IFT232

Laboratoire #0

Introduction à Java

Ce laboratoire comporte plusieurs exercices qui visent à vous donner une visite rapide du langage Java. On assume que vous avez un peu d'expérience avec C++ (IFT159) et que vous allez faire les exercices dans l'environnement Eclipse, avec Java 7 ou 8.

NOTES IMPORTANTES

Ce laboratoire est conçu en priorité pour ceux qui n'ont jamais fait de Java ou jamais utilisé Eclipse. Les premières pages sont remplies de détails qui ne seront pas utiles à une personne plus expérimentée. Cependant, les exercices à partir du numéro 4 devraient être utiles à tous pour réviser les bases du langage.

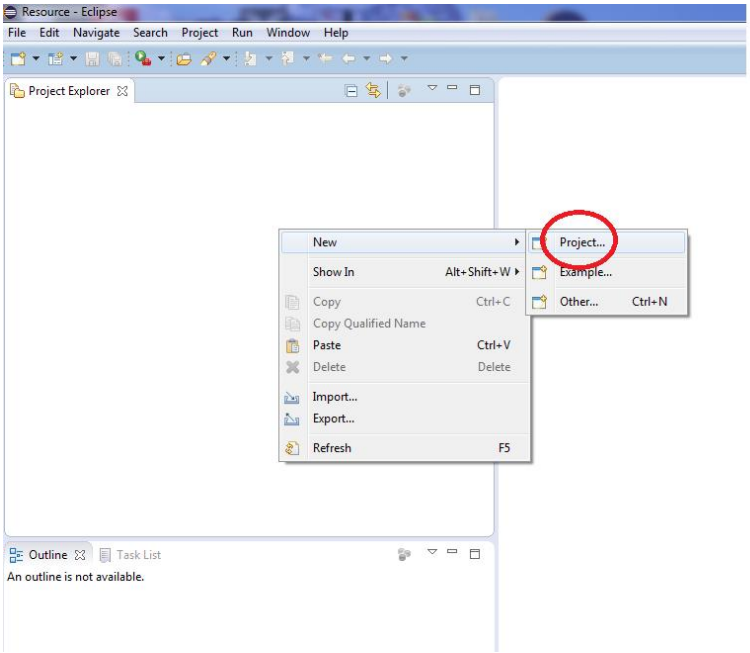
Les **encadrés verts** comme celui-ci contiennent des informations étendues sur la base du langage Java et l'environnement Eclipse (mots-clé, syntaxe, conventions, bonnes pratiques, raccourcis, trucs pratiques). Ceux-ci devraient être également lus par ceux qui ont de l'expérience en Java. Les informations qu'ils renferment devraient améliorer votre compréhension du langage et vous rendre plus efficient(e) dans l'environnement Eclipse. Cependant, vous devriez les lire plus tard si vous avancez dans le labo sans peine.

Exercice 0 : Démarrage

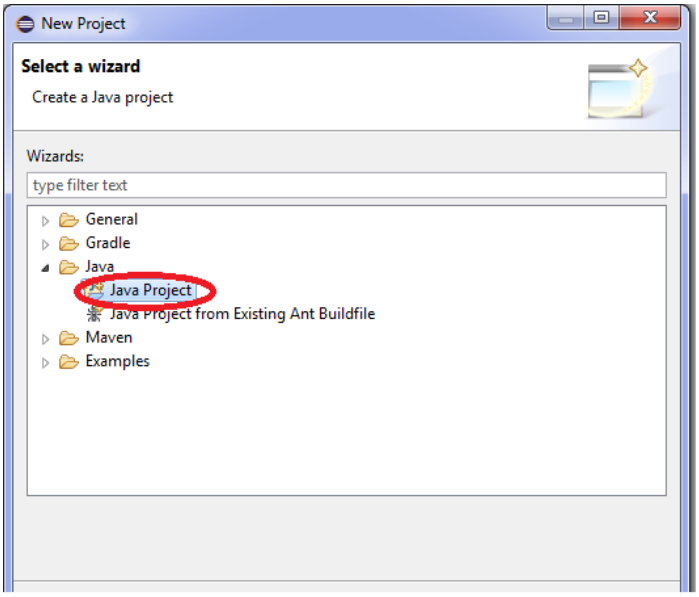
Au démarrage d'Eclipse, l'écran d'accueil vous est présenté.

Pour commencer à travailler, utilisez la flèche « **Go to Workbench** » qui se trouve en bas à droite de la page d'accueil.

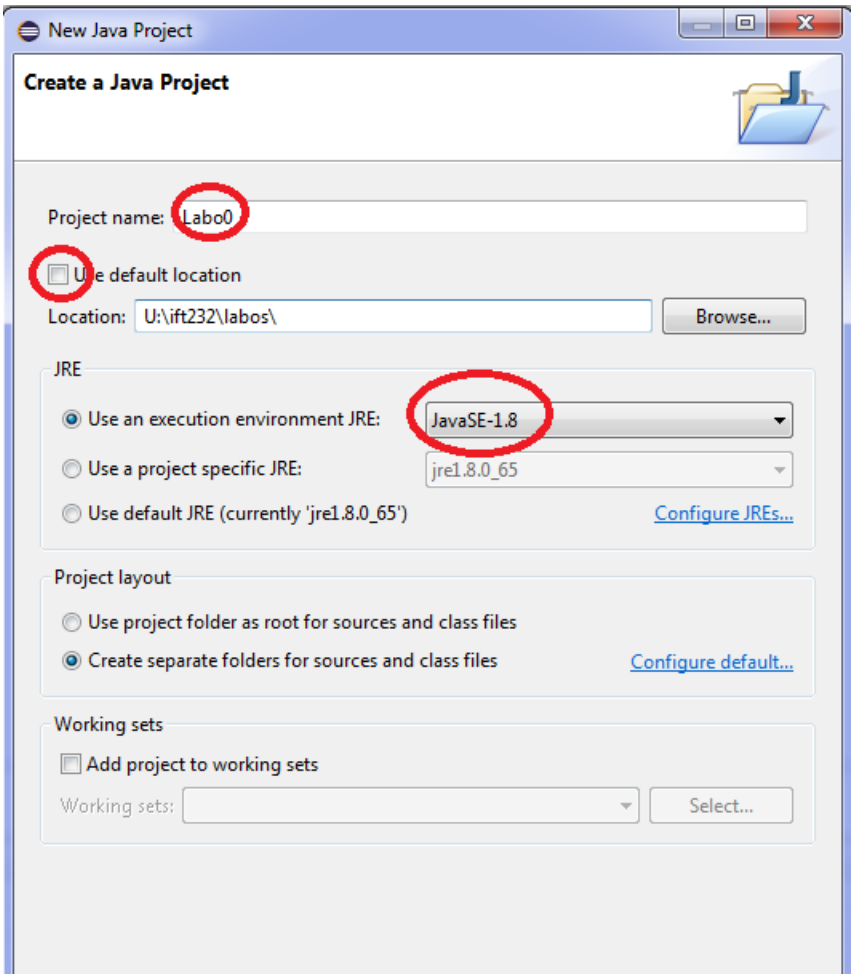
a) Créez un nouveau projet Java



Pour créer un nouveau projet, vous pouvez utiliser le menu **File** et l'option **New**, **Project**…, ou vous pouvez faire un clic droit dans l'onglet **Project Explorer** et choisir également **New**, **Project**… dans le menu contextuel.



Dans la fenêtre qui apparaît, ouvrez le dossier **Java** et choisissez **Java Project**, puis appuyez sur le bouton **Next**.



La fenêtre suivante vous permet de préparer votre projet.

-Choisissez un nom pour votre projet (pour ce labo, « Labo0 » fera l'affaire).

-Vous devriez vous assurer que votre projet est sauvegardé dans votre dossier personnel en décochant « Use Default Location » et en naviguant jusqu'au dossier de votre choix, sur le lecteur U : .

-Assurez-vous que la version de Java choisie est 1.7 ou 1.8.

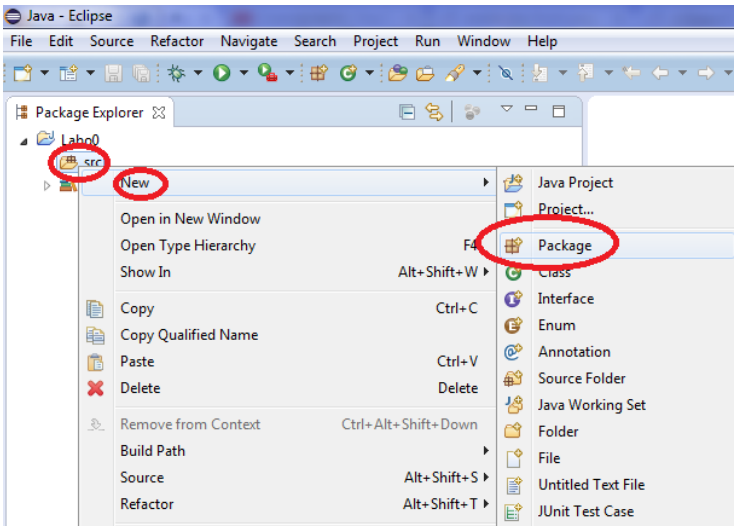
-Appuyez sur le bouton **Finish**.

Si une fenêtre apparaît vous suggérant de passer en perspective Java, c'est une bonne idée de dire «**Yes** », étant donné qu'on assume que votre Eclipse sera dans cette perspective pour la suite.

b) Créez et exécutez un petit programme

Commencez par ajouter un package à votre projet.

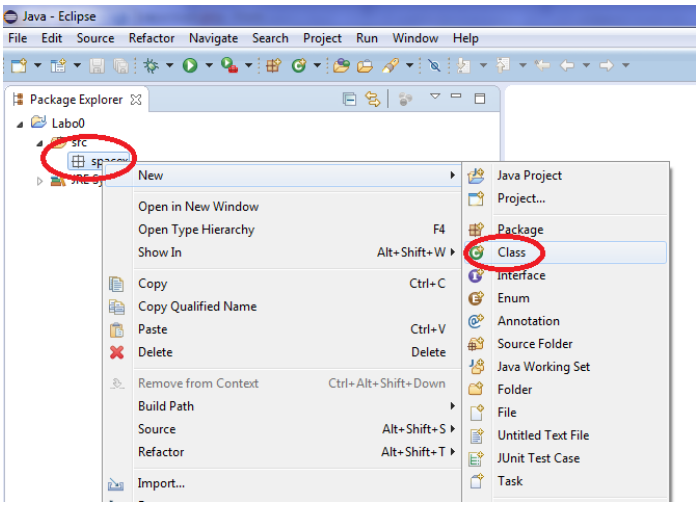
Un **package** est un espace de noms, une façon de regrouper plusieurs classes sous une même bannière. Ça aide à organiser votre code, ça évite les conflits de noms de classes et ça permet aux classes d'un même package de partager des informations via *la portée package* (voir encadré vert sur les portées plus loin).

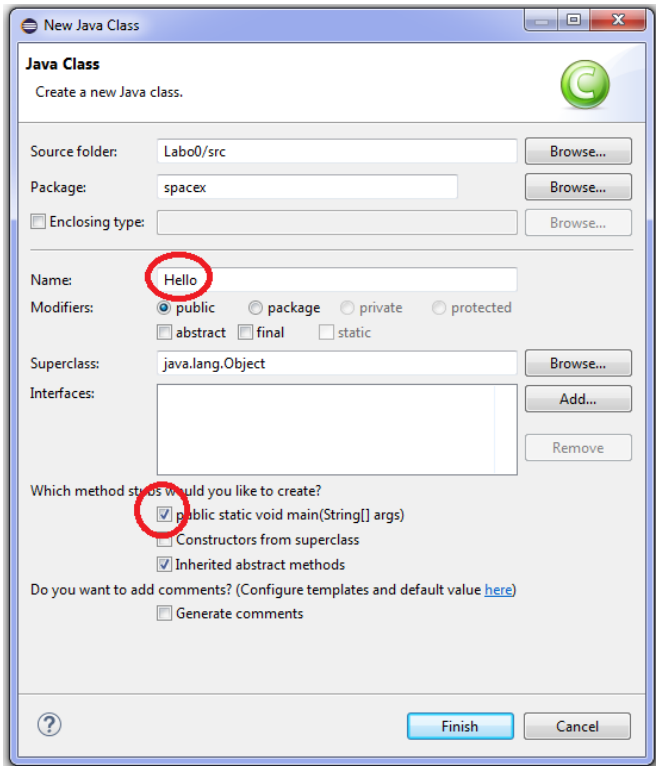


Pour créer un nouveau package, ouvrez d'abord le dossier de votre projet dans l'onglet **Package Explorer**, puis cliquez droit sur le dossier **src** pour obtenir le menu contextuel, duquel vous pouvez choisir **New**, **Package**.

Dans la fenêtre qui apparaît, il suffit de choisir un nom pour le package. Appelez-le **spacex**, étant donné que votre code servira à dessiner des fusées.

Ensuite, ajoutez une nouvelle classe à votre package, en effectuant un clic droit sur le package et en choisissant **New, Class** à partir du menu contextuel.





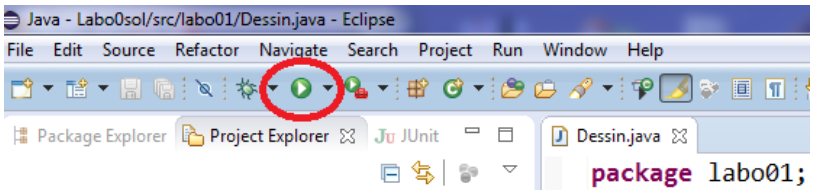
Donnez un nom à votre classe, puis assurez-vous que vous avez coché la case « **public static void main(String[] args)** ». Ceci indique à Eclipse d'ajouter un point d'entrée (équivalent de la fonction main en C++) à cette classe. Sans cela, il ne sera pas possible d'exécuter votre programme.

Finalement, le code de la classe devrait apparaître devant vous. Modifiez la méthode main en lui ajoutant un énoncé d'affichage :

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

    System.out.println("Hello, World!");
}
```

Pour exécuter votre programme, il ne vous reste qu'à appuyer sur le bouton **Run** (la flèche verte dans la barre d'outils en haut)



Vous devriez voir apparaître dans la console (fenêtre tout en bas) votre message.

OÙ SONT MES VARIABLES GLOBALES??

Tsk, tsk tsk, ne vous a-t-on pas enseigné que les variables globales sont la racine du mal, la cause des changements climatiques et devraient être évitées à tout prix?

Mais bon, pour ceux qui ne veulent pas sauver l'environnement (du programme), il devrait quand-même exister la possibilité d'en faire, non?

Pas dans le même sens qu'en C++. En Java, les variables et les fonctions doivent être déclarées à l'intérieur d'une classe. C'est obligatoire. Il est impossible de déclarer autre chose qu'une classe (ou quelque-chose de semblable) à l'extérieur des accolades d'une déclaration de classe.

Ceci implique qu'en Java, on ne parle jamais de fonctions, mais bien toujours de **méthodes**, le terme utilisé en programmation orientée-objet pour désigner les fonctions qui opèrent sur les données d'une classe.

Il est cependant justifiable de faire des **constantes** accessibles globalement (voir l'**exercice 4**).

De plus, le point de départ du programme (la fonction **main**) ne devrait-il pas être accessible globalement? Oui et non… (voir l'encadré sur le mot-clé **static** et les variables de classe).

Par contre, il existe quand-même quelque-chose de global : un exemplaire de chaque classe, auquel on peut s'adresser en écrivant son nom dans le code. On utilisera ce fait à notre avantage dans les rares cas où il est justifiable d'utiliser des fonctions globales (voir l'**exercice 5**).

Un exemple qui s'approche de l'utilisation d'une variable globale est l'énoncé *System.out.println()* que vous venez d'écrire. La variable « **out** » de la classe **System** existe en un seul exemplaire, et est accessible sans qu'on ait besoin de créer un objet de type System. C'est une **variable de classe**. Cette variable est de type **PrintWriter**, et possède la méthode println().

CONVENTIONS DE NOMMAGE
(classes, variables, packages, méthodes)

Habituellement, on applique les règles suivantes lorsqu'on choisit les noms des différents composants d'un programme en Java :

1. Les noms de packages sont complètement en minuscules.
2. Les noms de classes commencent par une majuscule.
3. Les noms de variables commencent par une minuscule.
4. Les noms de méthodes commencent par une minuscule.
5. Quand un nom comporte plusieurs mots, on met souvent en majuscule la première lettre de chaque mot, afin de rendre la lecture plus facile.

LES PORTÉES EN JAVA

Dans le langage Java, plusieurs modificateurs de portée sont disponibles. Ils sont représentés par des mots-clés qui sont utilisés à la déclaration de classes, de variables et de méthodes, pour déterminer quelle partie du code peut les utiliser.

Contrairement à C++, les classes possèdent une portée. De plus, les méthodes sont toujours définies à l'intérieur de la déclaration de la classe. Java exige également qu'on spécifie la **portée** de chaque variable, méthode et classe lors de sa déclaration. Si aucune portée n'est spécifiée, la portée **package** est utilisée. Voici la signification de chacune des portées :

public : La classe, la variable ou la méthode est accessible par n'importe quel segment de code du programme. Rendre une variable publique dans le but de l'accéder de n'importe où est à proscrire. Cependant, on a tendance à déclarer la majorité des méthodes comme publiques.

private : La classe, la variable ou la méthode n'est accessible que par les autres segments de code qui sont à l'intérieur de la portée (les accolades) de la classe dans laquelle elle est déclarée. Une classe ne peut être déclarée private que si c'est une classe déclarée dans une autre classe (**classe interne**). Cette portée devrait être utilisée la majorité du temps pour les variables déclarées dans une classe.

package : Si aucune portée n'est spécifiée, alors la portée package sera utilisée. Les classes, variables et méthodes qui possèdent cette portée sont accessibles de n'importe quel segment de code se trouvant dans le même package.

protected : La classe, variable ou méthode déclarée *protected* est accessible à tout le code qui se trouve dans le même package, ainsi que toutes les classes qui sont descendantes (par héritage) de la classe dans laquelle vous l'avez déclarée.

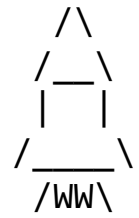
Exercice 1 : Dessin de base

Tout d'abord, renommez la classe que vous avez créée à l'exercice précédent et donnez-lui le nom **Dessin**.

Pour renommer une classe, il suffit de faire un clic droit sur la classe, puis de chercher dans le menu contextuel **Refactor, Rename**.

Vous pouvez aussi utiliser le raccourci clavier ALT-SHIFT-R.

Modifiez ensuite le code pour que votre affichage produise le dessin suivant :



N.B. Pour faire la barre oblique inverse, vous devez utiliser la séquence spéciale \\ (une barre oblique inverse seule signifiant qu'un caractère spécial suit).

C'est notre premier prototype de dessin de fusée, avec un cône contenant un cargo, une section cylindrique au centre contenant la citerne et finalement un moteur (en feu!!) à la base. Bon, ok ça prend de l'imagination mais à la fin du labo vous n'y verrez que du feu.

Vous pouvez utiliser plusieurs appels à la fonction **println** pour faire ce dessin. Chaque appel ajoute un saut de ligne à votre affichage automatiquement.

STRINGS ET CARACTÈRES SPÉCIAUX

À toutes les fois que vous placez du texte entre guillemets dans votre code, il est implicitement converti en un objet de type String. Plusieurs opérations

peuvent être faites avec ces objets (voir **exercice 2**) et ils sont très pratiques pour produire des traces et déverminer.

Tout comme en C++, les caractères spéciaux du code ASCII devraient être supportés, ainsi que les séquences d'échappement (codes commençant par \). Quelques codes d'échappements pratiques sont : **\n** (fin de ligne), **\t** (tabulation) **\"** (placer un guillemet à l'intérieur d'un String) et **** (placer une barre oblique inverse à l'intérieur d'un String).

Cependant, il est important de noter que les caractères (type de base **char**) en Java sont en unicode, avec l'encodage UTF-16, ils ont donc 16 bits. Contrairement à C++, ils ne devraient être utilisés comme des entiers de petite taille ou des octets (le type **byte** existe pour ça en Java). Vous ne devriez pas non plus faire des conversions implicites entre le type **char** et le type **int**.

Exercice 2 : Diviser en méthodes

Dans ce cours, vous aurez à développer des programmes en utilisant le paradigme orienté-objet. Il est donc pertinent de commencer à penser à notre dessin comme si c'était un objet, ou une collection d'objets.

Pour l'instant, vous pouvez imaginer que si on veut améliorer le dessin, il faudra considérer ses diverses parties séparément.

Le dessin se divise en trois parties :

-Le **cargo** (le triangle en haut de la fusée)

-La **citerne** (la section verticale au centre)

-Le **moteur** (la section la plus basse, comprend 2 lignes)

Écrivez donc trois méthodes : **dessinerCargo**, **dessinerCiterne** et **dessinerMoteur**, chacune étant chargée de dessiner une section de la fusée.

Appelez ensuite ces fonctions à partir de la méthode **main**().

Ici, vous allez rencontrer un problème : les fonctions sont des membres de la classe **Dessin** et ne peuvent être appelées qu'à partir d'un objet de type Dessin. Or, dans la fonction main, il n'existe pas d'objet de ce type présentement.

Déclarez un objet de type Dessin dans la fonction main, comme suit :

```
Dessin fusee = new Dessin();
```

Ensuite, vous serez libre d'appeler ses méthodes avec l'opérateur . (point), par exemple :

```
fusee.dessinerMoteur();
```

Il existe également une autre façon d'appeler ces fonctions, détaillée dans l'encadré suivant.

Finalement, notez que la portée que vous choisissez pour ces nouvelles méthodes peut être aussi restrictive que **private** et aussi permissive que **public**! La raison est que les appels à ces méthodes se trouvent dans la classe où elles sont définies. Lorsqu'on peut mettre une portée restrictive, on doit idéalement le faire, utilisez donc **private**, quitte à le changer plus tard si vous utilisez ces méthodes ailleurs dans le code.

MOT-CLÉ STATIC, VARIABLES ET MÉTHODES DE CLASSE

Le mot-clé **static** dans la déclaration d'une variable ou d'une méthode sert à indiquer que celle-ci sera une variable ou une méthode **de classe**.

Une variable de classe est une variable qui existe en un seul exemplaire, qui est accessible de tous les objets de cette classe.

Par exemple, dans le code suivant :

```
class Test{
    private static int x;
    private int y;
```

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

        Test t1 = new Test();
        Test t2 = new Test();
        t1.x = 5;
        t2.x = 10;
        t1.y = 7;
        t2.y = 13;

        System.out.println(t1.x);
        System.out.println(t2.x);
        System.out.println(t1.y);
        System.out.println(t1.y);
        System.out.println(Test.x);
    }
}
//Affiche: 10, 10, 7, 13, 10
```

La variable x est une variable de classe, la variable y est une variable d'instance. Ceci implique que t1 et t2 partagent la même instance de x, alors qu'ils possèdent chacun une instance de y. Ceci explique pourquoi t1.x affiche 10, c'est parce que la seule instance de x a été modifiée par l'énoncé t2.x=10 avant qu'on affiche. On peut également faire référence à une variable de classe en utilisant le nom de la classe comme préfixe (Test.x par exemple). Ceci implique que les variables de classes qui sont publiques sont en fait des variables globales, parce que les classes sont elles-mêmes globales. Il est acceptable d'utiliser ce mécanisme pour déclarer des constantes globales dans certaines circonstances, les variables globales restant à éviter.
À noter que les méthodes de classe sont donc également globales si elles sont publiques.

Dans ce second exemple :

```
class Test2{
    private static int x;
    private int y;

    public static void main (String[] args){

        Test.hello(); // appel typique si en dehors de Test
        hello(); // équivalent, si on est dans la classe Test
```

```
        Test.printy();

        Test t1 = new Test();

        t1.hello(); // légal, mais porte à confusion
        t1.printy(); // appel normal à une méthode d'instance

        y = 5; // Illégal, pas d'instance courante
        x = 5; // Légal, variable de classe
        this.y =5 //Illégal, this indéfini ici
    }


public static void hello (){

        System.out.println("Hello,World!");
}
public static void printy (){

        System.out.println(y);  //  Illégal,  pas  une  variable  de
                                //  classe
}
```

Dans cet exemple, la méthode **hello**() est une méthode de classe. Elle peut être appelée en utilisant le nom de la classe. Une instance peut appeler la méthode aussi, mais il est déconseillé de le faire, parce que ça peut donner l'impression que la méthode n'est pas une méthode de classe lorsqu'on lit le code qui l'appelle.

Dans une méthode de classe, les variables d'instance de la classe n'existent pas, parce qu'on n'a pas d'objet courant (**this**). Lorsqu'on écrit t1.printy(), alors dans printy, t1 est l'objet courant. Lorsqu'on écrit Test.hello() il n'existe pas d'objet courant dans la fonction hello.

Seules les variables de classe sont accessibles dans une méthode de classe et seules les méthodes de classe peuvent être appelées à partir d'une autre méthode de classe.

Cependant, dans une méthode d'instance, on peut utiliser des variables de classe ou des méthodes de classe.

N.B. La méthode main est nécessairement une méthode de classe. Elle est appelée au démarrage du programme avant que ne puissent exister des instances de la classe qui la contient (nulle part ou mettre Test t1 = new Test(), suivi de t1.main()…).

Exercice 3 : Opérations sur les Strings

Faire un dessin plus réaliste commence par une modification importante : les fusées ont des citernes énormes… en fait, la citerne est de loin la majeure partie de la fusée. Cependant, la hauteur et la largeur de la citerne dépendent de plusieurs facteurs un peu complexes à calculer. Avant de calculer, assurons-nous que l'on peut dessiner une citerne qui a une longueur déterminée par un paramètre.

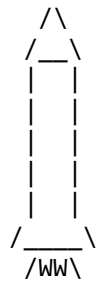
Pour ce faire, vous devriez d'abord modifier votre fonction main() pour qu'elle déclare une variable de type String, puis l'utilise pour accumuler le résultat des méthodes dessinerCargo, dessinerCiterne et dessinerMoteur et l'afficher à la toute fin. Le code devrait donc prendre la forme suivante :

```
public static void main(String[] args)
{
      Dessin fusee = new Dessin();
      String resultat=fusee.dessinerCargo();
      resultat+=fusee.dessinerCiterne(5);
      resultat+=fusee.dessinerMoteur();

      System.out.println(resultat);
}
```

Les méthodes ne devraient donc plus faire les affichages, mais bien retourner des objets de type String. Il est possible de concaténer des Strings à l'aide des opérateurs + et +=. Ceux-ci devraient être utiles lorsque vous voudrez dessiner une citerne qui a un nombre variable d'étages. De plus, vous devrez manuellement placer des sauts de ligne (caractère spécial **\n**) dans vos chaînes de caractères, parce que vous n'appelez plus println (qui

ajoute un saut de ligne) pour chaque ligne. Voici le résultat attendu lors de cet exercice :



À noter, pour la suite des exercices, les fusées risquent de devenir assez hautes, il serait peut-être utile de placer la fenêtre de la console à droite de votre environnement de travail pour mieux les voir. Ceci peut être réalisé en la saisissant par la barre d'onglets et en la faisant glisser à l'endroit désiré.

À noter, vous devrez utiliser le type **int** pour résoudre cet exercice, ainsi que la boucle **for**. Ceux-ci sont semblables à C++. Pour plus de détails sur les types de base comme **int** et les classes comme **String**, voir l'encadré suivant.

REFERENCES, CLASSES ET TYPES DE BASE

Vous aurez remarqué, au cours de cet exercice, que lorsqu'on crée un objet d'un type qui n'est pas un des types de base (comme int), il faut utiliser le mot-clé **new**.

La raison pour laquelle il faut utiliser new est que lorsqu'on déclare une variable d'un type comme Dessin, **aucun objet n'est créé**, contrairement à lorsqu'on déclare une variable d'un des types de base.

La variable en question est une **référence** à un objet, et si elle n'a pas été initialisée, elle ne réfère à rien du tout (**null**).

Par exemple :

```
Dessin ledessin;
```

```
ledessin.dessinerMoteur();
```

Ce code cause une erreur **NullPointerException**. En effet, ledessin ne réfère à rien, parce qu'aucun objet dessin ne lui a été assigné. Accéder à un membre d'un objet **null** cause systématiquement une erreur, et c'est habituellement parce qu'on tente de manipuler une variable non-initialisée.

Par contre :

```
Dessin ledessin;
ledessin=new Dessin();
ledessin.dessinerMoteur();
```

Fonctionnera comme un charme parce que ledessin réfère à un objet de type Dessin qu'on a construit en appelant le constructeur de la classe Dessin.

Qu'en est-il de l'opérateur delete?
En Java, il n'est pas nécessaire de gérer soi-même l'allocation dynamique. Lorsque la machine virtuelle Java détecte qu'il n'y a plus de références à un objet, l'espace qu'il occuppe se trouve libéré. Ceci est réalisé à l'aide d'un procédé qu'on appelle le ramasse-miettes.

Comment est-ce qu'un objet peut être référencé plusieurs fois?
À chaque fois qu'un objet est passé en paramètre, ou qu'on retourne un objet, c'est une référence à celui-ci qui est passée ou retournée. Pour les types de base, c'est toujours une copie de la valeur qui sera passée ou retournée.

Contrairement à C++, où vous pouvez explicitement décider si vous passez :
-Une copie : void f(int x);
-Un pointeur : void f(int* x);
-Une référence : void f(int& x);

En Java, les types de base sont toujours passés par copie et les types complexes par référence, vous n'avez pas le choix.

Ça simplifie le code, mais ça peut poser des problèmes. Modifier des structures de données complexes peut être délicat, sans compter que les passer en paramètre en assumant qu'on a reçu une copie peut causer des problèmes majeurs (modifier l'original en croyant qu'on travaille sur une copie).

En C++, vous savez explicitement à quoi vous avez affaire, et il est également recommandé de bloquer la modifications de paramètres reçus par référence avec le mot-clé const, pour vous éviter des ennuis.

Exercice 4 : Morceaux de fusée

Le dessin de notre fusée est divisé en plusieurs méthodes, qui se comporteront selon les résultats de calculs visant à obtenir les dimensions de la fusée. Afin de mieux séparer le travail, il convient de penser à chacune des parties de la fusée de façon indépendante. Au lieu d'un Dessin, on aurait donc maintenant une Fusee, composée de 3 objets : un cargo, une citerne et un moteur. Chacun de ces morceaux est capable de produire le dessin le représentant. Le dessin représentant la fusée serait donc l'accumulation des dessins de chacun des morceaux, effectués dans le bon ordre.

-Changez le nom de la classe Dessin pour Fusee.

> *N.B. : Il est impropre de mettre des accents dans les noms de classes, méthodes ou variables en java (problèmes de compilation).*

-Créez trois nouvelles classes : **Cargo**, **Citerne** et **Moteur**.

-Ajoutez à la classe **Fusee** une variable d'instance de chacun de ces types.

-Déplacez chaque méthode de dessin vers la classe correspondante. Elles peuvent maintenant toutes s'appeler « **dessiner** » seulement.

-Ajoutez une méthode **dessiner** à la classe **Fusee** également, qui appelle les méthodes de chacune de ses parties (cargo, citerne et moteur), puis accumule les résultats et retourne une **String**.

-Ajoutez un constructeur à la classe **Fusee**. Celui-ci prend la hauteur de la fusée en paramètre. Le constructeur doit également initialiser chacun des morceaux de la fusée en créant un objet de chaque type (opérateur **new**).

-La classe **Citerne** devrait posséder une variable hauteur et dessiner en considérant cette hauteur. Écrivez également un constructeur pour cette classe qui initialise la hauteur.

-La méthode main demeure dans la classe **Fusee**, mais contient désormais :

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

        Fusee fusee = new Fusee(5);

        System.out.println(fusee.dessiner());
    }
```

N.B. Le résultat de l'exécution de votre programme devrait être identique à celui de l'exercice précédent, mais vous avez transformé le code en vue d'accommoder les modifications à venir plus facilement. Cette technique se nomme réusinage (refactoring en anglais).

CLASSES ET ENCAPSULATION

Habituellement, lorsqu'on écrit un programme orienté objet, on assigne des responsabilités claires à chaque classe qu'on crée. Les méthodes des classes devraient principalement opérer sur les variables de cette classe.

La communication avec les objets des autres classes devrait se faire en appelant leurs méthodes, pas en accédant directement leurs variables d'instance. On dit toujours que c'est un grave faux-pas que de mettre une variable d'instance publique, mais la justification n'apparaît clairement que lorsque le code évolue et fait partie d'un ensemble plus grand.

En général, il est souhaitable qu'on puisse modifier le fonctionnement interne d'une classe en continuant de présenter la même interface au code qui se servira d'un objet de cette classe.

Tout ce qui est public, donc visible et utilisable par le reste du code, est l'interface de la classe. Si on encourage le reste du code à accéder directement les variables d'instance, alors on ne peut plus modifier leur type, mettre des contraintes sur les valeurs qu'elles peuvent prendre, ou carrément les faire disparaître et les remplacer par des calculs faits sur d'autres variables qui les simulent. Le code qui utilisait directement les variables d'instance de notre classe devra être modifié rétroactivement, et ce n'est pas toujours possible!

Si le code qui utilise notre classe ne peut qu'appeler des méthodes, alors on est libre de changer comment fonctionnent ces méthodes et quelles variables d'instance elles utilisent dans la mesure où elles produiront le même résultat.

Exercice 5 : Rocket science 101

Maintenant que l'on sait dessiner une citerne de hauteur variable, il est temps de calculer quelle serait sa taille réelle à partir de données techniques provenant de vrais modèles de fusées.

Lorsqu'on veut impartir une certaine vitesse à une fusée, on doit calculer la quantité de carburant qui sera nécessaire à l'aide de la célèbre équation de fusée idéale de Tsiolkosky :

$$\Delta v = I_{sp} \times g_0 \times \ln\left(\frac{m_0}{m_f}\right) \text{---}$$

où :

Δv est la vitesse qu'on désire impartir à notre fusée en mètres par seconde;

I_{sp} est l'impulsion spécifique, représentant la vitesse d'expulsion des gaz par le moteur en secondes;

g_0 est l'accélération gravitationnelle de la terre au niveau du sol (environ 9,8 m/s^2);

ln est le logarithme naturel en base **e**;

$\mathbf{m_0}$ est la masse totale de la fusée, incluant le carburant, en kilogrammes;

$\mathbf{m_f}$ est la masse finale de la fusée, une fois tout le carburant dépensé, aussi nommée la « masse sèche ».

Ce qui nous intéresse ici, c'est d'utiliser cette équation pour déterminer, à partir d'une certaine masse de cargo à envoyer en orbite, quelle quantité de combustible sera nécessaire. On peut facilement obtenir la proportion de masse qui sera du cargo :

a) $\frac{m_f}{m_0} = e^{\left(-\Delta v / I_{sp} \times g_0\right)}$

À partir de ce résultat, il suffit de constater que la proportion de la masse totale qui sera du combustible est alors :

b) $\left(1 - \frac{m_f}{m_0}\right) \times m_0$

À l'aide des équations a) et b) et sachant que :

g_0 = 9,8 m/s^2, l'accélération gravitationnelle au niveau du sol;

Δv = 9700 m/s, la vitesse nécessaire pour se placer en orbite basse;

I_{sp} = 282 s, l'impulsion spécifique du moteur Merlin 1-D de SpaceX;

m_0 = la masse du cargo que vous recevrez en paramètre, plus la masse du moteur Merlin 1-D (470 kg). La masse du reste de la fusée sera ignorée pour l'instant.

1. Écrivez des constantes pour g_0, Δv, $I_{sp,}$ et la masse du moteur dans les classes appropriées. Utilisez le type **double** pour les nombres réels. Une constante se déclare ainsi :

```
public static final double MASSE = 470;
```

2. Ajoutez une variable masse à la classe **Cargo**, ainsi qu'un constructeur qui prend cette valeur en paramètre, puis initialisez le cargo de votre fusée avec la valeur 0 pour l'instant. Ajoutez une méthode getMasse() à la classe **Cargo** qui vous permet d'obtenir la valeur de cette variable à partir de la classe **Fusee**.

3. Modifiez la classe **Fusee** pour que son constructeur reçoive la masse du cargo en paramètre, et le passe au constructeur de **Cargo**.

4. Ajoutez à la classe **Fusee** deux méthodes : **calculerMasseSeche** qui retourne la valeur de m_0, et **calculerMasseCombustible**, qui obtient la masse de combustible à l'aide des équations a) et b), des constantes et de la méthode calculerMasseSeche.

5. Ajoutez un affichage à la fin de votre méthode main qui donne la quantité de carburant requise pour lancer votre fusée en orbite. Les valeurs obtenues en fonction de la masse du cargo devraient suivre le tableau suivant :

Masse Cargo	Masse Combustible
0 kg	15 249 kg
1 000 kg	47 694 kg
5 000kg	177 476 kg
10 000 kg	339 703 kg
15 000 kg	501 930 kg
22 000 kg	729 048 kg

Pour réaliser vos calculs, utilisez la méthode **exp** de la classe **Math**, comme ceci :

```
Math.exp(valeur);
```

C'est une méthode de classe qui calcule la fonction exponentielle.

N.B. Une fusée qui vise à atteindre une vitesse orbitale sans être séparée en plusieurs sous-fusées, même dans des conditions idéales, va pouvoir réserver une quantité très faible de sa masse initiale à son cargo. Ici, on ignore même le poids de la citerne et du châssis et seulement 3% de la fusée constitue la masse du cargo! Avec des fusées plus complexes en 2 ou 3 étages, il est possible d'obtenir des ratios plus près de 4 ou 5%.

Accesseurs (Getters + setters)

Il est habituellement souhaité d'avoir accès à la valeur d'une variable. Il arrive toutefois que la valeur ne devrait pas pouvoir être modifiée directement par une autre classe ou méthode. Afin de pouvoir obtenir ce comportement, les variables doivent être déclarées comme étant privées. Pour accéder aux valeurs des variables, il devient pertinent de définir des méthodes d'accès en lecture et en écriture, le cas échéant.

Une méthode d'accès en lecture s'appelle un **Getter**. C'est une méthode qui a comme type de retour le même type que la variable qu'il accède. Comme c'est une méthode, il est impossible d'assigner une nouvelle valeur à la variable à l'aide d'un **Getter**. Il est possible aussi d'utiliser le **Getter** pour effectuer certaines validations avant de retourner la valeur. Attention cependant, lorsque le type de retour est une référence, si vous modifiez une propriété de l'objet récupéré en référence par le **Getter**, vous modifiez aussi la variable originale! Vous n'aurez pas ce problème avec les types primitifs (**int**, **char**, etc).

Lorsqu'on veut permettre la modification d'une variable privée, on doit définir une méthode qui s'appelle un **Setter**. Cette méthode reçoit en paramètre la nouvelle valeur à assigner à la variable privée. Cette valeur doit être du même type que la variable. Il arrive souvent que dans un **Setter** il y ait des validations supplémentaires sur la nouvelle valeur, ou même du traitement sur des variables connexes.

Les **Getters** et les **Setters** permettent de contrôler la lecture et la modification des valeurs des variables privées et vous permettent d'assurer une meilleure cohérence de vos données d'instances. Ils vous permettent

également de modifier ultérieurement la composition exacte de vos variables sans que ça ait un impact sur le code qui avait besoin d'accéder celles-ci.

Lorsque vous déclarez des méthodes de lecture et d'écriture de variables, vous devriez appeler ces méthodes **getXXXX** et **setXXXX** pour la lecture et l'écriture respectivement, avec les XXXX pour le nom de votre propriété (variable privée).

Exercice 6 : Citerne représentative

À partir du moment où l'on sait calculer la masse de carburant, il est également possible d'en obtenir le volume et ainsi en connaître plus sur les dimensions de notre citerne, afin de faire un dessin plus réaliste.

Le moteur Merlin 1-D utilise une combinaison d'oxygène liquide et de kérosène (RP-1), créant une réaction chimique qui expulse les gaz d'échappement avec une grande puissance.

Pour déterminer la taille de la citerne, il faut connaître la quantité de la masse de combustible qui est de l'oxygène liquide et celle qui est du kérosène, parce que les deux produits n'ont pas la même densité.

La réaction chimique s'opère dans une proportion de 2.56 :1 en faveur de l'oxygène. On peut donc en déduire que :

$m_k = m_c / (1+2.56)$

$m_{lox} = m_c - m_k$

où

m_k est la masse du kérosène;

m_{lox} est la masse de l'oxygène liquide;

m_c est la masse totale de combustible.

Sachant que :

- La densité de l'oxygène liquide est de 1.141 g/cm^3
- La densité du kérosène est de 1.020 g/cm^3
- Le diamètre du moteur Merlin 1-D est de 1.013 m

1. Écrivez une méthode qui calcule le volume total de liquide contenu dans la citerne et détermine sa hauteur en sachant que le cylindre qui constitue la citerne a le même diamètre que le moteur.
2. La formule pour calculer le volume d'un cylindre est : $V = \pi \cdot r^2 \cdot h$ Où V est le volume, r est le rayon, et h est la hauteur. N'oubliez pas d'ajouter des constantes aux classes appropriées pour faire votre calcul.

3. Votre méthode devrait faire partie de la classe **Citerne**. Dorénavant, lorsqu'on crée une citerne, on reçoit en paramètre la masse de combustible qu'elle contiendra et on calcule sa hauteur en utilisant la nouvelle méthode. La hauteur doit être conservée dans une variable d'instance.

4. Modifiez la fonction **dessiner()** de la classe **Citerne** pour tenir compte de la hauteur réelle de la citerne. Pour obtenir un affichage intéressant, assumez que une ligne de texte représente une hauteur de 1m. Affichez également la hauteur de la citerne dans votre fonction main, et comparez-la avec le tableau suivant :

Masse Cargo	Hauteur Citerne
0 kg	17 m
1 000 kg	53 m
5 000kg	199 m
10 000 kg	381 m
15 000 kg	564 m
22 000 kg	819 m

Faites attention aux conversions d'unités!! Faites vos conversions lorsque vous êtes en unités linéaires, pas carrées ou cubiques! Par exemple, 1 m^3 = 1 000 000 cm^3 !! Vos résultats risquent d'être très loin des bonnes valeurs si vous vous trompez dans vos conversions.

La constante Pi est accessible via la classe **Math** de la façon suivante :

```
Math.PI
```

Exercice 7 : Fusée ou pétard mouillé?

Bien que l'on ait calculé la hauteur de la citerne, vous avez probablement constaté que de conserver le diamètre d'un seul moteur donnait des fusées ridiculement hautes…

Il faudrait corriger cette lacune pour rendre notre dessin un peu plus réaliste. Le diamètre de la fusée dépend de la quantité de moteurs qu'on installera à la base de celle-ci.

Le nombre de moteurs, quant à lui, dépend du poids de la fusée qu'on désire soulever.

En effet, les calculs qu'on a faits à date ne tiennent pas compte du poids de la fusée sur terre en comparaison de la poussée exercée par les moteurs.

Pour déterminer de combien de moteurs on aura besoin, il faut connaître la masse totale de la fusée, et la diviser par la puissance des moteurs :

$N_{moteurs} = (m_{totale} / P_{moteur}) + 1$

La masse totale se calcule en faisant la somme de la masse de la citerne, des moteurs et du cargo. Il serait sage d'ajouter à chaque classe une méthode **masse()** qui calcule et retourne la masse de chaque partie.

La poussée des moteurs Merlin 1-D est de 180,1 fois leur propre masse.

On ajoute 1 moteur pour faire un arrondi simplifié.

Ensuite, pour déterminer le diamètre de la fusée, il faut savoir le diamètre du cercle le plus petit pouvant contenir le nombre de moteurs désiré.

Dans le meilleur des cas pour 3 moteurs ou plus, on arrive à occuper 80% de la surface de la base de la fusée avec l'aire de la base des moteurs. La configuration exacte des petits cercles dans le grand cercle pour obtenir la proportion optimale est un problème ouvert en mathématiques.

Si on assume que les moteurs occuperont 80% de l'aire de la base de la fusée, alors on peut déterminer la taille de cette base à partir de la taille des moteurs.

Connaissant le diamètre du moteur :

$$D_{base} = \frac{\sqrt{(D_m/2)^2 * N}}{0{,}80}$$

où N est le nombre de moteurs et D_m est le diamètre d'un moteur.

À partir du diamètre de la base de la fusée, on peut enfin calculer une taille plus réaliste pour la citerne.

1. Changez le nom de la classe **Moteur** pour **BlocMoteur**. Le constructeur prendra maintenant en paramètre la masse de la fusée excepté celle des moteurs, et calculera le nombre de moteurs, pour le conserver dans une variable d'instance. Une méthode **calculerNombreMoteurs** devrait faire ce travail. N'oubliez pas d'ajouter les constantes appropriées à votre classe.

2. Ajoutez une méthode permettant de calculer le diamètre de la fusée en fonction du nombre de moteurs nécessaires.

3. Modifiez la classe Citerne pour que celle-ci prenne en paramètre lors de la construction un BlocMoteur. Déclarez également une variable de ce type dans la classe Citerne et initialisez-la.

4. Modifiez le calcul de la hauteur de citerne en fonction du diamètre de la fusée que le bloc moteur peut vous calculer.

5. Ajoutez à la classe BlocMoteur une fonction qui détermine combien de moteur on va voir sur le dessin (calculerProfil). Ceci est déterminé comme étant combien de fois le diamètre d'un moteur entre dans le diamètre de la fusée, arrondi à l'entier précédent. Par exemple, une fusée à trois moteurs vous permettra de voir le profil de deux moteurs complets.
6. À l'aide de cette largeur de profil, modifiez la fonction qui dessine la citerne pour élargir votre dessin. Vous devrez générer à chaque ligne quatre espaces supplémentaires par moteur de plus que le premier moteur.

7. Modifiez la classe BlocMoteur pour faire le dessin de plusieurs moteurs. Ce dessin se divise en deux parties : le joint entre la citerne et les moteurs, et les moteurs eux-mêmes. La largeur du joint est maintenant augmentée de 4 lignes droites (____) par moteur supplémentaire. Les moteurs, quant à eux, sont simplement dessinés un à côté de l'autre.

8. Modifiez le dessin du cargo pour obtenir une forme triangulaire (en quelque-sorte) qui dépend de la largeur du profil des moteurs (même largeur qu'utilisée pour dessiner la citerne).

Déléguation

Lorsque le code d'une méthode devient difficile à lire, il est généralement souhaitable de la subdiviser en plusieurs petites méthodes qui font une partie claire et simple du travail.

Il en est de même avec les classes. Parfois, une classe évolue et se complexifie au point d'accumuler diverses responsabilités qui ne sont pas fortement liées. Dans ces cas, il convient de se demander si une autre

classe pourrait se charger de cette partie du travail, afin de rendre le code plus clair à lire et facile à modifier. Lorsqu'on donne une partie du travail à faire à une autre méthode ou une autre classe, on dit qu'on le délègue.

Apprendre à déléguer des parties judicieusement choisies du travail réalisé par vos méthodes et vos classes à d'autres méthodes ou classes est essentiel à rendre le code facilement réutilisable, modifiable, testable et compréhensible.

Le dernier exercice de ce labo en est un exemple : modifier l'algorithme de dessin de la citerne sans faire aucune délégation risque de produire du code difficile à lire.

Exercice 8 : Décoration

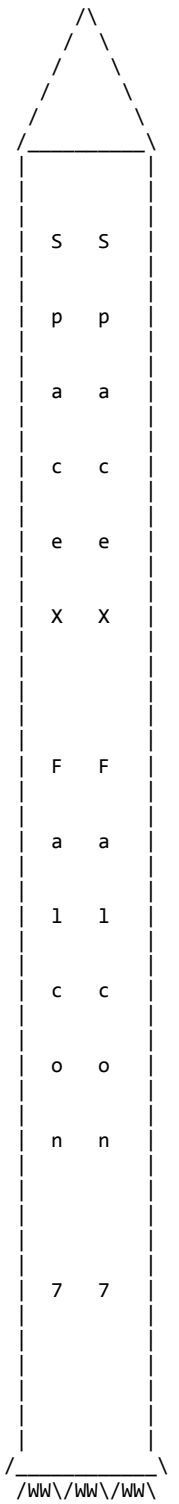
Finalement, il faudrait ajouter de la peinture sur la citerne, afin d'identifier la fusée.

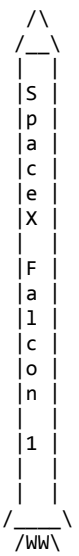
Modifiez la fonction qui dessine la citerne de telle sorte que pour chaque ligne, entre les deux barres droites qui constituent les rebords, on se réfère à une classe Peinture pour déterminer ce qui devrait y apparaître.

Un objet Peinture est initialisé avec la phrase à placer sur la fusée, ainsi que les dimensions de la citerne (en lignes et colonnes).

La classe Peinture fournit une méthode qui produit un String correspondant à ce qui devrait se trouver sur la citerne à une ligne particulière.

La phrase à écrire sur la fusée devrait être : « SpaceX Falcon XX » où XX est le nombre de moteurs de la fusée. De plus, le texte devrait apparaître verticalement, et être répété plusieurs fois si la fusée est plus large. Il ne devrait pas être directement adjacent aux rebords (haut, bas, gauche et droite) de la fusée, sauf si celle-ci est extrêmement mince. Regardez la page suivante pour voir deux exemples de dessins générés.


S S
p p
a a
c c
e e
X X
F F
a a
l l
c c
o o
n n
7 7
/WW\/WW\/WW\


SpaceX
Falcon
1
/WW\