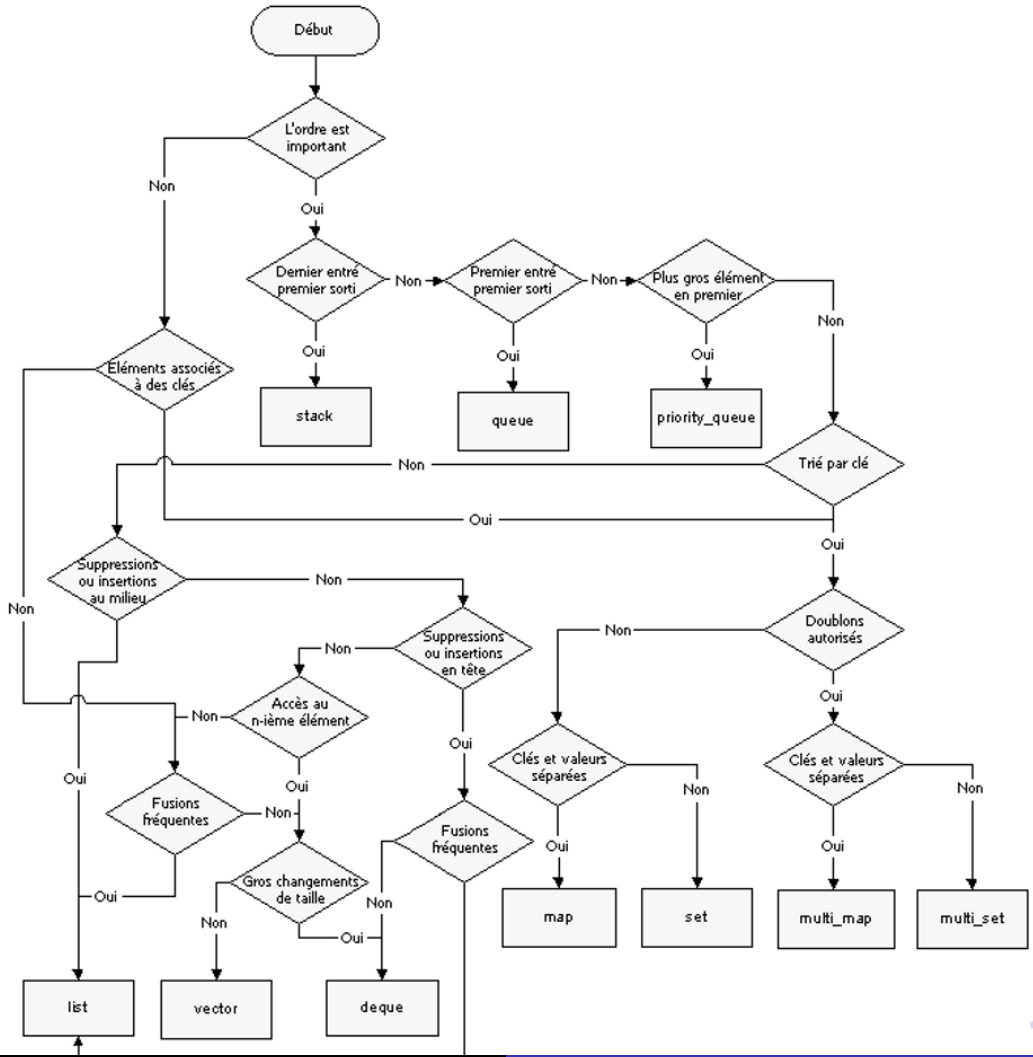
1. Ecrire une classe `LlNode` représentant les nœuds d'une liste chaînée d'`Integer` (idem que l'exercie précédent) ;
2. Ecrire le code permettant de tester la classe précédente. Pour celà, initialiser aléatoirement un tableau d'`int` puis remplir la liste chaînée avec (idem que l'exercie précédent) ;
3. Ecrire une classe `LinkedList` encapsulant `LlNode` et modifier le code de test. Il faudrait que `LinkedList` s'arrange à trier dans l'ordre croissant les données référencées ;
4. Rendre la donnée de chaque nœud générique en utilisant la classe `Object` et/ou `Comparable` ;
5. Rendre le tout générique en utilisant les types/classes génériques ;
6. Implémenter l'interface `iterable/iterator` (voir section suivante).

Solution - Exercice 02 - utiliser LlNode obtenu plus haut (Fichier LinkedList.java)

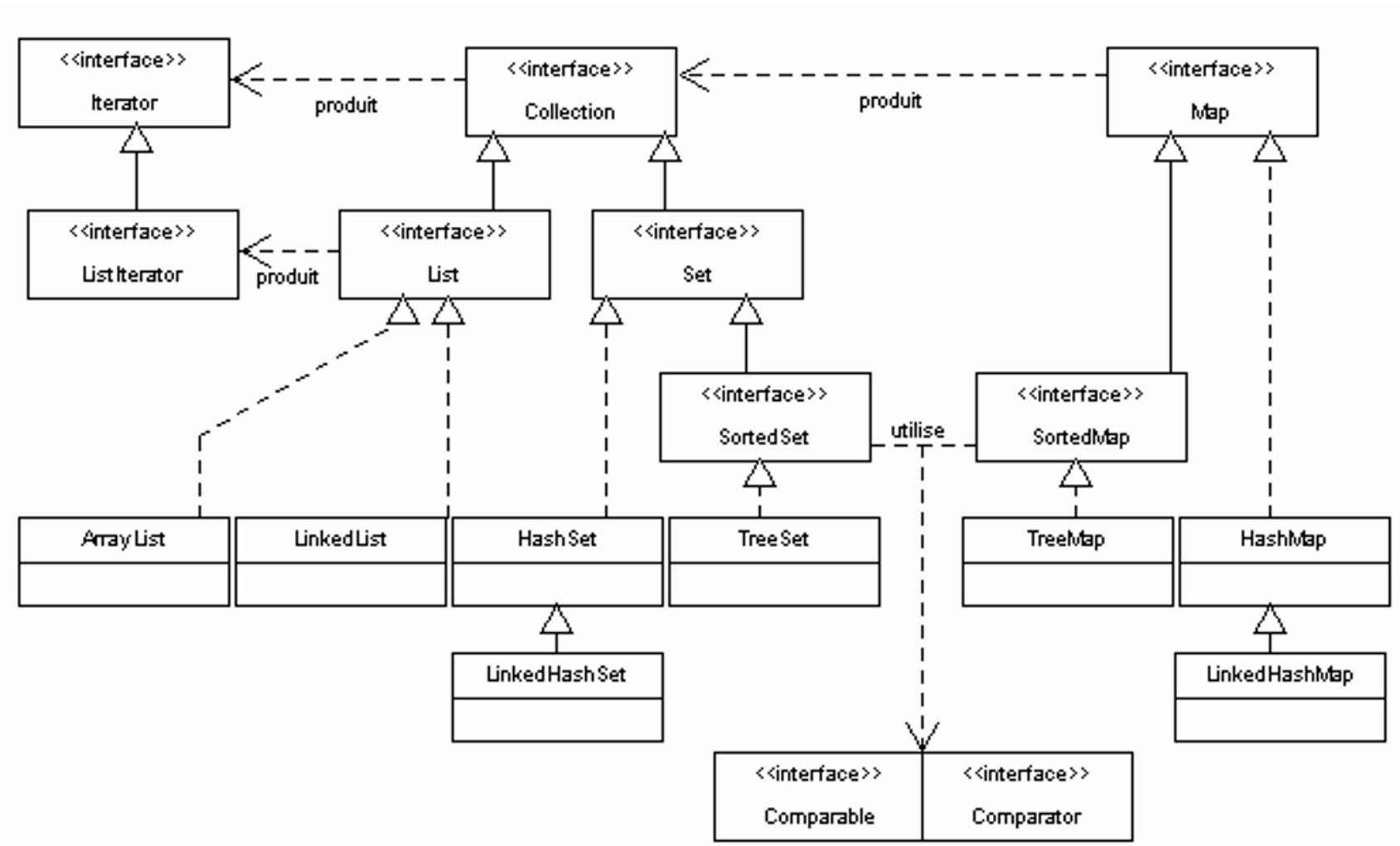
```
1   import java.util.Iterator;
    public class LinkedList<E extends Comparable<E>> implements Iterable<E> {
3           LlNode<E> _head = null;
            public void insert(E data) {
5                   if(_head == null)
                            _head = new LlNode<E>(data);
7                   else {
                            LlNode<E> ptr = _head, oldPtr = null;
9                           while(ptr != null && data.compareTo(ptr.getData()) > 0) {
                                    oldPtr = ptr;
11                                  ptr = ptr.getNext();
                            }
13                          if(oldPtr == null)
                                    _head = new LlNode<E>(data, _head);
15                          else
                                    oldPtr.setNext(new LlNode<E>(data, ptr));
17                  }
            }
19          @Override
            public String toString() {
21                  String S = "";
                    for(LlNode<E> ptr = _head; ptr != null; ptr = ptr.getNext())
23                          S = S + ptr.getData() + " - ";
                    return S;
25          }
            public Iterator<E> iterator() {
27                  return new Iterator<E>() {
                            LlNode<E> _ptr = _head, _oldPtr = null;
29                          public boolean hasNext() { return _ptr != null; }
                            public E next() {
31                                  assert(_ptr != null);
                                    E d = _ptr.getData();
33                                  if(_oldPtr == null) /* on lui laisse tjs un coup d'avance */
                                            _oldPtr = _ptr == _head ? null : _head;
35                                  else
                                            _oldPtr = _ptr == _oldPtr.getNext() ? _oldPtr : _oldPtr.getNext();
37                                  _ptr = _ptr.getNext();
                                    return d;
39                          }
                            public void remove() {
41                                  assert(_head != null);
                                    if(_oldPtr == null)
43                                          _head = _head.getNext();
                                    else
45                                          _oldPtr.setNext(_ptr);
                            }
47                  };
            }
49          static public void main(String [] str) {
                    LinkedList<Integer> ll = new LinkedList<Integer>();
51                  for(int i = 0; i < 10; i++)
                            ll.insert(new Integer((int)(Math.random() * 10)));
53                  for(Iterator<Integer> it = ll.iterator(); it.hasNext();) {
                            for(Integer d: ll) System.out.print(d + " - ");
55                          Integer d = it.next();
                            if(d.intValue() % 2 == 1) {
57                                  System.out.println("L'element " + d + " sera supprime");
                                    it.remove();
59                          } else
                                    System.out.println("L'element " + d + " sera garde");
61                  }
                    System.out.println(ll);
63          }
    }
```

Que choisir ?

logigramme concernant le C++ (source cpp.developpez.com)



Vue globale sur les collections en java



L'interface **Iterable**[Ite14a]

java.lang
Interface Iterable<T>

All Known Subinterfaces:
BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Collection<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>

All Known Implementing Classes:
AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BatchUpdateException, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DataTruncation, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, RowSetWarning, SerialException, ServiceLoader, SQLClientInfoException, SQLDataException, SQLException, SQLFeatureNotSupportedException, SQLIntegrityConstraintViolationException, SQLInvalidAuthorizationSpecException, SQLNonTransientConnectionException, SQLNonTransientException, SQLRecoverableException, SQLSyntaxErrorException, SQLTimeoutException, SQLTransactionRollbackException, SQLTransientConnectionException, SQLTransientException, SQLWarning, Stack, SyncFactoryException, SynchronousQueue, SyncProviderException, TreeSet, Vector

---

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

Since:
1.5

Method Summary

Iterator<T>	**iterator**() Returns an iterator over a set of elements of type T.

Method Detail

iterator

```
Iterator<T> iterator()
```

Returns an iterator over a set of elements of type T.

Returns:
an Iterator.

L'interface **Iterator**[Ite14b] (1/2)

java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>, XMLEventReader

All Known Implementing Classes:
BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

`public interface Iterator<E>`

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:
1.2
See Also:
Collection, ListIterator, Enumeration

L'interface **Iterator**[Ite14b] (2/2)

hasNext

```
boolean hasNext()
```

Returns `true` if the iteration has more elements. (In other words, returns `true` if `next` would return an element rather than throwing an exception.)

Returns:
`true` if the iterator has more elements.

---

next

```
E next()
```

Returns the next element in the iteration.

Returns:
the next element in the iteration.
Throws:
`NoSuchElementException` - iteration has no more elements.

---

remove

```
void remove()
```

Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to `next`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:
`UnsupportedOperationException` - if the `remove` operation is not supported by this Iterator.
`IllegalStateException` - if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method.

L'interface **Collection**[Col14] (1/3)

java.util

Interface Collection<E>

All Superinterfaces:
Iterable<E>

All Known Subinterfaces:
BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>

All Known Implementing Classes:
AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

---

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries comply.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the collection. For example, invoking the `addAll(Collection)` method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

L'interface **Collection**[Col14] (2/3)

It is up to each collection to determine its own synchronization policy. In the absence of a stronger guarantee by the implementation, undefined behavior may result from the invocation of any method on a collection that is being mutated by another thread; this includes direct invocations, passing the collection to a method that might perform invocations, and using an existing iterator to examine the collection.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `contains(Object o)` method says: "returns `true` if and only if this collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`." This specification should *not* be construed to imply that invoking `Collection.contains` with a non-null argument `o` will cause `o.equals(e)` to be invoked for any element `e`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two elements. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
`Set`, `List`, `Map`, `SortedSet`, `SortedMap`, `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`, `Vector`, `Collections`, `Arrays`, `AbstractCollection`

L'interface **Collection**[Col14] (3/3)

Method Summary	
boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns `true` if this collection contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns `true` if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Types basiques de collections

- `List` permet d'ajouter ou de supprimer des éléments à une position indiquée (indexée). L'ordre des éléments est respecté et les éléments peuvent être récupérés par itérateur et/ou indice ;
- `Queue` permet de stocker au début ou à la fin d'une liste d'éléments, de récupérer le début ou la fin. Cette structure permet de gérer les priorités ;
- `Set` permet d'insérer des données, par défaut ne respecte pas l'ordre mais empêche les doublons (unicité), dit si un élément y est ou pas ;
- `Map` comme un Set mais permet d'insérer des données par clé-valeur, ce qui permettrait de dupliquer la donnée (la valeur) si nécessaire.

L'interface **Map** (1/2)

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

Java™ Platform Standard Ed. 6

java.util

Interface Map<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values

All Known Subinterfaces:
Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:
AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

---

`public interface Map<K,V>`

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`." This specification should *not* be construed to imply that invoking `Map.containsKey` with a non-null argument `key` will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Since:
1.2

See Also:
HashMap, TreeMap, Hashtable, SortedMap, Collection, Set

L'interface **Map** (2/2)

Nested Class Summary	
static interface	**Map.Entry<K,V>** A map entry (key-value pair).

Method Summary	
void	**clear**() Removes all of the mappings from this map (optional operation).
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.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
boolean	**equals**(Object o) Compares the specified object with this map for equality.
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.
int	**hashCode**() Returns the hash code value for this map.
boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
V	**remove**(Object key) Removes the mapping for a key from this map if it is present (optional operation).
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.

L'interface **Set** (1/2)

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

Java™ Platform Standard Ed. 6

java.util

Interface Set<E>

Type Parameters:

E - the type of elements maintained by this set

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

NavigableSet<E>, SortedSet<E>

All Known Implementing Classes:

AbstractSet, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

---

```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Some set implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the set may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `List`, `SortedSet`, `HashSet`, `TreeSet`, `AbstractSet`, `Collections.singleton(java.lang.Object)`, `Collections.EMPTY_SET`

L'interface **Set** (2/2)

See Also:
Collection, List, SortedSet, HashSet, TreeSet, AbstractSet, Collections.singleton(java.lang.Object), Collections.EMPTY_SET

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Q1) Qu'imprime ce programme ?
Q2) Supprimez les *warnings*.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class Collections {
  /**
  * Exemple d'utilisation d'une liste chainee non typee
  */
  public static void main(String[] args) {
    List l = new LinkedList();
    l.add("Une chaine");
    l.add(new Integer(1));
    l.add(new Float(2.5f));
    System.out.println("Utilisation du size");
    for (int i = 0; i < l.size(); i++) {
      System.out.println(l.get(i));
    }
    l.set(1, new Double(3.1415));
    System.out.println("Utilisation du foreach");
    for (Object object : l) {
      System.out.println(object);
    }
    l.add(0, new Integer(0));
    System.out.println("Utilisation de l'iterator");
    for (Iterator iterator = l.iterator(); iterator.hasNext();) {
      Object object = (Object) iterator.next();
      System.out.println(object);
    }
  }
}
```

R1) Voici la trace :

```
1   Utilisation du size
    Une chaine
3   1
    2.5
5   Utilisation du foreach
    Une chaine
7   3.1415
    2.5
9   Utilisation de l'iterator
    0
11  Une chaine
    3.1415
13  2.5
```

R2) Solution 1, avec utilisation des annotations.

```
1  import java.util.Iterator;
   import java.util.LinkedList;
3  import java.util.List;
   public class Collections {
5    /**
     * Exemple d'utilisation d'une liste chainee non typee
7    */
     @SuppressWarnings({ "rawtypes", "unchecked" })
9    public static void main(String[] args) {
       ...
11   }
   }
```

R2) Solution 2, avec utilisation explicite du type Object.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class Collections {
  /**
  * Exemple d'utilisation d'une liste chainee d'Objects
  */
  public static void main(String[] args) {
    List<Object> l = new LinkedList<Object>();
    l.add("Une chaine");
    l.add(new Integer(1));
    l.add(new Float(2.5f));
    System.out.println("Utilisation du size");
    for (int i = 0; i < l.size(); i++) {
      System.out.println(l.get(i));
    }
    l.set(1, new Double(3.1415));
    System.out.println("Utilisation du foreach");
    for (Object object : l) {
      System.out.println(object);
    }
    l.add(0, new Integer(0));
    System.out.println("Utilisation de l'iterator");
    for (Iterator<Object> iterator = l.iterator(); iterator.hasNext();) {
      Object object = (Object) iterator.next();
      System.out.println(object);
    }
  }
}
```

Q3) Écrire une méthode permettant de parcourir tous les éléments de la liste et de :

- Rajouter 1 à chaque nombre ;
- De convertir toutes les chaînes de caractères (ie. `String`) en majuscules.

R3) Solution 1.

```
1   public static void change(List<Object> l) {
      System.out.println("Effectuons quelques changements ...");
3     for (Object object : l) {
        if(object instanceof Integer) {
5         object = new Integer(((Integer)object).intValue() + 1);
        } else if(object instanceof Float) {
7         object = new Float(((Float)object).floatValue() + 1.0f);
        } else if(object instanceof Double) {
9         object = new Double(((Double)object).doubleValue() + 1.0);
        } else if(object instanceof String) {
11        object = ((String)object).toUpperCase();
        }
13      System.out.println(object);
      }
15  }
```

La liste est-elle bien modifiée par la solution 1 ?
R3) Solution 1 - Bis.

```
1   public static void change(List<Object> l) {
      int id = 0;
3     System.out.println("Effectuons quelques changements ...");
      for (Object object : l) {
5       if(object instanceof Integer) {
          object = new Integer(((Integer)object).intValue() + 1);
7       } else if(object instanceof Float) {
          object = new Float(((Float)object).floatValue() + 1.0f);
9       } else if(object instanceof Double) {
          object = new Double(((Double)object).doubleValue() + 1.0);
11      } else if(object instanceof String) {
          object = ((String)object).toUpperCase();
13      }
        System.out.println(object);
15      l.set(id++, object);
      }
17  }
```

reflect (java.lang.reflect.Constructor) et Number ... R3) Solution 2.

```
1   public static void change2(List<Object> l) {
      int id = 0;
3     System.out.println("Effectuons quelques changements 2 ...");
      for (Object object : l) {
5       if(object instanceof Number) {
          Class<?> c = object.getClass();
7         try {
            double d = ((Number)object).doubleValue() + 1.0;
9           int i = (int)d;
            // Voir tous les catch explicites possibles:
11          // SecurityException, NoSuchMethodException
            Constructor<?> cons = c.getConstructor(String.class);
13          // Voir tous les catch explicites possibles:
            // IllegalArgumentException, InstantiationException, IllegalAccessException
15          // InvocationTargetException
            object = (double)i == d ?
17          cons.newInstance(new Integer(i).toString()) :
            cons.newInstance(new Double(d).toString());
19        } catch (Exception e) {
            // TODO Auto-generated catch block
21          e.printStackTrace();
          }
23      } else if(object instanceof String) {
          object = ((String)object).toUpperCase();
25      }
        System.out.println(object);
27      l.set(id++, object);
      }
29  }
```

Compréhension 1/4

```
1       HashMap<Integer, String> mois1 = new HashMap<Integer, String>() ;
        mois1.put(new Integer(1),  "janvier");
3       mois1.put(new Integer(12), "decembre");
        mois1.put(new Integer(2),  "fevrier");
5       mois1.put(new Integer(11), "novembre");
        mois1.put(new Integer(3),  "mars");
7       mois1.put(new Integer(10), "octobre");
        mois1.put(new Integer(4),  "avril");
9       mois1.put(new Integer(9),  "septembre");
        mois1.put(new Integer(5),  "mai");
11      mois1.put(new Integer(8),  "aout");
        mois1.put(new Integer(6),  "juin");
13      mois1.put(new Integer(7),  "juillet");
        HashMap<String, String> mois2 = new HashMap<String, String>() ;
15      mois2.put("un",     "janvier");
        mois2.put("douze",  "decembre");
17      mois2.put("deux",   "fevrier");
        mois2.put("onze",   "novembre");
19      mois2.put("trois",  "mars");
        mois2.put("dix",    "octobre");
21      mois2.put("quatre", "avril");
        mois2.put("neuf",   "septembre");
23      mois2.put("cing",   "mai");
        mois2.put("huit",   "aout");
25      mois2.put("six",    "juin");
        mois2.put("sept",   "juillet");
```

Compréhension 2/4

```
        System.out.println("Contenu de mois1");
 2      for (Iterator<String> iterator = mois1.values().iterator(); iterator.hasNext();) {
          String m = (String) iterator.next();
 4        System.out.println(m);
        }
 6      System.out.println("Contenu de mois2");
        for (Iterator<String> iterator = mois2.values().iterator(); iterator.hasNext();) {
 8        String m = (String) iterator.next();
          System.out.println(m);
10      }
        System.out.println("Les hashcode des clÃ©s de mois1");
12      for (Iterator<Integer> iterator = mois1.keySet().iterator(); iterator.hasNext();) {
          Integer m = (Integer) iterator.next();
14        System.out.println(m + " : " + m.hashCode());
        }
16      System.out.println("Les hashcode des clÃ©s de mois2");
        for (Iterator<String> iterator = mois2.keySet().iterator(); iterator.hasNext();) {
18        String m = (String) iterator.next();
          System.out.println(m + " : " + m.hashCode());
20      }
```

Compréhension 3/4

```
TreeMap<String, String> mois3 = new TreeMap<String, String>() ;
mois3.put("un",     "janvier");
mois3.put("douze",  "decembre");
mois3.put("deux",   "fevrier");
mois3.put("onze",   "novembre");
mois3.put("trois",  "mars");
mois3.put("dix",    "octobre");
mois3.put("quatre", "avril");
mois3.put("neuf",   "septembre");
mois3.put("cing",   "mai");
mois3.put("huit",   "aout");
mois3.put("six",    "juin");
mois3.put("sept",   "juillet");
TreeSet<String> mois4 = new TreeSet<String>() ;
mois4.add("janvier");
mois4.add("decembre");
mois4.add("fevrier");
mois4.add("novembre");
mois4.add("mars");
mois4.add("octobre");
mois4.add("avril");
mois4.add("septembre");
mois4.add("mai");
mois4.add("aout");
mois4.add("juin");
mois4.add("juillet");
```

Compréhension 4/4

```
        System.out.println("Contenu de mois3");
2       for (Iterator<String> iterator = mois3.values().iterator(); iterator.hasNext();) {
          String m = (String) iterator.next();
4         System.out.println(m);
        }
6       System.out.println("Contenu de mois3");
        for (Iterator<String> iterator = mois4.iterator(); iterator.hasNext();) {
8         String m = (String) iterator.next();
          System.out.println(m);
10      }
```

Trace 1/3

```
    Contenu de mois1
 2  janvier
    fevrier
 4  mars
    avril
 6  mai
    juin
 8  juillet
    aout
10  septembre
    octobre
12  novembre
    decembre
14  Contenu de mois2
    fevrier
16  decembre
    aout
18  mai
    juillet
20  mars
    avril
22  octobre
    septembre
24  janvier
    juin
26  novembre
```

Trace 2/3

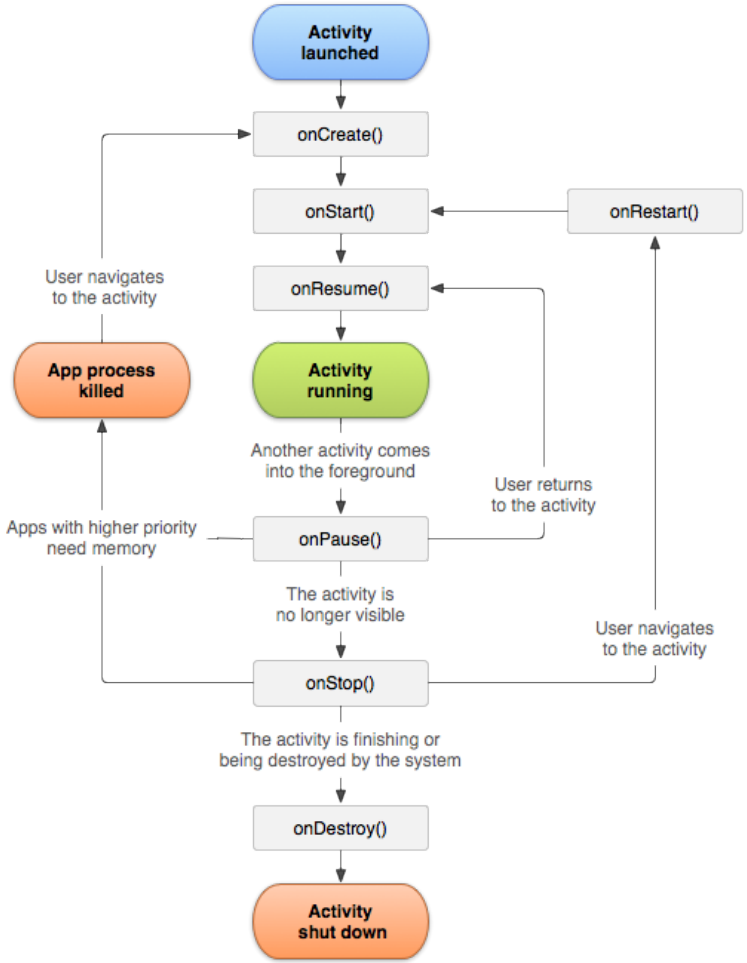
```
    Les hashcode des clÃ©s de mois1
 2  1 : 1
    2 : 2
 4  3 : 3
    4 : 4
 6  5 : 5
    6 : 6
 8  7 : 7
    8 : 8
10  9 : 9
    10 : 10
12  11 : 11
    12 : 12
14  Les hashcode des clÃ©s de mois2
    deux : 3079908
16  douze : 95775221
    huit : 3214072
18  cing : 3053727
    sept : 3526614
20  trois : 110634651
    quatre : -948816438
22  dix : 99475
    neuf : 3377800
24  un : 3737
    six : 113890
26  onze : 3416394
```

Trace 3/3

```
    Contenu de mois3
 2  mai
    fevrier
 4  octobre
    decembre
 6  aout
    septembre
 8  novembre
    avril
10  juillet
    juin
12  mars
    janvier
14  Contenu de mois3
    aout
16  avril
    decembre
18  fevrier
    janvier
20  juillet
    juin
22  mai
    mars
24  novembre
    octobre
26  septembre
```

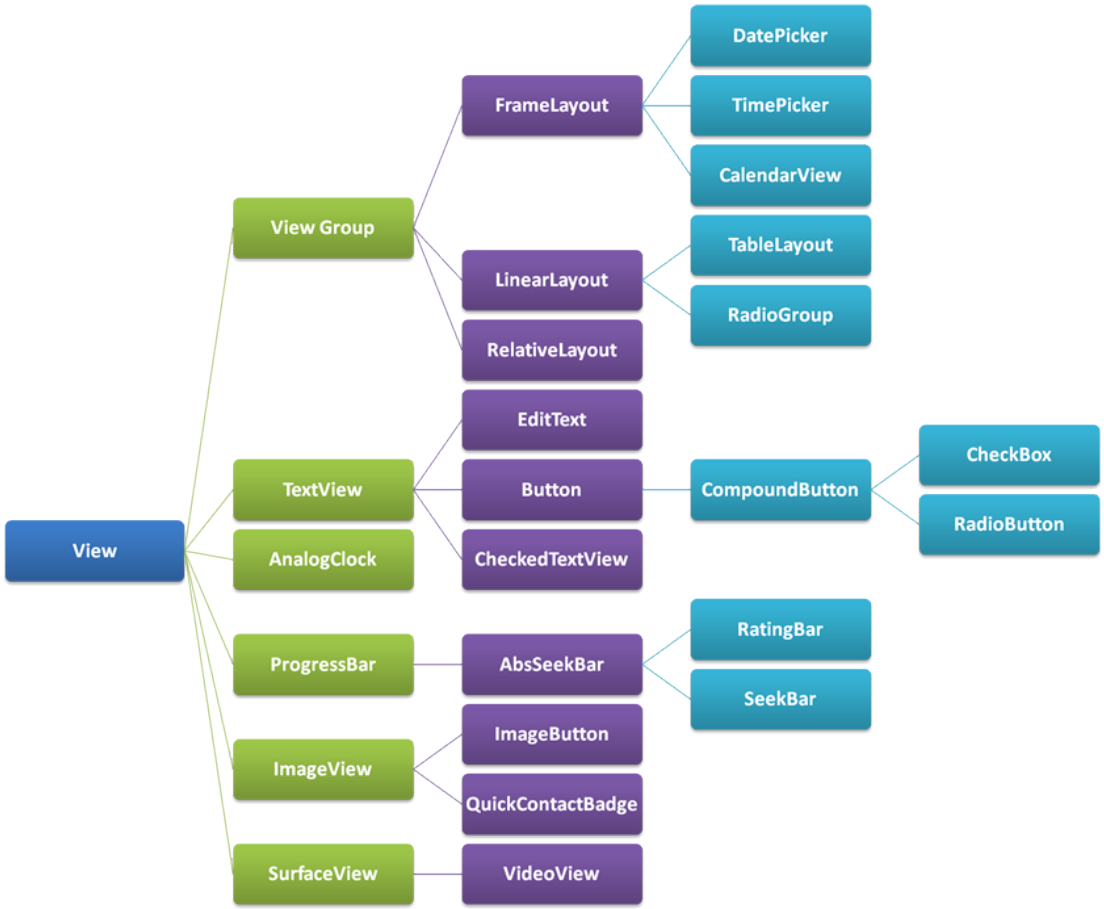
(Installation eclipse/Android SDK/Emulateur et premier exemple faits en cours)

Extrait de [And13]



Extrait de [And13]

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState); // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart(); // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume(); // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause(); // Another activity is taking focus (this activity is about to be "paused").
    }
    @Override
    protected void onStop() {
        super.onStop(); // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy(); // The activity is about to be destroyed.
    }
}
```



(image empruntée à `itcsolutions.eu`)

Exercice 1 Créer une application Android en mode *FullScreen* contenant :

1. un `RelativeLayout` identifié *rellay*, prenant la totalité de l'espace disponible pour l'activité (ie. `Activity`) et dont le fond est de couleur 0 en rouge, 255 en vert et 255 en bleu ; cette partie est à réaliser via le fichier xml rreprésentant le *Content View* de l'*Activity* ;
   (bis) Côté java, mettre la couleur de fond de *rellay* à jaune via l'appel d'une méthode statique à écrire : elle prend une quantité (`int`) de rouge, de vert et de bleu et renvoie le code couleur opaque (ie. alpha = 255) correspondant.
2. A l'évènement `OnCreate` de l'activité, nous ajouterons au layout *rellay* un `Button` contenant le texte "Hello 0" ; ce bouton doit être centré verticalement et horizontalement ;
3. Ce bouton écoute les évènements de type `onClick` et doit, à chaque clic modifier son contenu de manière à toujours avoir le texte "Hello" suivi du nombre de fois où le bouton a été cliqué ;
4. Ajouter un moyen automatique de décrémenter ce nombre de clics de 1 toutes les secondes.

Solution - exercice 1 - 1

```
<?xml version="1.0" encoding="utf-8"?>
<!-- pour le fill_parent : mettre match_parent pour 8+ -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#FF00FFFF"
    android:id="@+id/rellay">
</RelativeLayout>
```

Solution - exercice 1 - 1 (bis)

```
package fr.fbelhadj.hellobutton;

import android.app.Activity;
import android.os.Bundle;
import android.widget.RelativeLayout;

public class HelloButtonActivity extends Activity {
  public static int couleur(int red, int green, int blue) {
    return 0xFF000000 | (red << 16) | (green << 8) | blue;
  }
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    RelativeLayout rl = (RelativeLayout)findViewById(R.id.rellay);
    rl.setBackgroundColor(couleur(255, 255, 0));
  }
}
```

Solution - exercice 1 - 2

```
package fr.fbelhadj.hellobutton;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.RelativeLayout;

public class HelloButtonActivity extends Activity {
  private int nbClicks = 0;
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    RelativeLayout rl = (RelativeLayout)findViewById(R.id.rellay);
    Button button = new Button(getApplicationContext());
    button.setText("Hello " + nbClicks);
    RelativeLayout.LayoutParams lp = new RelativeLayout.LayoutParams(
    RelativeLayout.LayoutParams.WRAP_CONTENT,
    RelativeLayout.LayoutParams.WRAP_CONTENT );
    lp.addRule(RelativeLayout.CENTER_IN_PARENT);
    rl.addView(button, lp);
  }
}
```

Solution - exercice 1 - 3

- Ajout d'un *listener* d'évènement au bouton ;
- Un *listener* est une interface (une classe 100% abstraite) qui contient une méthode pouvant être appelée au moment où se déclanche l'évènement.

```
1       button.setOnClickListener(new View.OnClickListener() {
          @Override
3         public void onClick(View v) {
            if(v instanceof Button) // Pas necessaire, on le sait deja
5             ((Button)v).setText("Hello " + ++nbClicks);
          }
7       });
```

Solution - exercice 1 - 4

Quelle est le problème avec cette solution ?

```
1       ...

3       Button button = new Button(getApplicationContext()) {
          private int _nbClicks = -1;
5         private CharSequence _text = "";
          @Override
7         public void setText(CharSequence text, BufferType type) {
            _text = text;
9           super.setText(text + " " + ++_nbClicks, type);
          }
11        @Override
          public void draw(Canvas canvas) {
13          super.draw(canvas);
            try {
15            Thread.sleep(1000);
              _nbClicks -= 2;
17            setText(_text);
            } catch (InterruptedException e) {
19            setText("Echec sleep");
            }
21        }
        };
23      button.setText("Hello");

25      ...
```

Problème de la solution précédente :

La requète faite au processus demandant d'essayer de faire un *sleep* d'une seconde (cf. `Thread.sleep(1000)`) est faite dans (l'*override* de) la méthode `draw` ; par conséquent celà bloque l'affichage mais aussi toutes possibilités d'interaction de la *view* et par extension de l'*activity* pendant une seconde.

$\Longrightarrow$ La solution proposée est inappropriée.

Seconde solution - exercice 1 - 4

```
1       Button button = new Button(getApplicationContext()) {
          private long nbClicks = -1, t0 = System.currentTimeMillis();
3         private CharSequence _text = "";
          @Override
5         public void setText(CharSequence text, BufferType type) {
            super.setText((_text = text) + " " + ++nbClicks, type);
7         }
          @Override
9         public void draw(Canvas canvas) {
            super.draw(canvas);
11          long t = System.currentTimeMillis();
            if(t - t0 > 1000) {
13            t0 = t;
              nbClicks -= 2;
15            setText(_text);
            }
17          invalidate();
          }
19      };
        button.setText("Hello");
21      rl.addView(button, lp);
        button.setOnClickListener(new View.OnClickListener() {
23      @Override
        public void onClick(View arg0) {
25        if(arg0 instanceof Button)
            ((Button)arg0).setText("Hello");
27      }
        });
```

Problème de la solution précédente :

Solution satisfaisante, reste que l'`invalidate()` rajouté à la fin de la surcharge de draw demande le rafraîchissement systématique du bouton.

$\Longrightarrow$ La solution peut-être améliorée (voir plus loin).

Exercice 2 Créer une application Android en mode *FullScreen* contenant :

1. un `RelativeLayout` identifié *rellay*, prenant la totalité de l'espace disponible pour l'activité (ie. `Activity`) et dont le fond est de couleur 255 en rouge, 0 en vert et 0 en bleu ;
2. A l'évènement `OnCreate` de l'activité, nous ajouterons au layout *rellay* 9 (neuf) `Buttons` disposés en 3x3 et contenant leurs numéros compris entre 1 et 9 ; l'ensemble doit être centré verticalement et horizontalement ;
3. Chaque bouton écoute les évènements de type `onClick` et doit, à chaque clic disparaître pendant une seconde.

Solution - exercice 2 - 1

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=''http://schemas.android.com/apk/res/android''
    xmlns:tools=''http://schemas.android.com/tools''
    android:layout_width=''match_parent''
    android:layout_height=''match_parent''
    android:background=''#FFFF0000''
    android:padding=''40dp''
    android:id=''@+id/rellay''>
 </RelativeLayout>
```

Solution - exercice 2 - 2

```
1  package fr.fbelhadj.hellobutton;
   import android.os.Bundle;
3  import android.app.Activity;
   import android.view.Window;
5  import android.widget.Button;
   import android.widget.RelativeLayout;
7  public class MainActivity extends Activity {
     @Override
9    protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
11     requestWindowFeature(Window.FEATURE_NO_TITLE);
       setContentView(R.layout.activity_main);
13     RelativeLayout rl = (RelativeLayout)findViewById(R.id.rellay);
       int pH[] = {RelativeLayout.ALIGN_PARENT_LEFT,
15       RelativeLayout.CENTER_HORIZONTAL, RelativeLayout.ALIGN_PARENT_RIGHT};
       int pV[] = {RelativeLayout.ALIGN_PARENT_TOP,
17       RelativeLayout.CENTER_VERTICAL, RelativeLayout.ALIGN_PARENT_BOTTOM};
       for(int i = 0; i < 3; i++) {
19       for(int j = 0; j < 3; j++) {
           Button b = new Button(getApplicationContext());
21         RelativeLayout.LayoutParams lp = new RelativeLayout.LayoutParams(
           RelativeLayout.LayoutParams.WRAP_CONTENT,
23         RelativeLayout.LayoutParams.WRAP_CONTENT);
           lp.addRule(pV[i]);
25         lp.addRule(pH[j]);
           b.setText("" + (i * 3 + j + 1));
27         rl.addView(b, lp);
         }
29     }
     }
31 }
```

Recherche de Solution - exercice 2 - 3

Première étape : voir la classe `AsyncTask` [Asy14].

Recherche de Solution - exercice 2 - 3

Deuxième étape : ci-après un bout de code (dans la double boucle précédente) qui devrait fonctionner avec des *Thread* mais qui finalement ne fonctionne pas. Voir pourquoi et trouver une solution.

```
...
rl.addView(b, lp);
b.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(final View arg0) {
    arg0.setVisibility(View.INVISIBLE);
    Thread t = new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          Thread.sleep(1000);
          arg0.setVisibility(View.VISIBLE);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      }
    });
    t.start();
  }
});
...
```

Recherche de Solution - exercice 2 - 3

Regardons de plus près `AsyncTask` [Asy14] :

- **Résumé rapide :** Cette classe permet d'effectuer des opérations en tâche de fond et de passer les résultats à la tâche UI sans avoir à manipuler le couple `Thread/Handler` ;
- `AsyncTask<Params, Progress, Result>` est une classe abstraite pour laquelle il faut au minimum définir les types de `Params, Progress` et `Result` ainsi que la méthode `Result doInBackground(Params... params)` ; Où :
  - `Params` est le type défini pour les objets traités par cette `AsyncTask` ;
  - `Progress` est le type défini pour les objets indicant la progression de la tâche de fond (ie. tâche effectuée dans `doInBackground`) ;
  - `Result` est le type défini pour le résultat émis par la tâche de fond (ie. le retour de la méthode `doInBackground`) ;

  De plus `AsyncTask` définit d'autres méthodes *callback* agissant à différents stades de l'exécution de la tâche de fond. Exemple : `void onPreExecute()` ; `onPostExecute(Result result)` ; `void onProgressUpdate(Progress... values)` ; ...

Première vraie solution - exercice 2 - 3

Remplacer la partie du code utilisant la classe `Thread` par un code utilisant une dérivée de `AsyncTask`.

```
...
rl.addView(b, lp);
b.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(final View arg0) {
    arg0.setVisibility(View.INVISIBLE);
    AsyncTask<Void, Void, Void> at = new AsyncTask<Void, Void, Void>() {
      @Override
      protected Void doInBackground(Void... params) {
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {
          System.err.println("Echec du Thread.sleep");
        }
        return null;
      }
      @Override
      protected void onPostExecute(Void result) {
        super.onPostExecute(result);
        arg0.setVisibility(View.VISIBLE);
      }
    };
    at.execute();
  }
});
...
```

Deuxième solution - exercice 2 - 3

Version avec passage de paramètres entre la tâche de fond et le processus UI principal.

```
1   ...
    rl.addView(b, lp);
3   b.setOnClickListener(new View.OnClickListener() {
      @Override
5     public void onClick(final View arg0) {
        arg0.setVisibility(View.INVISIBLE);
7       AsyncTask<View, Void, View> at = new AsyncTask<View, Void, View>() {
          @Override
9         protected View doInBackground(View... params) {
            assert params.length == 1;
11          try {
              Thread.sleep(1000);
13            return params[0];
            } catch (InterruptedException e) {
15            System.err.println("Echec du Thread.sleep");
              return null;
17          }
          }
19        @Override
          protected void onPostExecute(View result) {
21          super.onPostExecute(result);
            if(result != null)
23          result.setVisibility(View.VISIBLE);
          }
25      };
        at.execute(arg0);
27    }
    });
29  ...
```

Exercice 3

Refaire la solution de l'exercice 1-4 en utilisant la classe `AsyncTask`.

Exercice 4

Partant de la solution de l'exercice 2, imaginons un jeu où l'utilisateur doit faire disparaître l'ensemble des boutons présents à l'écran avant de passer au niveau suivant. Les niveaux seront différenciés par un nombre et un positionnement différent des boutons ainsi qu'un temps de "réapparition" différent.

Sous la forme de TP $\Longrightarrow$ faire une application affichant du text (bouton ou autres) et gérant au minimum 2 langues (ex. EN et FR).

Téléchargement : Zoom sur Image

Récupérer l'archive ImgZoom.zip présente sur la page web du cours [Pag13] : extension d'une ImageView avec listeners d'évènements touch permettant de zoomer et parcourir une image en détails.


Android Activity.
Documentation relative à l'activity sous android.
http://developer.android.com/guide/components/activities.html, 2013.


AsyncTask.
Documentation relative à la classe asynctask.
http://developer.android.com/reference/android/os/AsyncTask.html, 2014.


Collection.
Documentation de l'interface collection.
http://docs.oracle.com/javase/6/docs/api/java/util/Collection.html, 2014.


Conventions d'écriture de code Java.
Code conventions for the java programming language.
http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html, 1999.


Iterable.
Documentation de l'interface iterable.
http://docs.oracle.com/javase/6/docs/api/java/util/Iterable.html, 2014.


Iterator.

Documentation de l'interface iterator.
http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html, 2014.

Page du cours.
Page web dédiée au présent support de cours.
http://www.ai.univ-paris8.fr/~amsi/CPAJAVA1314S1/, 2013.