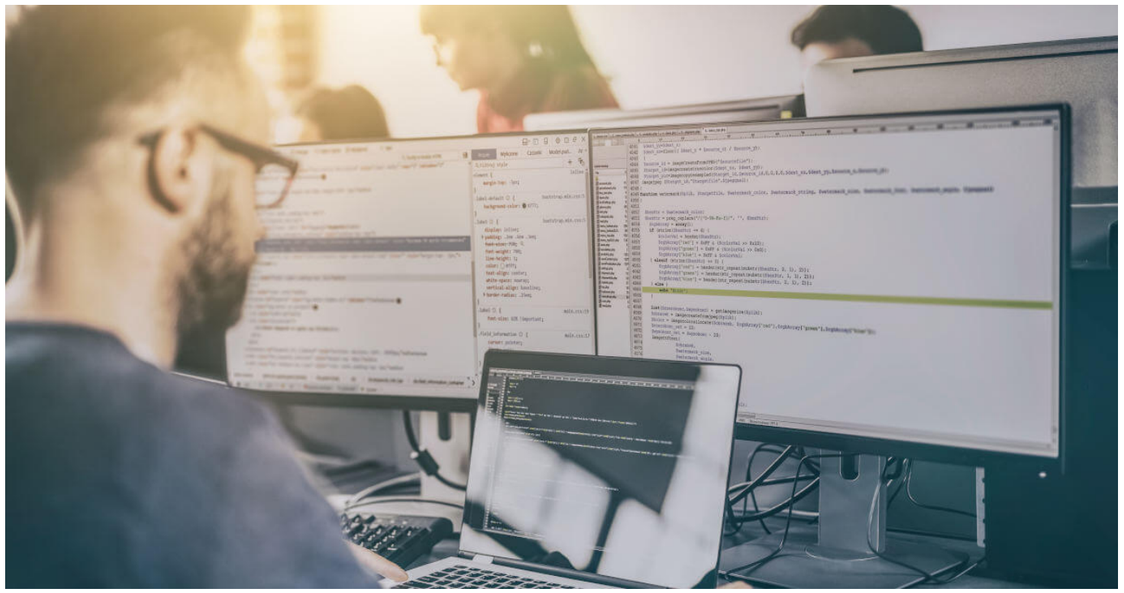
07.01.21
| Développement web (sites-internet/developpement-web/)



Tutoriel sur Rust

Rust est un langage de programmation de **Mozilla**. Il peut être utilisée pour écrire des outils en ligne de commande, des applications Web et des programmes de réseau. Le langage est également adapté à la programmation pour hardware.

Dans ce tutoriel sur Rust, nous vous montrons **les caractéristiques les plus importantes du langage**. Ce faisant, nous examinerons les similitudes et les différences avec d'autres langues similaires. Nous vous guiderons à travers l'installation de Rust et vous apprendrez comment écrire et compiler le code Rust sur votre propre système.

Sommaire

4. Apprendre les constructions de programmation supérieure de Rust (sites-internet/developpement-web/tutoriel-rust/#c322151)

Aperçu du langage de programmation Rust

Rust est un **langage compilé**, ce qui lui confère des performances élevées ; en parallèle, le langage propose des abstractions sophistiquées qui facilitent le travail du programmeur. Rust s'intéresse tout particulièrement à la sécurité de la mémoire. Cela donne au langage un avantage particulier par rapport aux anciens langages tels que C et C++.

Utiliser Rust sur son système

Comme Rust est un logiciel open-source gratuit (FOSS), tout le monde peut télécharger la chaîne d'outils Rust et l'utiliser sur son système personnel. Contrairement à Python ou JavaScript, Rust n'est pas un langage interprété. Au lieu d'un interpréteur, on utilise un compilateur, comme en C, C++ et Java. En pratique, cela signifie qu'il y a **deux étapes pour exécuter le code** :

1. **Compiler le code source**. Cela permet de produire un exécutable binaire.
2. **Exécuter le binaire** résultant.

Il est possible que les deux étapes soient simplement contrôlées depuis la ligne de commande.

Conseil

Dans un autre article du Digital Guide, nous examinons de plus près la différence entre un *compiler* et un *interpreter*.

Avec Rust, des bibliothèques peuvent être créées en plus des fichiers binaires exécutables. Si le code compilé est un programme directement exécutable, une fonction **main()** doit être définie dans le code source. Comme en C / C++, cela sert de point d'entrée dans l'exécution du code.

Installer Rust pour le tutoriel sur le système local

Pour utiliser Rust, il faut d'abord effectuer une installation locale. Sous macOS, vous pourrez utiliser le gestionnaire de paquets Homebrew (https://brew.sh/index_fr). Le homebrew fonctionne également sous Linux. Ouvrez une ligne de commande (« Terminal.App » sur Mac), copiez la ligne de code suivante dans le terminal et exécutez-la :

```
1 brew install rust
```

📌 Remarque

Pour installer Rust sur Windows ou tout autre système sans Homebrew, utilisez l'outil officiel Rustup (https://rustup.rs/).

Pour vérifier que **l'installation de Rust a réussi**, ouvrez une nouvelle fenêtre sur la ligne de commande et exécutez le code suivant :

```
rustc --version
```

Si Rust est correctement installé sur votre système, **la version du compilateur Rust** vous sera présentée. Si un message d'erreur apparaît à la place, redémarrez l'installation.

Compiler du code Rust

Pour compiler le code Rust, vous avez besoin d'un fichier de code source Rust. Ouvrez la ligne de commande et exécutez les codes suivants. Nous allons d'abord créer un dossier pour le tutoriel Rust sur le bureau et passer sur ce dossier :

```
cd "$HOME/Desktop/"
mkdir tutoriel-rust && cd tutoriel-rust
```

Ensuite, nous créons **le fichier de code source Rust** pour un exemple simple avec « Hello, World » :

```
cat << EOF > ./tutoriel-rust.rs
fn main() {
    println!("Hello, World!");
}
EOF
```

📌 Remarque

Les fichiers de code source Rust se terminent par l'abréviation *.rs*.

Enfin, nous allons **compiler le code source Rust et exécuter le binaire** qui en résulte :

```
# Compiler du code source Rust
rustc tutoriel-rust.rs
# Exécuter le binaire résultant
./tutoriel-rust
```

Conseil

Utilisez la commande *rustc tutoriel-rust.rs && ./tutoriel-rust*, pour combiner les deux étapes. Pour compiler et exécuter de nouveau votre programme avec la ligne de commande, appuyez sur la flèche qui part vers le haut puis la touche Entrée.

Gérer les paquets Rust avec Cargo

Outre le langage Rust proprement dit, il existe un certain nombre de paquets externes. Ces **« crates »** peuvent être obtenues dans le Rust Package Registry (https://crates.io/). L'outil Cargo installé avec Rust est alors utilisé. La commande *cargo* est utilisée en ligne de commande pour installer des paquets et en créer de nouveaux. Vérifiez que le Cargo a été installé correctement :

```
cargo --version
```

Apprendre les bases de Rust

Pour apprendre Rust, nous vous recommandons d'essayer vous-même les exemples de codes. Vous pouvez utiliser le fichier *tutoriel-rust.rs* déjà créé à cet effet. Copiez un échantillon de code dans le fichier, compilez-le et exécutez le binaire résultant. Pour que cela fonctionne, l'extrait de code doit être inséré dans la fonction *main()* !

Vous pouvez également utiliser le Rust Playground (https://play.rust-lang.org/) directement dans votre navigateur pour essayer le code Rust.

Instructions et blocs

Les instructions constituent la base du code Rust. Une instruction **se termine par un point-virgule (;)** et, contrairement à une expression, ne renvoie pas de valeur. Plusieurs instructions peuvent être regroupées en un seul bloc. Les blocs sont délimités par des accolades "{}", comme en C/C++ et Java.

Commentaires dans Rust

Les commentaires sont une caractéristique importante de tout langage de programmation. Ils sont utilisés à

la fois pour documenter le code et pour planifier le déroulement futur de votre code. Rust utilise **la même syntaxe de commentaire que C, C++, Java et JavaScript** : tout texte après une double barre oblique est interprété comme un commentaire et ignoré par le compilateur :

```
// Ceci est un commentaire
// Un commentaire,
// peut être écrit
// sur plusieurs lignes.
```

Variables et constantes

Dans Rust, on utilise le mot-clé « let » pour déclarer une variable. Une variable existante peut être déclarée de nouveau dans Rust et donc « éclipsée ». Contrairement à de nombreuses autres langues, **la valeur d'une variable ne peut pas être modifiée** facilement :

```
// Déclarer la variable « age » et y associer la valeur « 42 »
let age = 42;
// La valeur de la variable « age » ne peut pas être modifiée
age = 49; // Erreur compiler
// avec de nouveau « let », la variable peut être écrasée
let age = 49;
```

Pour marquer la valeur d'une variable comme **modifiable ultérieurement**, Rust a défini le mot-clé « mut ». La valeur d'une variable déclarée avec « mut » se modifie donc facilement :

```
let mut poids = 78;
poids = 75;
```

Le mot-clé « const » génère une constante. La valeur d'une constante de Rust doit être connue au moment de la compilation. Le type doit également être spécifié explicitement :

```
const VERSION: &amp;str = &quot;1.46.0&quot;;
```

La **valeur d'une constante ne peut pas être modifiée** - une constante ne peut pas non plus être déclarée comme « mut ». En outre, une constante ne peut pas être re-déclarée :

```
// Définir une constante
const MAX_NUM: u8 = 255;
MAX_NUM = 0; // Erreur de compilation, car la valeur d'une constante ne peut être
const MAX_NUM = 0; // Erreur de compilation, car la constante ne peut pas être dé
```

La notion de propriété dans Rust

L'une des caractéristiques déterminantes de Rust est le concept de propriété (angl. « Ownership »). La propriété est **étroitement liée à la valeur des variables**, à leur durée de vie et à la **gestion de la mémoire des objets** en tas (heap). Lorsqu'une variable quitte son champs d'application (scope), sa valeur est détruite et la mémoire est libérée. Rust peut donc se passer du garbage collection, ce qui permet d'accroître ses performances.

Chaque valeur dans Rust appartient à une variable - le propriétaire. Il ne peut y avoir **qu'un seul propriétaire pour chaque valeur**. Si le propriétaire transmet la valeur, alors il n'est plus propriétaire :

```
1  let nom = String::from("Jean Dupont");
2  let _nom = nom;
3  println!("{}, world!", nom); // Erreur de compilation, car la valeur de « nom » a
```

Un soin particulier doit être apporté à la définition des fonctions : si une variable est passée à une fonction, le propriétaire de la valeur change. La variable **ne peut pas être réutilisée après l'appel de fonction**. Ici, Rust utilise une astuce : au lieu de passer la valeur à la fonction elle-même, une référence est déclarée avec le symbole de l'esperluette (&). Cela permet d'"emprunter" la valeur d'une variable. Voici un exemple :

```
1  let nom = String::from("Jean Dupont");
2  // le type du paramètre « nom » est défini comme « String » et non « &String »
3  // la variable « nom » ne peut plus être utilisée après l'appel de la fonction
4  fn hello(nom: &String) {
5    println!("Hello, {}", nom);
6  }
7  // l'argument de la fonction doit également être
8  // marqué avec "&" pour référence
9  hello(&nom);
10 // cette ligne sans utilisation de la référence conduit à une erreur de compilat
11 println!("Hello, {}", nom);
```

Structures de contrôle

Une particularité fondamentale du langage est de rendre **le déroulement du programme non linéaire**. Un programme peut se ramifier, et les composants du programme peuvent être exécutés plusieurs fois. Ce n'est qu'à travers cette variabilité qu'un programme devient vraiment utile.

Rust dispose des structures de contrôle disponibles dans la plupart des langages de programmation. Il s'agit notamment des boucles "**for**" et "**while**", ainsi que des ramifications "**if**" et "**else**". Rust présente également des caractéristiques particulières. La construction "match" permet d'assigner des modèles, tandis que "loop" crée une boucle sans fin. Dans la pratique, cette dernière sera utilisée avec « break ».

L'itération

L'exécution répétée d'un bloc de code au moyen de boucles est également connue sous le nom d'itération. L'itération est souvent effectuée sur les éléments d'un conteneur. Comme Python, Rust connaît le concept d'"**itérateur**". Un itérateur abstrait l'accès successif aux éléments d'un conteneur. Prenons un exemple :

```
// Liste de noms
let noms = ["Jim", "Jack", "John"];
// Boucle « for » avec itérateur dans la liste
for nom in noms.iter() {
    println!("Hello, {}", nom);
}
```

Et maintenant, comment écrire une boucle "for" dans le style C/C++ ou Java ? Vous allez donc spécifier **un chiffre de début et un chiffre de fin et faire défiler toutes les valeurs intermédiaires**. Dans ce cas, il existe l'objet "Range" dans Rust, comme dans Python. Cela déclenche à son tour un itérateur sur lequel le mot-clé "for" fonctionne :

```
// Emettre les chiffres de 1 à 10
// Boucle « for » avec pour itérateur « range »
// Attention : le dernier chiffre n'est pas compris !
for nombre in 1..11 {
  println!("Nombre: {}", nombre);
}
// autre code pour un même résultat
for nombre in 1..=10 {
  println!("Nombre: {}", nombre);
}
```

Une boucle « while » fonctionne dans Rust comme dans la plupart des autres langages. Une condition est fixée et le corps de la boucle est exécuté tant que la condition est vraie :

```
// Emettre les chiffres de 1 à 10 avec la boucle ‚while'
let mut nombre = 1;
while (nombre <= 10) {
  println!(Nombre: {}, nombre);
  nombre += 1;
}
```

Il est possible pour tous les langages de programmation de créer une boucle sans fin avec « while ». Normalement, il s'agit d'une erreur, mais il y a aussi des cas d'utilisation qui l'exigent. Rust propose la procédure suivante dans ce cas :

```
// Boucle infinie avec « while »
while true {
  // …
}
// Boucle infinie avec « loop »
loop {
  // …
}
```

Dans les deux cas, le mot-clé « break » peut être utilisé pour sortir de la boucle.

Conditionnel

Les ramifications avec « if » et « else » fonctionne également dans Rust comme dans d'autres langages similaires :

```
1  const limit: u8 = 42;
2  let nombre = 43;
3  if nombre < limit {
4    println!("Sous la limite.");
5  }
6  else if nombre == limit {
7    println!("Limite atteinte…");
8  }
9  else {
10   println!("Au-dessus de la limite!");
11 }
```

Le **mot-clé « match » de Rust** est très intéressant. Il a une fonction similaire à celle de « switch » présent dans d'autres langues. Pour exemple, regardez la fonction carte_symbole() dans la section « Types de données composées » (voir plus loin dans cet article).

Fonctions, procédures et méthodes

Dans la plupart des langages de programmation, **les fonctions sont la base de la programmation modulaire**. Les fonctions sont définies dans Rust avec le mot-clé « fn ». Aucune distinction stricte n'est faite entre les concepts connexes de fonction et de procédure. Les deux sont définis de manière presque identique.

Une fonction au sens propre du terme renvoie une valeur. Comme beaucoup d'autres langages de programmation, Rust comprend aussi des **procédures**, c'est-à-dire des fonctions qui ne renvoient pas de valeur. La seule restriction fixe est que le type de retour d'une fonction doit être explicitement spécifié. Si aucun type de retour n'est spécifié, la fonction ne peut pas renvoyer une valeur ; elle est alors définie comme une procédure.

```
1  fn procedure() {
2    println!("Cette procedure ne retourne pas de valeur.");
3  }
4  // Type de retour après l'opérateur ,->'
5  fn moins(chiffreentier: i8) -> i8 {
6    return chiffreentier * -1;
7  }
```

En plus des fonctions et des procédures, Rust connaît également les méthodes de la programmation orientée

En plus des fonctions et des procédures, Rust connaît également les méthodes de la programmation orientée objet. **Une méthode est une fonction qui est liée à une structure de données**. Comme en Python, les méthodes Rust sont définies avec pour premier paramètre « self ». Une méthode est appelée selon le schéma habituel objet.methode(). Voici un exemple de la méthode surface(), liée à une structure de données « struct » :

```
// Définition « struct »
struct Rectangle {
    largeur: u32,
    longueur: u32,
}
// Impémentation « struct »
impl Rectangle {
    fn surface(&self) -> u32 {
        return self.largeur * self.longueur;
    }
}
let rectangle = Rectangle {
    largeur: 30,
    longueur: 50,
};
println!("La surface du rectangle est de {}.", rectangle.surface());
```

Types de données et structures de données

Rust est une langue de typage statique. Contrairement aux langages dits dynamiques comme Python, Ruby, PHP ou JavaScript, Rust exige que le type de chaque variable soit connu au moment de la compilation.

Types de données élémentaires (Primitives)

Comme beaucoup de langages de programmation, Rust connaît aussi quelques types de données élémentaires. Les instances de types de données élémentaires ou *primitives* sont distribuées sur la mémoire du stack, ce qui permet d'augmenter les performances. De plus, les valeurs des types de données élémentaires peuvent être définies en utilisant une syntaxe « littérale ». Cela signifie que les valeurs peuvent être simplement écrites.

Type de données	Explications	Annotation
Integer	Nombre entier	i8, u8, etc.
Floating point	Nombre à virgule	f64, f32
Boolean	Valeur True ou False	bool
Character	Lettre Unicode unique	char
String	Chaîne de caractères Unicode	str

Bien que Rust soit une langue de typage statique, le type d'une valeur ne doit pas toujours être déclaré explicitement. Dans de nombreux cas, le type peut être déduit par le compilateur grâce au contexte (« type inference »). Autrement, le type est explicitement spécifié par une annotation. Dans certains cas, cela peut même être obligatoire :

- Le type retour d'une fonction doit toujours être clairement spécifié.
- Le type d'une constante doit toujours être clairement spécifié.
- Les chaînes littérales doivent être spécialement manipulées pour que leur taille soit connue au moment de la compilation.

Voici quelques exemples clairs d'instanciation de types de données élémentaires avec une syntaxe littérale :

```
// ici, le compilateur reconnaît automatiquement le type de variable
let cents = 42;
// Type annotation : chiffre positif (‚u8' = "unsigned, 8 bits")
let age: u8 = -15; // Erreur de compilation, car la valeur donnée est négative
// Chiffre à virgule
let angle = 38.5;
// équivalent à
let angle: f64 = 38.5;
// valeur true
let connexion_utilisateur = true;
// équivalent à
let connexion_utilisateur: bool = true;
// guillemets simples pour les lettres
let lettre = 'a';
// Guillemets doubles pour les chaînes statiques
let nom = "Dupont";
// avec type explicite
let nom: &'static str = "Dupont";
// autrement : « String » dynamique avec ‚String::from()'
let nom: String = String::from("Dupont");
```

Types de données composées

Les types de données élémentaires correspondent à des valeurs individuelles, tandis que les types de données composées regroupent **plusieurs valeurs**. Rust fournit au programmeur une poignée de types de données composées.

Les instances de types de données composées sont attribuées au stack comme les instances de types de données élémentaires. Pour que cela fonctionne, les instances doivent avoir une taille fixe. Cela signifie également qu'elles ne peuvent pas être modifiées arbitrairement après l'instanciation. Voici un aperçu des types de données composées les plus importants de Rust :

Type de données	Explications	Type d'éléments	Syntaxe littérale
Array	Liste de plusieurs valeurs	Type similaire	[a1, a2, a3]
Tuple	Arrangement de plusieurs valeurs	Tout type	(t1, t2)
Struct	Regroupement de plusieurs valeurs nommées	Tout type	–
Enum	Liste	Tout type	–

Examinons d'abord une structure de données avec « struct ». Nous définissons une personne avec trois champs nommés :

```
struct Personne = {
  prénom: String,
  nomfamille: String,
  age: u8,
}
```

Pour représenter une personne concrète, nous instancions « struct » :

```
let joueur = Personne {
  prenom: String::from("Jean"),
  nomfamille: String::from("Dupont"),
  age: 42,
};
// accéder au champs d'une instance « struct »
println!("L'age du joueur est: {}", joueur.age);
```

« enum » (abréviation de « énumération ») représente **les variantes possibles d'une propriété**. Nous l'illustrons ici par un exemple simple des quatre couleurs possibles d'une carte à jouer :

```
enum CouleurCarte {
  Trefle,
  Pique,
  Coeur,
  Caro,
}
// la couleur d'une carte à jouer spécifique
let couleur = CouleurCarte::Trefle;
```

Rust comprend le mot clé « match » comme **pattern matching**. La fonctionnalité est comparable à la mention « switch » d'autres langues. Voici un exemple :

```
// déterminer le symbole appartenant à la couleur d'une carte
fn carte_symbole(Couleur: CouleurCarte) -> &'static str {
  match couleur {
    CouleurCarte::Trefle => "♣",
    CouleurCarte::Pique => "♠",
    CouleurCarte::Coeur => "♥",
    CouleurCarte::Caro => "♦",
  }
}
println!("Symbol: {}", carte_symbole(CouleurCarte::Trefle)); // renvoie le symbo
```

Un tuple est un arrangement de plusieurs valeurs, qui peuvent être de différents types. Chacune des valeurs du tuple peut être attribuée à plusieurs variables grâce à **une déstructuration**. Si l'une des valeurs n'est pas nécessaire, le trait de soulignement (_) est utilisé comme caractère de remplacement, comme il est d'usage dans Haskell, Python et JavaScript. Voici un exemple :

```
// Définir un jeu de cartes comme tuple
let jeu: (CouleurCarte, u8) = (CouleurCarte::Coeur, 7);
// attribuer les valeurs d'un tuple à plusieurs variables
let (couleur, valeur) = jeu;
// si l'on n'a besoin que de la valeur
let (_, valeur) = jeu;
```

Comme les valeurs des tuples sont ordonnées, elles sont également accessibles via **un index**. L'indexation ne se fait pas entre crochets, mais à l'aide d'une syntaxe de points. Dans la plupart des cas, la déstructuration devrait permettre d'obtenir un code plus lisible :

```
let nom = ("Jean", "Dupont");
let prenom = nom.0;
let nomfamille = nom.1;
```

Apprendre les constructions de programmation supérieure de Rust

Structures de données dynamiques

Les types de données composées déjà introduits ont en commun que leurs instances sont assignées sur le stack. La bibliothèque standard de Rust contient également un certain nombre de structures de données dynamiques couramment utilisées. Les instances de ces structures de données sont attribuées sur le heap.

Cela signifie que la taille des instances peut être modifiée par la suite. Voici un bref aperçu des structures de données dynamiques fréquemment utilisées :

Type de données	Explications
Vector	liste dynamique de plusieurs valeurs du même type
String	séquence dynamique de lettres Unicode
HashMap	attribution dynamique de paires clés-valeurs

Voici un exemple de vecteur en croissance dynamique :

```
// Déclarer le vecteur comme modifiable avec « mut »
```

```
let mut noms = Vec::new();
// ajouter une valeur au vecteur
noms.push("Jim");
noms.push("Jack");
noms.push("John");
```

La programmation orientée objet (POO) avec Rust

Contrairement aux langages tels que C++ et Java, Rust **ne connaît pas la notion de classe**. Néanmoins, il est possible de programmer selon la méthodologie POO. Cette dernière a pour base les types de données déjà présentés. Le type « struct », en particulier, peut être utilisé pour définir la structure des objets.

De plus, Rust a aussi des « traits ». Un trait regroupe **un ensemble de méthodes** qui peuvent ensuite être mises en œuvre de n'importe quel type. Un trait comprend les déclarations de méthodes, mais peut aussi contenir des implémentations.

Un trait existant peut être implémenté par différents types. En outre, un type peut implémenter plusieurs traits. Rust permet donc de composer des fonctionnalités pour différents types sans avoir un héritage commun.

Métaprogrammation

Comme de nombreux autres langages de programmation, Rust permet d'écrire du code pour la métaprogrammation. On peut le résumé par du **code qui génère un autre code**. Dans Rust, cela inclut d'une part les « macros » que vous connaissez peut-être en C/C++. Les macros se terminent par un point d'exclamation (!); la macro « println! » pour la sortie de texte sur la ligne de commande a déjà été utilisée plusieurs fois dans cet article.

D'autre part, Rust connaît aussi les « generics ». Ceux-ci vous permettent d'écrire des codes qui résument plusieurs types. Les génériques sont assez comparables aux templates en C++ ou à ce qui est également désigné par le terme generics en Java. Un générique souvent utilisé dans Rust est « Option<T> » , qui reprend la dualité « None »/ « Some(T) » pour tout type "T".

En résumé

Rust, en tant que langages de programmation système performant, a le potentiel de remplacer les favoris bien implantés que sont C et C++.

07.01.21

| Développement web (sites-internet/developpement-web/)

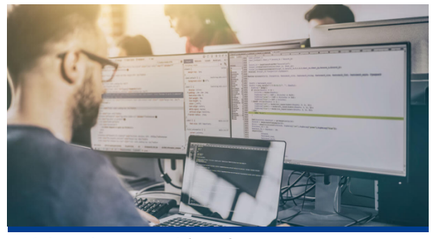
Articles similaires

(sites-internet/developpement-web/tutoriel-gitlab/)

Tutoriel GitLab : installation et premiers pas dans GitLab (sites-internet/developpement-web/tutoriel-gitlab/)

21.10.2020 | Développement web (sites-internet/developpement-web/)

GitLab compte parmi les outils les plus appréciés en développement logiciel agile et collaboratif. Dans ce tutoriel GitLab qui s'adresse aux débutants, vous apprendrez tout ce dont vous avez besoin pour réussir votre prise



(sites-internet/developpement-web/tutoriel-haskell/)

Tutoriel Haskell : apprendre Haskell en toute simplicité (sites-internet/developpement-web/tutoriel-haskell/)

13.10.2020 | Développement web (sites-internet/developpement-web/)

Haskell est un langage incontournable lorsque l'on s'intéresse à l'art de la programmation fonctionnelle. Cependant, l'apprentissage de ce langage constitue une étape conséquente pour les débutants, lorsque les premiers

(sites-internet/developpement-web/tutoriel-programmation-dart/)

Tutoriel DART : les premiers pas (sites-internet/developpement-web/tutoriel-programmation-dart/)

16.10.2020 | Développement web (sites-internet/developpement-web/)

Ce tutoriel sur DART vous permet d'acquérir les bases de la programmation avec DART. Si vous maîtrisez déjà d'autres langages de programmation, vous remarquerez que DART se concentre sur une syntaxe simple et

en main de GitLab. Vous découvrirez notamment quels sont les avantages du logiciel, comment l'installer et comment se déroule concrètement la collaboration en équipe.

obstacles sont surmontés, les progrès se font rapidement sentir. Ce tutoriel Haskell regroupe les étapes, conseils et astuces les plus importants pour vous permettre de débuter…

claire afin d'éviter certaines faiblesses fondamentales de JavaScript. Découvrez les exemples de ce tutoriel sur DART et essayez de les utiliser.