TUTORIEL DART
SESSION 01 LES BASES

" JavaScript tel qu'il est aujourd'hui n'est pas une solution viable à long terme et quelque chose doit changer ", ce quelque chose c'est la naissance d'un langage de programmation objet orienté Web appelé DART.

Introduction

Dart est un langage de programmation structuré **pour le web** purement **orienté objet**, les objets qu'il manipule peuvent être des classes ou des interfaces (que nous verrons plus loin). Si vous êtes habitué à développer en Java, vous n'aurez aucun problème avec ce langage assez prometteur.

Objectifs de Dart

1. Créer un langage structuré mais flexible pour la programmation Web.
2. Rendre DART familier et naturel pour les programmeurs, et donc facile à assimiler.
3. S'assurer que DART fournit de très hautes performances sur tous les navigateurs et les environnements modernes, des appareils mobiles aux fermes de serveurs.

Mise en marche

Il y a plusieurs méthodes permettant d'"exécuter" un script Dart en attendant que les navigateurs supportent pleinement le langage.

1. Par Installation: dart_bin

- Windows. (*attention dart_bin n'est pas stable sur WIndows XP*)
- Linux (Ubuntu 11.04 32bits).
- Mac OS X.

Les fichiers Dart ont l'extension ".dart" et peuvent être exécutés en lançant la commande:

```
$> ./dart_bin [options] monscript.dart
```

Voici les options possibles:

Options du compilateur	
–batch	Mode " Batch " pour les tests unitaires
–check-only	Ne fait que de l'analyse, sans génération de résultat
–expose_core_impl	Import automatique de dart:coreimpl library
–warn_no_such_type	traite les détections de test de type comme un avertissement et non une erreur fatale
–enable_type_checks	Génère le runtime (moteur d'exécution) test de type
–disable-type-optimizations	Désactive l'optimisation des types (for debugging)
–documentation-lib	Génère la documentation pour une bibliothèque donnée
–documentation-out	Répertoire pour la librairie
–generate-documentation	Génération du document de la source
–generate-isolate-stubs	Vérification d'état isolé
–generate_source_maps	Généré la carte des sources
–human-readable-output	Ecriture en langage humain compréhensible
–ignore-unrecognized-flags	Ignore les attributs inconnus
–isolate-stub-out	Fichier qui reçoit la vérification d'état
–jvm-metrics-detail	Affichage de données sommaire (verbose/stats)
–jvm-metrics-format	Modifie l'affichage des métriques
–jvm-metrics-type	Liste des sous argument du contrôle de l'affichage:+ " all: (par défaut) tout type de statistique + " gc: montre les collections de statistique " garbage " + " mem: montre des statistiques sur l'état de la mémoire + " jit: montre les statistique " jit "
–noincremental	Désactive la compilation incrémentale

Optimisation du Javascript généré	
–optimize	Produit un code optimisé
–out	Généré le code JavaScript dans un fichier
–help	Affiche le message d'aide
–jvm-metrics	Affiche les données JVP en fin de compilation
–metrics	Affiche les données de compilation
–fatal-type-errors	Considérer les erreurs de types comme fatale
–fatal-warnings	Traite les avertissement de non typage comme fatale
Options acceptées par le DartRunner	
–compile-only	Compilation sans exécution
–verbose	Affichage des données de diagnostic
–prof	Active l'analyse
–rhino	Utiliser Rhino comme interpréteur JavaScript

2. Compilation en ligne: le "Dartboard"

Il suffit d'aller sur try.dartlang.org, l'interface est simple, c'est celle que nous utiliserons pour ces sessions, elle est composée de 7 parties:

Règles de base

Analyse du code

Tout script Dart ne peut être lancé que s'il passe par des étapes préliminaires d'analyse et de gestion parallèle des erreurs. Les étapes d'analyses sont les suivantes:

- **Analyse lexicale (lexer ou scanner)** – elle reconnait les caractères (ou chaines de caractères) d'entrée (les identificateurs, les mots réservés [while, print...], les opérateurs arithmétique et les affectations) et produit un flux d'unités lexicales (ou lexème) que vous reconnaitrait sur Dart par le terme de ***tokens***. ces chaines de caractères sont délimitées par les séparateurs suivant:

    - les espaces et tabulations (règle dite " skip " ou chaine ignorée)
    - les sauts de lignes (règle dite " skip " ou chaine ignorée)
    - les commentaires (règle dite " special_token ")

Exemple de quelques token Dart:

LPAREN	(
RPAREN	)
LBRACE	{
RBRACE	}
COLON	:
SEMICOLON	;
ASSIGN	=
ASSIGN_ADD	+=
OR	||
AND	&&
ADD	+
SUB	-
MUL	*
EQ	==
NE	!=
LT	<
GT	>
NOT	!
INC	++
CATCH	catch
FINAL	final
FOR	for
IF	if
TRY	try
VAR	var
VOID	void

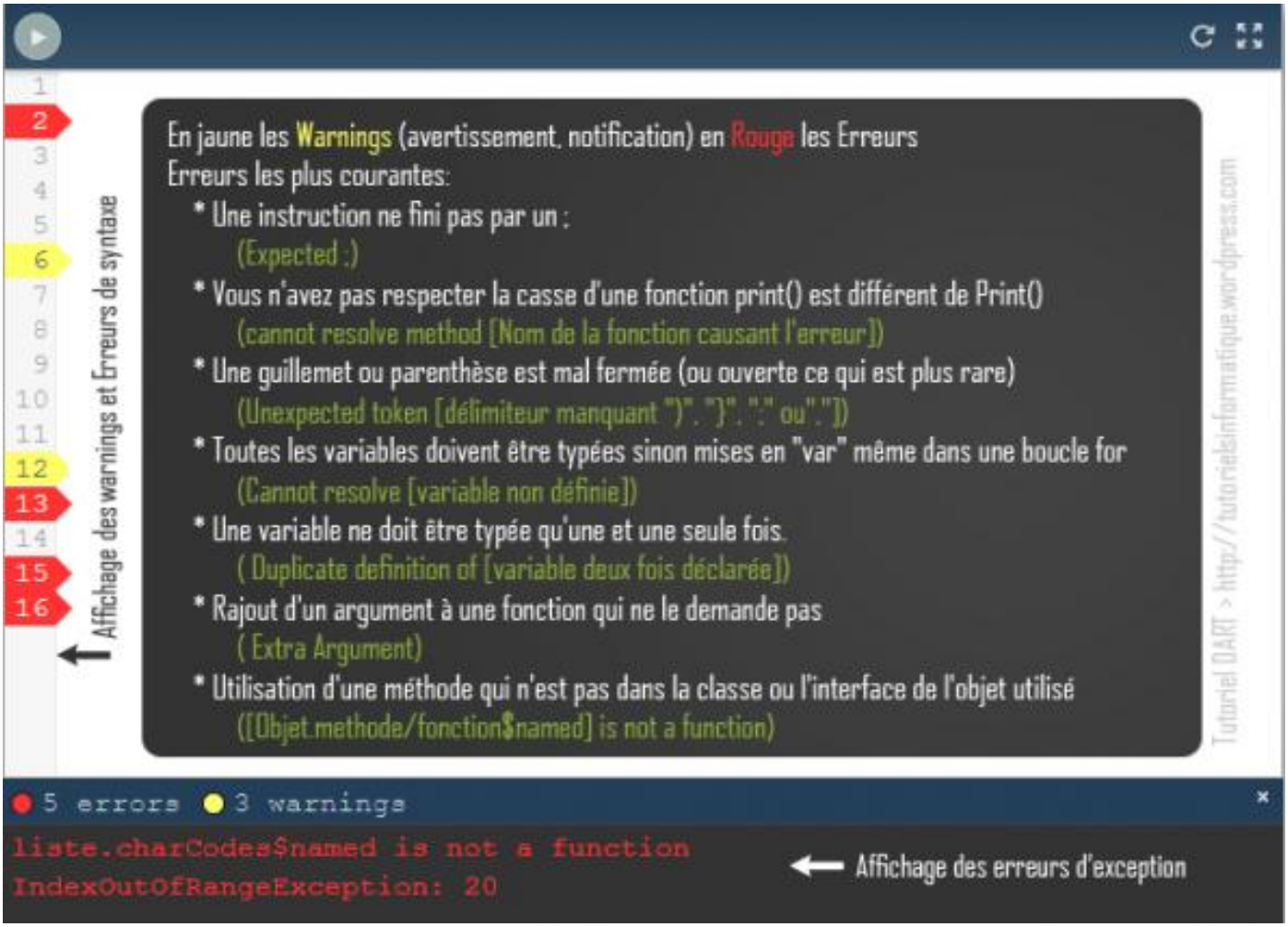
- **Analyse syntaxique (parser)** – elle s'occupe de l'ordonnancement des unités lexicales, cela consiste à prendre ce flux de lexème (ayant une structure linéaire) et à le transformer en un arbre de syntaxe abstraite plus simple à interprété. Un arbre est composé de nœud, chaque nœud est une opération et chaque " fils " d'un nœud représente les arguments de cette opération. Cette arborescence incite a ce que l'interprétation d'un nœud ne se fasse qu'après avoir interprété les sous arbre de ce nœud.
- **Analyse sémantique** – elle vérifie la cohérence des instructions en se basant sur des règles qui garantiront que le script pourra s'exécuter normalement.

Gestion des erreurs

La gestion des Erreurs est particulièrement bien implantée sur Dartboard, comme le montre l'image en dessous. Dans la partie supérieure, on trouve les warning ou avertissements (en jaune) et les erreurs de pre-exécution (en rouge). Dans la partie inférieur on à l'affichage des erreurs plus profondes et imprévisibles. Ce sont des erreurs que l'on peut gérer manuellement (nous dédierons une session sur les options de compilation et la gestion des erreurs).

TUTORIEL DART

SESSION 01 LES BASES



Notre premier script Dart Vierge

Comme en Java, Dart commence par la méthode main()

```
main(){
}
```

[Tester le code]

L'affichage à l'écran

*L'affichage sur Dart est réalisée par la fonction **print()**, fonction qui n'a qu'un seul argument (le contenu à affiché sur la sortie standard). L'élément à afficher doit être mis entre simple quote ou double quote, vous verrez qu'on peut faire des concaténations de chaine de caractère, utiliser des constantes mathématiques, insérer de l'interpolation d'opérations arithmétique, des fonctions ou utiliser un codage ascii tel que \n, \t et \uxxxx, l'utilisation des variables sera vu plus loin.*

```
main(){
print ('Bonjour le monde!');
print (" Hello the World! ");
print (" bienvenue " + " sur " + " tutorielsinformatique.wordpress.com ");
print ('nous verrons plus tard' + 'comment intégrer des variables' + " simplement ");
print ('je rajoute ici une tabulation \t voila 😀');
print ('Et la un saut de ligne \n et Hop! 😀');

print (" on peut faire plusieurs saut de ligne \n \n \n fin du message          " );
print (" Ligne 1 \nLigne2 \n Lign3 \u00A0 Ligne 4 \n Ligne 5");
print (" pour les caractère unicode on fait: (Bonjour) ");
print (" \u0042\u006f\u006e\u006a\u006f\u0075\u0072");
print (" Pour consulter tous les codes, visitez: ");
print (" http://www.unicode.org/charts/ ");
print (" on peut faire de l'interpolation entre accolades précédées d'un \$ ");
print ('on peut utiliser des fonctions en interpolation:');
print (" Nous sommes ${5+5-(8%5)} milliard d'habitant sur Terre ");
print ('ou en concaténation avec la chaine à afficher:');
print (" le nombre PI est égale à: "+ Math.PI);
print (" que l'on peut arrondir à: "+Math.PI.toStringAsFixed(2));
print (" on peut afficher toutes sortes de caractères: ");
print (" ☮ ✈ ♋ 웃 유 ☠ ☯ ♥ ✌ ✖ ☢ ☣ 🍸 🀰 ❁ ♨ ❖ ❤ ❥ ❦ ❧ ♡ ✗ ✘ ⊗ ♒ Ω ♦ ♠ ♥ ♣ ♢ ♤ ♡ ♧ ✦ ✧ ♕ ♕ ♔ ♛ ★ ☆
 ✯ ✭ ☄ ☾ ☽ ☼ ☀ ☘ ☂ ☃ ☻ ☺ 🙾 🎵 🎵 🎵 ✄ ✂ ✆ ✉ ∞ ♂ ♀ ☿ ▲ ▼ △ ▽ ◆ ◇ ◕ ◔ ◡ ϟ ᘐ 回 ₪ ✓ ✔ ✕ ☥ ☨ ☧ ☦ ☩ C·
 ⍟ ⍦ 🕭 ™ © ® ¿¡ №⇨ ❝ ❞ ∃ ⊥ ∀ Ξ ∞ Σ ∏ ⌥ ⌘ 文 Ψ ");
}
```

[Tester le code]

```
main(){
print(" \u00ce\u00f1\u0163\u00e9r "
+ " \u00f1\u00e5\u0163\u0129\u014d\u00f1\u01fb\u013c\u00ee\u015f "
```

```
+ " \u00e5\u0163\u00ee\u1edd\u019e ");
print(" Îñţérñåţĩōñăļîşåţîờn ");
}
```

[Tester de l'internationalisation]

Les commentaires

Les lignes de commentaires sont des lignes que l'on rajoute pour expliquer son code, pour marquer un endroit, pour rajouter des informations de licence (cc). Sur Dart comme sur JavaScript tout ce qui est après deux slash " // " est considéré comme un commentaire et tout ce qui est entre / et */ est considéré comme un bloc de commentaires. Ils font partie dans la compilation / interprétation des éléments qui seront omis par l'analyse syntaxique.*

```
// tutorielsinformatique.wordpress.com
// les lignes 3, 5, 6, 7, 8 et 10 ne seront pas affichées
// ce sont soit des lignes de commentaires
// soit un bloc commenté
main(){
print('ligne n°1 affichée');
print('ligne n°2 affichée');
// print('ligne n°3 non affichée');
print('ligne n°4 affichée');
/* print('groupe de lignes (n°5) non affichée');
print('groupe de lignes (n°6)non affichée');
print('groupe de lignes (n°7)non affichée');
print('groupe de lignes (n°8)non affichée');*/
print('ligne n°9 affichée');// print('ligne n°10 non affichée');
}
```

[Tester le code]

Les variables

- *On peut déclarer les variables tout comme en JavaScript via le terme " **var** " (qui donne le type implicit any) ou en donnant le type exact (**num, int, double, bool, String**)*
- *Ou alors en définissant explicitement la variable par exemple: var i=1; et var j= "1"; la variable (i) est un entier tandis que la variable (j) est une chaine de caractère String (S en majuscule).*
- *Une variable ne peut être déclaré (je parle du type et non de la valeur) qu'une et une seule fois!*
- *Une mauvaise définition de type de variable crée généralement un avertissement (warning) sans pour autant que la compilation ou l'exécution du code soit stoppée.*
- *Une variable de type **final**, ne pas être initialisé que lors de sa déclaration.*
- *Une variable non initialisée est égale à **null**.*

```
main(){
var text0 = "DART";
String text1 = "DART";
int text2 = "DART";
print(text0);
print(text1);
```

```
print(text2);
}
```

En exécutant ce code on obtient un warning disant qu'on ne peut pas assigner de texte dans la variable text2 qui est un entier

[Tester le code]

Remarque: la fonction ***print*** n'accepte qu'un seul argument (donc pas de virgule de séparation d'arguments sinon cela causera le warning " extra argument ") [Tester le code]

Le code suivant montre que Dart est souple au niveau des types mais il faut quand même faire attention àBenfarhat ne pas trop se laisser aller en donnant n'importe quel type ce qui pourrait donner un résultat différent de celui escompté.

```
main(){
int a=1,b=2;
String c="1",d="2";
print (a+b);
print (c+d);
}
```

[Tester le code]

Rappelez vous qu'une variable marquée final doit absolument être initialisée lors de la déclaration !

```
main() {
// la variable i peut être initialisé plus tard
int i;
i=1;
// la variable j (final) doit être initialisé lors de la déclaration
final j;
j=1;
print (" $i $j ");
}
```

[Tester l'exemple]

Affichage d'un texte

Elle s'opère grâce à la fonction " print "! Vous remarquerez que chaque instruction finit par un point-virgule " ; "

```
main(){
var blog="tutorielsinformatique.wordpress.com";
print('ceci est une ligne');
print('ceci est une autre ligne');
print ('bienvenue sur '+blog);
}
```

[Tester le code]

Plus loin avec les variables String

Un triple " double quote " (ou triple quotes simple également) permet de délimiter une variable texte sur plusieurs lignes. Vous constaterez dans l'exemple ci dessous que l'on peut même intégrer une autre variable à l'intérieur (ce n'est pas une variable dynamique!!! le contenu de la variable multi-lignes est affecté une seule fois).

```
main(){
String nom= "Benfarhat elyes ";
var message = "  "" Bonjour $nom
+————————————————————————+
| Vous êtes invité à vous présenter                  |
| .. à nos locaux  pour faire le point sur la mise en place |
| d'applications Dart                                |
+————————————————La Direction———+
"  "";
nom= "Thomas Ben ";
print(message);
}
```

[Tester le code]

Interpolation et évaluation d'une variable

En condition normal les variables ne sont pas précédés d'un dollar '$' sauf à l'intérieur d'une chaine de caractère pour permuter avec le contenu de la variable. Il peut être intéressant de rajouter des guillemets si l'on désire avant interpolation faire une évaluation de la valeur de ce qui est entre guillemet. Ce qui est l'équivalent en JavaScript de ces lignes de code

```
var str = " 8 + 12)

document.write(eval(str))
```

Dans cette exemple j'ai mis plusieurs combinaisons qui je l'espère seront claire quant à l'utilisation du dollar et de l'importance du " *typage* " lors d'une évaluation.

```
main(){
int age=36;
String text="36";
print("Cette année j'ai "+age+" ans");
print("Cette année j'ai age ans");
print("Cette année j'ai $age ans");
print("L'année prochaine j'aurais $age +1 ans");
print("L'année prochaine j'aurais ${age +1} ans");
print("L'année prochaine j'aurais ${text +1} ans");
}
```

[Tester le code]

Appel d'une fonction sans argument

la fonction est définie avant main()

```
int mafonction(){
print('message écrit dans la fonction "mafonction"');
}
main(){
mafonction();
}
```

[Tester le code]

Appel d'une fonction avec argument

```
int mafonction(final nom){
print('bonjour '+nom+'!');
}

main(){
mafonction('toi');
mafonction('vous');
mafonction('dart');
}
```

[Tester le code]

Pour plus d'infos

- Site de Google Dart: http://www.dartlang.org/
- Site de dartboard: http://try.dartlang.org/
- Suivi de l'évolution du code de Dart: http://code.google.com/p/dart/

TUTORIEL DART
SESSION 02 Les Nombres

Dart! Langage de programmation structuré pour le web purement orienté objet, a nous deux! 😃 Allez! dans cette session on va essayer d'explorer le monde des nombres toujours en nous basant sur le DartBoard. Si vous venez d'arriver je vous conseil vivement de lire la première session

Manipulation des Nombres

Déclaration et Initialisation

Comme d'habitude on commence avec notre main() et les types que nous utiliserons ici seront soit num (numérique), int (entier), double (nombre à virgule flottante),bool (valeur booléenne tel que vrai/faux) et sinon par défaut var.

```
main() {
num a=1,b=2, bb=-2, num pi = 3.14159265;
int i=1,y=3;
double z=65.423
……
```

Les instructions d'affectations

Une variable est identifié par son nom qui doit être unique et son type qui permet de dire ce que cette variable représente (un entier, un flottant, une chaine de caractère, un liste etc…). L'opération la plus courante pour une variable est l'affectation qui consiste à donner à une variable, une valeur qui en principe est en adéquation avec le type.

L'exemple le plus simple est l'affectation simple:

```
int a=1; double b=3,14, c=13,1019;
```

On peut stocker dans une variable le résultat d'un calcul ou d'une fonction ce qui donne par exemple:

```
a=5+3;
```

TUTORIEL DART
SESSION 02 Les Nombres

Théorie

Une ***opération mathématique*** est un processus visant à obtenir un résultat en utilisant un ou des symboles spécifiques appelés opérateurs et nécessitant au moins une ***expression***, aussi appelée un ***argument*** ou "***une opérande***" .

Une opération aura donc cette forme:

Résultat [opérateur d'affectation] opérande1 [opérateur1] opérande2 [opérateur2] .. opérandeN

par exemple:

```
x = a + b – 2 * 3 + c / 2
```

L'***opérateur*** d'une opération est le symbole qui désigne le calcul a faire avec les arguments qui l'entourent, il y a de nombreux opérateur, citons en quelque un:

```
+ , – , * , / , % , ! , & , ^ , >> , << , AND etc…
```

Une opération munie d'une seule opérande est appelé ***opération unaire***, avec deux opérandes elle est appelée ***binaire*** et enfin (nous le verrons dans la session relatif aux instructions de contrôles) il existe des opérations à trois opérandes qu'on appelle ***opérations ternaire***.

Opérateurs simplifié d'affectation

Si une variable se trouve dans l'opérande de gauche et dans l'une des deux opérandes de droite d'une affectation alors il est possible de faire une réécriture simplifiée de l'opération, par exemple:

```
a= a+5; peut être simplifiée par a+=5;
```

il ne faut pas uniquement deux opérandes à droites mais deux opérandes (et donc l'usage d'un seul opérateur mathématique) à la suite d'une simplification, un exemple tout bête serait le suivant:

```
a=2+5+6+a; se simplifie par a+=13;
```

Voici quelques exemples de réécritures simplifiées:

```
Soit la déclaration suivante: int x=4; y=2;
x += y est équivalent à: x = x+y résultat: 6
x -= y est équivalent à: x = x-y résultat: 2
x *= y est équivalent à: x = x*y résultat: 8
x /= y est équivalent à: x = x/y résultat: 2
x%=y est équivalent à: x=x%y résultat: 0
```

Les opérateurs incrémentaux

Il s'agit d'une autre écriture de x+=1 et x-=1

```
Soit x=4;
x++ équivaut à l'incrémentation par 1 résultat: 5
++x équivaut à l'incrémentation par 1 résultat: 5
x-- équivaut à la décrémentation par 1 résultat: 3
--x équivaut à la décrémentation par 1 résultat: 3
```

Simple non maintenant voyons la différence entre x++ et ++x (ou x-- et --x)

1èr cas x++

```
var x=1;
var y;
y=x++; // ici on réalise l'instruction puis on incrémente => y=1 et x=2
```

2ème cas --x

```
x=1;
y=++x; // ici on incrémente puis on réalise l'instruction => y=2 et x=2
```

Division et modulo

En arithmétique, la division euclidienne (dite aussi division) entière est une opération qui, à deux entiers naturels appelés dividende et diviseur, associe deux entiers appelés quotient et reste. Selon le format: **Dividende=diviseur * quotient + Reste: Dividende=diviseur * quotient + Reste.**

Par exemple: 36=7*4+8

Le signe "/" représente la division et le signe "%" représente le modulo (reste de la division Euclidienne)

Dans l'exemple précédent 8 qui est le reste représente le résultat de

*Dividende [modulo] (diviseur*quotient)*

Soit D=dq+r
Si le reste (r) est nul on dit que D (dividende) est divisible par d (diviseur) ou que d est un diviseur de a

Soit le code suivant:

```
main(){
int x=4,y=3;
```

```
print (x/y);
print (x%y);
}
```

Cela nous affichera:

```
1.3333333333333333

1
```

Remarque: le modulo est une fonction très importante en informatique (nous l'utiliserons dans la session relative aux dates), elle peut être décrite comme suit:

```
x % y = x – y * (x/y).floor()
```

Oops floor??? Dance floor?? C'est quoi ??et bien nous allons le voir de suite

Les fonctions mathématiques basiques ABS, ROUND, CEIL, FLOOR et TRUNCATE

- La fonction **abs** rend la valeur absolue ce qui reviens en gros a enlever le signe " - " s'il existe. En langage mathématique cela correspond à " ***x*** " si x est positif et " ***-x*** " si x est négatif

```
// -4 -> 4 , 5 -> 5
print (a.abs());
print (bb.abs());
```

- La fonction **round** arrondi à la valeur inférieur si le nombre après la virgule est strictement inférieur à 0.5 et arrondi à la valeur supérieur si le nombre après la virgule et supérieur ou égale à 0.5

```
// 4.3 -> 4 , 4.5 -> 5 , 4.9 -> 5 , -1.51 -> -2
// -1.49 -> -1 , -1.50 -> -1 , -1.51 -> -2
print (pi.round());
```

- La fonction **ceil** donne un arrondissement toujours à la valeur supérieure

```
// -4.3 -> -4 , 4.3 -> 5 , -1.50 -> -1
print (pi.ceil());
```

- La fonction **floor** renvoie la valeur inférieure

```
// -4.3 -> -5 , 4.3 -> 4 , -1.50 -> -2
print (pi.floor());
```

- La fonction **truncate** quant à elle renvoie la valeur sans la partie après la virgule

```
// -1.49 -> -1 , 1.50 -> 1
print (pi.truncate());
```

Les constantes mathématiques

Les constantes sont des valeurs qu'on dit remarquables, elle existe dans tous les domaines, théorie des nombres, théorie de l'information, en analyse, en combinatoire, en physique …

Vous pouvez très facilement définir une constante sur Dart grâce au type final (Voir plus haut Session 01).

```
print(Math.E); // Nombre d'Euler
print(Math.LN10); // logarithme de 10 équivalent à Math.log(10.0)
print(Math.LN2); // logarithme de 2 équivalent à Math.log(2.0)
print(Math.LOG2E); // Base 2 du logarithme de E (Nombre d'Euler)
print(Math.LOG10E); // Base 10 du logarithme de E (Nombre d'Euler)
print(Math.PI); // valeur de Pi 3.141592
print(Math.SQRT1_2);// Racine carrée de 0.5 (1/2)
print(Math.SQRT2); // Racine carrée de 2
```

Tester le code

un mot sur le nombre d'Euler

```
print(Math.exp(1)); // Renvoie E (2.718281)
print(Math.log(Math.E)); // Renvoie 1
print (Math.exp(Math.LN10)); // Renvoie 10.000000000000002
print (Math.exp(Math.LN2)); // Renvoie 2
// Déclaration d'une valeur epsilon
final double EPSILON = 1e-10;
```

Valeur Maximale et valeur minimale

```
// renvoie la valeur maximale
print (Math.max(a,b));
// renvoie la valeur minimale
print (Math.min(a,b));
```

La fonction random " récupérer " un nombre au hasard

La fonction **random** renvoie un nombre au hasard entre 0 et 1.

```
print (Math.random());
```

Pour le renvoi d'un nombre au hasard entre 0 et 10 on peut faire ceci

```
print ((Math.random()*10).truncate());
```

La puissance (waw !) et la Racine carrée

- **Les Puissances:**

```
print (Math.pow(4,2)); // affiche 4 au carré (soit 16)
print (Math.pow(4,3)); // affiche le résultat de 4 à la puissance 3 (soit 64)
```

- **Les Racines carrée**

```
print (Math.sqrt(16)); // retourne 4
print (Math.sqrt(64)); // retourne 8
print (Math.sqrt(-64)); // retourne NaN, comme dans beaucoup de langage Nan veut dire que le résultat retourné n'est pas un nombre (Not a Number )
```

Les fonctions trigonométriques

Sachez que les fonctions trigonométrique cosinus, sinus, tangente, arc cosinus arc sinus arc tangente prennent comme argument une valeur en radian et non en degré.

- **Relation entre le degré et le radian**

1 degre = 0,0174532925199 radian
1 degre = radian * Math.PI / 180 => var radian = angle * Math.PI / 180

- **Les fonctions trigonométriques**

```
print (Math.cos(90));
print (Math.cos(90));
print (Math.sin(90));
print (Math.tan(90));

print (Math.acos(90)); // NaN (Not A Number)
print (Math.asin(90)); // NaN (Not A Number)

print (Math.acos(0.4));
print (Math.asin(0.4));
print (Math.atan(90));
print (Math.atan2(90,45));
```

Les tests

- Tester le **signe** avec isNegative()

```
num nombre=-1.5;
String resultat= "positive ";
if (nombre.isNegative())resultat= "negative ";
print (" le nombre $nombre est $resultat ");
```

- Tester la **parité** avec odd() & even()

```
for (var i=0; i <10; i++) if (i.isEven()) print(i);
for (var i=0; i <10; i++) if (i.isOdd()) print(i);
```

- Tester si le résultat **n'est pas un numérique** NaN

```
    if(Math.acos(90).isNaN()) print (" Attention le résultat n'est pas un nombre ");
if(Math.cos(90).isNaN()) print (" Attention le résultat n'est pas un nombre "); else print (" Pas de NaN détécté ");
var testNan = " Excalibur 😀" ;
if(Math.pow(testNan,3).isNaN()) print (" Attention le résultat n'est pas un nombre ");
```

- Tester si un nombre est **infini**, quand le nombre est supérieur à 1.7976931348623157E+10308. (positif ou négatif)

```
var division=Math.parseInt("1")/Math.parseInt("0");
if (division.isInfinite()) print ("Résultat Infini!");
division=Math.parseDouble("1")/Math.parseDouble("0");
if (division.isInfinite()) print ("Résultat Infini!");
```

- Un mot sur **parseInt**, **parseDouble** et **Expect.equals**

1. La fonction **Expect.equals** " Expect.equals(a,b) " est un test qui verifie que le résultat de l'expression B donne A, si c'est faux il retourne une alerte, sinon il ne fait rien. Par exemple: *Expect.equals(8, 4+5);* retournera le message d'erreur suivant: *Expect.equals(expected: <8>, actual: <9>) fails.*
2. La fonction **ParseInt** permet de convertir une chaine en un entier int (ou " NaN " [Not a Number] si c'est impossible)
3. La fonction **ParseDouble** permet de convertir une chaine en un décimal (ou " NaN " si c'est impossible)

```
Expect.equals(0, Math.parseInt(" 0"));
Expect.equals(0, Math.parseInt(" +0"));
Expect.equals(0, Math.parseInt(" -0"));
Expect.equals(0, Math.parseInt( " 0 " ));
Expect.equals(0, Math.parseInt( " +0 " ));
Expect.equals(0, Math.parseInt( " -0 " ));
Expect.equals(10, Math.parseInt(" 010"));
```

```
Expect.equals(-10, Math.parseInt(" -010"));
Expect.equals(10, Math.parseInt( " 010 " ));
Expect.equals(-10, Math.parseInt( " -010 " ));

Expect.equals(499.0, Math.parseDouble(" 499"));
Expect.equals(499.0, Math.parseDouble(" 499.0"));
Expect.equals(499.0, Math.parseDouble(" 499.0"));
Expect.equals(499.0, Math.parseDouble(" +499"));
Expect.equals(-499.0, Math.parseDouble(" -499"));
Expect.equals(499.0, Math.parseDouble( " 499 " ));
Expect.equals(499.0, Math.parseDouble( " +499 " ));
Expect.equals(-499.0, Math.parseDouble( " -499 " ));

Expect.equals(0.0, Math.parseDouble(" 0"));
Expect.equals(0.0, Math.parseDouble(" +0"));
Expect.equals(-0.0, Math.parseDouble(" -0"));

Expect.equals(true, Math.parseDouble(" -0").isNegative());

Expect.equals(0.0, Math.parseDouble( " 0 " ));
Expect.equals(0.0, Math.parseDouble( " +0 " ));
Expect.equals(-0.0, Math.parseDouble( " -0 " ));
Expect.equals(10.0, Math.parseDouble(" 010"));
Expect.equals(-10.0, Math.parseDouble(" -010"));
Expect.equals(10.0, Math.parseDouble( " 010 " ));
Expect.equals(-10.0, Math.parseDouble( " -010 " ));

Expect.equals(0.1, Math.parseDouble(" 0.1"));
Expect.equals(0.1, Math.parseDouble( " 0.1 " ));

Expect.equals(0.1, Math.parseDouble( " +0.1 " ));
Expect.equals(-0.1, Math.parseDouble( " -0.1 " ));

Expect.equals(0.1, Math.parseDouble(" .1"));
Expect.equals(0.1, Math.parseDouble( " .1 " ));

Expect.equals(0.1, Math.parseDouble( " +.1 " ));
Expect.equals(-0.1, Math.parseDouble( " -.1 " ));

Expect.equals(1234567.89, Math.parseDouble(" 1234567.89"));
Expect.equals(1234567.89, Math.parseDouble( " 1234567.89 " ));

Expect.equals(1234567.89, Math.parseDouble( " +1234567.89 " ));
Expect.equals(-1234567.89, Math.parseDouble( " -1234567.89 " ));
```

Les opérations Binaires

On retrouve dans les opérations binaires, les **opérations bits à bits** tel que (!, &, |, ^), les **opérateurs de décalages** (>>, <<, >>>) et les **opérateurs logiques** (&&, ||)

- La fonction AND imaginez deux robinets l'un après l'autre, le robinet A et le robinet B, si A est fermé et B fermé il n'y a pas d'eau, Si A est ouvert et B fermé il n'y a pas d'eau (puisque B va couper l'arrivé), Si B est ouvert et A fermé même chose, par contre si A et B sont ouvert alors il y a de l'eau! et bien c'est la fonction AND, deux robinet en série l'un après l'autre

Entrée		Sortie
A	B	A ET B
0	0	0
0	1	0
1	0	0
1	1	1

- La fonction OR c'est deux robinets en parallèle il suffit qu'un des deux robinets soit ouvert pour qu'on ai de l'eau

Entrée		Sortie
A	B	A OU B
0	0	0
0	1	1
1	0	1
1	1	1

- La fonction NOT inverse l'état de 0 à 1 et de 1 à zéro, si un robinet est ouvert il le ferme, s'il est fermé il l'ouvre :

Entrée	Sortie
A	NON A
0	1
1	0

- Les fonctions NAND, NOR, XOR et XNOR

NAND: deux robinets en série, ici c'est vrai (valeur 1) si l'eau n'arrive pas donc si un des robinets est fermé donc si A ou B est à 0.

Entrée		Sortie
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

NOR: deux robinets en parallèle, C'est N pour Non donc on veut que l'eau n'arrive pas (valeur 1) que si les deux robinets sont fermés (donc A et B sont égale à 0)

Entrée		Sortie
A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

XOR: Pour s'en rappeler dites vous qu'en gros si les robinets n'ont pas le même état alors la valeur est à 1 (avec plus d'éléments binaire c'est un peu plus compliqué)

Entrée		Sortie
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR: L'inverse de XOR, si les deux robinets ont le même état (la même valeur) alors la sortie est un 1

Entrée		Sortie
A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Le code source

```
print(2 & 5); // 2 en binaire c'est 0010 et 5 c'est 0101
// en lés mettant l'un en dessous de l'autre, aucun 1 l'un sous l'autre -> resultat 0
print(7 & 13);// 7 en binaire c'est 0111 et 13 c'est 1101, la fonction AND donne 00101 -> resultat 5
print(2 | 5); // 0010 OR 0101 donne 0111 (7)
print(7 | 13);// 0111 OR 1101 donne 1111 (15)
print(2 ^ 5); // 0010 XOR 0101 donne 0111 (7)
print(7 ^ 13);// 0111 XOR 1101 donne 1010 (10)
print(~9); // le resultat (sur 32 bits) négative veut dire qu'au lieu de lire avec des 1 lisez avec des zeros
// 00000000000000000000000000001001 – > 11111111111111111111111111110110
// donc sans faire de calcul -> print(~x); affichera -(x + 1)
print(~7);print(~16);print(~291); // on aura ici -8, -17 -292
print(7<<1); // on décale 7 (0000111) d'un cran vers la gauche ce qui donne 14 (0001110)
print(12>>1); // on décale 12 (00001100) d'un cran vers la droite ce qui donne 6 (000000110)
print(12>>2); // on décale 12 (00001100) de deux crans vers la droite ce qui donne 3 (000000011)
print(1>>1); // retourne 0! -> pas de boucle pour dire que ce qui sort lors du décalage est perdu
```

[Tester le code des fonctions et opérateurs présentés jusqu'ici]

Les conversions

Un nombre hexadécimal commence toujours par 0x, par exemple l'hexadécimal EF7 sera noté 0xef7

- **Convertir un hexadécimal en décimal**

```
main(){
    print(Math.parseInt(" 0xef7")); // affiche 3831
}
```

Remarque: l'octal est aujourd'hui abandonnée au profit de la base 16

- **convertir un décimal en hexadécimal**

```
var nombre=3831;
print(nombre.toRadixString(16)); //ef7
// Sans passage par une variable
print((255).toRadixString(16));
```

Les conversions de systèmes et simplification (décimal et exponentiel)

- Le nombre de chiffre après la virgule peut être fixé par la fonction toStringAsFixed()

```
// soit le nombre Pi
print(Math.PI); // retourne 3.141592653589793
print((Math.PI).toStringAsFixed(2)); // retourne 3.14
print((Math.PI).toStringAsFixed(3)); // retourne 3.142
print((16).toStringAsFixed(0)); // retourne 16
print((16).toStringAsFixed(1)); // retourne 16.0
print((16).toStringAsFixed(4)); // retourne 16.0000
```

- La Précision ou nombre de chiffre affichés, ici que ce soit avant ou après la virgule on arrondi (fonction ceil) et on coupe via l'utilisation de la fonction toStringAsPrecision().

```
// Imaginons le nombre 1607 et voyons l'utilité pour la division en millier

// print((1607).toStringAsPrecision(0)); // ne fonctionne pas => precision 0 out of range

print((1607).toStringAsPrecision(1)); // retourne 2e+3
print((1607).toStringAsPrecision(2)); // retourne 1.6e+3
print((1607).toStringAsPrecision(3)); // retourne 1.61e+3

// imaginons à présent un nombre inférieur à 1000 (par exemple 80) et voyons le résultat

print((80).toStringAsPrecision(1)); // retourne 8e+1
print((80).toStringAsPrecision(2)); // retourne 80
print((80).toStringAsPrecision(3)); // retourne 80.0
```

- Forcer l'affichage de la précision avec ou sans exponentiel

Utilisation de la fonction toStringAsExponential().

```
// si on veut forcer l'affichage de l'exponentiel
print((80).toStringAsPrecision(2)); // ici cela affiche 80 (sans exponentiel)
print((80).toStringAsExponential(2)); // retourne 8.00e+1
```

La méthode remainder()

La fonction remainder est l'équivalent de l'opérateur modulo " % "

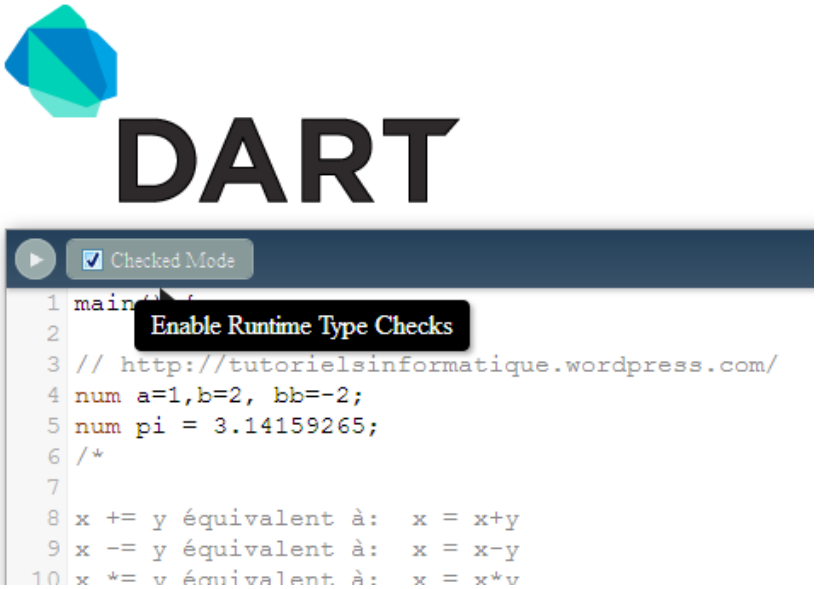
```
print(1%5); print((1).remainder(5));
print(2%5); print((2).remainder(5));
print(3%5); print((3).remainder(5));
print(4%5); print((4).remainder(5));
print(5%5); print((5).remainder(5));
print(6%5); print((6).remainder(5));
print(7%5); print((7).remainder(5));
print(216%5); print((216).remainder(5));
```

Un marge d'erreur à prendre en compte

Pour revenir aux fonctions trigonométrique, nous avions précisé que celles ci utilisent le radian qui peut être obtenu a partir du degré en multipliant par Pi et divisant par 180 or Pi transcendant et surtout irrationnel. La conversion et conversion inverse ne donnera pas le même résultat et causera une marge d'erreur. par exemple cosinus de 90° est égale à 0 or:

```
print(Math.cos((90*Math.PI)/180));  // Dart retourne 6.123233995736766e-17
```

En java il existe les fonctions Math.toRadians(x) and Math.toDegrees(x)qui permettent de convertir d'une unité de mesure radian vers l'unité degré et vice et versa. Seulement elle n'est pas " encore? " implémentée dans Dart. Qui sait ce que l'équipe de développement nous réserve? Nous observons déjà des améliorations du DartBoard depuis la mise en ligne de la session 1 qui se voit rajouter un case à coché permettant d'activer la compilation avec l'option " –enable_type_checks "



[Tester le code]

Sessions à suivre …

Rendez vous sur http://tutorielsinformatique.wordpress.com/ pour le reste des sessions