

Algorithmique et UML

CM3 : Java

Mickaël Martin Nevot

V1.15.0



Cette œuvre de [Mickaël Martin Nevot](#) est mise à disposition selon les termes de la [licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Partage à l'Identique 3.0 non transposé](#).

Algorithmique et UML

- I. Prés. du cours
- II. Init. à la prog.
- III. Algo.
- IV. APP
- V. Java
- VI. Java avancé
- VII. Algo. avancée
- VIII. UML
- IX. Génie log.

Java

- Sun Microsystem (1995)
- Langage :
 - **Orienté objet et fortement typé**
 - **Héritage simple**, interface, polymorphisme
- JRE :
 - JVM : **machine virtuelle** qui interprète le code
 - API : bibliothèques standards
- JDK :
 - Compilateur
 - JVM : débogueur



Philosophie

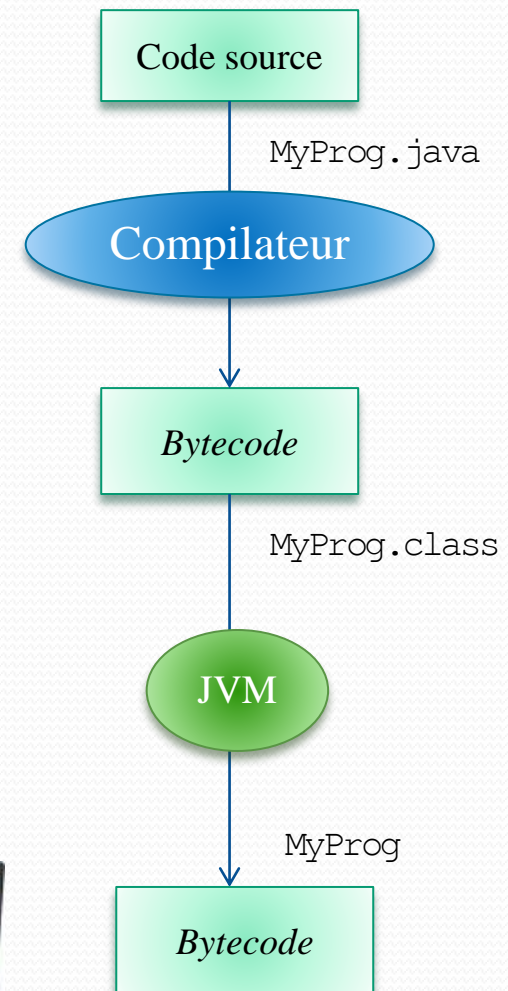
- **Simple** et familier
- **Robuste** et sûr
- **Indépendant** de la machine employée pour l'exécution
- Très **performant**
- Interprété, multitâches et dynamique
- Pourquoi apprendre Java ?
 - Plus de 4,5 milliards de périphériques
 - Programmes Web et services Web
 - Programme sur téléphone portable

Duke, la mascotte de Java



Fonctionnement

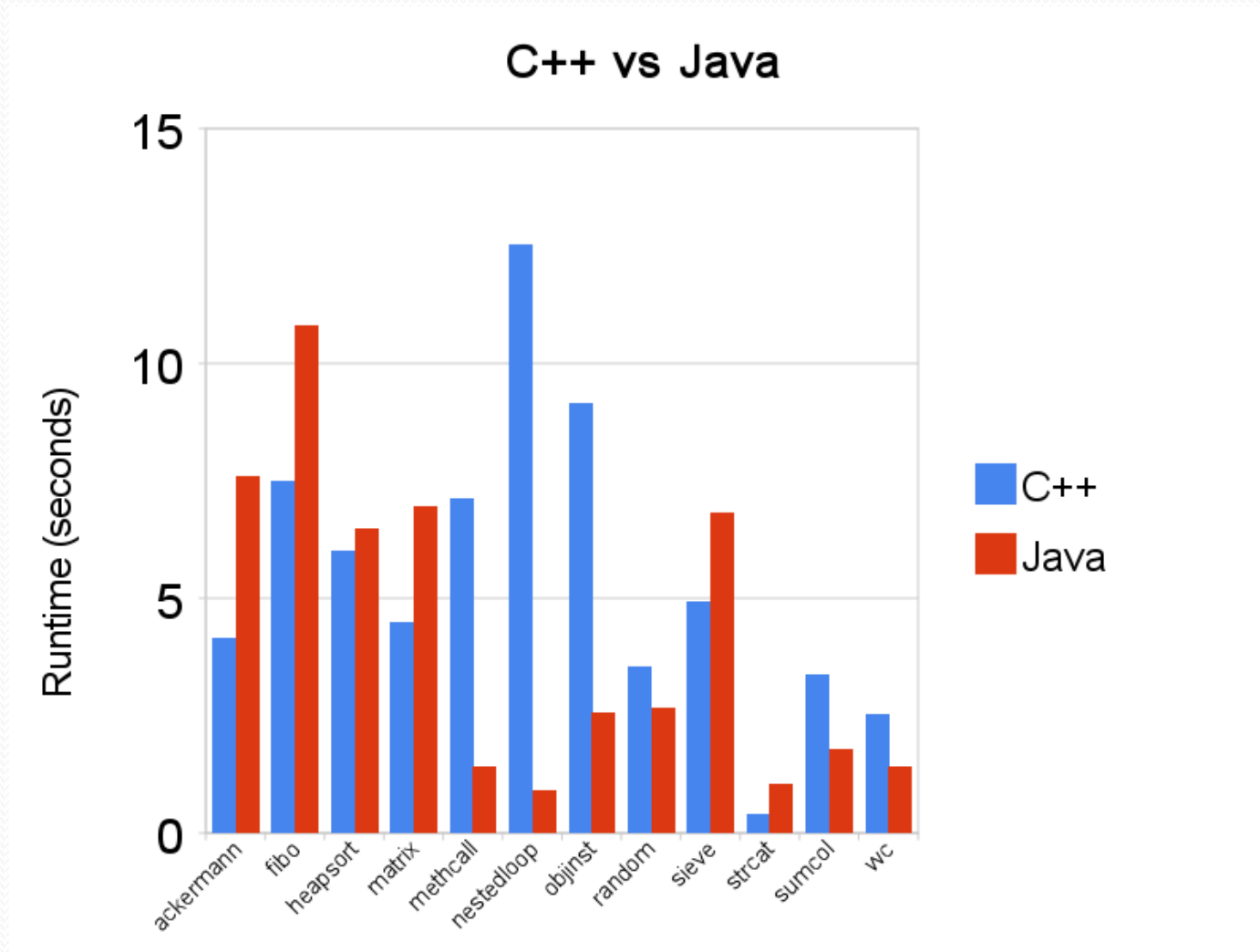
- Création du code source
- **Compilation** en *bytecode* :
 - À partir du code source
 - Code exécutable sur toute JVM
- Exécution :
 - Interprète le *bytecode*
 - *Bytecode* indépendant de la plateforme



Différences par rapport à C++

- Il n'y a que des classes (pas de struct, enum, union)
- Il n'y a que des références, pas de pointeurs
- Pas besoin de gérer la mémoire : *garbage collector*
- Java est indépendant de la plate-forme utilisée
- Java gère nativement les *threads*

Différences par rapport à C++



Structure du code source

- Un fichier source Java contient une **classe**
- Une classe contient des **attributs** et des **méthodes**
- Une méthode contient des **instructions**

```
// Déclaration de classe.  
class MyClass {  
    // Déclaration d'attribut.  
    int att1;  
  
    //Déclaration de méthode.  
    void meth1(int i) {  
        instruction1  
        instruction2  
        ...  
    }  
}
```

Structure du code source

- Méthode :

`typeDeRetour myMeth(paramètre(s))`

Signature

L'ordre des paramètres est déterminant

Délimitation de la classe

`// Déclaration de classe.`

`class MyClass {`

`int att1; // Déclaration d'attribut.`

Paramètre (typé)

`float meth1(int i) { //Déclaration de méthode.`

`instruction1`

`instruction2`

`...`

`}`

Délimitation de la méthode

Valeur de retour
(void : aucune)

Utilisation

- Fichier source : extension `.java`
- Fichier binaire : extension `.class`
- **Un** fichier source contient **une** classe
- Le nom du fichier est identique au nom de la classe
- On lance l'exécution par la classe principale :
 - Contient un point de commencement d'exécution du code
- Sensible à la casse : `MyClass` est différent de `Myclass`
- Respecter les mots-clefs réservés

Instruction

- Se termine par ;
- Type d'instruction :
 - Déclaration : `String att1;`
 - Affectation : `a = 10;`
 - Appel de fonction/méthode : `myMeth() ;`
 - Instruction conditionnelle : `if (a == b) { ... }`
 - Instruction vide : ;
 - Bloc (d'instructions) :

```
{  
    instruction1  
    ...  
}
```

Mise en forme / commentaires

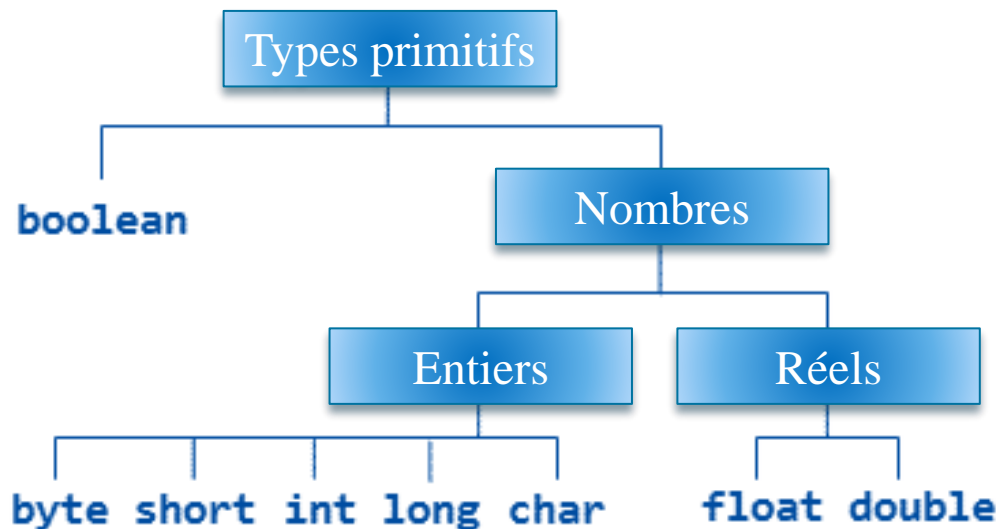
- Mise en forme :
 - Indentation à chaque niveau de bloc
 - Convention de nommage :
 - `mypackage`
 - `MyClass`
 - `myMethod`
 - `myVar`
 - `MY_CONST`

- Commentaires de type C/C++ :

```
// Commentaire (une seule ligne)
/* Autre commentaire (une ligne). */
/*
    Autre commentaire (sur plusieurs lignes).
*/
```

Type primitif

- N'est pas un objet
- Occupe une place fixe en mémoire
- Dispose d'un *alter ego* objet et d'une méthode de conversion
- Est converti automatiquement en référence (*autoboxing*)
- Conversion de type explicite (*cast*) : (**type**)



Types primitifs

- Entiers :

• byte	-128 à 127	1 octet
• short	-32768 à 32768	2 octets
• int	-2147483648 à 2147483647	4 octets
• long	-9223372036854775808 à 9223372036854775807	8 octets

- Flottants :

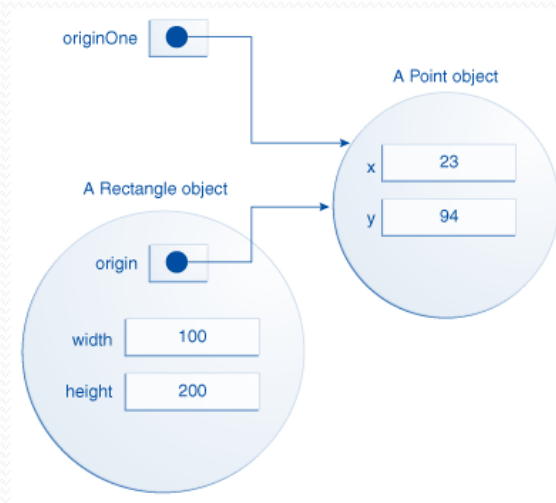
• float	variable	4 octets
• double	variable	8 octets

- Autres :

• boolean	true, false	selon la JVM
• char (Unicode, c.-à-d. a)	0 à 65535	2 octets

Référence

- Référence vers un objet : il n'existe **pas de variable objet**
- Une référence déclarée pour un type d'objet ne peut référencer que des objets de ce type
- Une référence ne référence qu'un seul objet à la fois
- Un objet peut être référencé par plusieurs références
- Aucune référence : null



Variable et constante

- Variable (deux catégories : **primitive** ou **référence**) :
 - Identifiant
 - Type
- Constante :
 - Variable ne pouvant avoir qu'une seule affectation
 - **Non modifiable**
 - Mot clef `final`

```
final int n = 5;  
final int t;  
...  
t = 8;  
n = 10; // Erreur !
```



Tableau

- Considéré comme un objet
- Un seul type par tableau (primitif/objet)
- Indices commencent à zéro (comme en C/C++)
- Mot clef `new` :
 - Alloue la mémoire en fonction de la taille (fixe)
 - Initialise à 0 (type primitif)
- Multidimensionnel (tableau de type tableau) :

Pas de dimensions à la déclaration

```
int[] myTab; // Déclaration.  
myTab = new int[3]; // Dimensionnement.  
myTab[0] = 1;  
myTab[2] = 5;
```

```
int myTab[] = {1, 0, 5};
```

=

Opérateurs

- Unaires :
 - Arithmétiques : +, −
 - Incrémentation / décrémentation (pré, post) : ++, --
 - Transtypage : (`type`)
- Binaires :
 - Arithmétiques : +, −, *, /, %
 - Affectations (élargies) : =, +=, -=, *=, /=
 - Comparaisons : ==, >, >=, <, !=
 - Logiques : &&, ||, &, |
 - Concaténation : +

Structure de contrôle

- Conditionnelle :
 - `if (condition) { ... } else { ... }`
- Branchement conditionnel :
 - `switch (ident) { case val0 : ... case val1 : ... default: ... }`
- Boucles :
 - `for (initialisation ; condition ; modification) { ... }`
 - `for (Type var : Collection) { ... }`
 - `while (condition) { ... }`
 - `do { ... } while (condition)`
- Mots clefs break/continue :
 - `break` : permet de sortir du bloc (boucle, branchement conditionnel, etc.)
 - `continue` : « saute » à l'itération suivante d'une boucle

Variable/méthode d'instance

- Variable d'instance (attribut) :
 - **Varie d'une instance (objet) à l'autre**
 - Initialisation non obligatoire. Valeur par défaut :
 - Entier `0`
 - Flottant `0.0`
 - Booléen, caractère `false`
 - Référence `null`
- Méthode d'instance (passage par valeur) :
 - Type primitif : ne modifie pas la valeur d'une variable
 - Référence : c'est l'objet qui est modifié par la référence

Notation pointée / this

- Notation pointée :

- Pour spécifier une hiérarchie de paquets :

```
package package1.package2;  
import java.lang.*;
```

- Pour accéder à un attribut :

```
myObj.att1; // Accès à un attribut.
```

- Pour accéder à une méthode :

```
myObj.meth1(); // Accès à une méthode.  
myObj.meth2(1, 2); // Accès à une méthode.
```

- Mot clef `this` (lecture seule) :

- Désigne l'**objet courant** (celui dans lequel on code) :

```
this.att1; // Accès à un attribut.  
this.meth1(); // Accès à une méthode.
```

- Fortement conseillé même si le sens n'est pas équivoque

Paquetage

- Groupe de classes associé à une fonctionnalité
- À chaque paquetage correspond un répertoire
- Un paquetage peut contenir un autre paquetage
- Mot clef package (première instruction d'une classe) :

```
package package1.package2;  
// MyClass appartient au paquetage package2  
// qui appartient lui-même à package1.  
class MyClass { ... }
```

- Mot clef import :

```
// Importe la classe MyClass (ci-dessus).  
import package1.package2.MyClass;  
// Importe tous les éléments du paquetage lang  
// qui appartient lui-même au paquetage java.  
import java.lang.*;
```


Affectation, recopie, comparaison

- Affectation/recopie : =

- Type primitif : modification distincte
- En cas de recopie, les deux opérandes restent distincts
- Objet : modification commune :

```
a = 8; // Affectation.
```

```
b = c; // Recopie (type primitif ou référence).
```

- Recopie de contenus d'objets : clone()

- Modification distincte :

```
a = b.clone();
```

- Comparaison : ==, equals(Objet o)

```
a == 8; // Comparaison.
```

```
b == c; // Comparaison (type primitif ou référence d'objet).
```

```
a.equals(b) // Comparaison de contenus d'objets.
```

Surcharge

- Possible pour n'importe quelle méthode
- Méthode avec le même nom mais pas avec la même liste de paramètres (**signature non identique**)
- Type de retour possiblement différent si les listes des paramètres sont différentes

```
int   meth1()      { ... }  
int   meth1(int p) { ... }  
float meth1(float p) { ... }  
float meth1(int p)  { ... } // Impossible !
```



Le type de retour ne fait pas partie de la signature

Constructeur

- **Même nom que la classe**
- Méthode spéciale appelée à l'**instanciation d'un objet** pour initialiser son état
- **Pas de valeur de retour**
- Si aucun constructeur n'existe, un **constructeur par défaut** (sans paramètre et qui ne fait rien) est défini
- Mot clef **new** (création et allocation mémoire)

```
class MyClass { // Déclaration de classe.  
    MyClass(int p) { ... }  
    ...  
}  
...  
MyClass myObj1; // Déclaration.  
// Création et allocation mémoire : instanciation.  
myObj1 = new MyClass(5);
```

Constructeurs multiples

- Surcharge de constructeurs
- Mot clef `this`


```
class MyClass {  
    MyClass() {  
        this(5, new Objet( ... )); // Qu'en première instruction d'un constructeur.  
    }  
    MyClass(int p) {  
        this(p, new Objet( ... ));  
    }  
    MyClass(int p, Objet a) {  
        instruction1  
        ...  
    }  
    ...  
}
```

Encapsulation

- Concerne : classe, constructeur, attribut et méthode
- Accessibilité :
 - `public` : accessible de partout et sans aucune restriction (la classe principale doit nécessairement être publique)
 - `protected` : accessible aux classes du paquetage et à ses classes filles
 - `private` : accessible uniquement au sein de sa classe
 - (par défaut) : accessible aux classes du paquetage
- Accesseurs/mutateurs :

```
private int cpt;  
public int getCpt(){ return this.cpt; }  
public void setCpt (int p){ this.cpt = p; }
```

Héritage

- Héritage simple (une seule super-classe et unidirectionnelle)
- Mot clef `extends`
- Surcharge
- Redéfinition : 
- Même signature de méthode (ne peut pas être moins accessible)
- Réécriture du code
- Redéfinition d'attributs (les attributs redéfinis sont ajoutés)

```
public class MySuperClass {  
    int meth1(int a) { instruction1 }  
}  
...  
public class MyClass extends MySuperClass {  
    int meth1(int a) { instruction2 }  
}
```


Méthode : super

- Permet de **réutiliser le code de la méthode de la super-classe** :

```
class MySuperClass {  
    meth1(int a) { instruction1 }  
}  
...  
class MyClass extends MySuperClass {  
    meth1(int a) {  
        ...  
        super.meth1(a);  
        ...  
    }  
}
```



Constructeur : super()

- Permet de **réutiliser le code du constructeur de la super-classe** :
 - Doit être la **première instruction**
 - Appel implicite `super()` (sans paramètre) par défaut
 - Pas compatible avec `this()`

```
class MySuperClass {  
    MySuperClass (int a) { instruction1 }  
}  
...  
class MyClass extends MySuperClass {  
    MyClass(int a) {  
        super(a - 1);  
        ...  
    }  
}
```

S'il y a au moins un constructeur explicite avec au moins un paramètre dans la super-classe : il y a désactivation du constructeur par défaut et un appel implicite de ce dernier dans la classe-fille générera une erreur

Classe Object / méthode main()

- **Object :**

- **Classe de plus haut niveau** dans la hiérarchie d'héritage
- Toute classe autre que **Object** possède une super-classe
- Toute classe hérite directement ou pas de **Object**
- Toute classe qui n'a pas de clause extends hérite de **Object**

- **main(...)** :

- Point de commencement d'exécution du code
- Au moins une par application (**classe principale**) :

```
public static void main(String[] args) {  
    // Le code va commencer par s'exécuter ici.  
}
```

Portée et variable locale

- **Portée** (d'une variable) :
 - Début : à partir de sa déclaration
 - Fin : la fin du bloc d'instructions dans lequel elle se trouve (ou celui du corps de la méthode pour un paramètre)
- **Variable locale** :
 - Déclarée dans une méthode ou un bloc d'une méthode
 - Durée de vie : sa portée
 - **Visible qu'à l'intérieur du bloc**
 - Pas de valeur par défaut

Destruction et garbage collector

- Destruction implicite en Java
- *Garbage collector* (ou ramasse-miettes):
 - Appel automatique :
 - Si plus aucune variable ne référence l'objet
 - Si le bloc dans lequel l'objet est défini se termine
 - Si l'objet a été affecté à `null`
 - Appel manuel :
 - Instruction : `System.gc()` ;
- Usage de **pseudo-destructeur** :
 - Classe utilisateur : `protected void finalize()`
 - Appelée juste avant le *garbage collector* ? Pas certain !

Pour être sûr que `finalize()` soit appelée, il faut appeler manuellement le *garbage collector*

Statique

- **Variable de classe :**

- Donnée commune à tous les objets d'une même classe
- Existe même s'il n'y a aucune instance de la classe :

```
public static int att1; // Définition.
```

```
...
```

```
MyClass.att1 = 3;
```

```
myObj.att1 = 3; // Fortement déconseillé !
```

- Constante de classe :

```
public static final int ATT1 = 3; // Définition et initialisation.
```

- **Méthode de classe :**

- Ne s'intéresse pas à un objet particulier :

```
public static int meth1() { ... }; // Définition.
```

```
...
```

```
MyClass.meth1();
```

```
myObj.meth1(); // Fortement déconseillé !
```

Incompatible avec this !

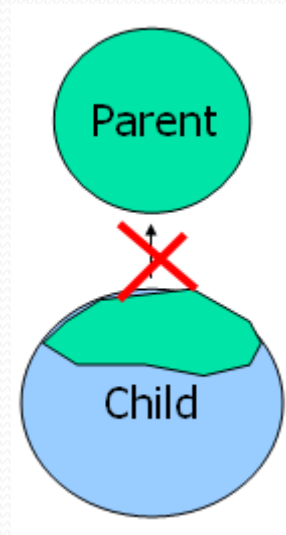
Mot clef final

- Constante
- Constante de classe
- Méthode (pour interdire toute redéfinition) :

```
public final int meth1() { ... }
```

- Classe (pour interdire tout héritage) :

```
public final class MyClass { ... }
```



A savoir

- Classe String :

```
String myString = "Hello!";  
myString += "How are you?";
```

- Classe File :

```
File myFile = new File("file.txt");
```

- Affichage (y compris types primitifs / références) :

```
System.out.println("a = " + a);
```



Bonnes pratiques

- Penser à l'initialisation pour éviter une erreur
- Penser à construire les objets avant de les utiliser
- Penser à l'utilisation de `break` dans un `switch`
- Attention à l'encapsulation
- Utiliser le mot clef `this` autant de fois que possible
- Pas de mot clef `then` (en relation avec un `if`)
- Pas d'utilisation de variable d'instance ni du mot clef `this` dans une méthode de classe
- Pas d'héritage multiple

API

Paquetages

Classes

Description
Attributs
Méthodes

**Java™ 2 Platform
Std. Ed. v1.4.2**

[All Classes](#)

Packages
[java.applet](#)
[java.awt](#)
[java.awt.color](#)
[java.awt.datatransfer](#)
[java.awt.dnd](#)
[java.awt.event](#)
[java.awt.font](#)
[java.awt.geom](#)
[java.awt.im](#)
[java.awt.im.spi](#)
[java.awt.image](#)
[java.awt.image.renderable](#)

All Classes
[ARG_IN](#)
[ARG_OUT](#)
[AWTError](#)
[AWTEvent](#)
[AWTEventListener](#)
[AWTEventListenerProxy](#)
[AWTEventMulticaster](#)
[AWTException](#)
[AWTKeyStroke](#)
[AWTMouseEvent](#)
[AbstractAction](#)
[AbstractBorder](#)
[AbstractButton](#)
[AbstractCellEditor](#)
[AbstractCollection](#)
[AbstractColorChooserPanel](#)
[AbstractDocument](#)
[AbstractDocument.AttributeContext](#)
[AbstractDocument.Content](#)
[AbstractDocument.ElementEdit](#)
[AbstractInterruptibleChannel](#)
[AbstractLayoutCache](#)
[AbstractLayoutCache.NodeDimensions](#)
[AbstractList](#)
[AbstractListModel](#)
[AbstractMap](#)
[AbstractMethodError](#)
[AbstractPreferences](#)
[AbstractSelectableChannel](#)
[AbstractSelectableKey](#)
[AbstractSelector](#)
[AbstractSequentialList](#)
[AbstractSet](#)
[AbstractSpinnerModel](#)
[AbstractTableModel](#)
[AbstractUndoableEdit](#)
[AbstractWriter](#)
[AccessControlContext](#)
[AccessControlException](#)
[AccessController](#)
[AccessDeniedException](#)
[Accessible](#)
[AccessibleAction](#)
[AccessibleBundle](#)
[AccessibleComponent](#)

Overview Package Class Use **Tree** **Deprecated** **Index** **Help**

PREV NEXT

[FRAMES](#) [NO FRAMES](#)

Java™ 2 Platform, Standard Edition, v 1.4.2

API Specification

This document is the API specification for the Java 2 Platform, Standard Edition, version 1.4.2.

See:
[Description](#)

Java 2 Platform Packages

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing <i>beans</i> -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.nio.channels	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
java.nio.channels.spi	Service-provider classes for the java.nio.channels package.
java.nio.charset	Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters.
java.nio.charset.spi	Service-provider classes for the java.nio.charset package.
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garbage-collection (DGC).
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server side of RMI.

Outils

- Éditeur :

- Eclipse : <http://www.eclipse.org>



- Ressources Java :

- API :

- <http://download.oracle.com/javase/1.5.0/docs/api>

- Convention de nommage :

- <http://java.sun.com/docs/codeconv/CodeConventions.pdf>

- Mots clefs réservés :

- http://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Liens

- Documents électroniques :

- <http://nicolas.baudru.perso.esil.univmed.fr/Enseignement/enseignement.html>

- Documents classiques :

- Livres :

- Claude Delannoy. *Programmer en Java 2ème édition.*

- Cours :

- Francis Jambon. *Programmation orientée application au langage Java.*

Crédits

Auteur

Mickaël Martin Nevot

mmartin.nevot@gmail.com



Carte de visite électronique

Relecteurs

Cours en ligne sur : www.mickael-martin-nevot.com

