Apprendre l'informatique avec Javascool à travers la programmation de petits jeux

Benoît Crespin

Dernière mise à jour : 29 février 2012

Version en ligne :
http://javascool.gforge.inria.fr/documents/crespin-etal.pdf

Table des matières

1 Préambule

1.1 Avertissement

Ce document est encore en développement, certaines parties sont en cours de rédaction. Toute remarque ou suggestion (ou correction des exercices !) est la bienvenue : `benoit.crespin@unilim.fr`

1.2 Qu'est-ce qu'un langage de programmation ?

Un langage de programmation peut être vu comme un moyen permettant à un programmeur humain de faire exécuter à un ordinateur des tâches complexes. Il s'agit donc, un peu comme une langue étrangère, d'un ensemble d'éléments (vocabulaire, syntaxe, grammaire, etc.) implantés dans l'ordinateur et que le programmeur doit apprendre à maîtriser pour arriver à ses fins. Accompagnant l'évolution matérielle des ordinateurs depuis les années 1950, il existe à présent des milliers de langages de programmation [1], en général créés pour répondre à des besoins précis et bénéficiant d'une audience internationale plus ou moins grande. On peut citer comme principaux langages largement utilisés actuellement : Java, C/C++, Python et Visual Basic.

1.2.1 Langages compilés vs interprétés

Une distinction usuelle entre les différents langages de programmation concerne l'exécution des programmes par l'ordinateur. Un programme écrit dans un langage **compilé** est transformé par un logiciel spécifique, le **compilateur**, en un ensemble d'instructions directement exécutables par la machine. Ce type de langage requiert donc une **phase de compilation** pour réaliser cette transformation avant de pouvoir exécuter le programme. A l'inverse, avec un langage **interprété** le programme est exécuté directement par la machine en transformant les instructions **à la volée** : aucune phase de compilation n'est nécessaire.

Cette différence essentielle fait que les langages compilés sont généralement préférés par les informaticiens, pour plusieurs raisons :

- lors de la phase de compilation, il est possible de détecter certaines erreurs dans le programme afin d'éviter l'exécution d'un programme erroné. Tant que ces erreurs ne sont pas corrigées l'exécution du programme est impossible.
- Si la compilation réussit, les instructions seront exécutées plus rapidement par la machine que le programme équivalent écrit dans un langage interprété.

Pourtant les choses ne sont pas forcément aussi simples, et de nombreux autres aspects sont à considérer concernant les différences entre langages. Le langage Java par exemple, que nous allons utiliser avec Javascool, peut être vu à la fois comme un langage compilé et interprété. Pour plus de détails sur le sujet consulter : `http://fr.wikipedia.org/wiki/Java_(langage)`

1.2.2 Javascool

Comme on peut le lire sur son site Web [2], **Java's Cool (alias Javascool) est un logiciel conçu pour l'apprentissage des bases de la programmation**. Il se base sur le logiciel Java, qui est le seul élément qui doit être installé sur la machine pour permettre à Javascool de fonctionner. Les avantages de Javascool pour l'apprentissage de l'algorithmique et de la programmation sont multiples :

- L'utilisation de la syntaxe de Java, elle-même proche de la syntaxe de nombreux langages tels que le C/C++, permet de donner aux élèves des bases qu'ils retrouveront fréquemment s'ils poursuivent des études dans le domaine Informatique.
- Cette syntaxe est également assez proche des *méta-langages* (ou "langages algorithmiques") traditionnellement utilisés pour enseigner l'algorithmique, par conséquent la traduction en langage Javascool d'un algorithme exprimé en méta-langage est plus simple qu'avec d'autres langages, même si ce document montre justement qu'il y a quelques pièges à éviter...
- Java étant multi-plateformes, Javascool peut fonctionner indifféremment sous Linux, Windows ou Mac

1. `http://en.wikipedia.org/wiki/Programming_language`
2. `http://javascool.gforge.inria.fr/`

- C'est un logiciel "tout-en-un", intégrant à la fois un éditeur pour écrire ses programmes et une fenêtre pour les exécuter
- Javascool est utilisé par de nombreux enseignants, et commence à proposer un nombre important de ressources pédagogiques accessibles sur son site web

On peut néanmoins lui trouver quelques inconvénients :

- Les messages d'erreurs ne sont pas toujours "parlants", et le logiciel ne propose pas réellement de fonctionnalités pour faciliter l'élimination des erreurs (ce qui est parfois vu comme un avantage étant donné que cela oblige à se poser plus de questions)
- Javascool contient des fonctionnalités permettant de simplifier l'apprentissage de la programmation en "cachant" certaines fonctionnalités du langage Java ; cela peut entraîner une confusion chez certains élèves lorsqu'ils sont confrontés plus tard dans leur cursus à des langages réellement utilisés dans le monde industriel ou académique.
- Java reste un langage relativement peu rapide par rapport à d'autres, ce qui n'empêche quand même pas de l'utiliser même pour résoudre des problèmes complexes.

Il n'en reste pas moins que tout langage de programmation a ses avantages et inconvénients, et le choix n'est pas facile parmi les milliers de langages utilisés actuellement sur la planète. Enseigner comment traduire un algorithme dans un langage relativement simple comme Javascool permet de toute à façon de préparer les élèves à apprendre plus tard comment passer facilement d'un langage de programmation à un autre et comment choisir le langage le plus approprié selon le type d'application envisagé, comme tout bon programmeur qui se respecte.

On peut aussi considérer Javascool comme un tremplin vers des plateformes de programmation professionnelles comme Eclipse [3] ou Netbeans [4].

1.3 Objectifs et structure de ce support

Ce document a pour objectif principal de former les enseignants pour leur permettre d'aider ensuite leurs élèves à utiliser Javascool, ce qui inclut :

- des notions basiques et avancées liées à la traduction d'un algorithme en "langage Javascool" (*ie Java simplifié)*
- *une description des outils de débogage et de mise au point des programmes*

On trouvera notamment ici des explications complémentaires sur certaines activités de base déjà proposées dans Javascool, d'autres idées d'activités à mener avec les élèves, et, dans certains cas, des notions liées à d'autres langages de programmation pour ceux qui souhaitent éventuellement aborder l'algorithmique sous d'autres formes. En revanche, de nombreuses activités proposées dans Javascool ne seront que peu abordées ici, mais elles sont en général suffisamment bien documentées.

La structure de ce document est la suivante :

- La section 2 donne les éléments de base permettant la prise en main de Javascool, pour aider à comprendre les grands principes et comment démarrer.
- La section 3 est une sorte de guide pour découvrir les ingrédients des algorithmes, c'est-à-dire la base qui permet de passer des algorithmes à la programmation.
- La section 4 est une analyse didactique des erreurs de compilation et d'exécution les plus courantes, c'est donc un contenu essentiel pour ne plus être "victime" des bugs.
- Les sections 5 et 6 permettent d'approfondir les éléments essentiels de programmation correspondant au programme ISN de l'enseignement de spécialité de TS.
- Enfin les sections 7 et 8 offrent la possibilité d'aller plus loin en explorant les structures usuelles de données de l'informatique au delà du programme ISN.

1.4 Fil rouge : programmer des jeux

Les livres ou cours de programmation illustrent souvent les notions abordées par une application complète, permettant de montrer l'intérêt de ces notions dans un cadre concret. Ici on tentera de mettre en pratique ces différents éléments à travers le développement de petits jeux. Ce choix d'une application ludique a de nombreux avantages : l'expérience du développement d'un jeu est souvent très motivante et permet de faire passer plus facilement concepts avancés et liens avec des domaines tels que

3. `http://www.eclipse.org/`
4. `http://netbeans.org/`

les mathématiques ou la physique. Programmer un jeu de type “casse-briques” par exemple implique de comprendre les notions de vélocité et de collision. On peut retrouver en fait dans les jeux un panorama complet des notions abordées en informatique :

- Algorithmique classique (comment fonctionne le jeu, que se passe-t-il si le joueur entre telle ou telle valeur)
- Structures de données (comment stocker et accéder efficacement aux données du jeu telles que les scores ou les positions des joueurs)
- Affichage, graphismes 2D ou 3D (comment afficher et éventuellement dessiner des données à l'écran)
- Son (comment générer des sons au cours du jeu)
- Aide à la décision (comment programmer un adversaire performant face au joueur humain)
- Réseaux (comment faire communiquer des joueurs au travers du jeu)
- etc.

1.5 Versions de Javascool

Les exemples donnés dans ce document utilisent la version 4 de Javascool, téléchargée le 29/02/2012 (5.9 Mo). Il est possible que certains ne fonctionnent pas avec des versions différentes (notamment des versions antérieures), ou se comportent différemment par rapport aux résultats attendus.

1.6 Pré-requis : des notions d'anglais

Cela peut paraître étonnant, mais un aspect parfois rebutant pour les élèves par rapport à la pratique de la programmation tient à ce que ce domaine est fortement lié, pour des raisons historiques, à la langue anglaise. Les mots-clefs employés par Javascool, comme dans la plupart des langages de programmation, sont en anglais et sont évidemment plus faciles à comprendre pour les élèves ayant une bonne maîtrise de cette langue. Par exemple, le concept de “programme principal” est immédiatement compréhensible pour qui sait que le terme anglo-saxon “main” se traduit par “principal” en français.

De même, les messages d'erreur sont parfois en anglais et peuvent apparaître très mystérieux pour les élèves. On s'efforcera dans ces documents d'apporter des précisions sur le sens des messages rencontrés le plus fréquemment mais il est important de faire passer auprès des élèves que la maîtrise de l'anglais, en informatique comme pour toute discipline scientifique, devient de plus en plus nécessaire.

2 Principes de base

2.1 La philosophie Javascool

Javascool est plus qu'un langage de programmation : comme mentionné plus haut c'est un logiciel complet intégrant un éditeur pour écrire ses programmes et une fenêtre pour les exécuter. De façon plus générale, Javascool se veut un support permettant d'apprendre à la fois l'algorithmique et sa traduction immédiate, en permettant au professeur de faciliter la vie des élèves grâce à la possibilité d'utiliser des “proglets” existants ou qu'il programmera lui-même. Une “proglet” est un ensemble de fonctionnalités écrites en Javascool et de documents expliquant comment les utiliser dans le cadre d'une activité, en passant par une interface graphique ou en écrivant du code Javascool [5].

Ainsi, on peut éventuellement proposer à l'élève d'utiliser une fonctionnalité (par exemple la fonction **load** permettant de charger une image en mémoire) en lui cachant les détails concernant la façon dont cette fonctionnalité est programmée.

Cette “philosophie” Javascool permet de simplifier la prise en main par l'élève de certains aspects techniques pour qu'il puisse se concentrer sur les aspects purement algorithmiques. Pour reprendre l'exemple de la manipulation d'images, l'élève pourra ainsi manipuler une image comme un tableau à deux dimensions dont les cases représentent les intensités des pixels, sans avoir à comprendre comment sont réellement codées les images au format JPG ou autre.

Nous nous concentrons dans ce document sur les aspects techniques de Javascool : la syntaxe du langage (similaire à celle du langage Java), les fonctionnalités du logiciel, les causes les plus fréquentes d'erreurs de traduction entre version algorithmique et version Javascool d'un programme. Les aspects plus

5. Les “educlets” sont des “proglets” restreintes à l'utilisation d'une interface graphique

FIGURE 1 – Fenêtre de lancement de Javascool

abstraits orientés vers la conception des algorithmes eux-mêmes, écrits dans un langage algorithmique, ne sont pas traités dans ce document.

2.2 Installation et lancement

Le seul prérequis est d'avoir le logiciel Java installé sur la machine, en suivant les instructions données sur : `http://www.java.com/fr/`

Sur une machine connectée à Internet, la méthode la plus simple consiste à :

- ouvrir dans un navigateur la page `http://javascool.gforge.inria.fr/`
- cliquer sur "Lancer" dans la barre horizontale située sous le bandeau Javascool
- selon la configuration du navigateur, cette action entraînera le lancement immédiat de Javascool ou la possibilité d'enregistrer le fichier **javascool-proglets.jar** sur la machine
- le logiciel peut ensuite être lancé en général par un double-clic sur ce fichier. Si cela ne fonctionne pas, il peut être nécessaire de le lancer par un clic-droit sur le fichier puis "Ouvrir avec", puis en choisissant "Sun Java 6 Runtime" ou une commande équivalente
- sous Linux, le logiciel peut aussi être lancé en tapant dans un terminal la commande **java -jar javascool-proglets.jar**

Sur une machine non-connectée, ou pour éviter que le fichier **javascool-proglets.jar** ne soit téléchargé sur chaque machine, il vaut mieux mettre ce fichier dans un dossier partagé accessible à toutes les machines. Alternativement, ce fichier peut aussi être sauvegardé par le professeur sur une clef USB et copié sur chaque machine en début de séance.

La figure 1 montre ce que vous devez voir au lancement du logiciel.

2.3 Pour commencer

A partir de la fenêtre de lancement, cliquer sur "abcdAlgos". Vous devriez alors voir l'éditeur tel que le montre la figure 2. Cette description des différents composants peut aussi être obtenue en cliquant sur "Mémo". Il est également utile de consulter cette page pour revoir en un coup d'œil les commandes et les raccourcis-clavier utilisables dans l'éditeur.

Voir ensuite l'onglet "Aide de la proglet" pour découvrir les cinq ingrédients de base que nous allons détailler dans la section suivante.

2.4 Éléments de syntaxe élémentaire

Par la volonté des concepteurs de Java et de Javascool, la syntaxe d'un programme Javascool est proche de celle d'un programme écrit en langage algorithmique sur papier avant de passer sur machine. On peut parler de *traduction*, avec le même objectif que la traduction du français vers une langue étrangère. Le langage Javascool a pour traits principaux :

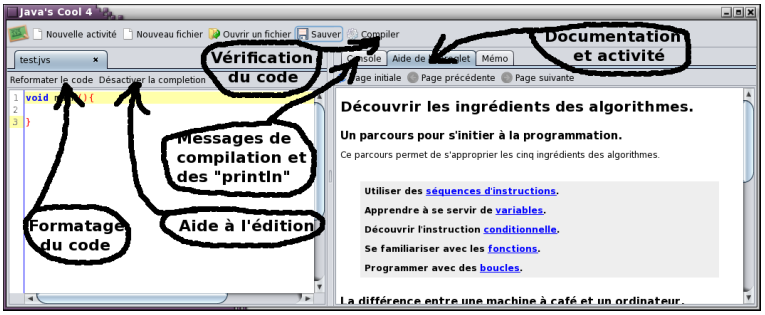


FIGURE 2 – Présentation des éléments de l'interface Javascool

- l'emploi fréquent de points-virgules en fin de ligne
- peu de contraintes sur les identificateurs (noms de variables ou de fonctions), qui doivent être constitués de chiffres et de lettres (sans accents), et peuvent contenir le caractère _ mais aucun autre symbole [6]. Il est recommandé de choisir des identificateurs "parlants" (par exemple, une fonction qui calcule le carré d'un nombre devrait s'appeler **carre** plutôt que **toto**), mais attention à ne pas utiliser les mots-clefs du langage que vous allez découvrir ensuite
- les espaces ont en général peu d'importance, il ne faut pas hésiter à "aérer" ses programmes en laissant des espaces et des lignes vides plutôt qu'essayer d'écrire le programme le plus court du monde à chaque fois
- les lignes commençant par // sont des commentaires que l'on peut ajouter dans le programme pour bien expliciter son fonctionnement
- cliquer sur l'onglet "Mémo" pour voir ou revoir les instructions de base du langage

3 La programmation impérative avec Javascool

Nous reprenons la structure du parcours proposé dans Javascool pour plus de simplicité, en explicitant certaines notions et en proposant des exercices supplémentaires plus particulièrement orientés vers la traduction en langage Javascool d'algorithmes existants. Vous rencontrerez parfois des erreurs en essayant de résoudre ces exercices ; vous pouvez vous reporter à la partie 4 où sont abordés les principaux cas d'erreur. A noter : le bouton "Insertion" de l'éditeur permet d'ajouter dans le code certaines instructions sans avoir à les taper soi-même caractère par caractère, ce qui réduit les risques d'erreurs.

3.1 HelloWorld

En cliquant sur "séquences d'instructions" on découvre le tutoriel "HelloWorld", qui montre le programme le plus simple que l'on peut écrire avec Javascool :

```
void main() {
    println("Hello World !");
}
```

Ce programme correspond, en langage algorithmique, à :

```
Algorithme principal
Début
  Afficher("Hello World !")
Fin
```

Vous pouvez copier le code de ce programme Javascool dans l'éditeur à gauche (en le retapant entièrement ou en utilisant Ctrl-C puis Ctrl-V), puis cliquer sur "Compiler", ce qui nécessite de sauvegarder votre premier fichier Javascool. Une fois l'emplacement du fichier choisi, la *console* doit afficher "Compilation réussie !".

6. Et par convention, ces identificateurs commencent en général par une minuscule.

Reste enfin à exécuter le programme (en cliquant sur “Exécuter” donc), pour voir s'afficher le texte voulu dans la console.

3.1.1 Exercices du tutoriel

Comme énoncé dans le tutoriel, modifier ce programme pour changer le texte qui s'affiche et ajouter de nouvelles phrases qui s'afficheront les unes après les autres.

3.1.2 Exercice : Avec des étoiles

Écrire un programme qui affiche le résultat ci-dessous. Pour le sauvegarder sous un nom différent du précédent (et ainsi éviter d'écraser le contenu de celui-ci), cliquer sur “Sauver” avant de le compiler puis de l'exécuter.

```
*****************
*               *
* Hello World ! *
*               *
*****************
```

3.2 Variables

Revenir sur le “Parcours d'initiation”, cliquer sur “Page initiale” puis “variables”. On trouve ici la traduction de l'algorithme :

```
Algorithme principal
Variable texte: chaîne de caractères
Début
    Afficher("Bonjour, quel est ton nom ?")
    Saisir(texte)
    Afficher("Enchanté ")
    Afficher(texte)
    Afficher(", et ... à bientôt !")
Fin
```

qui s'écrit en langage Javascool :

```
void main() {
    println("Bonjour, quel est ton nom ?");
    String texte = readString();
    println("Enchanté " + texte + ", et ... à bientôt !");
}
```

La “variable” dont il est question dans ce tutoriel s'appelle ici **texte**.

Comme on le voit en compilant puis en exécutant ce programme, une petite fenêtre “Entrée au clavier” s'ouvre pour permettre la saisie de **texte**, qui est utilisée ensuite pour l'affichage. Cet affichage avec la fonction **println** implique un retour à la ligne. L'opérateur “+”, quand il s'applique à un affichage (une ou plusieurs variables ou bien une phrase entourée par des guillemets) sert à créer une seule chaîne de caractères à partir de ces données : on parle alors de *concaténation*.

Alternativement, la fonction **print**, sans retour à la ligne, peut être utilisée pour obtenir le même résultat sans nécessiter de concaténation :

```
void main() {
    println("Bonjour, quel est ton nom ?");
    String texte = readString();
    print("Enchanté ");
    print(texte);
    println(", et ... à bientôt !");
}
```

3.2.1 Exercice : Afficher une ligne vide

Modifier le programme précédent pour faire afficher une ligne vide juste avant la dernière ligne

3.2.2 Exercices du tutoriel

Réaliser les exercices proposés dans le tutoriel pour bien comprendre qu'il est possible de donner à **texte** un nom différent, et qu'il existe des types de variables autres que *string* (ou *chaîne de caractères* en langage algorithmique) : les types numériques *int* et *double*

3.2.3 Exercice : Autour du cercle

Avec les types numériques, on peut utiliser les opérateurs arithmétiques usuels notés $+$, $-$, $*$ et $/$. En utilisant uniquement ces opérateurs, écrire le programme équivalent à l'algorithme suivant :

```
Algorithme principal
Variables rayon, circ, aire: nombres réels
Constante Pi: nombre réel <- 3.14159
Début
 Afficher("Entrer le rayon: ")
 Saisir(rayon)
 circ <- 2 * rayon * Pi
 aire <- Pi * rayon^2
 Afficher("Circonférence du cercle de rayon ", rayon, ": ")
 Afficher(circ)
 Afficher("Aire: ")
 Afficher(aire)
Fin
```

Attention, pour calculer le carré d'un nombre vous devez le multiplier par lui-même.

3.2.4 Exercice : Pow

Le carré d'un nombre peut aussi être obtenu par la fonction **pow**, décrite dans l'onglet "Mémo". Modifier le programme précédent pour l'utiliser.

3.2.5 Exercice : Racines

Écrire un nouveau programme se comportant comme ci-dessous, en utilisant la fonction **sqrt** qui calcule la racine carrée d'un nombre :

```
Entrer les coeffs d'une équation du 2nd degré
(le déterminant doit être positif)
Entrer a: (on entre -1)
Entrer b: (on entre 1)
Entrer c: (on entre 2)
Déterminant: 9.0
Racine x1: -1.0
Racine x2: 2.0
```

On ne vérifiera pas si le déterminant est réellement positif. Que se passe-t-il si ce n'est pas le cas ?

3.3 Instructions conditionnelles

Le début de ce tutoriel résume la traduction des "SI ... ALORS ... SINON" du langage algorithmique par des **if ... then ... else** en langage Javascool. Ainsi l'algorithme :

```
Algorithme principal
Variable temperature: réel
Début
 Afficher("Entrer la température: ")
 Saisir(temperature)
 Si (temperature < 15) Alors
  Afficher("allumer chauffage")
 FinSi
 Sinon
  Si (temperature > 100) Alors
   Afficher("appeler les pompiers")
  FinSi
  Sinon
   Afficher("ne rien faire")
  FinSinon
 FinSinon
Fin
```

se traduira en Javascool par :

```
void main() {
 println("Entrer la température: ");
 double temperature = readFloat();
 if(temperature < 15) {
  println("allumer chauffage");
 }
 else if(temperature > 100) {
  println("appeler les pompiers");
  }
 else {
  println("ne rien faire");
 }
}
```

On peut vérifier en compilant puis en exécutant ce nouveau programme qu'il change d'affichage en fonction de la valeur saisie pour la température.

Le tutoriel permet ensuite de comprendre comment traduire les conditions d'égalité entre nombres ou entre chaînes de caractères, et également comment on peut combiner plusieurs conditions pour en obtenir une seule. Par exemple l'algorithme :

```
Algorithme principal
Variable temperature: réel
Début
 Afficher("Entrer la température: ")
 Saisir(temperature)
 Si ((temperature > 15) ET (temperature < 21)) Alors
  Afficher("Température idéale !")
 FinSi
Fin
```

se traduira par :

```
void main() {
 println("Entrer la température: ");
 double temperature = readFloat();
 if((temperature > 15) && (temperature < 21))  {
  println("Température idéale !");
 }
}
```

Il est important de remarquer que les conditions d'égalité stricte ont des formes particulières, d'abord l'égalité entre deux nombres : le test écrit en langage algorithmique "Si (x = 2) Alors ..." se traduira par **if (x==2) ...**. Cette double égalité est souvent cause d'erreurs dans d'autres langages proches de la syntaxe de Java, notamment C et C++. Pour l'égalité entre chaînes de caractères, on traduira "Si (texte="Bonjour") Alors ..." par **if equal(texte,"Bonjour") ...**.

Un autre élément remarquable dans les exemples ci-dessus tient à l'utilisation des espaces en début de ligne, permettant de décaler les instructions quand on entre dans un bloc correspondant à un "Alors" ou un "Sinon", ce qui donne tout de suite une vision claire du code bien utile pour le débogage. Ce point est parfois fastidieux à respecter, heureusement il est possible de laisser Javascool faire le travail en cliquant sur le bouton "Reformater le code". N'hésitez pas à l'utiliser !

La dernière partie du tutoriel aborde la notion de variable *booléenne*, qui ne peut prendre comme valeur que **true** ou **false**. Un lien cliquable après le dernier exercice du tutoriel permet d'en savoir plus.

Enfin, on peut noter qu'il manque ici la définition de la structure **switch** (équivalent à *SELON* en algorithmique), qui permet de ne pas avoir à enchaîner les tests, comme dans l'exemple suivant :

```
void main() {
 println("Entrez un numéro de mois:");
 int n = readInt();
 switch (n) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
   println("Mois à 31 jours");
   break;
  case 4:
  case 6:
  case 9:
  case 11:
   println("Mois à 30 jours");
   break;
  case 2: println("Mois à 28 ou 29 jours");
   break;
  default: println("Erreur de saisie");
 }
}
```

Le mot-clef **break** permet de séparer les cas, comme ici pour traiter de façon unique les mois à 30 ou 31 jours.

3.3.1 Exercices du tutoriel

Réaliser les exercices proposés dans le tutoriel pour bien comprendre les mécanismes décrits ci-dessus.

3.3.2 Exercice : Toujours le cercle

Reprendre le programme 3.2.3 pour faire afficher un message indiquant si la circonférence est inférieure, égale ou supérieure à l'aire.

3.3.3 Exercice : Ordre de 3 nombres

Traduire en Javascool l'algorithme suivant :

```
Algorithme principal
Variables a, b, c: entiers
```

```
Début
 Afficher("Entrer a, b et c")
 Saisir(a,b,c)
 Si (a < b < c) Alors Afficher("a < b < c")
 FinSi
 Sinon Si (a < c < b) Alors Afficher("a < c < b")
       FinSi
       Sinon Si (b < a < c) Alors ...
 ...
Fin
```

3.3.4 Tester les entrées

Améliorer le programme 3.2.4 pour vérifier que le rayon est bien un nombre strictement positif. Améliorer ensuite le programme 3.2.5 pour vérifier que le déterminant n'est pas négatif, et pour afficher le résultat adéquat si le déterminant est nul.

3.4 Fonctions

Le tutoriel "Fonctions" de Javascool permet de bien comprendre la notion de structuration d'un programme en différentes fonctions, pour le rendre plus lisible et plus facilement ré-utilisable. Il est notamment important de noter que nous avons en fait déjà utilisé des fonctions prédéfinies de Java : **println**, **pow**, etc.

Nous insistons ici encore une fois sur la traduction d'une fonction écrite en langage algorithmique, en reprenant la fonction **int abs(int x)** décrite au début du tutoriel. L'algorithme suivant :

```
Fonction abs
Entrée: x: entier
Sortie: entier
Début
 Si (x > 0) Alors Retourner x
 Sinon Retourner -x
Fin_abs

Procédure affichage_abs
Entrée: x: entier
Début
 Afficher("Valeur absolue: ", abs(x))
Fin_affichage_abs

Algorithme principal
Variable y: entier
Début
 Afficher("Entrer y: ")
 Saisir(y)
 affichage_abs(y)
Fin
```

se traduit en Javascool par :

```
int abs(int x) {
   if (x > 0) {
      return  x;
   } else {
      return - x;
   }
}
```

```
void affichage_abs(int x) {
   println("Valeur absolue: " + abs(x));
}
void main() {
   println("Entrer y: ");
   int y = readInteger();
   affichage_abs(y);
}
```

On voit dans cet exemple plusieurs éléments remarquables :

- une même variable x peut être utilisée dans deux fonctions différentes, ce sont deux variables différentes qui n'ont aucun rapport entre elles pour l'ordinateur : on dit que ces variables sont *locales* à la fonction qui les déclare
- il peut y avoir zéro, une ou plusieurs entrées ; en revanche il y a zéro ou une sortie au maximum
- les variables d'entrée sont placées entre parenthèses après le nom de la fonction, en précisant leur type
- on parle généralement en algorithmique de *procédure* lorsqu'une fonction ne retourne pas de valeur en sortie, ce qui se traduit par le mot-clef **void** placé devant le nom de la fonction
- comme indiqué dans le tutoriel, l'instruction **void main()** traduit par conséquent le fait que l'algorithme principal ne prend aucune entrée, et qu'il ne retourne aucune valeur en sortie

Supposons maintenant que nous avons écrit une fonction **int soustrait(int a, int b)** qui retourne la valeur $a - b$. Pour l'utiliser, nous devons faire attention à l'ordre dans lequel les données sont transmises en entrée de la fonction. Par exemple, dans le programme ci-dessous on comprend aisément qu'il n'est pas équivalent d'écrire **soustrait(x1, x2)** ou **soustrait(x2, x1)** :

```
int soustrait(int a, int b) {
   return  a - b;
}
void main() {
   println("Entrer x1: ");
   int x1 = readInteger();
   println("Entrer x2: ");
   int x2 = readInteger();
   println("Résultat: " + soustrait(x1, x2));
}
```

3.4.1 Exercices du tutoriel

Réaliser les exercices proposés dans le tutoriel pour bien comprendre les mécanismes décrits ci-dessus. On notera que pour la fonction "maximum de trois entiers" **int max(int x, int y, int z)**, on peut s'inspirer de l'exercice 3.3.3.

3.4.2 Exercice : Ordre de 3 nombres (bis)

Traduire en Javascool l'algorithme ci-dessous, qui est une ré-écriture du programme 3.3.3 avec une fonction.

```
Fonction ordre
Entrée: x,y,z: entiers
Sortie: entier
Début
 Si (x < y < z) Alors retourner 1
 FinSi
 Sinon Si (x < z < y) Alors retourner 2
       FinSi
       Sinon ...
Fin_ordre
```

```
Algorithme principal
Variables a, b, c, d: entiers
Début
 Afficher("Entrer a, b et c")
 Saisir(a,b,c)
 d <- ordre(a,b,c)
 Si (d = 1) Alors Afficher("a < b < c")
 FinSi
 Sinon Si (d = 2) Alors Afficher("a < c < b")
       FinSi
       Sinon Si ...
 ...
Fin
```

3.4.3 Exercice : Réécrire des algorithmes en utilisant des fonctions

Sur le même principe, reprendre les programmes vus à la section 3.3.4 et séparer les saisies de données au clavier (qui resteront dans le programme principal) des différents tests et calculs (qui seront placés dans des fonctions avec les entrées/sorties appropriées)

3.5 Boucles

Ce tutoriel montre deux formes de boucles, la forme **while** équivalente en langage algorithmique à **tant que**, et la forme **for** équivalente à **Pour** (soit une traduction littérale).

La première forme permet, à partir de l'algorithme :

```
Algorithme principal
Variable n: entier
Début
    n <- 0
    Tant que (n < 10) Faire
    Début
        Afficher ("Hello World !")
        n <- n + 1
    FinTantQue
Fin
```

d'obtenir en JavaScool :

```
void main() {
    int n = 0;
    while( n < 10) {
        println("Hello World !");
        n = n + 1;
    }
}
```

On voit que l'équivalence est réellement immédiate : si le principe algorithmique de la boucle **Tant que** est acquis, alors son application en Javascool est très simple.

Les choses sont un peu plus compliquées en ce qui concerne la boucle **Pour**. Ici, la forme algorithmique utilisant une boucle **Pour** équivalente à l'algorithme précédent sera :

```
Algorithme principal
Variable n: entier
Début
    Pour n de 0 à 9 (par pas de 1) Faire
    Début
```

```
        Afficher ("Hello World !")
    FinPour
Fin
```

ce qui se traduit en Javascool par :

```
void main() {
    for(int n = 0; n < 10; n = n + 1) {
       println("Hello World !");
    }
}
```

On voit ici que l'équivalence n'est plus aussi immédiate : la boucle **for** est en fait une forme de boucle permettant de compacter la forme **while**, mais on ne retrouve pas les notions de :

- initialisation automatique : en algorithmique le compteur n démarre à 0 par définition, alors qu'en Javascool l'instruction **int n=0** placée à l'intérieur de la boucle **for** est obligatoire
- comptage automatique : en algorithmique la boucle **Pour** autorise à définir le pas (de 1 en 1, 2 en 2, etc.), mais ceci nécessite en Javascool une instruction explicite **n = n + 1** pour modifier la valeur du compteur
- fin de boucle automatique : on considère que la répétition de la boucle **Pour** s'arrête lorsque le compteur dépasse la valeur définie comme borne d'arrêt (fixée à 9 dans l'exemple ci-dessus). Par contre, la forme **for** oblige à écrire un test permettant de vérifier que le compteur n'a pas atteint cette valeur. Ici on aurait aussi pu écrire `n <= 9` à la place de `n < 10`

Une boucle **for** est en fait définie de façon générale par 3 types d'instructions, placées entre parenthèses et séparées par des points-virgules :

```
for(initialisation; test_de_fin; increment_avant_de_recommencer) {
    corps_de_la_boucle
}
```

Comme l'écrit l'auteur du tutoriel, la forme **for** "est plus concise... mais ne change rien sur le fond". Elle a en effet été introduite dans les langages de programmation plutôt comme un moyen d'écrire plus vite du code. Il est toujours possible de ré-écrire avec la forme **while** un programme écrit initialement avec **for**, et inversement.

Néanmoins, la boucle **for** étant souvent utilisée par les programmeurs, il est utile de s'y familiariser pour comprendre comment fonctionnent des programmes existants. On peut aussi mentionner l'incrémentation **n++**, équivalente en Javascool à **n=n+1**. Ceci n'est pas spécifique aux boucles mais on trouvera souvent le programme précédent plutôt écrit sous la forme :

```
void main() {
    for(int n = 0; n < 10; n++) {
       println("Hello World !");
    }
}
```

3.5.1 Exercices du tutoriel

Réaliser les exercices proposés dans le tutoriel, d'abord ceux consistant à modifier le premier programme ci-dessus, pour lesquels on tiendra compte de la dernière remarque du tutoriel indiquant comment faire pour arrêter un programme qui boucle indéfiniment (cette notion de boucle infinie sera revue dans la section 4). La deuxième série d'exercice peut être réalisée d'abord sur papier en langage algorithmique, puis traduite en Javascool ensuite. Néanmoins c'est tout l'intérêt de l'outil informatique de pouvoir permettre à l'élève de parfois mieux comprendre un concept en le programmant directement : par exemple pour l'exercice du nombre d'or, ou celui des décimales de π, il peut être intéressant de voir à l'écran le résultat du programme pour éventuellement le corriger si on voit que le résultat n'est pas celui attendu.

3.5.2 Exercice : Vérifier les entrées

Les boucles sont très utiles pour vérifier les entrées d'un programme, et faire en sorte qu'elles soient correctes. Traduire par exemple l'algorithme ci-dessous :

```
Algorithme principal
Variables rayon, circ, aire: nombres réels
Constante Pi: nombre réel <- 3.14159
Début
 Afficher("Entrer le rayon: ")
 Saisir(rayon)
 Tant que (rayon < 0) Faire
 Début
    Afficher("Erreur, rayon négatif. Entrer le rayon: ")
    Saisir(rayon)
 FinTantQue
 circ <- 2 * rayon * Pi
 aire <- Pi * rayon^2
 Afficher("Circonférence du cercle de rayon ", rayon, ": ")
 Afficher(circ)
 Afficher("Aire: ")
 Afficher(aire)
Fin
```

En suivant le même principe, modifier le programme écrit pour l'exercice 3.2.5 de façon à obliger l'utilisateur à rentrer des coefficients a, b et c donnant un déterminant positif.

3.5.3 Exercice : Encore des étoiles

Modifier le programme principal **main** de l'exercice 3.1.2 de façon à afficher les lignes avec des étoiles ou les espaces vides en utilisant des boucles **for**. Modifier ensuite ce programme en créant une fonction d'affichage d'une ligne d'étoiles, une fonction d'affichage d'espaces vides, puis en faisant appel à cette fonction pour afficher une ligne au-dessus et en-dessous de la phrase "Hello World".

3.5.4 Exercice : Fibonacci

Créer différents programmes permettant :
- d'afficher les 30 premiers termes de la suite de Fibonacci
- d'afficher les termes de la suite de Fibonacci inférieurs à 100
- d'afficher les termes de la suite de Fibonacci compris entre 100 et 200

Pour la dernière question on doit combiner une boucle (pour calculer chaque terme), qui elle-même contiendra un test (pour décider si le terme doit être affiché). On reviendra en section 4 sur les problèmes liés à l'imbrication de tests dans les boucles, de boucles à l'intérieur de tests, de boucles à l'intérieur de boucles, etc.

3.6 Fil rouge : deviner un nombre

Dans le tutoriel sur les boucles, dernière partie du parcours d'initiation, on trouve la phrase indiquant qu'il est dorénavant possible de "programmer efficacement tous les algorithmes possibles". Si la notion d'efficacité en informatique est difficile à définir, il est néanmoins vrai que toutes ces briques de base permettent de commencer à mettre au point des programmes un peu plus ambitieux que les exercices vus jusqu'ici.

A titre d'exemple, voici le code en Javascool permettant de jouer au jeu du nombre-mystère : l'ordinateur choisit un nombre au hasard entre 1 et 1000, et le joueur cherche à le retrouver en faisant des propositions de nombres auxquelles l'ordinateur répond en indiquant si le nombre-mystère est plus grand ou plus petit.

```
// Fonction permettant de saisir un nombre entre 1 et 1000
int saisieUnMille ()
{
 print("Entrez un nombre: ");
 int n = readInt();
 while ((n < 1) && (n > 1000)) {
  println("Erreur, nombre hors de l'intervalle [0;1000]");
  print("Entrez un nombre: ");
  n = readInt();
 }
 return n;
}

// Programme principal
void main ()
{
 int nbemystere = random(0,1000);
 int x = saisieUnMille ();
 while (x != nbemystere) {
  if (x < nbemystere) {
   println(x+" est trop petit...");
  }
  else {
   println(x+" est trop grand...");
  }
  x = saisieUnMille ();
 }
 println("Bravo, c'était bien "+x);
}
```

On voit ici apparaître tous les éléments vu jusqu'ici : affichage à l'écran et saisie au clavier, variables, tests, fonctions et boucles.

3.6.1 Exercice : Nombre de coups

Ajouter au programme un mécanisme permettant de garder en mémoire le nombre de coups joués, et d'arrêter le programme si ce nombre dépasse 10 en affichant "Perdu !"

3.6.2 Exercice : Faire jouer l'ordinateur

Écrire un programme qui, au lieu de faire jouer l'utilisateur, fait jouer l'ordinateur : l'utilisateur entre un nombre, puis l'ordinateur cherche ce nombre en tenant compte des réponses de l'utilisateur. On peut s'amuser à faire adopter à l'ordinateur différentes stratégies :

- une stratégie purement aléatoire (à chaque fois l'ordinateur propose un nombre au hasard)
- une stratégie aléatoire gardant en mémoire les nombres déjà proposés
- une stratégie aléatoire mais limitée à l'intervalle de recherche donnée par l'utilisateur (par exemple si l'ordinateur propose 500 et que l'utilisateur indique que le nombre-mystère est plus grand, le prochain nombre proposé par l'ordinateur sera choisi aléatoirement entre 500 et 1000)
- la "bonne" stratégie dichotomique consistant à essayer de diviser à chaque fois l'intervalle de recherche par 2

4 Débogage, Mise au point

L'apprentissage de la programmation a évolué au rythme des capacités des machines. Dans les années 1970, il était inconcevable de lancer la compilation puis l'exécution d'un programme sans avoir exhaus-

tivement testé toutes les possibilités d'erreur ou de comportement non prévu en raison de la lenteur et du coût de fonctionnement des ordinateurs. Aujourd'hui, la rapidité d'exécution du moindre ordinateur personnel fait qu'il est au contraire conseillé d'utiliser l'ordinateur comme un outil de mise au point, sans forcément passer par une phase de test manuelle (sur papier). Pour autant, il est parfois difficile d'enseigner des concepts liés à l'algorithmique directement sur machine, tant les élèves ont tendance à se focaliser sur leur écran en n'écoutant pas ou peu les consignes : ainsi, il ne faut pas négliger les séances **sans ordinateur** pendant lesquelles l'élève doit imaginer le comportement de ses algorithmes et de leurs traductions sur machine, ce qui permet justement de se mettre à à la place de l'ordinateur et de mieux comprendre son fonctionnement. Le site web de Javascool propose des activités "unplugged" adaptées à ce type de fonctionnement [7].

Un autre écueil de l'apprentissage de la programmation sur machine vient du fait que, dès qu'une erreur est rencontrée, certains élèves ont tendance à corriger au hasard, sans essayer d'utiliser les informations pertinentes délivrées ou qui pourraient être délivrées par l'ordinateur. D'autres s'arrêtent tout simplement, estimant que c'est la machine qui commet une erreur. Il peut également arriver que la correction d'erreurs ne soit pas pertinente, et change complètement le comportement du programme ensuite, même si celui-ci fonctionne.

Il est d'ailleurs difficile d'inculquer aux élèves la notion de vérification, consistant à tester si le comportement d'un programme est conforme à celui attendu dans différentes situations. Par exemple, que se passe-t-il si lors de la saisie d'une valeur l'utilisateur entre une valeur négative (comme à l'exercice 3.2.3) ? Cela a-t-il été prévu par le programme ? Si non, ne risquons-nous pas d'avoir des problèmes si nous reprenons le code plus tard dans un autre programme ? Ce type de questionnement, même si le but n'est pas de former des informaticiens directement opérationnels après le bac, reste intéressant pour l'apprentissage de l'informatique.

Par conséquent, il est souvent pertinent de proposer des exercices où l'on doit essayer de trouver, sans ordinateur, les erreurs qui pourraient se produire. Nous proposons ci-dessous un récapitulatif d'erreurs classiquement rencontrées par les débutants, sans toutefois prétendre à l'exhaustivité. Ces exemples peuvent servir de supports d'exercices, mais ils vous seront surtout utiles pour votre propre apprentissage de Javascool.

4.1 Interpréter les erreurs du compilateur

Il est important de bien comprendre que le compilateur est un programme chargé d'analyser le code pour le transformer en instructions exécutables par la machine [8], en tenant compte de la *grammaire* et de la *syntaxe* définies par le langage. S'il est en général facile de détecter les erreurs, indiquer pourquoi elle a lieu et comment la résoudre devient beaucoup plus compliqué : par exemple dans l'instruction **x=3*(y-x+4*5 ;**, on voit aisément qu'une parenthèse est manquante, mais seul le concepteur du code peut dire si l'expression correcte est **x=3*(y-x)+4*5 ;** ou **x=3*(y-x+4)*5 ;**

De même, la nature de l'erreur déterminée par le compilateur peut être ambiguë, comme dans l'exemple suivant où l'on tente de compiler le programme **erreursyntaxe.jvs** :

```
void main() {
 double x = 2y;
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 3 :
  Un ';' est attendu (il peut manquer, ou une parenthèse être incorrecte, ..)
   double x = 2y;
            ^
```

Ici l'erreur vient de l'instruction **x=2y ;**, qui suit la notation habituellement utilisée en mathématiques mais qui devrait s'écrire en Javascool **x=2*y ;** et pourtant, le compilateur indique :

7. http://javascool.gforge.inria.fr/v4//index.php?page=resources&action=link-sujets-mathinfo

8. C'est en fait un peu plus compliqué dans le cas de Javascool : un programme Javascool est d'abord traduit en programme Java, et c'est ce dernier qui est compilé. Pour plus d'explications concernant Java voir : http://java.developpez.com/cours/

- qu'à la ligne 3 un point-virgule est attendu (Javascool ajoute une première ligne vide quand on reformate le code)
- que ce point-virgule devrait être placé après **x** (le symbole ^ étant censé indiquer l'endroit exact de l'erreur sur la ligne).

Le compilateur considère en fait qu'il faudrait plutôt écrire **x;**, ce qui est effectivement un moyen de corriger l'erreur mais ne répond probablement pas à ce que souhaitait le concepteur... qui a de toute façon oublié de déclarer la variable **y** : après avoir remplacé l'instruction fautive par **x=2*y;** une nouvelle erreur apparaîtra à la compilation.

Par conséquent l'information la plus utile reste le plus souvent le numéro de ligne, mais même cette information peut être trompeuse comme ci-dessous où le compilateur indique une erreur à la ligne 4 alors que celle-ci est clairement due à une parenthèse manquante au début :

```
void main( {
double x = 2*y;
}
-- Affichage de la sortie --------
Attention: il faut un block "void main()" pour que le programme puisque se compiler

 Erreur de syntaxe ligne 4 :
  (illegal start of expression) L'instruction (ou la précédente) est tronquée ou mal
  écrite
void main({
^
```

Une difficulté supplémentaire avec Javascool est que les messages d'erreurs peuvent être une combinaison de messages délivrés en français et en anglais. Ceci s'explique par le fait que les erreurs sont générées par Java, puis traitées par Javascool pour en donner si possible une explication plus claire en français. Les messages "bruts" en anglais sont donc plutôt dûs à Java, et ceux en français à Javascool. Néanmoins on retrouve souvent les mêmes messages, ce qui permet à la longue de s'y adapter (et de toute façon une traduction systématique en français des messages du compilateur ne serait pas forcément beaucoup plus simple à comprendre).

Les erreurs les plus fréquemment rencontrées sont liées à des erreurs de syntaxe : fautes de frappe, accolades ou points-virgules manquants ou en trop, etc. Nous montrons ci-dessous des exemples de programmes comportant ce type d'erreurs et tentons d'expliquer les messages affichés lors de la phase de compilation.

4.1.1 Points-virgules

Pas ambiguïté ci-dessous, le point-virgule manquant en fin de ligne est à l'origine de l'erreur, et il est correctement détecté :

```
void main() {
 println("Hello World !")
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 3 :
  Un ';' est attendu (il peut manquer, ou une parenthèse être incorrecte, ..)
   println("Hello World !")
                           ^
```

Pour résoudre le problème la tentation est grande d'ajouter systématiquement un point-virgule à la fin de chaque ligne. Parfois ce n'est pas très grave, par exemple le programme ci-dessous n'entraînera pas d'erreurs :

```
void main() {;
 println("Hello World !");;
};
```

Nous verrons dans la partie 4.2 que cela peut tout de même devenir problématique dans certains cas et que ce n'est pas un comportement à encourager.

4.1.2 Mots-clefs incorrects

Les mots-clefs du langage doivent être toujours écrits correctement et en minuscules, si ce n'est pas le cas Javascool les considère comme des variables ou des fonctions définies par l'utilisateur et tente d'expliquer comment celles-ci devraient être utilisées, par exemple ici en considérant qu'ajouter un point-virgule pourrait être la solution :

```
Void Main() {
 println("Hello World !");
}
-- Affichage de la sortie --------
Attention: il faut un block "void main()" pour que le programme puisque se compiler

 Erreur de syntaxe ligne 4 :
  Un ';' est attendu (il peut manquer, ou une parenthèse être incorrecte, ..)
Void Main() {
         ^
```

Dans ce deuxième exemple le message est plus "parlant" puisque la fonction **printl** n'existe pas (ce serait plutôt **print** ou **println**) :

```
void main() {
 printl("Hello World !");
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 3 :
  Il y a un symbole non-défini à cette ligne : ≪method printl(java.lang.String)≫
  (utilisez-vous la bonne proglet ?)
   printl("Hello World !");
   ^
```

4.1.3 Accolades et parenthèses

C'est bien une accolade ouvrante qu'il faudrait placer après **void main()**, et non un point-virgule :

```
void main()
 println("Hello World !");
}
-- Affichage de la sortie --------
 Erreur de syntaxe ligne 2 :
  Un ';' est attendu (il peut manquer, ou une parenthèse être incorrecte, ..)
void main()
           ^
```

Ci-dessous il manque une accolade fermante, ce qui est relativement bien détecté par le compilateur qui indique qu'il a analysé (ou *parsé*) le code et qu'il a atteint la fin du fichier sans rencontrer cette accolade pourtant nécessaire :

```
void main() {
 println("Hello World !");
-- Affichage de la sortie --------
Erreur de syntaxe ligne 5 :
```

```
  reached end of file while parsing
}
 ^
```

Sur le même principe, attention à ouvrir et fermer correctement les parenthèses.

4.1.4 Utilisation incorrecte des variables

Il est indispensable de déclarer les variables avant de pouvoir les utiliser (ci-dessous on devrait plutôt écrire **int x=2 ;**) :

```
void main() {
 x = 2;
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 3 :
  Il y a un symbole non-défini à cette ligne : ≪variable x≫
  (utilisez-vous la bonne proglet ?)
   x = 2;
   ^
```

De la même façon, il n'y aucun sens à afficher le contenu de la variable **x** si on ne lui pas affecté préalablement une valeur. C'est un problème classique d'**initialisation de variable** :

```
void main() {
 int x;
 println(x);
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 4 :
  variable x might not have been initialized
   println(x);
           ^
```

Attention toutefois à ne pas déclarer plusieurs fois la même variable :

```
void main() {
 int x = 2;
 int y = x+x;
 int x = 3;
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 5 :
  x is already defined in main()
   int x = 3;
       ^
```

Il n'y a aucun sens à vouloir affecter à une variable une valeur non-adaptée. Ci-dessous par exemple on affecte à **x** le résultat donné par l'instruction **println("Hello World !")**, alors que **println** ne sert qu'à afficher du texte à l'écran :

```
void main() {
 int x = println("Hello World !");
}
-- Affichage de la sortie --------
```

```
Erreur de syntaxe ligne 3 :
  Vous avez mis une valeur de type void alors qu'il faut une valeur de type int
   int x = println("Hello World !");
                   ^
```

De façon similaire, il faut être très vigilant avec les types numériques qui ne sont pas équivalents entre eux :

```
void main() {
 double x = 2;
 int y = 2*x;
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 4 :
  possible loss of precision
found   : double
required: int
   int y = 2 * x;
             ^
```

Dans cet exemple l'erreur est réellement difficile à comprendre : il n'y a clairement ici aucun risque de perte de précision (*possible loss of precision*) avec l'opération **y=2*x** puisque **x** a été initialisé avec une valeur entière. Mais ce serait le cas si on avait écrit **x=2.5**, et le compilateur ne veut donc prendre aucun risque...

4.1.5 Comparaisons

Le test d'égalité en Javascool s'écrit avec un double symbole = (ci-dessous il faudrait donc plutôt écrire **x==3**). Si ce n'est pas le cas le compilateur indique une erreur relativement compréhensible : il indique que le résultat attendu pour une comparaison est plutôt de type *boolean*, c'est-à-dire une valeur **vraie** ou **fausse**.

```
void main() {
 int x = 2;
 if (x = 3) {
  println("Erreur");
 }
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 4 :
  Vous avez mis une valeur de type int alors qu'il faut une valeur de type boolean
   if (x = 3) {
         ^
```

Et bien sûr attention à comparer ce qui est comparable, ici on essaie de comparer sans succès un nombre avec du texte :

```
void main() {
 int x = 2;
 if (x == "Hello world !") {
  println("Erreur");
 }
}
-- Affichage de la sortie --------
Erreur de syntaxe ligne 4 :
```

```
incomparable types: int and java.lang.String
 if (x == "Hello world !") {
       ^
```

4.2 Erreurs d'exécution

Une fois les erreurs corrigées, le compilateur affiche un victorieux "Compilation réussie !" qui ne signifie pourtant pas que tous les problèmes sont résolus. En effet, certaines erreurs sont indétectables lors de la phase d'analyse du code et ne surviennent que lorsque le programme s'exécute.

Ces erreurs sont plus difficiles à corriger que les erreurs rencontrées lors de la phase de compilation car il sera parfois impossible de savoir quelle instruction est précisément incriminée. On peut distinguer deux types d'erreurs à l'exécution :

- les erreurs **bloquantes**, qui vont **interrompre l'exécution** et qui seront donc visibles par le concepteur qui devra trouver un moyen de les corriger.
- les erreurs **non-bloquantes**, qui n'empêchent pas l'exécution de se dérouler mais qui entraînent des résultats ou un comportement erronés. Ce sont les plus difficiles à détecter et à corriger, puisqu'elles sont en quelque sorte visibles uniquement de façon indirecte.

Comme dans la section précédente, nous répertorions ici quelques types d'erreurs fréquemment rencontrés sans prétendre à l'exhaustivité. Certaines peuvent être considérées comme des erreurs plutôt *numériques* (un calcul ne donnant pas le résultat escompté en raison d'une méconnaissance du langage), d'autres comme des erreurs plutôt *algorithmiques* (une mauvaise conception du programme dès le départ).

4.2.1 Division par zéro

La division par zéro étant indéfinie, elle est a priori interdite mais n'est détectée qu'à l'exécution :

```
void main() {
 int x = 5/0;
 println("x = "+x);
}
-- Affichage de la sortie --------
Erreur lors de l'exécution de la proglet
java.lang.ArithmeticException: / by zero
.main(JvsToJavaTranslated26.java:2)
.run(JvsToJavaTranslated26.java:1)
```

Ici l'erreur vient clairement de l'instruction `5/0`, ce qui cause une erreur entraînant l'arrêt du programme, puisque la ligne suivante n'est pas exécutée. Une erreur en Java se dit **Exception**, et l'affichage indique son type, ici **ArithmeticException**.

Ce type d'erreur peut aussi survenir lorsque le dénominateur est une variable atteignant la valeur zéro :

```
void main() {
 for (int i = 10; i >= 0; i--) {
  int x = 100/i;
  print(x+" ");
 }
}
-- Affichage de la sortie --------
10 11 12 14 16 20 25 33 50 100
------------------
Erreur lors de l'exécution de la proglet
java.lang.ArithmeticException: / by zero
.main(JvsToJavaTranslated27.java:4)
.run(JvsToJavaTranslated27.java:1)
```

Et pourtant la division par zéro n'est parfois pas détectée... Si dans les exemples précédents l'instruction incriminée est facile à déterminer, cela peut être plus compliqué :

```
void main() {
 double x = 5/0.0;
 println("x = "+x);
}
-- Affichage de la sortie --------
x = Infinity
```

On voit ici que la division par zéro n'a pas provoqué l'arrêt du programme, qui a bien exécuté l'instruction `5/0.0` puis affiché la valeur obtenue. La différence avec les exemples précédents est que le type **double** de Javascool intègre une dimension *symbolique* : une variable de type **double** peut avoir la valeur **Infinity**, ce qui est cohérent avec la définition mathématique de la division. Et on peut inversement... obtenir la valeur zéro en divisant par **Infinity**, comme le montre l'exemple suivant.

```
void main() {
 double x = 5/0.0;
 println("x = "+x);
 double y = 3 + x;
 println("y = "+y);
 double z = 1 / y;
 println("z = " + z);
}
-- Affichage de la sortie --------
x = Infinity
y = Infinity
z = 0.0
```

4.2.2 Division entière

La division peut aussi provoquer un comportement inattendu, lorsqu'elle est appliquée sur des entiers :

```
void main() {
 for (int i = 1; i <= 10; i++) {
  int x = 1/i;
  print(x+" ");
 }
}
-- Affichage de la sortie --------
1 0 0 0 0 0 0 0 0 0
```

On voit avec ce résultat que le choix du type **int** n'est évidemment pas adapté pour représenter des valeurs entre 0 et 1, qui dans ce cas sont tronquées à la valeur 0. Ceci est dû au fait que Javascool applique ici la *division entière*, car le calcul n'implique que des entiers. Ce n'est donc pas à proprement parler une erreur, mais il faut absolument veiller à bien choisir le type **double** quand c'est nécessaire.

4.2.3 Dépassement de capacité

Une autre erreur courante peut se produire si on dépasse la capacité de stockage pour les types **int** ou **double**. En effet comme on peut s'en douter cette capacité n'est pas infinie :

```
void main() {
 int x = 0;
 for (int i = 1; i <= 7; i++) {
  x = (x+1)*(x+2);
  print(x+" ");
 }
```

```
}
-- Affichage de la sortie --------
2 12 182 33672 1133904602 1044523572 -1265919698
```

Là encore l'erreur n'est pas signalée, au programmeur donc de faire attention. Par définition les valeurs pour les **int** vont de -2147483648 à 2147483647, et pour les **double** de $4,9.10^{-324}$ à $1,7.10^{308}$

4.2.4 Point-virgule derrière un test ou une boucle

Le point-virgule systématique en fin de ligne peut être générateur d'erreurs d'exécution difficiles à repérer, quand ces lignes sont des boucles ou des tests. En effet le point-virgule sert dans ce cas à *ne rien faire*, c'est-à-dire à exécuter la boucle ou le test mais pas le bloc d'instructions qui suit. Considérons l'exemple suivant :

```
void main() {
 int x = 0;
 for (int i = 1; i <= 7; i++); {
  x = (x+1)*(x+2);
  print(x+" ");
 }
}
-- Affichage de la sortie --------
2
```

Comme on le voit, la boucle **for** est ici terminée ici par un point-virgule. Ceci entraîne une exécution donnant la fausse impression que la boucle n'est exécutée que pour la première valeur de i. Pourtant, le point-virgule a comme conséquence que la boucle "tourne" bien jusqu'à $i = 7$, mais cela n'a comme conséquence que l'incrémentation du compteur. Le bloc de calcul de la valeur de x n'est en fait exécuté *qu'une fois la boucle terminée*, et ce code Javascool aurait plutôt comme équivalent en langage algorithmique :

```
Algorithme principal
Variable x: entier
Début
 x <- 0
 Pour i de 1 à 7 Faire
 Début
  // Bloc vide !
 FinPour
 x <- (x+1)*(x+2)
 Afficher (x, " ")
Fin
```

On observe le même problème dans le cas d'un test, comme dans l'exemple suivant :

```
void main() {
 int x = 0;
 if (x == 3); {
  print(x+" ");
 }
}
-- Affichage de la sortie --------
0
```

Encore une fois, le comportement qui semble attendu (afficher la valeur de x si celle-ci est égale à 3) n'est pas celui observé, ce qui est dû au point-virgule en fin de test. L'équivalent en langage algorithmique est plutôt ici :

```
Algorithme principal
Variable x: entier
Début
 x <- 0
 Si (x = 3) Alors
 Début
  // Bloc vide !
 FinSi
 Afficher (x, " ")
Fin
```

Il est évident que ce comportement n'est pas en général celui désiré, par conséquent **les points-virgules en fin de boucle ou de test doivent la plupart du temps être évités**.

4.2.5 Boucles infinies

La *boucle infinie* est un problème très courant, qui relève la plupart du temps d'une erreur de conception de l'algorithme mais qui par définition n'est détectable qu'à l'exécution. L'exemple le plus simple est le suivant :

```
void main() {
 while (true);
}
-- Version en langage algorithmique
Algorithme principal
Début
 Tant que (vrai) Faire
 Début
  // Bloc vide !
 FinTantQue
Fin
```

Ce programme, comme l'indique le terme *infini*, ne s'arrête... que si l'utilisateur l'interrompt manuellement (avec Javascool cliquer sur le bouton **Arrêter**), sinon il continue indéfiniment son exécution. Évidemment la plupart du temps ce n'est pas dans l'intention du programmeur de créer une boucle infinie, qui provient souvent d'une erreur de test ou d'incrémentation dans la boucle, comme dans les deux exemples suivants :

```
-- Version en langage algorithmique
Algorithme principal
Variable x: entier
Début
 x <- 0
 Tant que (x < 10) Faire
 Début
  Afficher (x, " ")
  x <- x+1
 FinTantQue
Fin
-- Première traduction erronée en Javascool
void main() {
 int x = 0;
 while (x >= 0) {
  println(x+" ");
  x++;
 }
}
-- Affichage de la sortie --------
```

```
0
1
2
...
-- Seconde traduction erronée en Javascool
void main() {
 int x = 0;
 while (x < 10) {
  println(x+" ");
 }
}
-- Affichage de la sortie --------
0
0
0
...
```

Dans le premier cas, le test utilisé pour terminer la boucle est mal traduit, et implique que celle-ci ne s'arrêtera jamais puisque la condition $x >= 0$ est toujours vérifiée. Dans le second cas, c'est l'erreur vient de l'oubli de l'incrémentation de la variable. Encore une fois la boucle ne s'arrêtera jamais puisque x reste toujours à 0 et la condition $x < 10$ n'a aucune chance de ne plus être vérifiée.

Évidemment les cas les plus complexes à résoudre sont ceux impliquant d'autres types d'erreurs, par exemple le point-virgule en fin de test :

```
-- Troisième traduction erronée en Javascool
void main() {
 int x = 0;
 while (x < 10); {
  println(x+" ");
  x++;
 }
}
```

Ici le point-virgule fait que la boucle reste bloquée puisque le compteur x n'est jamais incrémenté, par conséquent rien ne s'affiche à l'écran...

4.2.6 Récursivité non contrôlée

Comme les boucles infinies, la récursivité non contrôlée est en général plutôt due à une erreur de conception de l'algorithme initial, cette fois en utilisant une fonction qui s'appelle elle-même. Voici un exemple de programme correct :

```
int sommeTousEntiers (int x) {
 if (x != 0) return x+sommeTousEntiers(x-1);
 return 0;
}
void main() {
 println("Somme des entiers de 1 à 10: " +
         sommeTousEntiers(10));
}
```

Ici l'exécution se déroule sans problème, mais une petite erreur et on entre dans une boucle infinie, par exemple en écrivant *sommeTousEntiers(x+1)* comme appel récursif. Et même si la fonction récursive est correcte, que se passerait-il si on souhaitait calculer *sommeTousEntiers(-1)* ?

En tout état de cause au Lycée on peut préférer, comme l'écrivent les concepteurs de Javascool, "éviter de définir des fonctions récursives, sauf dans les cas indispensables", même si dans certains cas cela permet "des calculs très complexes et très intéressants"...

4.3 Mise au point de programmes

La plupart des langages de programmation disposent de logiciels appelés EDI (pour *Environnement de développement intégré*) tels que Visual Studio, Eclipse, Netbeans, etc. Ces logiciels permettent d'écrire des programmes tout en réduisant le risque d'erreur, car ils proposent notamment une aide lors de la saisie à la manière d'un correcteur orthographique dans un logiciel de traitement de texte. Ils sont aussi plus complexes à utiliser que Javascool, qui offre moins de fonctionnalités de mise au point mais permet néanmoins d'aider à la recherche d'erreurs. Il faut de plus essayer, pour tout développement informatique, de se tenir à des règles simples qui aideront beaucoup à se rapprocher du programme final désiré.

4.3.1 Assertions et traçage de variables

Confronté à une erreur d'exécution, la première option pour le programmeur consiste à vérifier que certaines variables ont bien la valeur attendue. Ceci peut se faire de façon systématique, comme dans l'exemple suivant :

```
void main() {
 int x = 0;
 for (int i = 1; i <= 10; i++) {
  x = x + i;
  println("Pour i = "+i+", j'obtiens x = "+x);
 }
 println("Somme des entiers de 1 à 10: "+x);
}
-- Affichage de la sortie --------
Pour i = 1, j'obtiens x = 1
Pour i = 2, j'obtiens x = 3
Pour i = 3, j'obtiens x = 6
...
Somme des entiers de 1 à 10: 55
```

Ainsi, si le résultat n'est pas conforme à celui attendu, voir la valeur des variables au cours de l'exécution peut donner une éventuelle indication pour déterminer quelles instructions sont problématiques.

Les *assertions* sont un mécanisme permettant de "garantir" l'arrêt de l'exécution si une condition n'est pas satisfaite dans le programme. On peut voir une mise en œuvre dans l'exemple suivant :

```
void main() {
 println("Entrez le numérateur et le dénominateur: ");
 int n = readInt();
 int d = readInt();
 assertion((d != 0), "Le dénominateur est non nul !");
 println("n/d="+(n/d));
}
-- Affichage de la sortie --------
Entrez le numérateur et le dénominateur:
1
0
Arrêt du programme :{
  L'assertion(≪Le dénominateur est non nul !≫) est fausse.
  Pile d'exécution: {
     org.javascool.macros.Macros.assertion(Macros.ligne : 147)
     main(ligne : 6)
  }
}
-------------------
Erreur lors de l'exécution de la proglet
java.lang.RuntimeException: Programme arrêté !
.main(JvsToJavaTranslated35.java:6)
```

```
-------------------
```

On voit ici que si la condition ($d! = 0$) n'est pas vérifiée, alors le programme s'arrête en affichant le message spécifié par le programmeur. Si au contraire cette condition est satisfaite alors l'exécution continuera normalement.

4.3.2 Jeux d'essai

Les affichages de variables ou les assertions sont utiles lors de la mise au point du programme mais peuvent en général être supprimées dans la version finale qui n'a pas besoin de ces affichages intempestifs. Néanmoins cela ne doit pas empêcher de se poser les bonnes questions lors de la conception de l'algorithme initial :

- Quelles sont les valeurs pouvant poser problème dans mon programme ?
- Quelles sont les instructions critiques pouvant interrompre l'exécution ?

On parle en général de "jeux d'essai", qui sont normalement définis et simulés sur papier, sans l'aide de l'ordinateur. Néanmoins avoir la machine à disposition peut être un avantage pour tester rapidement un nombre important de possibilités, éventuellement aléatoires : on peut ainsi vérifier l'exactitude du résultat sur ces cas, puis supposer que le programme fonctionnera correctement quelles que soient les valeurs fournies. Dans l'exemple suivant on teste de façon aléatoire le comportement d'une fonction **racine** permettant de calculer la racine carrée d'un nombre :

```
double racine (double h) {
 double res = 1;
 for (int cpt = 1; cpt <= 7; cpt++) {
  double aux = res;
  res = aux - ((aux*aux-h)/(2*aux));
 }
 return res;
}
void main() {
 for (int i = 1; i < 5; i++) {
  double test = random()*1000;
  println("Racine de "+test+" calculée avec sqrt: "
          +sqrt(test));
  println("Racine de "+test+" calculée avec racine: "
          +racine(test));
 }
}
-- Affichage de la sortie --------
Racine de 876.1992731341172 calculée avec sqrt: 29.60066339010187
Racine de 876.1992731341172 calculée avec racine: 29.61101484265385
Racine de 145.45852354847744 calculée avec sqrt: 12.060618705044838
Racine de 145.45852354847744 calculée avec racine: 12.060618718938137
Racine de 758.5861021282223 calculée avec sqrt: 27.54244183307323
Racine de 758.5861021282223 calculée avec racine: 27.547484266118722
...
```

4.3.3 Calcul de la vitesse d'exécution

Déterminer la vitesse d'exécution d'un programme peut s'avérer utile dans certains cas, notamment pour les applications numériques où l'objectif est d'obtenir la plus grande vitesse possible. Ainsi on peut comparer différentes stratégies pour déterminer laquelle amène le meilleur gain de temps. Un autre intérêt peut être de détecter, si un programme s'exécute de façon anormalement lente, les parties du programme problématiques.

Le programme qui suit reprend l'exemple précédent et ajoute un calcul de la vitesse :

```
void main() {
```

```
 long heureDebut = System.currentTimeMillis();
 for (int i = 1; i < 5; i++) {
  double test = random()*1000;
  println("Racine de "+test+" calculée avec sqrt: "
          +sqrt(test));
  println("Racine de "+test+" calculée avec racine: "
          +racine(test));
 }
 long heureFin = System.currentTimeMillis();
 println("Temps d'exécution: "+ (heureFin - heureDebut)+ "ms");
}
-- Affichage de la sortie --------
Racine de 876.1992731341172 calculée avec sqrt: 29.60066339010187
...
Temps d'exécution: 101ms
```

Le principe est de stocker dans une variable de type **long** la valeur de l'horloge de l'ordinateur avant le démarrage des calculs, puis de ré-itérer cette opération à la fin. Ainsi, en faisant la différence entre ces deux valeurs on obtient le temps, exprimé en *ms*, qui s'est écoulé entre ces deux opérations. Il est important de noter cependant que ce temps peut être plus ou moins long en fonction d'autres événements, par exemple si de nombreux logiciels sont en train de fonctionner sur l'ordinateur, ou bien si celui-ci est entré en mode "mise à jour", etc. Par conséquent il est probable que si on exécute de nouveau ce programme le temps d'exécution sera proche de $101ms$ mais pas tout à fait identique.

On peut évidemment s'intéresser seulement à certaines parties du programme, par exemple ici pour déterminer si le calcul de la racine carrée est plus rapide avec **sqrt** ou avec notre fonction **racine** :

```
void main() {
 long heureDebut, heureFin;
 for (int i = 1; i < 5; i++) {
  double test = random()*1000;
  heureDebut = System.currentTimeMillis();
  println("Racine de "+test+" calculée avec sqrt: "
          +sqrt(test));
  heureFin = System.currentTimeMillis();
  println("Temps d'exécution pour sqrt: "+ (heureFin -  heureDebut)+ "ms");
  heureDebut = System.currentTimeMillis();
  println("Racine de "+test+" calculée avec racine: "+racine(test));
  heureFin = System.currentTimeMillis();
  println("Temps d'exécution pour racine: "+ (heureFin -  heureDebut)+ "ms");
 }
}
-- Affichage de la sortie --------
Racine de 210.81480147628383 calculée avec sqrt: 14.519462850817996
Temps d'exécution pour sqrt: 14ms
Racine de 210.81480147628383 calculée avec racine: 14.519463472506295
Temps d'exécution pour racine: 73ms
Racine de 186.84659004566151 calculée avec sqrt: 13.669183956830105
Temps d'exécution pour sqrt: 4ms
Racine de 186.84659004566151 calculée avec racine: 13.669184151203385
Temps d'exécution pour racine: 3ms
Racine de 352.80059504672334 calculée avec sqrt: 18.782986851050165
Temps d'exécution pour sqrt: 3ms
Racine de 352.80059504672334 calculée avec racine: 18.78303152345477
Temps d'exécution pour racine: 3ms
...
```

On peut remarquer ici que l'exécution lors des premiers calculs semble prendre plus de temps. Il s'agit en fait d'opérations d'initialisations exécutées par Javascool, impossibles à maîtriser par le programmeur.

On voit ensuite que les temps de calculs sont semblables, on pourrait donc dire que les deux calculs sont équivalents. Pourtant, considérons ce nouvel exemple :

```
void main() {
 long heureDebut, heureFin;
 long tpssqrt, tpsracine;
 tpssqrt = tpsracine = 0;
 for (int i = 1; i < 500000; i++) {
  double test = random()*1000;
  heureDebut = System.currentTimeMillis();
  double valsqrt = sqrt(test);
  heureFin = System.currentTimeMillis();
  tpssqrt += (heureFin - heureDebut);
  heureDebut = System.currentTimeMillis();
  double valracine = racine(test);
  heureFin = System.currentTimeMillis();
  tpsracine += (heureFin - heureDebut);
 }
  println("Temps d'exécution total pour sqrt: " + tpssqrt + "ms");
  println("Temps d'exécution total pour racine: " + tpsracine + "ms");
}
-- Affichage de la sortie --------
Temps d'exécution total pour sqrt: 685ms
Temps d'exécution total pour racine: 620ms
```

Ce dernier exemple donne des résultats plus significatifs, d'abord en enlevant du calcul de vitesse les opérations d'affichage à l'écran, et en considérant un grand nombre d'opérations (500000). Par conséquent on peut conclure que **racine** fonctionne de façon légèrement plus rapide que **sqrt**, ce qui est relativement normal puisque notre calcul de racine carrée renvoie des résultats un peu moins précis.

4.3.4 Soigner la présentation

La mise au point d'un programme nécessite de prendre en compte des aspects "esthétiques" avec comme objectif principal de faciliter la lecture du programme par une personne autre que le programmeur lui-même. On peut signaler que pour le développement de gros logiciels ces aspects sont consignés en début de projet et doivent être respectés par tous les programmeurs intervenant dans le projet.

L'intérêt d'écrire du code lisible et donc plus facilement compréhensible est aussi qu'il permet de détecter plus rapidement d'où peuvent venir les erreurs éventuelles. Nous ne reviendrons pas ici sur le choix des noms de variables et de fonctions, qui doivent absolument être "parlants" et ne pas se restreindre à **toto** ou **tutu**, voire à des noms choisis aléatoirement.

Un autre aspect concerne les explications que l'on peut donner directement dans le code : le mécanisme des **commentaires** permet d'ajouter des lignes, commençant par `//`, pour donner des éléments factuels en début de programme, des explications concernant les fonctions ou les boucles utilisées, etc. Ceci est illustré dans l'exemple ci-dessous où de nombreux commentaires ont été ajoutés au programme vu précédemment.

```
// Tests sur la racine carrée
// Programme développé par Benoît
// Dernière modification: 25 juin 2011

// Racine: fonction qui calcule la racine carrée
//         de la valeur h
double racine (double h) {
 double res = 1;
 for (int cpt = 1; cpt <= 7; cpt++) {
  double aux = res;
  res = aux - ((aux*aux-h)/(2*aux));
```

```
 }
 return res;
}

// Programme principal: Calcul de la racine carrée
//   de 5 valeurs choisies aléatoirement en comparant
//   les résultats entre sqrt et notre fonction
void main() {
 for (int i = 1; i < 5; i++) {
  // Calcul de la valeur aléatoire
  double test = random()*1000;
  println("Racine de "+test+" calculée avec sqrt: "
          +sqrt(test));
  println("Racine de "+test+" calculée avec racine: "
          +racine(test));
 }
}
```

On pourrait pousser le raisonnement jusqu'à écrire un commentaire avant chaque ligne de programme, mais ceci doit rester à l'appréciation du programmeur (ou du chef de projet) : il est d'usage de placer des commentaires pour expliquer les parties du code dont on estime qu'elles peuvent être difficile à comprendre. Il ne faut pas non plus "noyer" le code au milieu des commentaires puisque l'objectif doit rester la lisibilité.

Un dernier aspect important, que les programmes présentés dans ce document tentent de respecter, concerne l'**indentation** des programmes, c'est-à-dire l'alignement des instructions les unes sous les autres, notamment dans le cas où on utilise plusieurs niveaux d'imbrication. Reprenons par exemple le programme vu à la section 3.6, sous une forme absolument équivalente en terme d'exécution :

```
void main ()
{
int toto = random(0,1000);
int tutu = saisieUnMille ();
while (tutu != toto) {
if (tutu < toto) {
println(x+" est trop petit...");
}
else {
println(tutu+" est trop grand...");
}
tutu = saisieUnMille ();
}
println("Bravo, c'était bien "+tutu);
}
```

En enlevant l'indentation (et en renommant les variables), on obtient un programme peu lisible, puisqu'il est difficile de voir par exemple où s'arrête la boucle **while**. Ce problème est évidemment de plus en plus critique à mesure qu'on augmente le nombre de lignes dans un programme.

Tout ceci n'est bien sûr pas propre à Javascool, et la rédaction de programmes dans un langage quelconque devrait de toute façon respecter ces principes.

4.4 Le jeu du calcul mental

Appliquer toutes les notions de débogage et de mise au point vues dans cette section pour réaliser le jeu du calcul mental :

1. L'ordinateur affiche deux nombres aléatoires entre 10 et 40
2. Le joueur doit entrer le résultat de la multiplication de ces deux nombres le plus vite possible

3. L'ordinateur vérifie le résultat. Si le joueur n'a pas trouvé la bonne réponse, un message d'encouragement s'affiche et le programme recommence à l'étape 1.
4. Si le joueur a bien trouvé la bonne réponse, l'ordinateur le félicite et affiche le temps mis pour répondre (grâce à la fonction `currentTimeMillis` vue précédemment) avant de repartir à l'étape 1.

Améliorer ensuite en ajoutant la notion de **high-score**, c'est-à-dire stocker dans une variable le temps de réponse du joueur le plus rapide, qui peut éventuellement être battu...

5 Manipuler l'information

On s'intéresse ici à la manipulation avec Javascool d'entités algorithmiques usuelles (les tableaux et les caractères), et d'autres éléments peu développés dans un cours d'algorithmique mais qui sont à connaître pour mieux comprendre le fonctionnement de Javascool et pouvoir réaliser plus facilement ses propres programmes. On insistera notamment ici sur les aspects liés à la représentation des nombres (entiers ou réels), au calcul binaire et à la logique booléenne.

Les différents éléments développés ci-dessous sont accessibles à partir du tutoriel "Faire des exercices d'application : les algorithmes mathématiques en 2nd" accessible sur `http://javascool.gforge.inria.fr/documents/sujets-mathinfo/index.htm`.

5.1 Tableaux

Un **tableau** (appelé aussi **vecteur**) est un ensemble de valeurs du même type stockées les unes à la suite des autres. Nous nous basons ici sur l'activité Javascool sur les tableaux accessible depuis le tutoriel, qui s'ouvre par l'exemple suivant :

```
void main()
{
 // Déclaration et remplissage du tableau "noms"
 String[] noms = {"Alice", "Bob", "Samia", "Zheng-You"};
 // Affichage du contenu du tableau (première méthode)
 print("Contenu du tableau noms: ");
 print(noms[0] + " ");
 print(noms[1] + " ");
 print(noms[2] + " ");
 print(noms[3] + " ");
 println("");
 // Modification de l'une des valeurs
 noms[2] = "Samiha";
 // Affichage du contenu du tableau (deuxième méthode)
 print("Contenu du tableau noms: ");
 for (int i = 0; i < noms.length; i++) {
  print(noms[i] + " ");
 }
 println("");
}
-- Affichage de la sortie --------
Contenu du tableau noms: Alice Bob Samia Zheng-You
Contenu du tableau noms: Alice Bob Samiha Zheng-You
```

On comprend avec cet exemple que :

- un tableau est déclaré avec son type (ici **String**), et peut être rempli directement avec des valeurs (ici 4). On obtient donc ici un tableau de 4 chaînes de caractères.
- les différentes valeurs stockées dans le tableau sont accessibles grâce à leur **index**, cet index démarrant à 0 pour la première valeur, 1 pour la seconde, etc.
- on peut modifier très facilement une valeur stockée dans le tableau

– comme indiqué dans le tutoriel, la variable **noms.length** stocke de façon automatique le nombre de valeurs du tableau (ici 4)
– les boucles, et notamment la boucle **for**, sont très bien adaptées à la manipulation des tableaux. Dans cet exemple, on affiche les valeurs du tableau grâce à une boucle, sans avoir à se préoccuper du nombre de valeurs.

Il faut également noter que les exemples présentés ici fonctionnent avec des tableaux contenant des **String**, mais nous verrons dans les exercices qui suivent qu'on peut de la même façon manipuler des **int**, **double**, etc.

Le tutoriel pointe un problème récurrent avec les tableaux qui concerne l'accès à une valeur qui n'existe pas. En effet, le compilateur autorise les instructions du type **noms[-1]**, mais ces instructions vont générer des erreurs à l'exécution :

```
void main()
{
 String[] noms = {"Alice", "Bob", "Samia", "Zheng-You"};
 // Affichage de la valeur noms[-1]
 println(noms[-1]);
}
-- Affichage de la sortie --------
Sauvegarde de tableaux.jvs ..
Compilation réussie !
---------------------------------
java.lang.ArrayIndexOutOfBoundsException: -1
```

L'erreur signalée ici indique que l'index utilisé (-1) est "hors-limite" (ou *out of bounds*) : en effet l'index doit être obligatoirement **compris entre 0 (inclus) et noms.length (exclus)**. Cela n'a donc pas de sens non plus de vouloir accéder à la valeur **noms[4]** ici.

Il est possible de déclarer un tableau sans le remplir immédiatement, mais dans ce cas on devra indiquer le nombre de valeurs :

```
void main()
{
 int i;
 // Déclaration du tableau "noms"
 String[] noms = new String[4];
 // Remplissage par l'utilisateur
 for (i = 0; i < noms.length; i++) {
  println("Entrez la valeur d'index " + i + " : ");
  noms[i] = readString();
 }
 // Affichage du contenu du tableau (deuxième méthode)
 print("Contenu du tableau noms: ");
 for (i = 0; i < noms.length; i++) {
  print(noms[i] + " ");
 }
 println("");
}
-- Affichage de la sortie --------
Entrez la valeur d'index 0 :  (on entre "Bill")
Entrez la valeur d'index 1 :  ...
Entrez la valeur d'index 2 :  ...
Entrez la valeur d'index 3 :  ...
Contenu du tableau noms: Bill Steve Mark Georges
```

La manipulation de tableaux à travers des fonctions mérite un exemple pour bien remarquer qu'un tableau passé en paramètre d'une fonction devient une variable d'entrée-sortie :

```
void remplissage (String tab[]) {
```

```
 for (int i = 0; i < tab.length; i++) {
  println("Entrez la valeur d'index " + i + " : ");
  tab[i] = readString();
 }
}
void affichage (String tab[]) {
 for (int i = 0; i < tab.length; i++) {
  print(tab[i] + " ");
 }
 println("");
}
void main()
{
 // Déclaration du tableau "noms"
 String[] noms = new String[4];
 // Remplissage par l'utilisateur
 remplissage (noms);
 // Affichage du contenu du tableau
 print("Contenu du tableau noms: ");
 affichage (noms);
}
```

Le programme se comporte de la même façon que l'exemple précédent, mais une forme utilisant des fonctions est préférable puisqu'elle permet une réutilisation plus simple du code déjà écrit : une fonction réalisant un calcul complexe et mise au point pour un programme précis peut sûrement être reprise dans un autre programme plus tard. On voit ici qu'un tableau passé en paramètre d'une fonction est bien une variable d'entrée-sortie : **tab** est utilisé en *entrée* dans la fonction **affichage** (on affiche simplement les valeurs qu'il contient) et en *sortie* dans la fonction **remplissage** (on modifie ces valeurs). Nous reparlerons de cet aspect dans la section 6, mais il est important de noter que toute fonction comportant la modification de valeurs d'un tableau doit être utilisée avec précaution, car ces modifications seront effectives également en dehors de cette fonction.

L'exemple suivant garde l'affichage identique mais remplace le remplissage par une fonction équivalente :

```
String[] remplissage_bis (int n) {
 String[] tab = new String[n];
 for (int i = 0; i < tab.length; i++) {
  println("Entrez la valeur d'index " + i + " : ");
  tab[i] = readString();
 }
 return tab;
}
 ...
void main()
{
 // Déclaration du tableau "noms"
 String[] noms;
 // Remplissage par l'utilisateur
 noms = remplissage_bis(4);
 ...
}
```

On voit ici qu'une fonction peut également renvoyer un tableau. Dans ce cas, une différence notable avec le programme précédent est que la déclaration du tableau **noms** dans la fonction **main** ne comporte pas d'instruction à droite. Cette instruction de *construction* (ou *allocation*) est effectuée dans ce cas par la fonction de remplissage. Une question subsidiaire : quelles seraient les valeurs affichées dans le cas d'un tableau créé mais non rempli ?

Enfin, un dernier exemple pour montrer que les tableaux peuvent avoir autant de dimensions que nécessaire. Voyons ici un programme qui affiche les valeurs d'une matrice **h** de taille $5x3$:

```
void main()
{
 String[][] h = { {"#", " ", "#"},
                  {"#", " ", "#"},
                  {"#", "#", "#"},
                  {"#", " ", "#"},
                  {"#", " ", "#"}};
 for (int i = 0; i < 5; i++) {
  for (int j = 0; j < 3; j++) {
   print(h[i][j]);
  }
  println("");
 }
}
-- Affichage de la sortie --------
# #
# #
###
# #
# #
```

Bien sûr les mêmes règles s'appliquent (notamment l'interdiction de sortir des limites du tableau) ; nous verrons d'autres exemples d'utilisation des matrices dans la section 8.

5.1.1 Exercices du tutoriel

Réaliser les 4 premiers exercices proposés dans le tutoriel en utilisant les informations données ci-dessus.

5.1.2 Exercice : Avec des fonctions

Avec le même énoncé que l'exercice 1 du tutoriel, modifiez votre programme pour faire appel à une fonction **remplir** qui doit remplir un tableau d'entiers avec les dix premiers carrés, et une fonction **affiche** qui affiche un tableau d'entiers. De la même façon, reprendre votre programme de l'exercice 4 en utilisant des fonctions de remplissage et d'affichage.

5.1.3 Exercice : Pendu et Mastermind

L'exercice 5 du tutoriel montre un programme permettant de jouer au pendu, qui utilise un tableau de chaînes de caractères et un tableau d'entiers. Quelques explications/questions supplémentaires :

- `(int) (x * Math.random())` : pour obtenir un entier entre 0 et $x - 1$
- `charAt(i)` : appliqué à une chaîne de caractères, donne le caractère en i-ème position, le premier caractère étant en position 0. Pourquoi l'utilise-t-on alors dans l'instruction `char c = readString().charAt(0);` ?
- pouvez-vous modifier ce programme pour compter le nombre de coups joués, et afficher ce nombre en fin de partie ?

Pour programmer un Mastermind, vous pouvez utiliser :

- un tableau de 4 entiers de 1 à 8 (chaque entier correspondant à une couleur), déterminés aléatoirement au début du jeu par l'ordinateur, pour stocker la combinaison de pions que doit trouver le joueur
- pendant le déroulement de jeu, un tableau de 4 entiers pour stocker la proposition du joueur
- à chaque tour de jeu le programme doit comparer la proposition du joueur et celle de l'ordinateur puis en déduire le nombre de pions de la bonne couleur bien placés et le nombre de pions de la bonne couleur, mais mal placés.
- dans le jeu original le nombre de propositions est limité à 10

5.2 Calculs entiers et réels

A partir du même tutoriel on peut accéder au sujet "Réaliser une calculette d'indice de masse corporelle" et recopier le programme proposé dans l'éditeur. La formule est effectivement fausse, et doit être remplacée par
`imc=poids/(taille*taille)`. La question du type de la variable **imc** mérite quelques expérimentations. Lorsqu'on utilise **double** le résultat est correct, ce qu'on peut vérifier en compilant et en exécutant le programme : cliquer sur console et entrer les données 95 pour le poids et 1.81 pour la taille (et non $1,81$). Un essai avec *int* génère une erreur dont l'intitulé est assez parlant : une possible perte de précision... Cette erreur est en fait due à la division que l'on tente d'appliquer avec un entier comme numérateur (le poids) et un réel comme dénominateur (la taille au carré), le résultat étant un entier.

Si l'on souhaite ne pas en tenir compte, il est possible de *forcer la conversion* entre entiers et réels, grâce à `int imc=poids/(int)(taille * taille);` On convertit donc la taille au carré sous forme d'entier avant d'appliquer la division. Comme prévu le résultat est faux, mais pourquoi ? On peut déjà se rendre compte de l'erreur commise par la conversion en ajoutant une ligne qui affiche la valeur avant et après :

```
println("Taille*taille=" + (taille*taille) + " et (int)(taille*taille)="
  + (int)(taille * taille));
```

Pour 1.81 on obtient 3.2761 dans le cas réel, et 3 dans le cas entier, la valeur réelle est donc **tronquée**. De plus, **on a maintenant un numérateur et un diviseur qui sont des valeurs entières, et dans ce cas, la division appliquée est la division entière** : la valeur obtenue, au lieu d'être $95/3 = 31.6666667$, est encore une fois tronquée à 31.

On trouve à partir de l'onglet "Propositions d'exercices" un exemple de programme réalisé en classe appelé "Expérimenter la différence entre calcul en nombre entiers et flottants". Vous pouvez copier-coller cet exemple dans Javascool pour l'exécuter. On retrouve avec le choix 1 la division entière mentionnée ci-dessus. Le choix 2 permet d'effectuer la division exacte :

```
double res1 = nombre / Math.pow(diviseur, 1) * Math.pow(10, precision);
int res2 = (int) res1;
double resultat = res2 / Math.pow(10, precision);
println("Le resultat approché  est : " + resultat);
// Le langage permet aussi d'écrire directement:
// - double quotient = (double) nombre / diviseur;
// - System.out.printf("Le resultat approché est : %."+precision+"f", quotient);
```

On voit ici les 3 étapes permettant de diviser **nombre** par **diviseur**, avec **precision** chiffres après la virgule :

- calculer $\frac{\text{nombre}}{\text{diviseur}}.10^{\text{precision}}$
- tronquer cette valeur à sa valeur entière
- rediviser cette valeur par $10^{\text{precision}}$ pour obtenir le résultat avec la précision désirée

Comme mentionné en commentaire, on peut obtenir le même résultat grâce à la fonction **printf**, qui permet notamment d'afficher une variable réelle en spécifiant le nombre de décimales. Par exemple on peut afficher un réel x avec deux décimales par `System.out.printf("%.2",x);`, c'est le principe employé ici en remplaçant 2 par la précision souhaitée.

Avant de passer aux exercices, pouvez-vous dire en quoi ces deux méthodes sont quand même *très* différentes ?

5.2.1 Exercice du tutoriel

Le tutoriel "Réaliser une calculette d'indice de masse corporelle" propose un exercice très simple pour ajouter un message donnant une appréciation sur cet indice,

5.2.2 Exercice : Calcul du reste de la division euclidienne

Il existe un moyen très simple de calculer le reste de la division de deux entiers, grâce à l'opérateur `%`, appelé **modulo**. En écrivant par exemple `int r=a%b` où r, a et b sont des entiers, on obtient le reste r. Écrire un algorithme permettant de faire le même calcul *sans utiliser le modulo* à partir de deux entiers a et b saisis au clavier.

5.2.3 Exercice : Calcul de la racine carrée approchée

La racine carrée d'un réel est donnée par la fonction **sqrt** de Javascool, mais on peut aussi calculer une valeur approchée comme à la section 4.3.2. On peut aussi employer la méthode de Héron [9] : intégrer une nouvelle fonction à l'exemple de la section 4.3.2 pour réaliser ce calcul.

Réfléchir ensuite à la façon dont on peut déterminer, pour deux nombres réels a et b, jusqu'à quelle décimale a est une approximation de b. Pour cela on peut appliquer l'algorithme suivant pour écrire une nouvelle fonction prenant a et b en entrée :

1. Soit p le nombre de décimales identiques, fixé au départ à 0. Soit a' le résultat de la soustraction de sa partie entière au réel a, et b' calculé de la même façon avec b.
2. On multiplie a' et b' par 10, et on compare leurs parties entières : si elles sont identiques, alors $p = p + 1$ et on recommence cette étape.
3. Sinon la fonction s'arrête en renvoyant la valeur de p

Un paramètre d'entrée supplémentaire dira jusqu'à quelle décimale on souhaite effectuer ce calcul.

5.3 Calcul binaire

On trouve aussi dans le tutoriel plusieurs sujets dits *unplugged* (sans ordinateur) autour du "codage binaire" et du "codage des nombres". Ces notions sont mises en pratique dans "Un travail sur la conversion binaire/décimale". Dans ce dernier tutoriel on trouve les bases de la numération et du calcul binaire, qui permettent de comprendre comment on passe du binaire au décimal (et inversement). En stockant un nombre binaire dans un tableau de 4 entiers pouvant prendre comme valeur 0 ou 1, on peut effectivement représenter les entiers de 0 (le tableau contiendra les valeurs $0, 0, 0, 0$) à 15 (il contiendra $1, 1, 1, 1$).

Le dernier exercice de ce tutoriel consiste à programmer l'addition de 2 nombres codés sous forme binaire, ce qui oblige en fait à gérer la *retenue* propagée de droite à gauche. En réalité, tout nombre est stocké sous forme binaire dans l'ordinateur. On peut s'en rendre compte en utilisant de nouveau la fonction **printf**, qui autorise un affichage hexadécimal comme dans le code suivant :

```
int x = 1352;
System.out.printf("x en base 10 (entier): %d, en base 10 (réel): %.2f, en hexa: %h",
 x, (double)x, x);
-- Affichage de la sortie --------
x en base 10 (entier): 1352, en base 10 (réel): 1352,00, en hexa: 548
```

La fonction **printf** fonctionne en fait avec le principe du remplacement de motif : les données entre guillemets sont affichées à l'écran, sauf celles commençant par `%` qui sont remplacées par les éléments placés après les guillemets. Ici la variable x est utilisée 3 fois pour être affichée de 3 façon différentes (il en existe d'autres). La forme hexadécimale est traditionnellement utilisée en informatique plutôt que la base 10, notamment pour représenter les adresses en mémoire.

Un nombre en binaire n'étant constitué que de 0 et 1, on peut lui appliquer les opérateurs logiques issus de l'*algèbre de Boole*, notamment la conjonction ("ET") et la disjonction ("OU"). Les tables de vérité [10] donnent le comportement de ces opérateurs qui s'appliquent en Javascool à des entiers binaires : par exemple 0 ET 1 se note `0&1` en Javascool, et est égal à 0 (de la même façon la conjonction se note —). On peut appliquer ces opérateurs à des entiers décimaux, l'opération s'effectuant sur chaque bit composant leur représentation binaire. Par exemple, considérons l'opération `14 & 9` : $(14)_{10}$ et 9_{10} se notent respectivement $(1110)_2$ et $(1001)_2$, par conséquent le résultat est $(1000)_2 = 8_{10}$ puisque de gauche à droite $1\&1 = 1$, $1\&0 = 0$, $1\&0 = 0$ et $0\&1 = 0$

9. `http://fr.wikipedia.org/wiki/Methode_de_Heron`
10. `http://fr.wikipedia.org/wiki/Algebre_de_Boole_(logique)`

Il existe également des opérateurs de *décalage*, qui permettent de "pousser" les chiffres binaires vers la gauche ou la droite, notés respectivement `<<` et `>>`. Par exemple, $9 << 1$ est égal à 18, puisque $9_{10} = (1001)_2$ et que le décalage d'un bit vers la gauche ajoute un bit 0 à droite, ce qui donne $(10010)_2 = (18)_{10}$. On peut de la même façon décaler de plusieurs bits à la fois.

5.3.1 Exercices du tutoriel

Vous devriez, avec les connaissances acquises sur les tableaux (section 5.1), pouvoir réaliser sans problème les 3 exercices.

Ajouter le calcul de la disjonction et de la conjonction avec cette représentation sous forme de tableau.

5.3.2 Exercice : Hexadécimal

Sur le modèle des exercices du tutoriel, écrire les fonctions permettant de passer du décimal à l'hexadécimal, et vérifier les résultats grâce à l'affichage par la fonction **printf**[11]

Généraliser ensuite à n'importe quelle base, qui pourra être spécifiée au clavier au début du programme.

5.3.3 Exercice : Pair ou impair

En utilisant le principe des opérateurs binaires (`&` et —), écrire des fonctions permettant :
- de déterminer si un entier est pair ou impair (en regardant si le bit le plus à gauche est 0 ou 1)
- d'ajouter 1 à un entier si celui-ci est impair, mais ne rien ajouter s'il est pair (sans utiliser de `if`)
- multiplier un entier par 4 (sans utiliser la multiplication)

5.4 Logique et booléens

Un autre sujet concerne "Découvrir quelques notions de logique". On parle ici encore une fois ici d'opérateurs logiques, mais cette fois-ci appliqués à des prédicats tels que ceux manipulés par les instructions conditionnelles de type `if` comme on l'a vu à la section 3.3. Les opérateurs logiques indispensables pour pouvoir réaliser des programmes complexes sont && pour le ET, ||| pour le OU, pour le NON. Les tables de vérité sont les mêmes que l'on manipule des valeurs "logiques" (**vrai** ou **faux**) ou binaires (1 ou 0).

On peut ainsi rencontrer des programmes du type :

```
boolean x = false;
boolean y = (1 < 2);
boolean z = (((!x) && y) || (x && (!y)));
println("x="+x+", y="+y+", z="+z);
 if (z) {println("Z est vraie !!!");}
-- Affichage de la sortie --------
x=false, y=true, z=true
Z est vraie !!!
```

On voit ici la notion de *valeur booléenne*, essentielle en programmation et qui en Javascool peut donc correspondre à **true** ou **false** stockée dans une variable de type **boolean**.

On peut remarquer qu'un **if** accepte uniquement comme prédicat une variable de type **boolean**, le résultat d'une comparaison, ou bien directement une valeur **true** ou **false**. Cela n'a pas de sens par exemple d'écrire dans un programme `if (13)`

5.4.1 Exercices du tutoriel

Les tables de vérité devraient être faciles à retrouver. Les exercices concernant le OU *exclusif* et l'*implication* peuvent être programmés à travers des fonctions prenant en entrée des variables de type **boolean**.

11. Voir aussi `http://fr.wikipedia.org/wiki/Systeme_binaire`

5.4.2 Exercice : Tableaux de booléens

Reprendre les exercices de la section 5.3.1 en manipulant cette fois des tableaux de booléens et non d'entiers. Voyez-vous l'intérêt d'une telle démarche ? Pensez-vous qu'elle pourrait s'appliquer aussi dans le cas du jeu du pendu (section 5.1.3) ?

5.5 Bibliothèque mathématique

L'onglet "Mémo" de Javascool mentionne deux fonctions mathématiques : **pow** et **random** . Il en existe en fait beaucoup plus, qui peuvent être utilisées directement, recensées sur la page `http://download.oracle.com/javase/6/docs/api/java/lang/Math.html`.

Il existe également des classes Java permettant de représenter des grands nombres, notamment **BigDecimal** et **BigInteger**, pouvant être utilisées si la précision requise par le programme est très grande.

En guise d'exercice, vous pouvez après le lancement de Javascool choisir l'activité "algoDeMaths", et voir dans le tutoriel des activités autour du calcul de surfaces par tirage aléatoire. Où l'on voit que la fonction **random** a beaucoup d'applications...

5.6 Caractères et chaînes de caractères

Dans le tutoriel, accéder au sujet *unplugged* appelé "Le codage numérique ASCII". On y trouve notamment la table des caractères ASCII, qui établit la correspondance entre un caractère et un entier qui lui est associé : on voit par exemple que le caractère 'A' a le code 65, 'B' le code 66, etc.

On peut facilement afficher le code ASCII d'un caractère en utilisant de nouveau la fonction **printf** :

```
 char c = 'A';
 int x = 68;
 System.out.printf("Caractère c=%c ou %d", c, (int)c);
 println("");
 System.out.printf("Entier x=%d ou %c", x, (char)x);
 x = (int)c;
 c = (char)x;
-- Affichage de la sortie --------
Caractère c=A ou 65
Entier x=68 ou D
```

Cet exemple montre que les types **int** et **char** sont très proches, et qu'il est très facile de les convertir entre eux : le type **char** est en fait un entier "court" non-signé, stocké sur un seul octet et pouvant donc représenter des valeurs entières entre 0 et 127.

L'organisation du code ASCII fait qu'il est simple de tester si un caractères est une majuscule, étant donné que les codes des lettres 'A', 'B', ... se suivent jusqu'à 'Z', grâce au code suivant :

```
 char c = '8';
 if ((c < 'A') || (c > 'Z')) {
  println("Ce n'est pas une majuscule !");
 }
-- Affichage de la sortie --------
Ce n'est pas une majuscule !
```

On peut tester de la même façon si un caractère appartient à l'intervalle 'a'...'z', ou bien '0'...'9'

On a vu dans la partie 5.1 que les *chaînes de caractères* étaient en fait des tableaux de caractères, auxquels on peut accéder grâce à la fonction **charAt(i)** qui renvoie le $-i$ème caractère (attention, se rappeler que le premier caractère est en position 0). On a vu également à travers plusieurs exemples la *concaténation*, utilisée principalement pour l'affichage mais qui peut aussi permettre de créer une chaîne à partir de plusieurs chaînes, par exemple :

```
 String prenom = "Benoit";
 String message = "Bonjour " + prenom + ", ça va ?";
```

Encore quelques exemples (mais il existe de nombreuses autres fonctions applicables au type **String**[12]) :

- **indexOf(char c)** retourne la position de la première occurrence du caractère **c** dans la chaîne : par ex. `"bonjour".indexOf('o')` renvoie 1
- **replace(char oldChar, char newChar)** renvoie une nouvelle chaîne où toutes les occurrences de **oldChar** sont remplacées par **newChar** : par ex. `"bonjour".replace('o','i')` renvoie **"binjiur"**
- **substring(int beginIndex, int endIndex)** renvoie la *sous-chaîne* allant de **beginIndex** à **endIndex** (exclus) : par ex. `"bonjour".substring(1, 5)` renvoie **"onjo"**
- **toLowerCase()** et **toUpperCase()** renvoient une nouvelle chaîne contenant la chaîne initiale convertie en minuscules ou majuscules.

5.6.1 Exercice : retour sur le pendu

Reprendre le jeu du pendu vu dans la partie 5.1.3, en ajoutant l'interdiction de jouer 2 fois la même lettre. Pour cela on pourra créer un tableau de 128 booléens indiquant, pour chaque valeur du code ASCII, si le caractère correspondant a déjà été joué ou non.

5.6.2 Exercice : Conversions

Sans utiliser de fonction déjà définie dans Javascool, êtes-vous capable de convertir une majuscule en son équivalent en minuscule ? Sur le même principe comment peut-on convertir le caractère '8' en son équivalent entier ? Comment convertir une chaîne de caractères contenant des chiffres en un entier ? (par ex. **"1234"** sera converti en 1234)

5.6.3 Exercice : Vérification de données

Écrire un programme où l'utilisateur saisit dans une chaîne de caractères une date de naissance de la forme $JJ/MM/AAAA$, puis vérifier que la date est bien sous cette forme (on ne vérifiera pas sa validité, par ex. 31/02/1234 sera acceptée).

Sur le même principe vérifier une chaîne contenant une adresse e-mail, qui doit être de la forme :

- identifiant principal qui peut avoir n'importe quelle forme (par ex. *benoit.crespin* ou *bcrespin*)
- séparateur ''
- nom de domaine, de la forme **serveur.suffixe** (par ex. *unilim.fr*)

6 Aspects avancés

6.1 Structures et classes

La structure en algorithmique (ou *type composé*, ou encore *enregistrement*) est un type défini par l'utilisateur, permettant de regrouper au sein d'une même entité un ensemble de types de base. Un type *structuré* peut donc être utilisé pour stocker des données évoluées, par exemple on pourra définir le type **cartegrise** par :

```
structure carte_grise {
 nom : chaine de caractère
 prenom : chaine de caractère
 marque : chaine de caractère
 modele : chaine de caractère
 immatriculation : chaine de caractère
 annee_mise_en_service : entier
 puissance_fiscale : entier
}
```

La traduction en Javascool de cet exemple est directe grâce au mot-clef **class** :

12. `http://download.oracle.com/javase/6/docs/api/java/lang/String.html`

```
class cartegrise {
 String nom, prenom, marque, modele, immatriculation;
 int annee_mise_en_service, puissance_fiscale;
}
```

Pour utiliser une variable d'un type structuré, il est nécessaire de la créer avec le mot-clef **new** :

```
void main()
{
 cartegrise cg = new cartegrise();
 cg.nom = "Smith";
 cg.prenom = "Bill";
 ...
 println(cg);
}
-- Affichage de la sortie --------
my_class$cartegrise@3580ab
```

On voit ici le problème de l'affichage : Javascool n'est pas capable de déterminer ce que l'utilisateur souhaite exactement, il se contente d'afficher l'adresse de la variable en mémoire. Heureusement un type structuré peut s'accompagner de fonctions diverses, utilisables ensuite à la manière du type **String**, notamment pour l'affichage. On peut aussi bien sûr définir des fonctions en dehors du bloc **class** :

```
class triangle {
 double cote1, cote2, cote3;
 void saisie() {
  println("Saisir les valeurs des 3 cotés du triangle");
  cote1 = readDouble();
  cote2 = readDouble();
  cote3 = readDouble();
 }
 public String toString() {
  return ("Triangle de cotés "+cote1+", "
   +cote2+", "+cote3);
 }
 boolean estEquilateral () {
  return ((cote1 == cote2) && (cote2 == cote3));
 }
}

boolean sontIdentiques (triangle t1, triangle t2) {
 return ((t1.cote1 == t2.cote1) && (t1.cote2 == t2.cote2) && (t1.cote3 == t2.cote3));
}

void main()
{
 triangle t = new triangle();
 t.saisie();
 println(t);
 if (t.estEquilateral()) {
  println("Ce triangle est equilateral");
 }
 triangle u = t;
 println("u et t identiques ? "+sontIdentiques(t,u));
}
```

La fonction **toString** a une signature particulière qu'il faut respecter exactement comme sur l'exemple : c'est cette forme qui permet ensuite de pouvoir afficher les informations que l'on souhaite sur la variable

t de type **triangle** grâce à la fonction **println**. On verra dans les exercices qu'on peut définir des types structurés qui utilisent d'autres types structurés, des tableaux de types structurés, etc.

On peut noter que nous sommes ici dans le formalisme *objet*, qui sous-tend la mise en oeuvre du langage Java qui est derrière Javascool. Néanmoins nous n'irons pas au-delà, les notions avancées comme celle-ci étant plutôt abordées en général dans les cursus universitaires à partir de la 2ème ou 3ème année.

6.1.1 Exercice : triangles

Ajouter des fonctions (définies dans le bloc **class** ou non) pour :
- déterminer si un triangle est isocèle
- calculer le périmètre d'un triangle

Ajouter dans le programme principal des instructions pour tester ces fonctions.

6.1.2 Exercice : Date de naissance

Définir un type **datenaissance** composé de 3 entiers J, M et A et des fonctions pour :
- saisir et afficher une date de naissance
- tester si une date est avant une autre dans le temps (par exemple le 31/12/2011 est avant le 16/3/2012)
- tester si une date de naissance est valide, en excluant par exemple le 31/02/2012 (attention, l'algorithme est un peu compliqué [13])

Définir dans le programme principal un tableau de 5 dates à faire saisir par l'utilisateur en vérifiant leur validité, puis déterminer laquelle est la plus ancienne.

6.2 Sauvegarder des données dans un fichier

Il est possible de sauvegarder des données à tout moment pendant l'exécution d'un programme. Ceci peut se produire dans plusieurs cas de figures, par exemple :
- le programme réalise des calculs complexes, qu'on veut pouvoir utiliser ensuite avec un autre logiciel (typiquement des séries de données qu'on visualise de façon graphique avec un logiciel tel que *Gnuplot* [14], *Matlab* [15], ...)
- dans un jeu on souhaite conserver la position du joueur à un instant donné, s'il souhaite reprendre la partie plus tard
- toujours dans un jeu on souhaite conserver les meilleurs scores des joueurs
- le programme propose des messages adaptés à la langue de l'utilisateur, en stockant dans des fichiers les traductions de tous les messages dans différentes langues

Outre les opérations décrites ci-dessous, le lecteur intéressé par le sujet pourra aussi consulter les fonctions mises à disposition depuis la version 4 de Javascool sur `http://javascool.gforge.inria.fr/index.php?page=api&api=org/javascool/tools/FileManager.html`

Le mécanisme de sauvegarde de données proposé ici simplifie au maximum les choses, bien d'autres choses sont réalisables. On ne considérera ici par exemple que les sauvegardes au format *texte*, d'autres types de sauvegardes seront abordés plus tard avec les images.

```
import java.io.*;
void main ()
{
 try {
  PrintWriter fichier = new PrintWriter(new FileWriter("monfichier.txt"), false);
  fichier.println("Hello");
  fichier.println(1234);
  fichier.close();
 }
 catch(Exception e) {}
}
```

13. `http://fr.wikibooks.org/wiki/Structures_de_donnees/Dates`
14. `http://www.gnuplot.info/`
15. `http://fr.wikipedia.org/wiki/MATLAB`

Ce programme permet de sauvegarder, dans un fichier appelé **monfichier.txt**, plusieurs lignes de texte. Après l'exécution (qui n'affiche rien à l'écran), ce fichier qu'on peut visualiser avec un logiciel de type *Wordpad* ou *Notepad* contient :

```
Hello
1234
```

C'est donc un fichier ne contenant que du texte. Plusieurs lignes du programme méritent des explications :

- la ligne `import java.io.*;` indique au compilateur qu'on souhaite utiliser les fonctions spécifiques pour la gestion des fichiers en Java
- la ligne `PrintWriter fichier = new PrintWriter(..., false);` sert à "ouvrir" le fichier souhaité, avec en spécifiant avec **false** que si ce fichier existe déjà les données qu'il contenait seront perdues. Au contraire, si ce booléen a la valeur **true** cela indique que, si le fichier existe déjà, les données sauvegardées s'ajouteront automatiquement après.
- grâce à `println` on écrit des lignes de texte dans le fichier, comme on utilisait déjà cette fonction pour afficher du texte à l'écran
- la fonction `close` qui "ferme" le fichier n'a pas une grande utilité dans cet exemple. Elle est par contre obligatoire si on souhaite, plus tard pendant l'exécution du programme, ré-ouvrir le fichier.
- on observe l'apparition d'un bloc composé des lignes `try {` et `catch(Exception e) {}`, qui sans rentrer dans les détails servent à traiter les cas d'erreurs éventuelles. Ce bloc est obligatoire et doit entourer toutes les opérations sur les fichiers.

Les erreurs pouvant se produire concernent par exemple le nom choisi pour le fichier et le dossier où il sera sauvegardé sur le disque dur. Dans notre exemple, le nom fourni est donné sans indication de dossier : sous Linux il sera sauvegardé à la racine du compte de l'utilisateur, en général `/home/utilisateur`, sous Windows ce sera plutôt `C:\Users\utilisateur\Desktop\`

On peut aussi spécifier le dossier, par exemple le dossier `/tmp` sous Linux :

```
PrintWriter fichier = new PrintWriter(new FileWriter("/tmp/monfichier.txt"), false);
```

Sous Windows on utilisera plutôt `"C:/monfichier.txt"`, ou bien `"F:/monfichier.txt"` pour sauvegarder sur une clef USB, etc.

Le problème ensuite est de savoir comment lire des données à l'intérieur d'un fichier texte. Deux cas de figure se présentent selon qu'on connaît exactement le contenu du fichier, ou bien qu'il existe des inconnues, par exemple le nombre de lignes dans le fichier. Si le nombre de lignes est connu, on peut écrire un programme de ce type :

```
try {
 BufferedReader fichier2  = new BufferedReader(new FileReader("monfichier.txt"));
 String ligne;
 ligne = fichier2.readLine();
 println(ligne);
 ligne = fichier2.readLine();
 println(ligne);
 fichier2.close();
}
catch(Exception e) {}
```

Ceci va afficher à l'écran les deux lignes de **monfichier.txt**. La fonction **readLine** lit les données ligne par ligne, ce qui implique de bien séparer les données ligne par ligne lors de la sauvegarde.

Si le nombre de lignes est inconnu, on aura plutôt :

```
 BufferedReader fichier2  = new BufferedReader(new FileReader("monfichier.txt"));
 String ligne;
 ligne = fichier2.readLine();
 while(ligne != null) {
  println(ligne);
  ligne = fichier2.readLine();
 }
 fichier2.close();
```

Ici on comprend que la fonction **readLine** renvoie la valeur **null** s'il n'y a plus de données à lire dans le fichier : ainsi on peut utiliser une boucle qui lit chaque ligne, affiche cette ligne à l'écran puis tente de lire une nouvelle ligne.

6.2.1 Exercice : Création d'un fichier de carrés

Reprendre les spécifications de l'exercice 5.1.2 et, au lieu d'afficher les valeurs à l'écran, les entrer dans un fichier **carres.txt**. Améliorer le programme pour proposer à l'utilisateur de rentrer le nom du fichier qu'il souhaite. La fonction **Integer.valueOf**, appliquée à une chaîne de caractères, renvoie l'entier qu'elle contient : utiliser cette fonction pour écrire un autre programme capable de lire le contenu d'un tel fichier.

6.2.2 Exercice : Lecture d'un fichier structuré

On considère le fichier suivant, contenant sur la première ligne le nombre des lignes suivantes :

```
6
Assam
Bengale occidental
Bihar
Gujarat
Karnataka
Kerala
```

Écrire un programme permettant de lire ces données, qui doit fonctionner quel que soit le nombre de lignes.

6.2.3 Exercice : High-scores

Écrire une fonction qui prend en entrée deux chaînes de caractères **nomFichier** et **nomJoueur**, et un entier **score**. On considère que chaque ligne dans **nomFichier** contient un nom de joueur puis son score sur la ligne suivante, par exemple :

```
Benoit
1455
Henri
1434
Paul
1388
...
```

La fonction devra simplement afficher si **nomJoueur** mérite de rentrer dans le classement en fonction de son **score** ou non. Améliorer ensuite en mettant à jour le classement.

6.3 Données partagées, copies et effets de bord

On appelle généralement "effet de bord" le fait qu'une fonction modifie une ou plusieurs variables qui lui sont passées en paramètre, sans que cela soit forcément intuitif pour le programmeur. Ainsi, les effets de bord peuvent entraîner un fonctionnement inattendu du programme (on aurait donc pu placer ces considérations dans la partie 4).

Prenons d'abord le cas d'une fonction qui ne modifie pas ses arguments :

```
void f(int x) {
  x = 2;
}
void main () {
  int a = 1;
  println("a = " +a);
  f(a);
```

```
  println("a = " +a);
}
-- Affichage de la sortie --------
a = 1
a = 1
```

Ici la fonction **f** prend en paramètre un entier dont elle modifie la valeur, mais cette modification n'entraîne pas d'effet de bord puisqu'on voit que la variable **a**, dans le programme principal, conserve sa valeur initiale. Ceci s'explique par le fait que, quand la fonction **f** commence son exécution, le contenu de la variable **a** est d'abord dupliqué dans une nouvelle variable **x**. Ce comportement est donc logique, et vaut pour les *types de base* : **int**, **double** et **String**.

Pourtant, nous avons déjà vu que le comportement n'est pas le même lorsqu'on manipule des tableaux (voir section 5.1) :

```
void f(int tab[]) {
  tab[0] = 17;
}
void affichage(int tab[]) {
 for (int i = 0; i < tab.length; i++)
  print(tab[i]+" ");
 println("");
}
void main ()
{
  int a[] = {12, 7, 89, 32};
  affichage(a);
  f(a);
  affichage(a);
}
-- Affichage de la sortie --------
12 7 89 32
17 7 89 32
```

On voit ici que la modification amenée par la fonction **f** est permanente : lorsqu'on affiche de nouveau le tableau à l'intérieur de la fonction **main**, la modification est visible. Même si cela donne plus de souplesse au programmeur, il faut être très vigilant pour déterminer si l'appel à une fonction ne risque pas de modifier des données de façon inconsidérée. Ceci vaut aussi pour une variable de *type structuré*, comme dans l'exemple suivant.

```
class triangle {
 double cote1, cote2, cote3;
 public String toString() {
  return ("Triangle de cotés "+cote1+", "
   +cote2+", "+cote3);
 }
}
void f(triangle x) {
  x.cote1 = 2;
}
void main ()
{
  triangle a = new triangle();
  a.cote1 = 10; a.cote2 = 15; a.cote3 = 7;
  println(a);
  f(a);
  println(a);
}
```

```
-- Affichage de la sortie --------
Triangle de cotés 10.0, 15.0, 7.0
Triangle de cotés 2.0, 15.0, 7.0
```

Là encore, la modification apportée dans la fonction **f** est permanente et irréversible, donc méfiance !

Le même type de comportement peut être observé avec l'emploi de *variables globales*, c'est-à-dire des variables "partagées" par l'ensemble du programme :

```
int a;
void f() {
  a = 2;
}
void main () {
  a = 1;
  println("a = " +a);
  f();
  println("a = " +a);
}
-- Affichage de la sortie --------
a = 1
a = 2
```

Ici la variable **a** est déclarée en dehors de toute fonction, au début du programme. Par conséquent toute fonction peut afficher ou modifier sa valeur, ce que fait la fonction **f**. Cette fois-ci, à la différence du premier exemple montré ci-dessus, on voit que la modification est bien prise en compte.

6.3.1 Exercice : Trouvez les erreurs

On souhaite afficher à l'écran une forme composée d'étoiles :

```
*
**
***
****
*****
******
...
```

Voici un programme, qui comme vous pourrez le constater en faisant un copier-coller dans Javascool, ne fonctionne pas très bien... Comment le modifier ? Quels sont les effets de bord à l'oeuvre ici ?

```
// Inspiré par http://h-deb.clg.qc.ca/
int MAX_LIGNES = 10;
int Cpt_Lignes, // indique la ligne en train d'être dessinée
    Max_Car,    // indique le max. de caractères sur la ligne
    Cpt_Car;    // indique le caractère en train d'être dessiné
void Dessiner_Ligne () {
   Max_Car = Cpt_Lignes;
   Cpt_Lignes = 1;
   while (Cpt_Car <= Max_Car) {
      print("*");
      Cpt_Car++;
   }
   Cpt_Lignes++;
}
void main () {
   Cpt_Lignes = 1;
   while (Cpt_Lignes <= MAX_LIGNES) {
```

```
      Dessiner_Ligne ();
      println("");
      Cpt_Lignes++;
   }
}
```

6.3.2 Exercice : Enchaînement de conditions

```
// Inspiré de http://www-etud.iro.umontreal.ca/~blondimi/
class Entier {
 int valeur;
}
boolean Affecter(Entier a, int b) {
 boolean egaux = (a.valeur == b);
 a.valeur = b;
 return egaux;
}
void main() {
 Entier un = new Entier();
 Entier deux = new Entier();
 un.valeur = 1;
 deux.valeur = 2;
 if (Affecter(un, 3) && Affecter(deux, 4)) {
  deux.valeur = 3;
 }
 println("un="+un.valeur);
 println("deux="+deux.valeur);
}
```

Quel doivent être les valeurs affichées ici ? Même question si l'on remplace le symbole `&&` par `||` ? En déduire le comportement de ces opérateurs si le premier test a la valeur **false**

6.3.3 Exercice : plusieurs variables avec le même nom

```
int a;
void f() {
  int a;
  a = 2;
}
void main () {
  a = 1;
  println("a = " +a);
  f();
  println("a = " +a);
}
```

Expliquer ce qui doit s'afficher à la fin de ce programme et pourquoi ?

6.4 Activités Javascool pour le dessin 2D

Jusqu'ici la plupart de nos programmes fonctionnent en mode "console", c'est-à-dire avec de entrées au clavier et des sorties sous forme de texte affiché à l'écran. Pourtant il paraît évident qu'une interface utilisateur évoluée est plus intéressante à manipuler pour l'utilisateur, et plus motivante à programmer : les entrées peuvent être des actions liées à la souris (boutons à cliquer, cases à cocher, etc), et les sorties peuvent être des dessins géométriques comme dans la section 7.4, des messages s'affichant dans de nouvelles fenêtres, ...

L'activité "tortueLogo" définit des fonctions très simples permettant de dessiner à l'aide d'une petite tortue qui peut se déplacer dans un écran de taille 512x512, munie d'un stylo qu'elle peut abaisser ou relever. Avec les fonctions utilisables [16], on peut par exemple dessiner une magnifique spirale comme sur la figure 3 :

```
// Inspiré de
// http://en.wikipedia.org/wiki/Logo_(programming_language)
void spirale (double taille) {
 if (taille < 40) {
  forward(taille);
  rightward(15);
  spirale(taille*1.02);
 }
}
void main() {
 clear_all();
 set_background(9);
 set_color(0);
 pen_up();
 set_position(256,256);
 pen_down();
 spirale(10);
}
```

Le fait de pouvoir combiner des primitives de dessin très simples avec la puissance d'expression du langage Java permet souvent d'illustrer de manière plus motivante l'application réelle de l'algorithmique, comme dans l'exemple de la spirale qui utilise la récursivité. Nous reviendrons sur cet aspect dans les exercices. On voit aussi dans l'exemple de l'onglet "Documents de la proglet" que la tortue Logo peut être utilisée comme un classique grapheur de fonctions mathématiques.

L'activité "codagePixels" permet de manipuler des images et leur appliquer différents algorithmes, mais également de dessiner à l'écran. La tortue Logo pouvait se déplacer dans un espace de taille fixe, ici on pourra définir ses propres dimensions (inférieures à 500x500). L'onglet "Aide de la proglet" montre les différentes opérations disponibles, que nous allons utiliser tout d'abord pour du dessin :

```
void main() {
   int nb = 1500;
   reset(256, 256);
   int w = getWidth();
   int h = getHeight();
   int[] x = new int[nb];
   int[] y = new int[nb];
   int i;
   for (i = 0; i < nb; i ++) {
      x[i] = random( - w, w);
      y[i] = random( - h, h);
   }
   while (true) {
      for (i = 0; i < nb; i ++) {
         x[i] += random( - 3, 3);
         y[i] += random( - 3, 3);
         if ( ! setPixel(x[i], y[i], 0)) {
            x[i] = random( - w, w);
            y[i] = random( - h, h);
         }
      }
```

16. http://javascool.gforge.inria.fr/v3/api/proglet/tortuelogo/TortueLogo.html

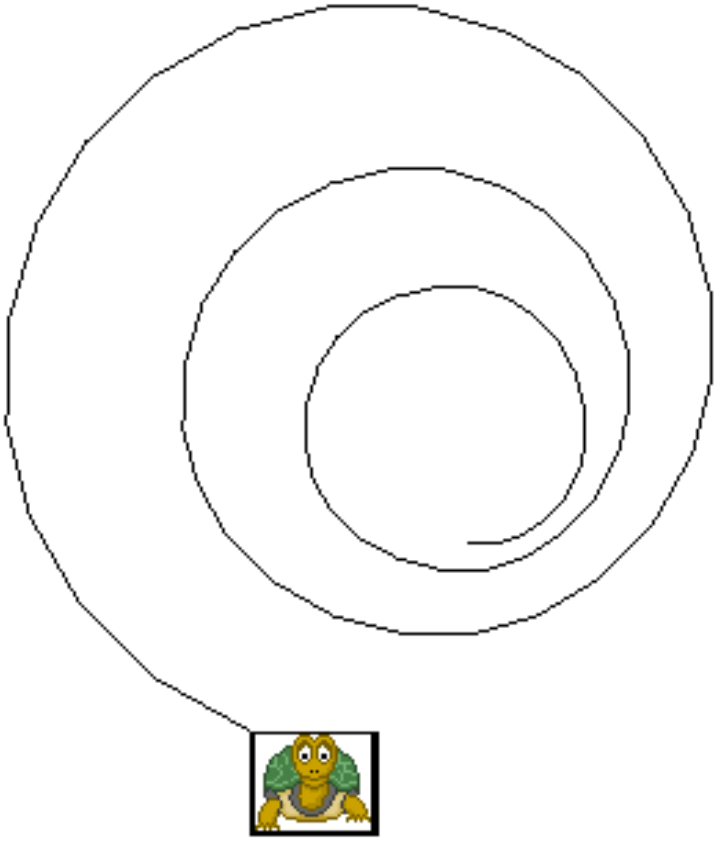


FIGURE 3 – Spirale obtenue avec la proglet “Logo”

```
      sleep(100);
      reset(256, 256);
   }
}
```

Si vous exécutez cet exemple vous verrez 1500 petits points qui suivent une sorte de mouvement brownien. La zone de dessin est fixée à la taille 512x512 puisqu'ici hauteur et largeur ne correspondent qu'à la moitié. Les deux tableaux utilisés pour stocker les points sont initialisés puis dans une boucle infinie, déplacés de façon aléatoire. On laisse les points s'afficher pendant 100 millisecondes, puis le dessin est effacé avant de recommencer. La fonction **setPixel** qui permet de dessiner un point renvoie un booléen indiquant si on est sorti des limites de l'écran, ce qui permet éventuellement de redonner une position aléatoire au point.

Les notions présentées en bas de page de cette même activité montrent la possibilité de pouvoir charger *n'importe quelle image trouvée sur le web, au format JPG et de taille inférieure à* $512x512$, et plutôt en niveaux de gris (*ie* pas d'images couleur). On peut ensuite appliquer des opérations à cette image en la considérant comme une matrice 2D : en effet les éléments de l'image, appelés *pixels*, sont en fait les cases d'une matrice dont les valeurs sont des intensités de gris, allant de 0 (noir) à 255 (blanc). Les explications données dans ce tutoriel permettent de comprendre comment on peut récupérer le niveau de gris pour un pixel donné avec **getPixel**, modifier cette valeur puis la ré-injecter dans l'image avec **setPixel**. Cette approche permet d'implémenter toutes sortes de filtres (inversion vidéo, flou, emboss). Voici un autre exemple permettant d'appliquer une symétrie à l'image verticale à une image (les pixels de gauche passent à droite et inversement).

```
void main() {
   load("http://farm3.static.flickr.com/2472/3988198655_d8b12c2328.jpg");
   int w = getWidth(), h = getHeight();
   int i, j;
   sleep(1000);
   // On attend un peu avant d'appliquer la symétrie
    for (i = - w; i < 0; i ++) {
      for (j = - h; j < h; j ++) {
         int val1 = getPixel(i, j);
         int val2 = getPixel( - i, j);
         setPixel(i, j, val2);
         setPixel( - i, j, val1);
      }
   }
}
```

Cette approche peut parfois se révéler un peu limitée, car toute modification sur l'image est immédiate. Pour certaines opérations on devra utiliser une matrice d'entiers de taille wxh pour stocker les résultats de façon temporaire.

6.4.1 Exercice : Logo

Programmer et tester une fonction pour dessiner un rectangle de dimensions nxm, en partant du coin supérieur gauche situé à la position (x, y). Sur le même principe écrire une fonction permettant de dessiner un cercle de rayon r et de centre (x, y). Utiliser ces fonctions pour dessiner un terrain de football [17]

6.4.2 Exercice : Dessiner sur une image

L'activité "paintBrush" est plus simple que l'activité "codagePixels" car elle se concentre sur le dessin. On y trouve un petit programme de dessin de type *Paint*, avec un mode démo dans lequel on peut tracer des points, des lignes et des rectangles, effacer des pixels, etc. L'activité consiste à reprogrammer les

17. Ou d'autres sports bien sûr, pour le football se référer à
http://fr.wikipedia.org/wiki/Fichier:Soccer_Field_Transparant.svg

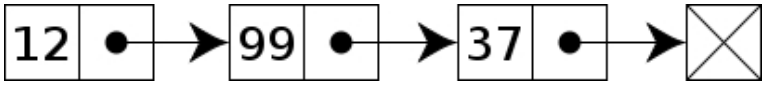


FIGURE 4 – Exemple de liste chaînée d'entiers

fonctions de tracé, de remplissage et de "gommage". Les plus audacieux pourront essayer, pour le tracé d'une ligne, d'appliquer l'algorithme de Bresenham [18].

6.4.3 Exercice : Manipulation d'images

Reprendre le dernier tutoriel sur la manipulation d'images et implémenter :
- le "signe de la paix" décrit dans l'onglet "Aide de la proglet"
- la "postérisation" d'une image, qui consiste à faire passer une image en noir et blanc en appliquant le critère suivant : tout pixel d'intensité inférieure à 128 devient noir, ou bien blanc sinon [19]
- les symétries horizontale et diagonale

7 Manipulation de listes, d'arbres et de graphes

On appelle en général structures de données "évoluées" des objets algorithmiques construits à partir des types de base et permettant de stocker des données de façon adaptée. Nous allons voir dans cette section des exemples de telles structures, et pourquoi il est souvent plus simple de les utiliser plutôt que de re-développer ses propres structures.

7.1 Structures chaînées et arborescentes

Il est classique en informatique de faire programmer des listes chaînées, sur le modèle de la figure 4. On considère en fait une suite de *noeuds*, composés d'un ou plusieurs champs contenant des données (ici un entier), et d'un champ pointant sur le noeud suivant. Il est nécessaire de conserver aussi une référence vers le premier noeud de la liste, comme avec l'implémentation ci-dessous :

```
class Noeud {
 int valeur;
 Noeud suiv;
}
class Liste {
 Noeud premier = null;
 void ajouterEnTete(int v) {
  Noeud n = new Noeud();
  n.valeur = v;
  n.suiv = premier;
  premier = n;
 }
 public String toString() {
  Noeud cour = premier;
  String s = "[";
  while (cour != null) {
   s += " "+cour.valeur;
   cour = cour.suiv;
  }
  return (s+" ]");
 }
}
void main() {
```

18. http://fr.wikipedia.org/wiki/Algorithme_de_trace_de_segment_de_Bresenham
19. Voir par exemple http://farm3.static.flickr.com/2800/4033907723_e52cdea975.jpg

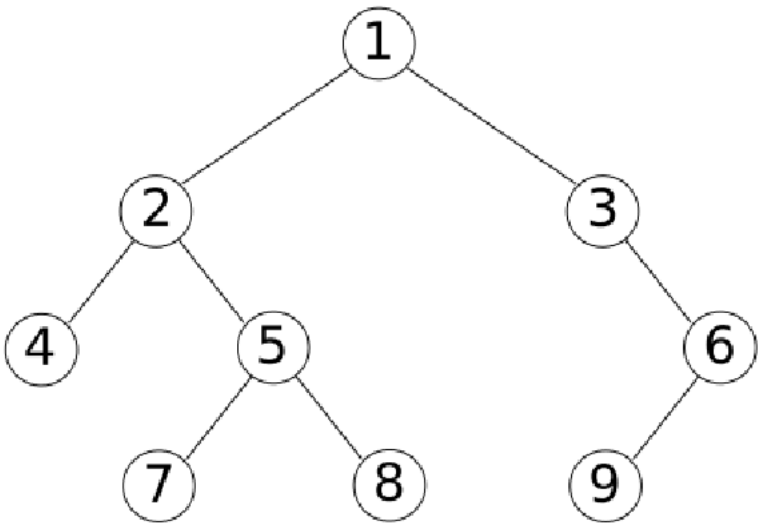


FIGURE 5 – Exemple d'arbre binaire

```
 Liste l = new Liste();
 l.ajouterEnTete(37); println(l);
 l.ajouterEnTete(99); println(l);
 l.ajouterEnTete(12); println(l);
}
-- Affichage de la sortie --------
[ 37 ]
[ 99 37 ]
[ 12 99 37 ]
```

Cet exemple, qui utilise les structures décrites à la section 6, montre une implémentation minimale dans laquelle les deux seules opérations sont l'insertion d'un nouveau noeud en tête de liste, et l'affichage d'une liste. On peut remarquer le mot-clef **null**, utilisé en début de programme pour exprimer le fait que le premier élément n'existe pas encore. On retrouve **null** pour arrêter le parcours de la liste lors de l'affichage. Lors de cette opération on peut également remarquer l'utilisation d'une variable **cour** pour parcourir la liste, évitant ainsi de modifier la variable **premier** qui elle ne doit surtout pas être modifiée ; en effet dans ce cas on perdrait tout simplement la référence au premier élément de la liste.

Le principe des noeuds comportant une ou plusieurs références vers d'autres noeuds peut être utilisé dans de nombreux autres cas, tels que des graphes ou les arbres. Par exemple, un arbre binaire est un arbre dans lequel chaque noeud a au plus deux fils, comme sur la figure 5. Voici une petite implémentation :

```
class Arbre {
 int valeur;
 Arbre gauche, droit;
 // Parcours prefixe
 public String toString() {
  String s = "[ " + valeur + " ";
  if (gauche != null) {
   s += " " + gauche + " ";
  }
  if (droit != null) {
   s += droit + " ";
  }
  return s + "]";
 }
}
Arbre creerArbre(int v, Arbre g, Arbre d) {
```

```
 Arbre a = new Arbre();
 a.valeur = v;
 a.gauche = g;
 a.droit = d;
 return a;
}
Arbre creerFeuille(int v) {
 return creerArbre(v, null, null);
}
void main() {
 Arbre a =
  creerArbre(1,
   creerArbre(2, creerFeuille(4),
                 creerArbre(5, creerFeuille(7),
                               creerFeuille(8))),
   creerArbre(3, null,
                 creerArbre(6, creerFeuille(9),
                               null)));
 println(a);
}
-- Affichage de la sortie --------
[ 1  [ 2  [ 4 ] [ 5  [ 7 ] [ 8 ] ] ] [ 3 [ 6  [ 9 ] ] ] ]
```

Ici la structure **Arbre** est munie de l'opération **toString** utilisée pour l'affichage, et qui implémente le parcours préfixe de l'arbre : on affiche d'abord la racine, puis le sous-arbre gauche, puis le sous-arbre droit. Encore une fois le mot-clef **null** symbolise le fait qu'un noeud peut ne pas avoir de fils à gauche ou à droite.

7.1.1 Exercice : listes chaînées

En vous inspirant éventuellement de `http://fr.wikipedia.org/wiki/Liste_chainee`, implémenter et tester les opérations suivantes sur une liste :

- déterminer si la liste est vide (la fonction renverra un booléen)
- calculer le nombre d'éléments dans la liste
- supprimer le premier élément
- ajouter un élément après un élément donné

7.1.2 Exercice : arbres binaires

Implémenter et tester les parcours infixes et postfixes d'un arbre binaire [20]. Implémenter ensuite un arbre binaire de recherche [21], qui est un arbre binaire particulier tel que la valeur à la racine est supérieure aux valeurs stockées dans le sous-arbre gauche, mais inférieure à celles du sous-arbre droit (cette règle est valide aussi pour les sous-arbres).

7.2 Collections : ArrayList, HashSet, HashMap

Il existe en Javascool de nombreuses implémentations des listes, arbres, etc. appelées *collections*. Nous allons voir ici 3 structures qui permettent de les manipuler sans avoir à les reprogrammer.

La structure **ArrayList** est une alternative aux listes chaînées vues à la section précédente et aux tableaux vus à la section 5.1, qui permet de stocker des valeurs dans une liste sans se préoccuper de leur nombre. En effet cette structure a comme propriété de pouvoir adapter sa taille aux données manipulées. Dans l'exemple qui suit on manipule une liste d'entiers (qu'il faut obligatoirement nommer "Integer"), mais tous les types peuvent être utilisés :

20. `http://fr.wikipedia.org/wiki/Arbre_binaire`
21. `http://fr.wikipedia.org/wiki/Arbre_binaire_de_recherche`

```
void main() {
 ArrayList<Integer> maListe = new ArrayList<Integer>();
 for (int i = 0; i < 10; i++) {
  maListe.add(random(15,20));
 }
 println(maListe);
 println("Valeur stockée en position 0: "+ maListe.get(0));
 println("1re apparition de la valeur 17 :"+ maListe.indexOf(17));
 println("Derniere apparition de 17 :"+ maListe.lastIndexOf(17));
 maListe.clear();
}
-- Affichage de la sortie --------
[19, 20, 16, 19, 17, 19, 16, 15, 20, 17]
Valeur stockée en position 0: 19
1re apparition de la valeur 17 :4
Derniere apparition de 17 :9
```

Ici la taille de la liste n'est pas fixée initialement, néanmoins on peut lui ajouter ensuite 10 valeurs (tirées au hasard entre 15 et 20). On voit que l'on peut facilement :

- ajouter une valeur (**add**) qui doit être du type spécifié, qui s'ajoute à la fin de la liste
- afficher le contenu de la liste directement grâce à **println**
- savoir quelle valeur est stockée à une position donnée (**get**) : ici on s'intéresse à la position 0, mais il faut comme avec les tableaux faire attention à ne pas essayer d'accéder à une position qui n'existe pas
- savoir à quelle position apparaît une valeur donnée, soit sa première apparition (**indexOf**) soit la dernière (**lastIndexOf**)
- effacer le contenu de la liste (**clear**)
- d'autres instructions ne sont pas montrées ici (notamment **size** et **remove**)

La structure **HashSet** est un peu différente : elle permet de stocker un ensemble de valeurs *sans permettre les doublons*. Elle est donc bien adaptée pour stocker des valeurs uniques, par exemple la table périodique des éléments, les pays du monde, les mois de l'année, ... L'exemple ci-dessous montre un **HashSet** stockant des chaînes de caractères :

```
import java.util.HashSet;
void main() {
 HashSet<String> monEnsemble = new HashSet<String>();
 monEnsemble.add("Bonjour");
 monEnsemble.add("Benoit");
 monEnsemble.add("Ca va");
 monEnsemble.add("Bonjour");
 println(monEnsemble);
 println("Contient Bonjour ? "+ monEnsemble.contains("Bonjour"));
 monEnsemble.remove("Bonjour");
 println(monEnsemble);
}
-- Affichage de la sortie --------
[Benoit, Bonjour, Ca va]
Contient Bonjour ? true
[Benoit, Ca va]
```

- la ligne `import ...` indique au compilateur qu'on souhaite utiliser les fonctions spécifiques pour **HashSet**
- là aussi on peut ajouter des valeurs avec **add**, et afficher directement l'ensemble des valeurs avec **println**. On se rend compte que les doublons ne sont pas acceptés, et que l'ordre dans lequel les valeurs sont stockées n'est pas celui dans lequel on a ajouté les valeurs
- **contains** permet de savoir si un élément donné existe dans l'ensemble, et **remove** permet de supprimer un élément

Les structures **ArrayList** et **HashSet** acceptent une forme particulière de la boucle **for** qui permet de parcourir les éléments contenus dans la liste ou l'ensemble. Le bout de code suivant affiche les éléments contenus dans un **HashSet¡String¿** :

```
for (String elt:monEnsemble) {
 println("Elt: "+elt);
}
```

Enfin, la structure **HashMap** permet de stocker des couples d'éléments : le premier est appelé la *clef*, le second la *valeur*. Les clefs doivent être uniques, et une seule valeur peut être associée à une clef. Cette structure est adaptée par exemple pour stocker des notes d'élèves dans une matière donnée :

```
void main() {
 HashMap<String,Double> maMap = new HashMap<String,Double>();
 maMap.put("Benoit",12.0);
 maMap.put("Hervé",17.5);
 maMap.put("Charles",15.0);
 maMap.put("Benoit",14.5);
 println(maMap);
 println("Note de Charles: "+ maMap.get("Charles"));
 maMap.remove("Hervé");
 println(maMap);
}
-- Affichage de la sortie --------
{Benoit=14.5, Hervé=17.5, Charles=15.0}
Note de Charles: 15.0
{Benoit=14.5, Charles=15.0}
```

Cette fois-ci c'est l'instruction **put** qui permet d'ajouter des couples (une chaîne de caractère pour le nom de l'élève et un réel pour sa note), mais si l'on essaie d'associer plusieurs notes au même élève c'est la dernière qui est conservée. On peut encore afficher le contenu par **println**, supprimer un couple par **remove**, et l'instruction **get** permet d'accéder à la valeur associée à une clef : ici la note associée à un élève.

7.2.1 Exercice : Remplacer les tableaux par des ArrayList

Reprendre les exercices de la section 5.1 et remplacer tous les tableaux par des **ArrayList**

7.2.2 Exercice : Sans duplication

Écrire une fonction **addSansDoublon** qui, avant d'ajouter un élément dans une **ArrayList**, vérifie qu'il n'est pas déjà présent en parcourant la liste et si c'est le cas, affiche un message d'erreur. De la même façon, écrire une fonction **putSansDoublon** qui permet, si une clef existe déjà dans une **HashMap**, de demander à l'utilisateur de confirmer s'il souhaite bien écraser la valeur précédente.

7.2.3 Exercice : vérificateur orthographique

Écrire un programme permettant de remplir un ensemble de type **Set** avec une liste de mots [22]. Utiliser ensuite cet ensemble pour vérifier si 10 mots tapés au clavier par l'utilisateur sont corrects.

7.3 Piles, Files

Les piles et les files sont des objets fréquemment rencontrés en algorithmique, et sont respectivement fondées sur le principe *Last in, First out* [23] et *First in, First out* [24].

22. http://www.freelang.com/download/misc/liste_francais.zip
23. http://fr.wikipedia.org/wiki/Pile_(informatique)
24. http://fr.wikipedia.org/wiki/File_(structure_de_donnees)

La structure **Stack** est une implémentation des piles vues en algorithmique, dotées des fonctions **push** (ajouter un élément au sommet de la pile) et **pop** (enlever l'élément présent au sommet de la pile) :

```
import java.util.Stack;
void main() {
 Stack<String> maPile = new Stack<String>();
 maPile.push("Benoit");
 maPile.push("Hervé");
 maPile.push("Charles");
 println(maPile.peek());
 println(maPile);
 println(maPile.pop());
 println(maPile);
 maPile.pop();
 maPile.pop();
 maPile.pop();
}
-- Affichage de la sortie --------
Charles
[Benoit, Hervé, Charles]
Charles
[Benoit, Hervé]
java.util.EmptyStackException
```

Ici la pile sert à stocker des chaînes de caractères, et l'on voit comment la déclarer et l'afficher avec **println**. On peut dépiler un élément avec **pop** ou simplement voir cet élément sans le supprimer avec **peek** [25]. Attention bien sûr à ne pas trop dépiler, comme à la fin de ce programme qui génère un message d'erreur.

De la même façon, la structure **Queue** propose une implémentation d'une file avec les opérations **add** (ajouter un élément à la fin de la file) et **remove** (enlever l'élément présent en tête de file) :

```
import java.util.*;
void main() {
 Queue<String> maFile = new LinkedList<String>();
 maFile.add("Benoit");
 maFile.add("Hervé");
 maFile.add("Charles");
 println(maFile);
 maFile.remove();
 println(maFile);
}
-- Affichage de la sortie --------
[Benoit, Hervé, Charles]
[Hervé, Charles]
```

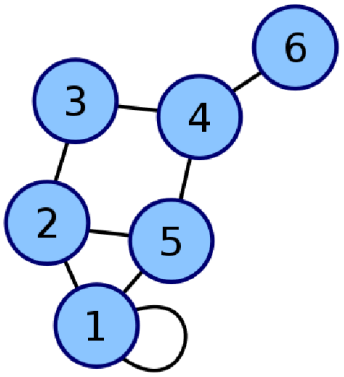
Sans rentrer dans les détails, on voit qu'on utilise en fait une structure appelée **LinkedList** mais dans la pratique cela n'a pas d'importance. Une fonction **peek** est également disponible pour voir l'élément en tête de file sans le supprimer.

Pour ces deux structures, on peut connaître le nombre d'éléments stockés grâce à la fonction **size()**, ce qui permet notamment d'éviter d'employer les opérations de suppression du premier élément si ce nombre est nul.

7.3.1 Exercice : Compter les parenthèses

Un problème très classique en informatique : on considère une chaîne contenant une expression arithmétique telle que "$((a + b)/(c - d))/(e + f)$". Comment vérifier à l'aide d'une pile que l'expres-

25. Le verbe *peek* peut se traduire par "jeter un coup d'oeil".



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

FIGURE 6 – Graphe et sa représentation par matrice d'adjacence

sion est *correctement parenthésée*, en interdisant par exemple l'expression "$)(a+b)/(d$" ?

7.3.2 Exercice : Inverser une pile

Encore un problème très classique en informatique : comment à partir d'une pile obtenir une pile avec les mêmes éléments dans l'ordre inverse ? Il suffit a priori de transférer les éléments de l'une à l'autre. Essayer ensuite, en utilisant une file, de résoudre le problème en ayant à la fin la pile de départ dans son état initial.

7.3.3 Exercice : Files

Implémenter le *parcours en largeur* d'un arbre binaire tel que celui présenté à la section 7.1 (se référer à `http://fr.wikipedia.org/wiki/Arbre_binaire`)

7.4 Graphes

On pourrait représenter un graphe avec un chaînage similaire à ceux vus à la section 7.1. On représente parfois aussi les graphes par une matrice d'adjacence, comme sur la figure 6. Il est évidemment possible d'implémenter de telles structures et des opérations pour les manipuler, par exemple pour calculer le plus court chemin d'un point à un autre d'un graphe [26].

Dans cette section nous préférons nous intéresser aux activités déjà présentes dans Javascool à propos des graphes, qui permettent de ne pas avoir à recoder ces structures et de s'intéresser plutôt aux algorithmes que l'on peut leur appliquer.

L'activité à choisir au lancement de Javascool s'appelle "gogleMaps", c'est la plus simple à utiliser. L'onglet "Aide de la proglet" explique d'abord comment afficher des positions et des chemins. Par exemple avec le code ci-dessous on obtient deux positions reliées par un segment sur la carte accessible par l'onglet "googlemap" :

```
void main()
{
 effaceCarte();
 affichePointSurCarte(2,47);
 affichePointSurCarte(4,46);
 afficheRouteSurCarte(2,47,4,46);
}
```

Les données géographiques (longitudes et latitudes des villes françaises et relations de voisinage entre elles) sont stockées dans des structures similaires à celles vues à la section 7.2. Par exemple, le code ci-dessous affiche sur la carte toutes les villes déjà enregistrées :

26. `http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`

```
 int i = 1;
 for (String ville:latitudes.keySet()) {
  double longitude = longitudes.get(ville);
  double latitude = latitudes.get(ville);
  affichePointSurCarte(longitude, latitude, i++);
  println(ville + "/" + longitude + "/" + latitude);
 }
-- Affichage de la sortie --------
Troyes/4.052795/48.287473
Lens/3.056121/50.381367
Nantes/-1.564819/47.240526
...
```

Il est aussi possible de tracer des segments de droite entre deux positions, et également d'accéder à la liste des villes voisines d'une ville donnée :

```
 String origine = "Limoges";
 double longOrigine = longitudes.get(origine);
 double latOrigine = latitudes.get(origine);
 for (String voisin:voisins.get(origine)) {
  double longitude = longitudes.get(voisin);
  double latitude = latitudes.get(voisin);
  afficheRouteSurCarte(longOrigine, latOrigine, longitude, latitude);
  print(voisin+" ");
 }
-- Affichage de la sortie --------
Orléans Clermont-Ferrand Angoulême
```

Le stockage des relations de voisinage définit de façon implicite un graphe non-orienté (pour toute relation de voisinage d'une ville A vers une ville B, il existe également une relation de voisinage de B vers A). Une fonction spécifique permet de déterminer le plus court chemin de A vers B, en prenant comme valuation des arêtes la distance euclidienne entre les villes. Cette fonction renvoie une liste de toutes les villes à traverser :

```
 List<String> villes = plusCourtCheminGogleMap("Pau", "Reims");
 int j = 1;
 for (String town:villes) {
  double longitude = longitudes.get(town);
  double latitude = latitudes.get(town);
  affichePointSurCarte(longitude, latitude, j++);
  print(town+" ");
 }
-- Affichage de la sortie --------
Pau Bayonne Bordeaux Angoulême Limoges Orléans Paris Reims
```

Les activités "grapheEtChemins" et "enVoiture" fonctionnent sensiblement sur le même principe, la seconde proposant un affichage 3D qui, même s'il ne fonctionne apparemment que sur une plateforme Windows [27], peut être plus motivant pour les élèves.

7.4.1 Exercice : Graphe complet

Afficher avec les fonctions ci-dessus le graphe complet des villes avec leurs relations de voisinages immédiates. Proposer ensuite un algorithme permettant de vérifier qu'il existe un chemin reliant n'importe quel couple de villes.

27. Voir `http://javascool.gforge.inria.fr/v4//index.php?page=proglets&action=show&id=enVoiture` en bas de page pour télécharger les bibliothèques à utiliser.

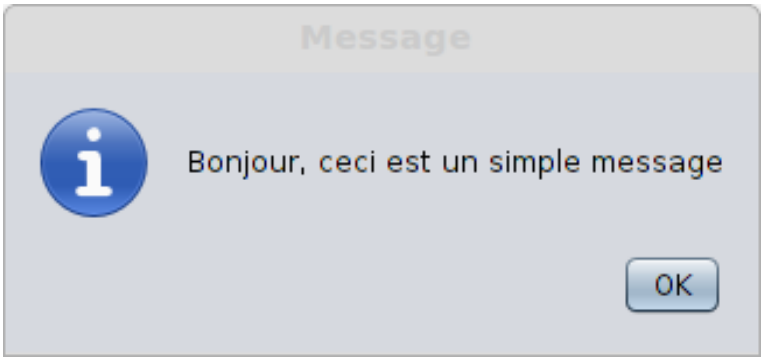


FIGURE 7 – Boîte de dialogue simple

7.4.2 Exercice : Plus court chemin

Recoder l'algorithme du plus court chemin [28] en considérant d'abord que les arêtes du graphe ont toutes un poids identique, puis en ajoutant la valuation des arêtes calculées par la distance euclidienne entre les deux sommets.

8 Dessin en deux dimensions et Interfaces graphiques

En plus des activités présentées dans la section 6.4, il est possible d'aller plus loin en programmant de réelles interfaces graphiques (au sens anglais de "*Graphical User Interface*"), où l'utilisateur peut déclencher des évènements en interagissant avec le programme grâce à la souris ou au clavier.

8.1 Boîtes de dialogue

Nous ne donnons ici qu'une brève introduction sur les fonctions permettant de remplacer les entrées-sorties au clavier et les affichages par des boîtes de dialogue assez simples à utiliser, grâce à l'activité "javaProg". Nous ne montrons que les boîtes de dialogue les plus utiles, d'autres sont aussi disponibles [29].

Le premier exemple est un simple message à l'écran, tel que représenté sur la figure 7. On utilise la fonction **getSwingPane()**, prédéfinie de l'activité "Programmer directement en Java", pour associer la boîte de dialogue à la fenêtre Javascool. La première ligne sert à charger en mémoire les bibliothèques nécessaires à l'exécution :

```
import javax.swing. *;
void main() {
   JLayeredPane frame = getPane();
   JOptionPane.showMessageDialog(frame,
    "Bonjour, ceci est un simple message");
}
```

Ensuite une boîte de dialogue permettant de poser une simple question, comme sur la figure 8 :

```
 int n = JOptionPane.showConfirmDialog(frame,
         "Maintenant, deux choix:", "Confirmation",
         JOptionPane.YES_NO_OPTION);
 if (n == JOptionPane.YES_OPTION) {
  // Traitement à appliquer s'il répond "Oui"
 } else if (n == JOptionPane.NO_OPTION) {
  // Idem pour "Non"
 } else {
  // S'il a fermé la boîte sans répondre
 }
```

28. http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra
29. Voir http://download.oracle.com/javase/tutorial/uiswing/components/dialog.html

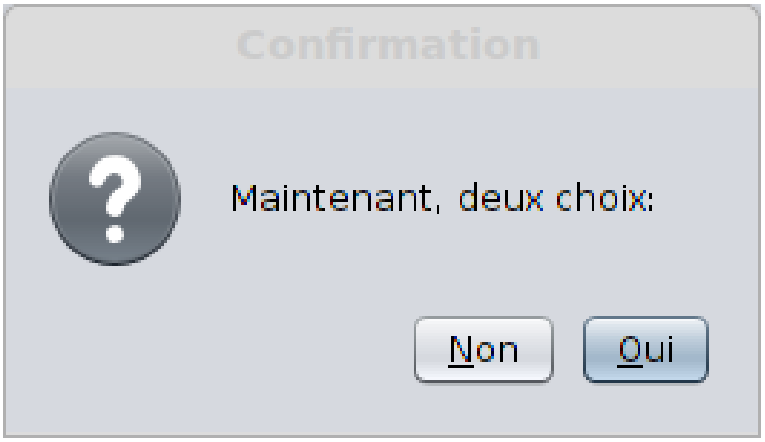


FIGURE 8 – Boîte de confirmation

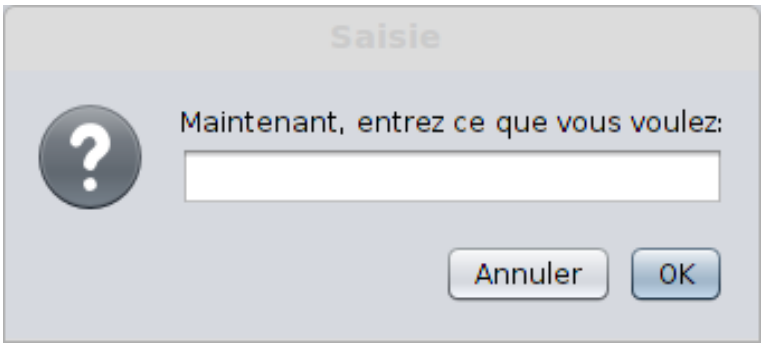


FIGURE 9 – Boîte de saisie

On peut également utiliser une boîte pour permettre la saisie d'un texte, comme sur la figure 9. La variable **reponse** contient le texte tapé, ou bien **null** si l'utilisateur a fermé la boîte sans répondre :

```
String reponse = JOptionPane.showInputDialog(frame,
 "Maintenant, entrez ce que vous voulez:", "Saisie",
 JOptionPane.QUESTION_MESSAGE);
```

Enfin, on peut également proposer une liste de choix pour l'utilisateur, comme sur la figure 10 (où les choix ne sont pas affichés, mais vous les verrez si vous exécutez le code). Là encore la variable **reponse** contient le texte choisi, ou bien **null** si l'utilisateur a fermé la boîte sans répondre. Le dernier paramètre de la fonction permet de sélectionner l'entrée qui sera proposée initialement à l'utilisateur, à partir de la liste de choix définie dans un tableau de chaînes de caractères :

```
String[] choix = {"Ben", "Hervé", "Sophie"};
String reponse = (String)JOptionPane.showInputDialog(frame,
 "Qui est votre préféré(e) ?", "Une super boîte de dialogue",
 JOptionPane.PLAIN_MESSAGE, null, choix, "Ben");
```

8.1.1 Exercice : le retour du mystère

Reprendre le jeu du nombre mystère (voir section 3.6) et remplacer toutes les saisies et affichages par des boîtes de dialogue.

8.1.2 Exercice : le retour du pendu

Même énoncé mais cette fois pour le jeu du pendu (section 5.1.3).

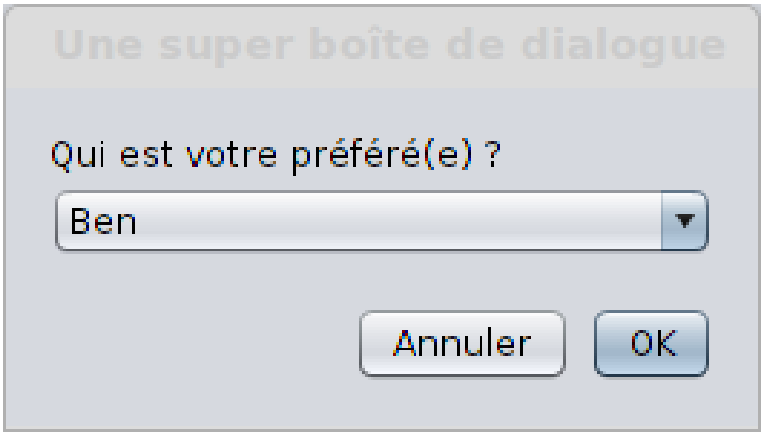


FIGURE 10 – Saisie à partir de choix prédéfinis

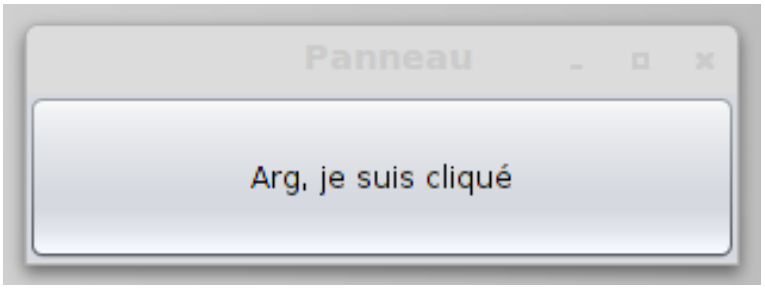


FIGURE 11 – Un panneau de contrôle à un seul bouton

8.2 Panneau de contrôle

On peut implémenter des comportements plus évolués qu'avec les boîtes de dialogues en créant des panneaux de contrôles et des boutons qui vont déclencher des actions. Commençons avec un exemple simple et un seul bouton :

```
import javax.swing.*;
import java.awt.event.*;
JButton bouton1;
JFrame panneau;
class GestionClic implements ActionListener {
 public void actionPerformed(ActionEvent e) {
  bouton1.setText("Arg, je suis cliqué");
 }
}
void main() {
 panneau = new JFrame("Panneau");
 bouton1 = new JButton("Du texte sur le bouton");
 bouton1.addActionListener(new GestionClic());
 panneau.add(bouton1);
 panneau.setSize(300,100);
 panneau.setVisible(true);
}
```

Dans cet exemple, on définit comme *variables globales* le bouton et le panneau, puis une structure **GestionClic** comportant une fonction **actionPerformed**. C'est cette fonction qui implémente ce qui doit se passer lors d'un clic (ici, on modifie simplement le texte du bouton). Dans le programme **main**, on crée les variables puis on associe, à l'aide de l'opération **addActionListener**, la structure **GestionClic**

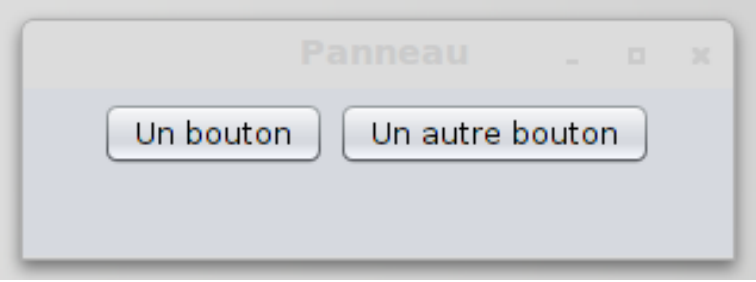


FIGURE 12 – Un panneau de contrôle avec deux boutons

au bouton. On ajoute le bouton au panneau, on lui donne une taille, puis (enfin) le panneau s'affiche à l'écran comme sur la figure 11. Si vous exécutez ce code vous aurez peut-être l'impression que seul un clic est pris en compte : c'est normal puisque le texte est changé une première fois, ensuite c'est le même texte qui est ré-affiché à chaque fois (mais tous les clics sont correctement gérés).

Notre second exemple propose maintenant deux boutons, pour obtenir un panneau tel que celui de la figure 12 :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
JButton bouton1, bouton2;
JFrame panneau;
class GestionClic implements ActionListener {
 public void actionPerformed(ActionEvent e) {
  if (e.getSource() == bouton1) {
   bouton1.setText("Arg, je suis cliqué");
  }
  else {
   bouton2.setText("Arg, je suis cliqué");
  }
 }
}
void main() {
 panneau = new JFrame("Panneau");
 bouton1 = new JButton("Un bouton");
 bouton2 = new JButton("Un autre bouton");
 GestionClic gc = new GestionClic();
 bouton1.addActionListener(gc);
 bouton2.addActionListener(gc);
 panneau.setLayout( new FlowLayout() );
 panneau.add(bouton1);
 panneau.add(bouton2);
 panneau.setSize(300,100);
 panneau.setVisible(true);
}
```

Les différences avec le premier exemple sont d'abord la déclaration de deux variables de type **JButton**, puis dans le code de la structure **GestionClic** on voit qu'un traitement permet de différencier si le clic a été déclenché sur le bouton 1 ou le bouton 2. Dans la partie **main** on voit que l'association entre boutons et structure **GestionClic** change un peu (on commence par créer une variable de type **GestionClic**, puis on l'associe aux deux boutons). Enfin, on voit apparaître une instruction **setLayout**, qui indique que les boutons sont placés horizontalement les uns à côté des autres.

Le dernier exemple, un peu plus évolué, montre une combinaison de la tortue LOGO (section 6.4) avec un panneau de contrôle permettant de bouger la tortue "à la main", comme sur la figure 13. Il faut

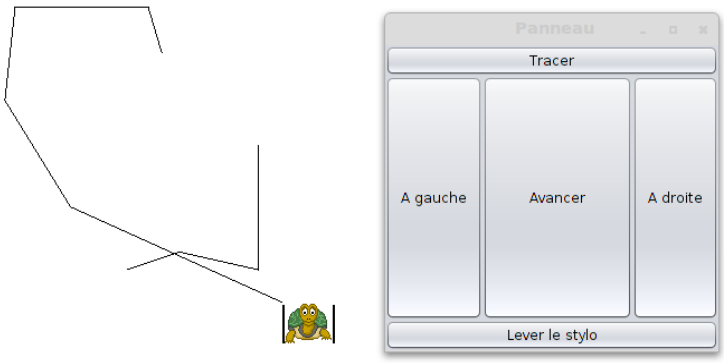


FIGURE 13 – Un panneau de contrôle avec cinq boutons

d'abord ouvrir l'activité "Programmer avec la tortue Logo" pour pouvoir exécuter ce code, qui utilise un autre type de **layout** permettant de placer les boutons dans 5 zones définies par les points cardinaux :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
JButton avancer, gauche, droite, tracer, lever;
JFrame panneau;
class GestionClic implements ActionListener {
 public void actionPerformed(ActionEvent e) {
  Object bouton = e.getSource();
  if (bouton == avancer) {forward(10);}
  else if (bouton == gauche) {leftward(10);}
  else if (bouton == droite) {rightward(10);}
  else if (bouton == tracer) {pen_down();}
  else {pen_up();}
 }
}
void main() {
 clear_all();
 set_background(9);
 set_color(0);
 pen_up();
 set_position(256,256);
 panneau = new JFrame("Panneau");
 avancer = new JButton("Avancer");
 gauche = new JButton("A gauche");
 droite = new JButton("A droite");
 tracer = new JButton("Tracer");
 lever = new JButton("Lever le stylo");
 GestionClic gc = new GestionClic();
 avancer.addActionListener(gc);
 gauche.addActionListener(gc);
 droite.addActionListener(gc);
 tracer.addActionListener(gc);
 lever.addActionListener(gc);
 panneau.setLayout(new BorderLayout());
 panneau.add(tracer, BorderLayout.NORTH);
 panneau.add(lever, BorderLayout.SOUTH);
 panneau.add(droite, BorderLayout.EAST);
```

```
 panneau.add(gauche, BorderLayout.WEST);
 panneau.add(avancer, BorderLayout.CENTER);
 panneau.setSize(300,300);
 panneau.setVisible(true);
}
```

9 Correction des exercices

9.1 Section 3

9.1.1 Hello World

Exercice du tutoriel “Hello World” :

```
void main() {
 println("Bonjour très cher:");
 println("Quel bon vent vous amène");
}
```

Exercice 3.1.2 :

```
void main() {
 println("*****************");
 println("*               *");
 println("*  HelloWorld!  *");
 println("*               *");
 println("*****************");
}
```

Pour afficher une ligne vide : `println("");`

9.1.2 Variables

Exercice du tutoriel sur les variables, un programme un peu amélioré :

```
void main() {
 println("Bonjour, quel est ton nom ?");
 String nom = readString();
 println("Et quel est ton âge ?");
 int age = readInteger();
 println("Enchanté " + nom + ", " +
  age + " ans est un bel âge !");
 println("Mais quelle est ta moyenne ?");
 double moy = readDouble();
 println(moy + " de moyenne c'est pas mal.");
}
```

Exercice 3.2.3 :

```
void main() {
 double rayon, circ, aire;
 // le mot-clef final sert à indiquer une constante
 final double Pi = 3.14159;
 println("Entrer le rayon: ");
 rayon = readDouble();
 circ = 2 * rayon * Pi;
 aire = Pi * rayon * rayon;
 println("Circonférence du cercle de rayon " +
  rayon + ": " + circ);
```

```
 println("Aire: " + aire);
}
```

Exercice 3.2.4 :

```
void main() {
 double rayon, circ, aire;
 println("Entrer le rayon: ");
 rayon = readDouble();
 // Autre amélioration: il existe une constante prédéfinie PI
 circ = 2 * rayon * PI;
 aire = PI * pow(rayon, 2);
 println("Circonférence du cercle de rayon " +
  rayon + ": " + circ);
 println("Aire: " + aire);
}
```

Exercice 3.2.5 :

```
void main() {
 double a, b, c, d;
 println("Entrer les coeffs d'une équation du 2nd degré (le déterminant doit être positif)");
 println("Entrer a:");
 a = readDouble();
 println("Entrer b:");
 b = readDouble();
 println("Entrer c:");
 c = readDouble();
 d = b*b - (4*a*c);
 println("Déterminant: " + d);
 println("Racine x1 = " + (-b + sqrt(d))/(2*a));
 println("Racine x2 = " + (-b - sqrt(d))/(2*a));
}
```

9.1.3 Instructions conditionnelles

Exercices du tutoriel sur les instructions conditionnelles :

```
void main() {
 println("Bonjour, quel est ton nom ?");
 String nom = readString();
 if((equal(nom, "Toto")) || (equal(nom, "Dieu"))) {
 println("Ce n'est pas crédible !");
 }
 else if(equal(nom, "Nadia")) {
 println("Super !");
 }
 println("Et quel est ton âge ?");
 int age = readInteger();
 if ((age >= 3) && (age <= 120)) {
  println("OK !");
 }
 else {
  println("Ce n'est pas sérieux !");
 }
}
```

Exercice sur le cercle (version améliorée) :

```
void main() {
 double rayon, circ, aire;
 println("Entrer le rayon: ");
 rayon = readDouble();
 if (rayon < 0) {
  println("Erreur, calcul impossible");
 }
 else {
  circ = 2 * rayon * PI;
  aire = PI * pow(rayon, 2);
  println("Circonférence du cercle de rayon " +
   rayon + ": " + circ);
  println("Aire: " + aire);
  if (circ < aire) {
   println("La circonférence est inférieure à l'aire");
  }
  else if (circ > aire) {
   println("La circonférence est supérieure à l'aire");
  }
  else {
   println("La circonférence est égale à l'aire");
  }
 }
}
```

Exercice 3.3.3 :

```
void main() {
 int a, b, c;
 println("Entrer a, b et c: ");
 a = readInteger();
 b = readInteger();
 c = readInteger();
 if (a < b) {
  if (b < c) {
   println ("a < b < c");
  }
  else if (a < c) {
   println ("a < c < b");
  }
  else {
   println ("c < a < b");
  }
 }
 else {
  if (a < c) {
   println ("b < a < c");
  }
  else if (b < c) {
   println ("b < c < a");
  }
  else {
   println ("c < b < a");
  }
 }
 // On ne peut pas écrire: if (a < b < c) ...
 // mais on peut combiner: if ((a < b) && (b < c)) ...
```

```
}
```

Exercice 3.3.4 :

```
void main() {
 double a, b, c, d;
 println("Entrer les coeffs d'une équation du 2nd degré (le déterminant doit être positif)");
 println("Entrer a:");
 a = readDouble();
 println("Entrer b:");
 b = readDouble();
 println("Entrer c:");
 c = readDouble();
 d = b*b - (4*a*c);
 println("Déterminant: " + d);
 if (d < 0) {
  println("Calcul des racines impossible");
 }
 else if (d > 0) {
  println("Racine x1 = " + (-b + sqrt(d))/(2*a));
  println("Racine x2 = " + (-b - sqrt(d))/(2*a));
 }
 else {
  println("Racine double = " + (-b/(2*a)));
 }
}
```

9.1.4 Fonctions

Exercices du tutoriel sur les fonctions :

```
// Exercice 1
int min (int x, int y) {
 if (x < y) {
  return x;
 } else {
  return y;
 }
}
int max (int x, int y) {
 if (x > y) {
  return x;
 } else {
  return y;
 }
}
int abs(int x) {
 return (max(-x, x));
}
void main() {
 int x = 12;
 int y = -14;
 println("Max = " + max(abs(x), abs(y)) +
  " et Min = " + min(abs(x), abs(y)));
}

// Exercice 2a
```

```
double div(double x, double y) {
 if (y == 0) {
  return Double.NaN;
 } else {
  return x/y;
 }
}
void main() {
 double a, b;
 println ("Entrer a et b: ");
 a = readDouble(); b = readDouble();
 println ("a/b = " + div(a, b));
}

// Exercice 2b
boolean xor (boolean x, boolean y) {
 if ((x) && (y)) {
  // Ou: if ((x == true) && (y == true))
  return false;
 }
 return true;
}

// Exercice 2c
int max (int x, int y) {
 if (x > y) {
  return x;
 } else {
  return y;
 }
}
int max (int x, int y, int z) {
 return max(x, max(y, z));
 // Javascool comprend qu'on veut faire appel
 // à la fonction max de 2 variables
}
void main() {
 int a, b, c;
 println("Entrer a, b et c: ");
 a = readInteger();
 b = readInteger();
 c = readInteger();
 println("Max = " + max(a, b, c));
}

// Exercice 2d
void infox(String question, boolean reponse) {
 println(question + ": true ou false ?");
 boolean choix = readBoolean();
 if (choix == reponse) {
  println("Bravo !");
 } else {
  println("Tu t'es gentiment trompé...");
 }
}
void main() {
 infox("Il y a environ 100 millions d'ordinateurs dans le monde", false);
```

```
 infox("Ada Byron-Lovelace était, dès le XIXème siècle, la 1ère femme informaticienne", true);
 infox("Toutes les fonctions mathématiques peuvent être calculées par un ordinateur", false);
 infox("Le cours de la 2nd guerre mondiale a changé plus vite grâce à ... un calcul informatique", t
}
```

Exercice sur l'ordre de 3 nombres (bis) :

```
int ordre(int x, int y, int z) {
 if (x < y) {
  if (y < z) {
   return 1;
  } else if (x < z) {
   return 2;
  } else {
   return 3;
  }
 } else {
  if (x < z) {
   return 4;
  } else if (y < z) {
   return 5;
  } else {
   return 6;
  }
 }
}
void main() {
 int a, b, c;
 println("Entrer a, b et c: ");
 a = readInteger();
 b = readInteger();
 c = readInteger();
 switch (ordre(a, b, c)) {
  case 1: println ("a < b < c");
          break;
  case 2: println ("a < c < b");
          break;
  case 3: println ("c < a < b");
          break;
  case 4: println ("b < a < c");
          break;
  case 5: println ("b < c < a");
          break;
  default: println ("c < b < a");
 }
}
```

Réécriture de l'exercice sur les équations du second degré :

```
void calcul (double c1, double c2, double c3) {
 double d = c2*c2 - (4*c1*c3);
 println("Déterminant: " + d);
 if (d < 0) {
  println("Calcul des racines impossible");
 }
 else if (d > 0) {
  println("Racine x1 = " + (-c2 + sqrt(d))/(2*c1));
```

```
  println("Racine x2 = " + (-c2 - sqrt(d))/(2*c1));
 }
 else {
  println("Racine double = " + (-c2/(2*c1)));
 }
}
void main() {
 double a, b, c;
 println("Entrer les coeffs d'une équation du 2nd degré");
 println("Entrer a:");
 a = readDouble();
 println("Entrer b:");
 b = readDouble();
 println("Entrer c:");
 c = readDouble();
 calcul (a, b, c);
}
```

9.1.5 Boucles

Exercices du tutoriel :

```
// Huit affichages
void main() {
 int n = 1;
 while( n <= 8) {
  println("Hello World !");
  n = n + 1;
 }
}

// Nombres impairs de 1 à 39
void main() {
 int n = 1;
 while( n <= 39) {
  println("n = " + n);
  n = n + 2;
 }
}

// Calcul du nombre d'or
void main() {
 double r = 1;
 int n = 1;
 while(n <= 30) {
  r = 1 + 1/r;
  println("r = " + r);
  n++;
 }
}

// Approximation de Pi
double archimede (int n) {
 // On démarre avec des polygones à 3 côtés
 int i = 3;
 // Périmètre du polygone inscrit
 double pInscrit = 2 * sqrt(2);
```

```
 // Périmètre du polygone circonscrit
 double pCircons = 4;
 // Moyenne des deux périmètres
 double moy = (pInscrit + pCircons)/2;
 while(i <= n) {
  pCircons = pCircons * pInscrit/moy;
  pInscrit = sqrt(pInscrit * pCircons);
  moy =  (pInscrit + pCircons)/2;
  i++;
 }
 return (2*pInscrit + pCircons)/3;
}
void main() {
 println("Entrez n: ");
 int n = readInteger();
 println("Approximation de Pi d'ordre " + n + ": " + archimede(n));
}
```

Exercice sur la vérification des entrées :

```
void main() {
 double rayon, circ, aire;
 println("Entrer le rayon: ");
 rayon = readDouble();
 while (rayon < 0) {
  println("Erreur, rayon négatif. Entrer le rayon: ");
  rayon = readDouble();
 }
 circ = 2 * rayon * PI;
 aire = PI * rayon * rayon;
 println("Circonference du cercle de rayon " +
  rayon + ": " + circ);
 println("Aire: " + aire);
}

void main() {
 // Les variables doivent être initialisées pour
 // que Javascool soit sûr de pouvoir faire les calculs
 double a = 1, b = 1, c = 1, d = -1;
 while (d < 0) {
  println("Entrer les coeffs: ");
  println("Entrer a:");
  a = readDouble();
  println("Entrer b:");
  b = readDouble();
  println("Entrer c:");
  c = readDouble();
  d = b*b - (4*a*c);
  if (d < 0) {
   println("Erreur, le déterminant est négatif");
  }
 }
 println("Déterminant: " + d);
 println("Racine x1 = " + (-b + sqrt(d))/(2*a));
 println("Racine x2 = " + (-b - sqrt(d))/(2*a));
}
```

Exercice sur l'affichage des étoiles :

```
void afficherEtoiles (int n) {
 for (int i = 1; i <= n; i++) {
  print("*");
 }
 println(""); // Retour à la ligne
}
void afficherLigneVide (int n) {
 print("*");
 for (int i = 1; i <= (n-2); i++) {
  print(" ");
 }
 println("*");
}
void main() {
 afficherEtoiles(17);
 afficherLigneVide(17);
 println("* Hello World ! *");
 afficherLigneVide(17);
 afficherEtoiles(17);
}
```

Exercice sur la suite de Fibonnaci :

```
// Affichage des 30 premiers termes
void fibo1 () {
 int a = 1, b = 1, c;
 println(a); println(b);
 for (int i = 1; i <= 28; i++) {
  c = a + b;
  println(c);
  a = b;
  b = c;
 }
}
// Affichage des premiers termes inférieurs à 100
void fibo2 () {
 int a = 1, b = 1, c = a + b;
 println(a); println(b);
 while (c < 100) {
  println(c);
  c = a + b;
  a = b;
  b = c;
 }
}
// Affichage des termes compris entre 100 et 200
void fibo3 () {
 int a = 1, b = 1, c = a + b;
 while (c < 200) {
  if (c > 100) {
   println(c);
  }
  c = a + b;
  a = b;
  b = c;
```

```
 }
}
void main() {
 fibo1();
 fibo2();
 fibo3();
}
```

9.1.6 Nombre mystère

Amélioration en ajoutant la prise en compte du nombre de coups :

```
// Programme principal
void main ()
{
 int nbemystere = random(0,1000);
 int x = saisieUnMille ();
 int cpt = 1;
 while ((x != nbemystere) && (cpt <= 10)) {
  if (x < nbemystere) {
   println(x+" est trop petit...");
  }
  else {
   println(x+" est trop grand...");
  }
  x = saisieUnMille ();
  cpt++;
 }
 if (x == nbemystere) {
  println("Bravo, tu as trouvé " + x + " en " +
   cpt + " coups !");
 } else {
  println("Perdu, c'était " + nbemystere);
 }
}
```

Faisons jouer l'ordinateur, qui utilisera la technique dichotomique :

```
// Version intelligente...
void main () {
 int borneInf = 1; int borneSup = 1000;
 int x = (borneInf + borneSup) / 2;
 int cpt = 0;
 boolean gagne = false;
 println("Pense à un nombre entre 1 et 1000 dans ta tête...");
 while ((gagne == false) && (cpt <= 10)) {
  println("Je propose " + x);
  println("Tape 1 si c'est trop petit, 2 si c'est trop grand, ou 3 si j'ai gagné !");
  int rep = readInteger();
  if (rep == 1) {
   borneInf = x + 1;
   x = (borneInf + borneSup) / 2;
  } else if (rep == 2) {
   borneSup = x - 1;
   x = (borneInf + borneSup) / 2;
  } else {
   gagne = true;
```

```
  }
  cpt++;
 }
 if (gagne) {
  println("J'ai trouvé " + x + " en " +
   cpt + " coups !");
 } else {
  println("J'ai perdu...");
 }
}
```

10 Annexe : Aide-mémoire Javascool

Syntaxe générale du langage Javascool / Java

1. Structure générale d'un programme

```
void main() {
	// Déclaration des variables
	// Programme principal
}
```

2. Déclaration des variables et des constantes

```
	type_1 nom_var1; (par exemple: int x;)
	type_2 nom_var2, nom_var3; (par exemple: double y, z;)
	…
	final type_1 nom_const_1 = valeur_constante;
		(par exemple: final double pi = 3.14159;)
```

3. Types

`int, double, boolean, char, String`

Rq : *Les caractères sont encadrés par des simples quotes :* `'o'`
Les chaînes sont délimitées par des doubles quotes : `"chaîne"`

4. Commentaires

```
// Exemple de commentaire
```

5. Séquence

```
{
   Action_1
   Action_2
   …
}
```

6. Action de lecture/écriture

```
variable = readInt(); (ou readDouble(), readString(), etc.)
println(liste des éléments à afficher);
```

7. Affectation

```
variable = expression;
```

8. Action conditionnelle

```
if (condition) {
	actions_si
} else {
	actions_sinon
}
```

9. Choix multiple

```
switch (variable) {
	case valeur_1: actions_1
			break;
	case valeur_2: actions_2
			break;
	…
	default: actions_sinon
}
```

10. Actions itératives

```
for (variable = indice_début; variable <= indice_fin; variable++) {
	actions
}

while (condition) {
	actions
}

do {
	actions
} while (condition)
```

11. Sous-programmes

Syntaxe d'une procédure :

```
void nom_procédure()                    void nom_procédure(paramètres)
{                                       {
  // Déclaration des variables  ou      // Déclaration des variables
  // Actions                            // Actions
}                                       }
```

Appel d'une procédure :

`nom_procédure()` *ou* `nom_procédure(variables ou constantes)`

Les paramètres sont séparés par des virgules et spécifiés sous la forme : `type nom_paramètre`

Les paramètres appartenant à un type de base (`int, double, boolean, char`) sont forcément des paramètres d'entrée, les autres sont des paramètres d'Entrée/Sortie

Syntaxe d'une fonction (les paramètres fonctionnent de manière similaire) :

```
type_retour nom_fonction()                  type_retour nom_fonction(paramètres)
{                                           {
  // Déclaration des variables      ou       // Déclaration des variables
  // Actions                                 // Actions
}                                           }
```

Pour « retourner » une valeur, on place dans le corps de la fonction une ou plusieurs actions de type :

```
return(valeur)
```

Appel d'une fonction :

`Variable = nom_fonction()` *ou* `variable = nom_fonction(variables ou constantes)`

12. Fichiers

Manipulation par des variables de ou type `PrintWriter` (écriture) ou `BufferedReader` (lecture)

Création d'un nouveau fichier (pour écriture) :

```
PrintWriter fichier = new PrintWriter(new FileWriter("monfichier.txt"), false);
```

(si le fichier existait déjà les données sont perdues)

Ajout de données dans un fichier existant (pour écriture) :

```
PrintWriter fichier = new PrintWriter(new FileWriter("monfichier.txt"), true);
```

Désignation d'un fichier existant (pour lecture) :

```
BufferedReader fichier2  = new BufferedReader(new FileReader("monfichier.txt"));
```

Fermeture : `fichier.close();`

Lecture : `String ligne = fichier2.readLine();`

Cette fonction renvoie **null** s'il n'y a plus de lignes à lire dans le fichier.

Ecriture : `fichier.println(données);`

13. Structures

```
class nom_structure
{
        …
}
```

Déclaration de variable : `nom_structure variable = new nom_structure();`

Accès à un élément d'une variable de type structure : `variable.élément`

14. Vecteurs et tableaux

Déclaration de variables :

```
type_éléments[] nom_vecteur = new type_éléments[nombre_éléments]; ou
type_éléments[][]...[] nom_tableau = new type_éléments[dim_1][dim_2]...;
```

Accès à une case à une position donnée :

`nom_vecteur[position]` *ou* *`nom_tableau[pos_1][pos_2][…]`*

15. Opérations diverses

Modulo : `entier_1 % entier_2`

Division entière : `entier_1 / entier_2` (division entière si les deux variables sont des entiers)

Nombre aléatoire entre deux entiers `a` et `b` inclus : `random(a,b)`

16. Pointeurs

En Javascool toute variable n'appartenant pas à un type de base (`int, double, boolean, char`) est forcément un pointeur.

Attention en modifiant les champs d'une variable de type structure...