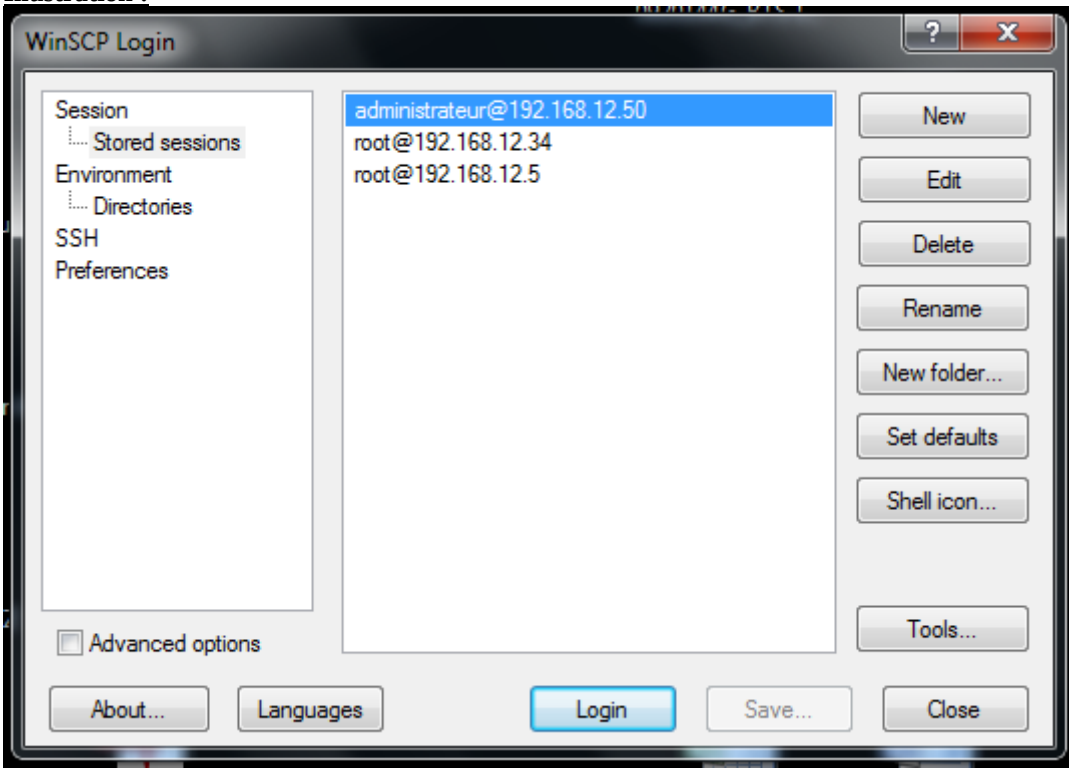
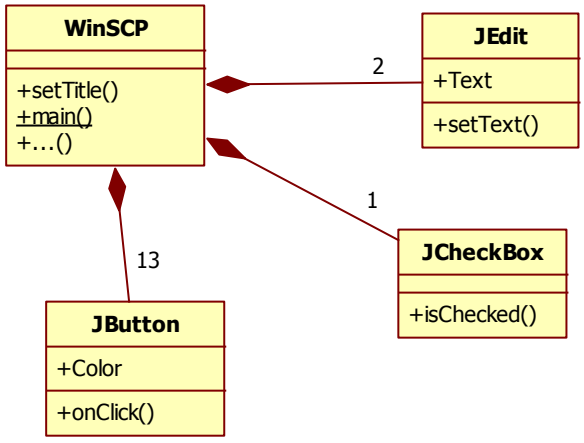
JAVA : les classe et les objets

1] Classes et objets

Les objets sont des entités logicielles ayant des caractéristiques (attributs ou données membres) et de méthodes permettant de manipuler ces membres.
Une classe est un modèle de conception d'un ou plusieurs objets de même nature.
Un objet est une **instance** de classe.
Illustration :



L'application fenêtrée winSCP, est un objet issu d'une classe **winSCP**. Cette application est composée de boutons eux même issus de la classe JButton, d'un objet JCheckBox, d'un objet JEdit etc…
Chaque objets (boutons, fenêtre, checkbox…) possède des caractéristiques (hauteur, largeur, couleur, titre…) manipulables par des méthodes de la classe dont ils sont issus (setTitle, setColor, appendText…).
On peut représenter cette application de la façon suivante :



2] Instanciation des objets et variables :

Toutes les instanciations d'objets doivent être faites par des références aux objets : initialisées lors de la déclaration.

2.1] Instanciation de types primaires :

```
int b=0;
byte[] tab = {5,8,7};     // tableau de 3 octets initialisés.
```

```
byte [] b = new byte [ 5 ];
```

2.2] Instanciation d'objets:

```
Byte oct = 0x25;
String chaine = new String ("coucou");
Thread toto = new Thread ( ) ;
Thread mth = null;
```

3] Utilisation des objets :

Lorsqu'un objet est instancié et correctement initialisé, on accède à ces membres par l'opérateur **point**.

`System.out.println("coucou");`	Appel de la méthode **println** de l'objet **out** du package **System**
`Individu inn = new Individu();`	Création d'un objet **Individu** nommé **inn**
`System.in.read(tab);`	Appel de la méthode **read** de l'objet **in** du package **System**
`inn.setNom(new String(tab));`	Appel de la méthode **setNom** de l'objet **inn**
`int u = Integer.parseInt(tmp);`	Création d'un entier

4] Les classes

4.1] Ecriture d'une classe :

Classe sans package :	Classe appartenant à un package :

`public class nomDeClasse {`
 	
`}`
	`package nomPack`
	
	`public class nomDeClasse {`
	`}`

4.2] Membres

Chaque classe possède une ou plusieurs données membres représentant les caractéristiques des objets issus de la classe.
Des méthodes appartenant à la classe permettront de manipuler ces données.
Les membres d'une classe sont :

- ⇨ Données membres appelées parfois **attributs**
- ⇨ Méthodes analogue aux fonctions

4.2.1] Les données membres doivent être typées et initialisées car elles sont définies par références.

- ⇨ Elles peuvent être de type natif ou des objets issus d'autres classes.
- ⇨ Elles doivent être définies dans la classe comme ceci :

```
modeAcces   Type nom = new  Type(…) ;                    pour des objets
modeAcces   typeXXX [ ] nom  = new typeXXX [TAILLE ] ;   pour les tableaux
modeAcces   Type nom = null ;
modeAcces   typeNatif    nom ;                           pour les types natifs
modeAcces   typeNatif    nom  = valeur ;
```

- ⇨ **null** permet de déclarer un objet qui sera réellement initialisé ultérieurement.
- ⇨ Les modes d'accès peuvent être **public**, **protected**, **private**

```
public class nomDeClasse {

      public      int age;
      private     String texte = null;
      public      JButton  b1 = new JButton("Valider");
                  }
```

4.2.2] Les méthodes :

• Les méthodes doivent être déclarées et définies de la manière suivante :

```
type_acces      type_retourné          nom     ( liste des types et noms des arguments)
                { corps de la méthode }
```

Ex :

```
public void Voir (int a , String b )
        {
        System.out.println ( b + " " + a);
        }
```

⇨ On accède aux méthodes de la classe d'un objet via l'opérateur '**.**'.
objet.setText ("nouveau texte");

⇨ Les méthodes statiques (static) peuvent être appelées via un objet ou bien de la manière suivante :
Nom_package.nom_classe.nom_méthode (...) ;
Ex : *java.lang.Thread.sleep (1000) ;*

⇨ Une **application est une classe** comme une autre, mais qui contient **une méthode main ( ) static et public** dans laquelle est instancié un objet de la classe nécessaire à l'application, le constructeur de cette classe instancie les autres objets nécessaires.

```
public static void main(String[] args){}
```

args est un tableau de String contenant les paramètres passés au programme.

4.2.3] Modificateurs d'accès : private protected public

Définissent les droits d'accès sur les membres et les packages. Ils permettent un accès public, protégé ou privé sur les membres de la classe (données ou méthodes).

modificateurs	classe	package	Sous classe	reste
public	oui	oui	oui	oui
protected	oui	oui	oui	non
Sans modificateurs	oui	oui	non	Non
private	oui	non	non	non

4.2.4] Les constructeurs

Le constructeur d'une classe est une méthode particulière, elle porte le **nom de la classe**, **ne retourne pas de valeur** et est **appelé automatiquement** par la JVM lors de l'instanciation d'un objet.
Il peut-y avoir plusieurs constructeurs s'ils sont de signatures différentes (liste des types des arguments différente).

```
public Individu(String r , int x){…}
public Individu(String n , int a){…}            //incorrect même signature

public Individu(){…}
public Individu(String n , int a){…}            //correct signature différente
```

⇨ En java, il n'existe pas de destructeur. Le nettoyage des objets ne possédant plus de référence dans le système incombe au **garbage collector (ramasse miettes).** Cependant on peut forcer la finalisation d'un objet grâce à la méthode **finalize**

```
@Override
protected void finalize()
{
      //code spécifique
}
```

Ce mécanisme est désormais obsolète et on préférera appeler explicitement le **garbage collector** ainsi :

```
System.gc();
```

4.2.5] L'auto-référence : this

Au sein de la classe, pour les membres non sttics,, il est possible de manipuler directement la référence à l'objet. Pour cela on utilise le mot réservé **this** de la manière suivante :

```
this.age = 5;
```

4.2.6] L'opérateur instanceof

Cet opérateur permet de savoir si un objet est issu d'une classe ou non.

Ex : `System.out.println("Objet de la classe ihmBase: " + (ihm instanceof ihmBase) );`

Ce code affichera : `Objet de la classe ihmBase:true`

4.2.7] Les mots clés final et static :

Static permet de définir une méthode ou une variable unique quelque soit le nombre d'objet créé :
Se déclare en ajoutant au prototype le mot réservé static.

Ex `public static void main ( String[] args ) { }`

- ⇨ Si c'est une méthode :
  - Impossible d'utiliser **this**. Pas d'attachement à un objet particulier
  - Impossible d'utiliser les méthodes non statiques sans passer par un objet
  - S'utilise de 2manières :
    - **`Objet.methodeStatique() ;`**
    - **`nomClasse.methodeStatique() ;`**
- ⇨ Si c'est une donnée :
  - Existe en un seul exemplaire et est commune à tous les objets
  - S'utilise de plusieurs manères :
    - **`objetX.donneesStatique . . .`**
    - **`nomClasse.donneesStatique . . .`**

final appliqué à une donnée empêche sa modification. Il permet de créer des constantes.

Ex :

```
public final float  pi = 3.1415927f;//constante identique pour chaque objet
```

Etant constant, même valeur pour tous les objets de la classe, il sera **static** afin de ne créer qu'un seul exemplaire.

```
public static final float  pi = 3.1415927f;//constant unique pour tous les objets
```

Ce mot réservé est utilisable sur des méthodes, classes et packages.

4.2.8] Exercices :

⇨ Créer un projet eclipse (ou autre) composé de la classe suivante :

```
package oiio.chapitre04

//import javax.swing.*;
//import java.awt.GridLayout;

public class ihmBase extends JFrame {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		ihmBase ihmm = new ihmBase();
		ihmm.setVisible(true);
	}

}
```

A) Ce code est-il correct ?
B) Quels sont les problèmes ?
C) Décommenter, constater et conclure.

D) exécuter ce programme. En déduire la nature de l'objet **ihmBase** créé.

⇨ Que représente la ligne
`ihmBase ihmm = new ihmBase();`
⇨ Que représente **ihmm** ?

⇨ Ajout d'objets : Compléter le code de la façon suivante :

```
public static void main(String[] args) {
	ihmBase ihmm = new ihmBase();
	ihmm.setVisible(true);
	ihmBase ihmm2 = new ihmBase();
	ihmm2.setVisible(true);
}
```

⇨ Exécuter et constater.
⇨ Combien d'objets **ihmBase** ont été créé ?

⇨ Enrichir la classe de base. Pour cela, on ajoute à la classe deux nouvelles données membres issues elles mêmes d'une autre classe (JButton). Modifier le code de la façon suivante :

```
public class ihmBase extends JFrame {

JButton b1 = new JButton("Valider");
JButton b2 = new JButton("Cancel");
public void init(int x , int y ,int w , int h)
	{
	JPanel p=new JPanel(new GridLayout(2,1));
	p.add(b1);
	p.add(b2);
	this.setTitle("Titre de la fenêtre");
	this.setBounds(x,y,w,h);
	this.setVisible(true);
	this.setContentPane(p);
	}
public static void main(String[] args) {
	ihmBase ihmm = new ihmBase();
	ihmm.init(10,10,200,300);
	ihmBase ihmm2 = new ihmBase();
	ihmm2.init(100,100,100,100);
	}
}
```

- ⇨ Exécuter et constater l'influence des paramètres de la méthode **init**.
- ⇨ Modifier le code pour obtenir **une seule** fenêtre carrée visible dans le quart **supérieur droit** de l'écran.
- ⇨ Numéroter ci-contre dans l'ordre chronologique les lignes de codes exécutées.

⇨ Mise en œuvre des constructeurs :
⇨ Modifier le code ainsi :

```
public class ihmBase extends JFrame {

	JButton b1 = new JButton("Valider");
	JButton b2 = new JButton("Cancel");

	public ihmBase()
	{
		init(100 , 100 ,400 , 500);
		System.out.println("objet : " + this);
	}
	public void init(int x ,int y,int w,int h)
	{
	JPanel p=new JPanel(new GridLayout(2,1));
	p.add(b1);
	p.add(b2);
	this.setTitle("Titre de la fenêtre");
	this.setBounds(x,y,w,h);
	this.setVisible(true);
	this.setContentPane(p);
	}
	public static void main(String[] args) {
		ihmBase ihmm = new ihmBase();
	}
}
```

- ⇨ Que fait le constructeur ?
- ⇨ Ajouter et tester un constructeur permet de modifier la taille de la fenêtre.
- ⇨ Ajouter et tester un autre constructeur permettant de choisir le titre de la fenêtre.
- ⇨ Numéroter ci-contre dans l'ordre chronologique les lignes de codes exécutées.

⇨ **Les modificateurs d'accès :** Créer une autre classe nommée lanceur, appartenant au package oiio.chapitre4 telle que :

```
package oiio.chapitre04;
public class lanceur {

	public static void main(String[] args)
{
		ihmBase ihm = new ihmBase();
		ihm.b1.setText("coucou");
	}
}
```

Ce code est-il correct ? justifier.
Modifier la classe **ihmBase** ainsi :

```
private JButton b1 = new
JButton("Valider");
```

Quelle est la nature de l'erreur?

⇨ Membres statiques : Ajouter une donnée membre de type numérique pour compter le nombre d'objets **ihmBase** créés. Implémenter le code adéquat dans les constructeurs pour que cette valeur s'incrémente à chaque nouvel objet.

4.3] Les classes internes :

Il est possible de définir des classes dans d'autres classes. On dit que ce sont des classes internes. Généralement utilisé pour implémenter les gestionnaires d'événements qui sont spécifiques à une classe et n'ont pas de raisons d'avoir leur propre fichier. Ces classes seront détaillées plus tard.

5] Importer des classes ou des packages :

Pour utiliser les classes contenues dans les packages, il faut importer ces packages au sein de la classe.
Le mot clé est :**'import'**

```
Ex :    import   java.swing. *;          //tout le package
        import   java.util.date ;        //uniquement la classe date du package java.util
```

Par exemple, pour utiliser les classes de gestion de la date, il est nécessaire d'importer le package *'java.util.date'*.

6] La classe java.lang.String

Classe de gestion de texte.

Constructor Summary
String() Constructeur sans paramètre
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, int offset, int length) Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.
String(byte[] bytes, int offset, int length, String charsetName) Constructs a new String by decoding the specified subarray of bytes using the specified charset.
String(byte[] bytes, String charsetName) Constructs a new String by decoding the specified array of bytes using the specified charset.
String(char[] value) Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
String(char[] value, int offset, int count) Allocates a new String that contains characters from a subarray of the character array argument.
String(int[] codePoints, int offset, int count) Allocates a new String that contains characters from a subarray of the Unicode code point array argument.
String(String original) Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
String(StringBuffer buffer) Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

Method Summary	
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.

String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(StringBuffer sb) Compares this string to the specified StringBuffer.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
static String	**copyValueOf**(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	**equals**(Object anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations.
static String	**format**(Locale l, String format, Object... args) Returns a formatted string using the specified locale, format string, and arguments.
static String	**format**(String format, Object... args) Returns a formatted string using the specified format string and arguments.
byte[]	**getBytes**() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[]	**getBytes**(Charset charset) Encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.
void	**getBytes**(int srcBegin, int srcEnd, byte[] dst, int dstBegin) **Deprecated.** *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the getBytes() method, which uses the platform's default charset.*
byte[]	**getBytes**(String charsetName) Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
boolean	**isEmpty**() Returns true if, and only if, length() is 0.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf**(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**lastIndexOf**(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
String	**replace**(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String	**replace**(CharSequence target, CharSequence replacement) Replaces each substring of this string that matches the literal target sequence with the specified

	literal replacement sequence.
String	**replaceAll**(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement.
String	**replaceFirst**(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
String[]	**split**(String regex) Splits this string around matches of the given regular expression.
String[]	**split**(String regex, int limit) Splits this string around matches of the given regular expression.
boolean	**startsWith**(String prefix) Tests if this string starts with the specified prefix.
boolean	**startsWith**(String prefix, int toffset) Tests if the substring of this string beginning at the specified index starts with the specified prefix.
CharSequence	**subSequence**(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toLowerCase**(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.
String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**toUpperCase**(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.
static String	**valueOf**(int i) Returns the string representation of the int argument. Existe pour tous les types natifs

6.1] Exercices :

A partir du texte suivant :

Bonjour%etudiant%de%lasection%SLAM%oiio%2014

On veut mettre l'ensemble du texte en majuscule, remplacer **etudiant** par **étudiant(es),** remplacer le '%' par ' ?', connaître la taille du texte, créer un tableau de String contenant chacun des mots délimités par ' ?' (split) et enfin, extraire le texte 2014, le convertir en entier, ajouter 10, créer un nouveau texte égal au résultat converti.
Le résultat final devra être comparé à celui-ci :

Bonjour ?étudiant(es) ?de ?lasection ?SLAM ?oiio ?2024

Est-ce le cas ?

7] la classe java.utils.arrays

Classe permettant de gérer les tableaux (tri, taille …): voir la documentation.

8] Les exceptions :

Une exception est levée (déclenchée) lorsque la JVM détecte une erreur au cours de l'exécution d'une partie de code. Dès lors, le déroulement du code est détourné au profit d'un gestionnaire d'exception.
Il existe de multiple cause d'erreurs, les plus courantes étant les problèmes d'entrées/sorties, de formats de données incompatibles, d'objet non initialisés… C'est pour cela que java propose un grand nombre d'exceptions implémentées.
Java impose donc pour un grand nombre de classes de surveiller (**try**) une partie de code et d'y adjoindre un gestionnaire spécifique à l'exception qui risque d'être levée.

La syntaxe est la suivante :

```
try     { … code à risque
          }
          catch ( Type_Exception   variable )
                  { …  traitement des erreurs
                   }
          finally  { …  code exécuté dans tous les cas ( optionnel)
                   }
```

variable est une variable de type **Type_Exception** munie de différentes méthodes, notamment **getMessage** ( ) qui permet de récupérer le message d'erreur envoyé sur la console lors de l'exécution.

Exemple : lecture du clavier :

`try {`	Essai
`. . .`	
`System.in.read(tab);`	Lecture clavier
`inn.visu();`	
`} catch (IOException e)`	Bloc traitement erreur
`{`	
`System.out.println("Erreur: " + e.getMessage());`	Affichage de la cause de l'erreur
`System.exit(-1);`	Sortie brutale du programme
`e.printStackTrace();`	Affichage de la cause de l'erreur
`}`	

La JVM essaie de réaliser le code dans le bloc **try**, en cas d'erreurs l'exécution est interrompue et le code dans le bloc **catch** est exécuté. La JVM passe en argument une variable du type de l'exception.
Enfin le code dans le bloc **finally** est exécuté.

Ex : Tester le code suivant :

```
byte[] b = new byte[5];
System.out.println("b=" + b[10]);
```

Puis celui-ci:

```
byte[] b = new byte[5];
try{
      System.out.println("b=" + b[10]);
}catch(Exception ie){
      System.out.println("Exception:" + ie.getMessage());
}
```

Conclusion :

L'API java :

En java, il existe un grand nombre de classes, toutes issues directement ou indirectement de la classe de base Object offrant déjà divers service tel que :

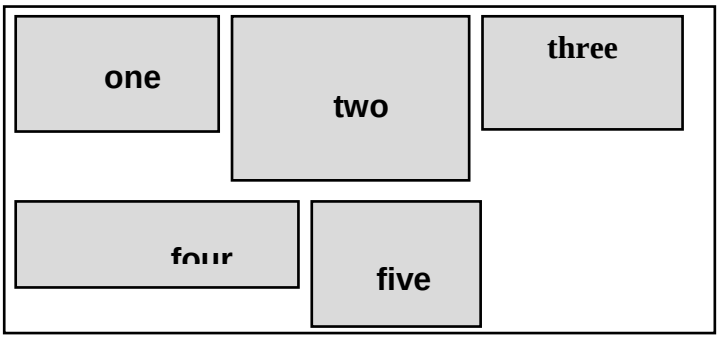
Method Summary	
protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this 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 the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	**wait**(long timeout) Causes the 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 the 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.

A partir de cette classe, un grand nombre de classes sont rangées dans un grand nombre de packages dont il est impossible de donner la liste exhaustive (voir la documentation http://docs.oracle.com/javase/6/docs/api/).

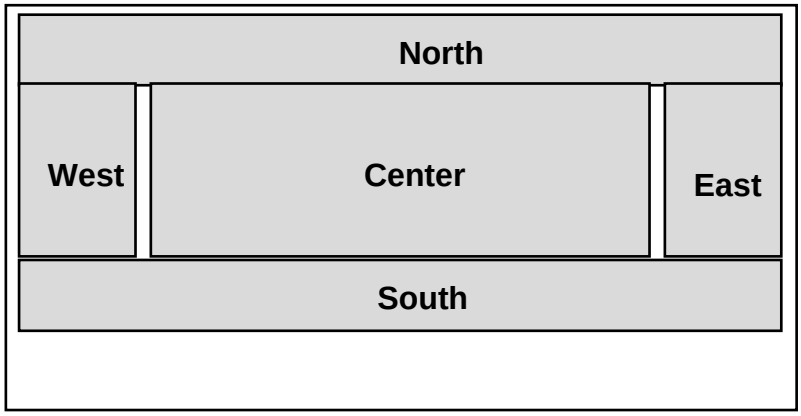
On peut cependant présenter les plus courus

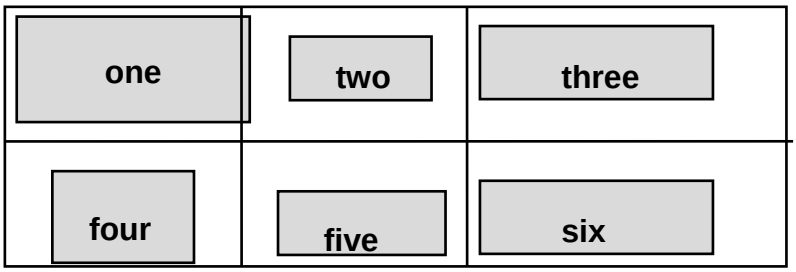
- **java.awt (bibliothèques de composants graphiques Abstarct Windows Toolkit) :**
  - **composant graphique** **Frame, Button, Edit**
  - **layout** **FlowLayout, BorderLayout, gridLayout**
  - **container** **Panel**
- **javax.swing (bibliothèques de composants graphiques plus récent que AWT) :**
  - **composant graphique** **JFrame, JButton, JEdit**
  - **container** **JPanel**

- **java.awt.FlowLayout**

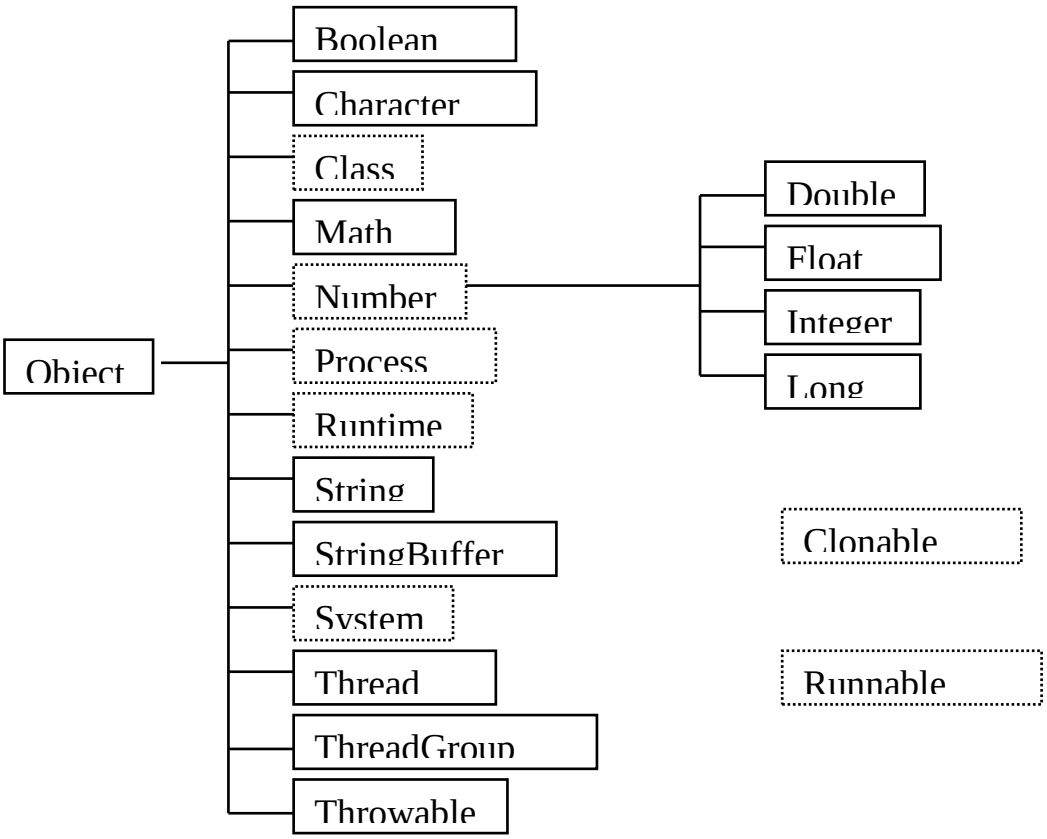


- **java.awt.BorderLayout**





- **java.awt.GridLayout**
- **java.lang :**



- java.lang.String gestion du texte en Unicode (caractère sur 2 octets)
- **java.io (accès aux fichiers) :**
- **java.io.File :**
- **java.io.File(Input|Output)Stream**
- **java.io.Data(Input|Output)Stream**
- **java.net (accès réseau) :**
- **java.net.Socket :**
- **java.net.ServerSocket :**
- **java.net.URL :**
- **javax.sql**
- **javax.xml**

.