

# Langage Clank : Référence

## 1. Démarrage rapide.

### Présentation

Le langage Clank est un langage permettant de décrire un modèle d'état de serveur dans un langage abstrait, ainsi que des interfaces pour que le serveur et des clients écrits dans des langages différents puissent communiquer par le biais de simples fonctions.

A partir du modèle d'état, et de la description des interfaces, le compilateur Clank génère du code serveur et client (dans les langages demandés) capable de gérer :

- Pour le serveur : la **réception d'une commande** (sous forme de string JSON), et le **renvoi d'une réponse** (sous un format arbitraire). Les mécanismes de réception/envoi de données par TCP ne sont **PAS pris en charge par le langage Clank**, ces mécanismes sont à la charge des développeurs du serveur, pour plus de flexibilité.
- Pour les clients : **l'envoi de commandes** au serveur et **la réception de réponses** par le biais de simples fonctions. L'envoi et la réception de données par TCP est générée (en partie) par le compilateur. **Le client** doit fournir des fonctions permettant :
  - o *D'initialiser la connexion TCP avec le serveur*
  - o *D'envoyer un string au serveur*
  - o *De recevoir un string vers le serveur.*

Pour chaque langage cible, une **classe de génération de code existe**, et cette classe peut être modifiée pour appeler ces fonctions (ou pour changer la manière de sérialiser le code par exemple). Les classes de génération de code sont les seules qu'un utilisateur final doit modifier pour modifier la manière dont le code de sérialisation / envoi par TCP dans un certain langage doit être généré. (Pour plus d'informations consulter la section 5 : Documentation Développeurs).

Le présent document recouvre à la fois une référence exhaustive sur la syntaxe, les fonctionnalités basiques et avancées du langage, certaines parties du code du compilateur, l'utilisation du compilateur, ainsi que des exemples d'utilisation.

### Obtenir le code

Le github du projet est disponible à cette adresse :

<https://github.com/Scriptopathe/clank>

C'est un repository privé, pour obtenir l'accès, demander à : [j\\_alvare@etud.insa-toulouse.fr](mailto:j_alvare@etud.insa-toulouse.fr)

## Démarrage rapide

Cette rubrique a pour but de se familiariser avec le langage et le compilateur.

### *Création d'un script basique*

Créez un fichier script.clank, et copiez-collez le script suivant :

```
main
{
    # Représente l'état du serveur.
    state
    {
        # Déclaration d'une classe qui doit être connue à la fois par le client
        # et le serveur.
        public class Car
        {
            # Une variable accessible par le client et le serveur s'ils disposent
            # d'une instance de Car.
            public string color;

            # Une variable privée.
            int m_speed;

            # Un constructeur
            public constructor Car new(string c)
            {
                color = c;
            }

            # Une fonction publique (exportée chez le client et le serveur)
            public void SetSpeed(int speed)
            {
                m_speed = speed;
                DoSmth();
            }

            public int GetSpeed()
            {
                return m_speed;
            }
            # Une fonction privée (utilisée seulement par les fonctions publiques
            # de cette classe).
            void DoSmth() { }
        }

        # Déclaration d'une classe qui ne sera exportée que sur le serveur.
        # Les fonctions des blocs read/write ne doivent pas retourner d'objet de
        # ce type.
        class PrivateClass
        {
            public int someVariable;
            int somePrivateVariable;
            int somePrivateFunction () { }
        }
        # Déclaration de variables utilisables par le serveur, mais
        # inconnues du client.
        int myVariable;
        Car myCar;
        PrivateClass mySecret;
    }

    # Bloc contenant toutes les fonction permettant au client de récupérer des données depuis le serveur.
    access
    {
```

```

# Les fonctions du block access doivent toutes être publiques.
public int GetMyVariable()
{
    return state.myVariable;
}

}

write
{
    # Exemples de code produisant des warnings
    public void DoBadThings()
    {
        state.mySecret.somePrivateVariable = 6;
        state.mySecret.somePrivateFunction();
    }

    # Une fonction qui modifie l'état du serveur.
    public void DoSecretThing()
    {
        state.mySecret = PrivateClass.New();
    }

    # Fonction qui va retourner et prendre en paramètre des objets dont la classe
    # est spécifiée dans le block state, et est PUBLIQUE !
    # Si la fonction retourne une variable de type PrivateClass, une exception sera
    # levée.
    public Car GetARedOne(Car c, int acc)
    {
        Car newCar = Car.New("red");
        newCar.SetSpeed(c.GetSpeed() + acc);
        return newCar;
    }
}

}

```

### Compilation du script

Exécutez la ligne de commande suivante :

```
Clank.Compiler.exe -server=CS:server.cs -clients=CS:client.cs|JAVA:client.java -src=script.clank
```

Le compilateur doit avoir produit 3 fichiers : server.cs, client.cs, client.java.

Le serveur contient 3 classes : PrivateClass, State et Car. On s'intéresse particulièrement à la classe State :

Server.cs (extrait)

```
/// <summary>
/// Contient toutes les informations concernant l'état du serveur.
/// </summary>
class State
{
    int myVariable;
    Car myCar;
    PrivateClass mySecret;

    /* .... */

    public Car GetARedOne(Car c, int acc, int clientId)
    {
        Car newCar = new Car("red");
        newCar.SetSpeed((c.GetSpeed() + acc));
        return newCar;
    }
    /// <summary>
    /// Génère le code pour la fonction de traitement des messages.
    /// </summary>
    public string ProcessRequest(string request, int clientId)
    {
        Newtonsoft.Json.Linq.JArray o =
(Newtonsoft.Json.Linq.JArray)Newtonsoft.Json.JsonConvert.DeserializeObject(request);
        int fonctionId = o.Value<int>(0);
        switch (fonctionId)
        {
            case 0:
                return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { GetMyVariable(clientId) });
            case 1:
                return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { DoBadThings(clientId) });
            case 2:
                return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { DoSecretThing(clientId) });
            case 3:
                Car arg3_0 = (Car)o[1][0].ToObject(typeof(Car));
                int arg3_1 = o[1].Value<int>(1);
                return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { GetARedOne(arg3_0, arg3_1,
clientId) });
        }
        return "";
    }
}
```

On note la présence de ProcessRequest, LA fonction importante qui à partir d'un message JSON, renvoie une réponse et modifie l'état du serveur.

Regardons désormais côté client : Il n'y a que 2 classes State et Car (la dernière étant privée, seul le serveur la connaît).

Client.cs (extrait)

```
/// <summary>
/// Contient toutes les informations concernant l'état du serveur.
/// </summary>
class State
{

    /* .... */

    public Car GetARedOne(Car c, int acc)
    {
        // Send
        List<object> args = new List<object>() { c, acc };
        int funcId = 3;
        List<object> obj = new List<object>() { funcId, args };
        TCPHelper.Send(Newtonsoft.Json.JsonConvert.SerializeObject(obj));
        // Receive
        string str = TCPHelper.Receive();
        Newtonsoft.Json.Linq.JArray o =
(Newtonsoft.Json.Linq.JArray)Newtonsoft.Json.JsonConvert.DeserializeObject(str);
        return (Car)o[0].ToObject(typeof(Car));
    }
}
```

Chaque fonction de la classe state est utilisable par le client pour récupérer des informations depuis le serveur. La classe TCPHelper doit être codée pour le client, et contenir de quoi : initialiser la connexion vers le serveur, lire et envoyer des données sous forme de string vers/depuis cette connexion.

### Utilisation de l'IDE

Un IDE a été développé pour le langage, il fonctionne sur Windows, et devrait bientôt être fonctionnel sur les systèmes Linux. Voir la section « Clank.IDE ».

### Intégration du script

#### Erreurs / Warning du compilateur

Lorsque le compilateur rencontre quelque chose d'anormal, il peut réagir de 2 manières :

**Envoyer une Error :** Cela se produit lorsqu'une erreur de syntaxe repérée est dans le script ou qu'une erreur empêchant la compilation de se poursuivre se produit. Par exemple, lorsque le compilateur ne peut pas typer une expression, il arrête la compilation et explique pourquoi. Après une erreur en général, aucun autre message ne sera affiché.

**Envoyer un Warning :** Cela se produit lorsqu'une erreur dans la « logique » du code est détectée. Cela inclut les erreurs de typage, d'accessibilité de variables / fonctions, etc... Ces warnings n'empêchent pas la génération des fichiers finaux, mais ceux-ci auront alors des risques de ne pas compiler (une erreur de typage va faire que le client C++ ne compile pas, alors que le client python va s'interpréter correctement, mais avec du code bien pourri :D). **Ils sont à considérer comme des erreurs !**

## 2. Référence Syntaxe

### 2.1. Instructions

#### Déclaration de block nommé

Un block nommé sert à identifier l'utilité d'une portion de code. Les blocs nommés reconnus par le langage Clank sont : *main*, *state*, *access*, *write*, *macro*. (détaillés après).

```
BlockName
{
    # Code
}
```

La totalité du code doit être contenue dans le bloc *main*. Voir la section « blocs nommés » pour plus d'informations sur les blocs *access*, *write*, *macro*, *state*.

#### Déclaration de classe

```
[modifiers] class ClassName<GenericParam1, ...> { }
```

Les modificateurs de classe peuvent être : *public*, *array*, *object*, *serializable* (détaillés plus loin). Les paramètres génériques ainsi définis peuvent être utilisés dans la classe pour représenter un type passé en paramètre à ce type.

Ex :

```
class Container<T> {
    T value ;
}
Container<int> c = Container<int>.new() ;
int val = c.value ;
```

#### Déclaration d'énumération

```
[modifiers] enum EnumName {
    Field1 = 1,
    Field2 = 2,
    ...
}
```

Le seul modificateur accepté est *public*.

#### Déclaration de constructeur de classe

```
public constructor ClassName<GenericParam1, ...> new(ArgType arg1, ArgType arg2, ...) { }
```

#### Déclaration de fonction

```
[modifiers] ReturnType FunctionName(ArgType arg1, ArgType arg2...) { }
```

Les modificateurs peuvent être : *public*, *constructor*, *static*.

#### Déclaration de variable

```
VariableType variableName;
```

#### Affectation de variable

```
variableName = <expression>;
```

Les expressions valides sont détaillées plus tard dans ce document.

## 2.2. Expressions.

Les expressions sont des morceaux de code évaluables, retournant une valeur typée.

### Littéraux

Les littéraux suivants sont supportés : int (digits 0-9), float (digits 0-9 . digits 0-9), string ("str"), bool (true/false).

Exemples :

```
int number = 9;
```

```
bool Boolean = true;
```

```
string str = "hahaha";
```

```
float floatingPointNumber = 0.65;
```

### Groupes d'expression

Les groupes d'expression sont constituées d'une ou deux opérandes et d'un opérateur.

[todo : tableau des opérateurs]

### Référence à une variable

Toute référence à une variable est une expression. Le type de l'expression correspond au type de la variable.

Exemple :

```
int var = 6 ;
```

```
int var2 = var ;
```

### Appel de fonction

Tout appel de fonction est une expression. Le type de l'expression correspond au type de retour de la fonction appelée.

## 2.3. Opérations sur les objets

Considérons la classe suivante :

```
class Dummy<T> {  
    public T variable ;  
    public T Function() { }  
    public static T StaticFunction() { }  
    public constructor Dummy new() { }  
}  
Dummy<int> dum ;
```

### Création d'une instance

```
dum = Dummy<int>.new()
```

### Appel de fonction

Un appel de fonction est considéré comme une expression.

```
dum.Function();
```

### Appel de fonction statique

Un appel de fonction statique est considéré comme une expression.

```
Dummy<int>.StaticFunction() ;
```

## Accès à une variable

dum.variable = dum.variable+1;

## 2.4. Structures de contrôle

### If/else/elsif

Syntaxe :

```
if(boolexpr) { }  
elsif(boolexpr) { }  
else { }
```

### while

Syntaxe :

```
while(boolexpr) { }
```

## 2.5. Tableaux

### Syntaxe des tableaux

Clank a un support built-in limité pour les tableaux.

`bool[]` arr sera évalué à : `Array<bool> arr`. Où `Array` est un type macro (donc customisable).

### Création d'un macro-type se comportant comme un tableau

Les macro-types se comportant « comme » des tableaux (enfin, plutôt des collections génériques) sont particuliers à sérialiser dans certains langages ne profitant pas de la réflexion (C++ par exemple). Il faut donc dire à Clank comment manipuler ces types, et des informations sur comment les sérialiser.

Exemple : la classe `Matrix<T>` :

```
# Représente un tableau bidimensionnel.  
public serializable array class Matrix<T>
```

Le mot clef `array` indique que la classe doit être représentée comme un `array json`. Le type des éléments doit être renseigné par une fonction `getArrayElementType()` :

```
# Type des éléments de l'array : List<T>.  
List<T> getArrayElementType() { }
```

Ces informations permettent à Clank de sérialiser le type correctement.

**Pour l'instant, le type C++ correspondant à ce type doit fournir la méthode *push\_back* permettant d'insérer un élément, ainsi que *begin()* et *end()* pour itérer sur ses éléments. Pour java, c'est la méthode *add()* qui sera utilisée.**



## 2.6. Types built-in

Les types suivants sont disponibles dans le langage :

string, int, float, bool, void, Type, State.

State : correspond au type contenant les déclarations contenues dans le block state.

Type : Représente un type en langage clank. Les référence au type sont typées avec ce type.

Ex : dans ClassName.New(), ClassName est de type Type.

Array : Représente un tableau.

## 2.7. Préprocesseur

Le langage inclut aussi un préprocesseur, fonctionnant de manière analogue au préprocesseur C.

### Directives #include

Syntaxe : #include filename

Le préprocesseur va utiliser un module de chargement afin de charger le script dont l'URI est « filename ». Par défaut, cela va charger le fichier « filename » à partir du dossier d'exécution du compilateur Clank.

*Note (avancé) :*

Il est aussi possible de charger des fichiers dans la mémoire (en modifiant le IncludeLoader du Preprocessor du Générateur, dans le code servant à la compilation) :

```
string generationLog;
GenerationTarget serverTarget = new GenerationTarget("CS", "Serveur.cs");
List<GenerationTarget> clientTargets = new List<GenerationTarget>() { new GenerationTarget("CS", "Client.cs"),
new GenerationTarget("Python", "Client.py") };
ProjectGenerator generator = new ProjectGenerator();

// Permet de charger les scripts depuis la mémoire
MemoryIncludeLoader loader = new MemoryIncludeLoader();

// #include myScript va être remplacer par le string s1.
loader.AddFile("myScript", s1);
loader.AddFile("myScript2", s2);
generator.Preprocessor.ScriptIncludeLoader = loader;

List<OutputFile> files = generator.Generate(script, serverTarget, clientTargets, out generationLog);
```

## 2.8. Commentaires

Il existe 2 syntaxes pour les commentaires : commentaire simple ligne ou multi-ligne. Les commentaires multi-ligne peuvent comporter des informations de documentation permettant de générer les commentaires appropriés dans les langages cibles, en vue de génération de doc automatique.

```
/**
 * @brief Ceci est la description de la fonction.
 * @param:param1 Description du param1
 * @returns Description de la valeur de retour.
 */
public int FonctionTest(int param1) { }

// Commentaire simple ligne
```

### 3. Blocs nommés

#### Généralités

Plusieurs ensembles de code peuvent être générés à l'aide de Clank.

Ces ensembles de code sont :

- macros : Représentent des types et fonctions considérées built-in des langages cibles. Par exemple, les classes telles que les listes, ou autres sont à renseigner dans les macros.
- state: Contient uniquement des classes et des variables. Les variables contenues dans un bloc state peuvent utiliser les classes déclarées dans ce même bloc ainsi que les classes définies dans les blocs "macros".
- access: Contient uniquement des fonctions permettant l'accès à des variables de State. Dans ce bloc, le mot clef state est réservé et est une instance d'une classe State contenant les variables définies dans les blocs "state". La variable *client\_id* est réservée et contient le numéro du client.
- write : Contient uniquement des fonctions permettant la modification de variables de State. Dans ce bloc, le mot clef state est réservé et se comporte comme dans le bloc access. La variable *client\_id* est réservée et contient le numéro du client.

Ces différents blocs seront détaillés dans leur propre section.

Les modifications de variables ou l'accès à des variables du mot clef state sont faits avec la syntaxe suivante :

state.machin	# Accès à la variable machin
state.machin = valeur;	# Modification de la variable machin

En réalité, ces opérations passent par le réseau ou se font directement en mémoire, mais le modèle en fait abstraction.

#### Block state

##### Généralités

Les variables / fonctions contenues dans le bloc state sont imbriqués dans une classe State disponible à la fois chez le client et le serveur.

Les classes contenues dans le bloc state seront toutes exportées chez le serveur. Seules les classes publiques sont cependant exportées chez le client.

A savoir : les fonctions contenues dans le bloc state n'ont pas accès aux variables contenues dans ce même block car les variables du block state ne seront pas exportées chez le client (seulement sur le serveur).

##### Sérialisation

La majorité des classes exportées chez le client doivent fournir un support pour la sérialisation, afin de pouvoir envoyer des instances de ces classes au serveur, et vice-versa. Pour qu'une classe soit serializable, il faut qu'elle soit [public](#) et [serializable](#). Une classe serializable subit des vérifications supplémentaires pour être « validée ». Entre autres, elle ne doit pas contenir de variable dont le type est un paramètre générique, ou un type non serializable.

## Bloc macro

Les classes contenues dans le bloc macro sont un peu particulières. Elles concernent des classes built-in dans les langages cibles, ayant des fonctionnalités similaires, mais un nom différent, et des fonctions ayant des noms différents entre langages, mais fournissant les mêmes fonctionnalités. Elles permettent de pouvoir utiliser de telles classes (pour pouvoir générer du code dans tous les langages cibles) sans en connaître le code.

Pour cela il faut :

1. Créer la classe normalement.
2. Créer une fonction string name() contenant une variable de type string du nom de chaque langage cible, et contenant le nom de la classe à utiliser dans le langage cible (pour les déclarations etc...).
3. Créer des fonctions "normalement" : c'est à dire, écrire leur signature correcte. En lieu et place du corps, pour chaque langage cible, créer une variable de type string du nom du langage cible contenant le code à exécuter dans le langage cible.

Pour que cela marche correctement, il faut remplacer les noms des arguments de la fonction par \$(argument), et si besoin, le nom de la variable sur laquelle la fonction est appelée par @self.

```
# Représente une "liste".
# Ce conteneur est à utiliser en lieu et place des array, et autres,
# car il est compatible avec la plupart des langages et facile à utiliser.
# Cependant, l'implémentation ne sera pas forcément optimale, mais ce n'est
# pas très important.
public serializable array class List<T>
{
    # Nom du type "List" dans les différents langages cibles.
    string name()
    {
        string cs = "List<$(T)>";
        string java = "ArrayList<$(T)>";
        string cpp = "std::vector<$(T)>";
        string python = "osef";
    }
    # Type des éléments de l'array.
    List<T> getArrayElementType() { }

    # Obtient un élément de la liste à l'index donné.
    T get(int index)
    {
        string cs = "@self[$(index)]";
        string java = "@self.get$(index)";
        string cpp = "@self.at$(index)";
        string python = "@self[$(index)]";
    }

    .....

    # Supprime l'élément de la liste à la position donnée.
    void removeat(int index)
```

```

    {
        string cs = "@self.RemoveAt($(index))";
        string java = "@self.remove($(index))";
        string cpp = "erase(@self.begin() + $(index))";
        string python = "del @self[$(index)]";
    }

}
}

```

Modificateurs :

- **Serializable** : des vérifications supplémentaires vont être faites sur la classe pour vérifier qu'elle remplit les conditions nécessaires pour être serialisable.
- Un des 2 modificateurs suivants :
  - o **array** la classe représente un conteneur du type « array ». (ex : std::vector de c++, ArrayList de java, List de C#). Voir l'utilité dans la section « Sérialisation ».
  - o **object** (par défaut) la classe représente un objet « classique », par opposition à array.

### Sérialisation

Une classe macro peut être **serializable** si elle répond à des contraintes précises :

- Elle est déclarée **public serializable array**.
- Elle comporte une méthode `getArrayElementType()` ayant pour type de retour le type de l'élément dans l'array.

Pour l'instant, la classe représentée doit fournir les méthodes suivantes pour être correctement sérialisée :

- C++ : `push_back(T elem)`
- Java : `add(T elem)`
- C# : `Add(T elem)`

Prévu : support de la sérialisation pour des classes non-array. Plus de flexibilité pour la sérialisation des classes non array.

## Blocks write/access

Les blocks write et access sont très similaires. Ils contiennent des fonctions qui permettent de lire ou d'écrire dans l'état. Ces fonctions sont traduites en code « natif » côté serveur, et en envoi/réception de données côté client. Le générateur de code génère aussi une fonction côté serveur qui se charge de recevoir les messages (typiquement JSON envoyés par TCP), de les décoder, et d'appeler la fonction adéquate avec les arguments encodés en JSON.

Les fonctions de ces blocks peuvent prendre en argument des types `public serializable` du block state/macro, ainsi que des types de base.

Les fonctions dans ces blocks contiennent deux mots-clefs :

`clientId`: id du client qui a demandé l'appel à la fonction.

`state`: contient l'instance de la classe State utilisée par le serveur.

## 4. Documentation développeurs

Doc pour les gens qui doivent utiliser le code.

Créer un générateur pour un nouveau langage

TODO

Modifier le langage / ajouter des fonctionnalités (avancé)

TODO

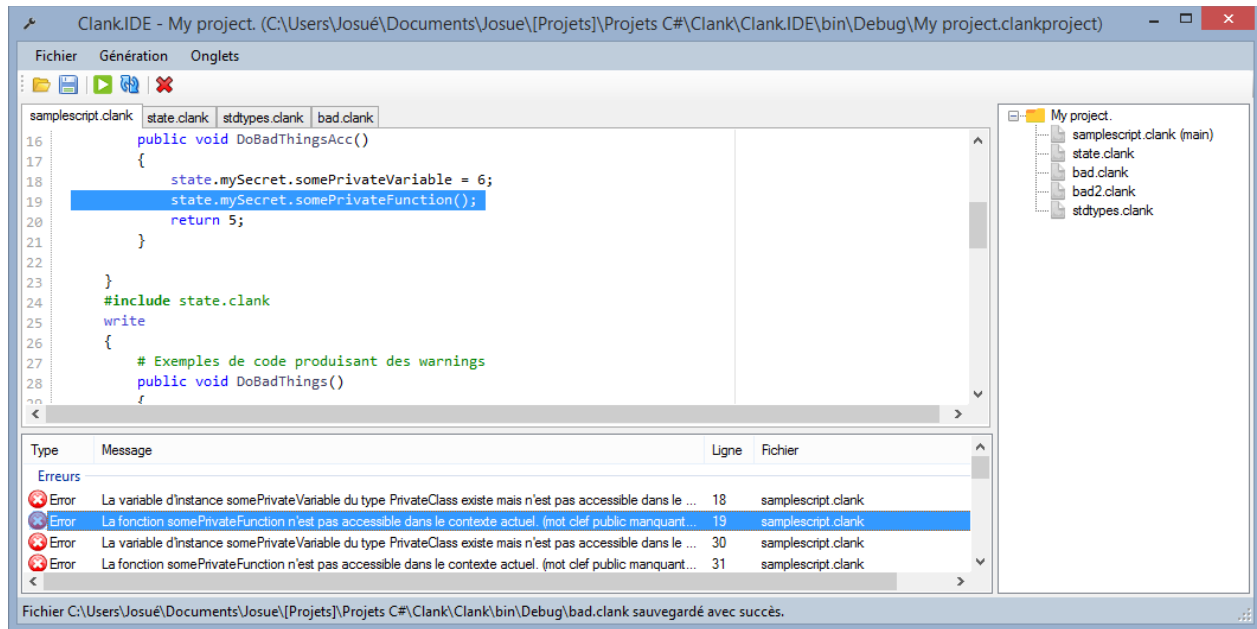
Compilation

TODO

## 5. Clank.IDE

### 1. Description

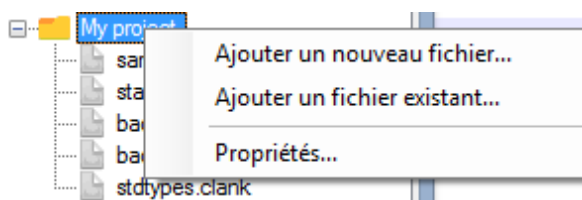
Clank.IDE est un petit outil permettant d'éditer du code Clank dans un environnement de développement qui permet de repérer les erreurs dans le code et les éliminer rapidement.



### 2. Démarrage rapide

Pour créer un nouveau projet : Fichier->Nouveau projet (CTRL+SHIFT+N). Choisissez l'endroit où créer le fichier du projet sur la boîte de dialogue qui apparaît.

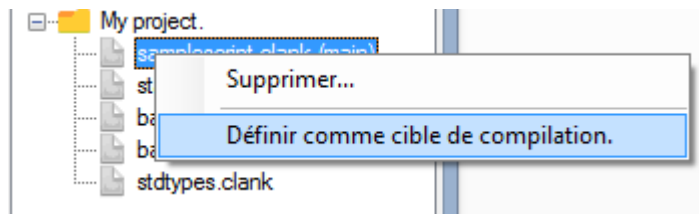
Sur le panneau latéral, faites un clic-droit sur le nœud principal du projet, et sélectionnez « Ajouter un nouveau fichier », ou « Ajouter un fichier existant ». Pour ouvrir le fichier ajouté au projet, il suffit de double cliquer dessus.



Une fois l'opération terminée, appuyez sur CTRL+SHIFT+S (ou Fichier->Enregistrer le projet) pour enregistrer le projet.

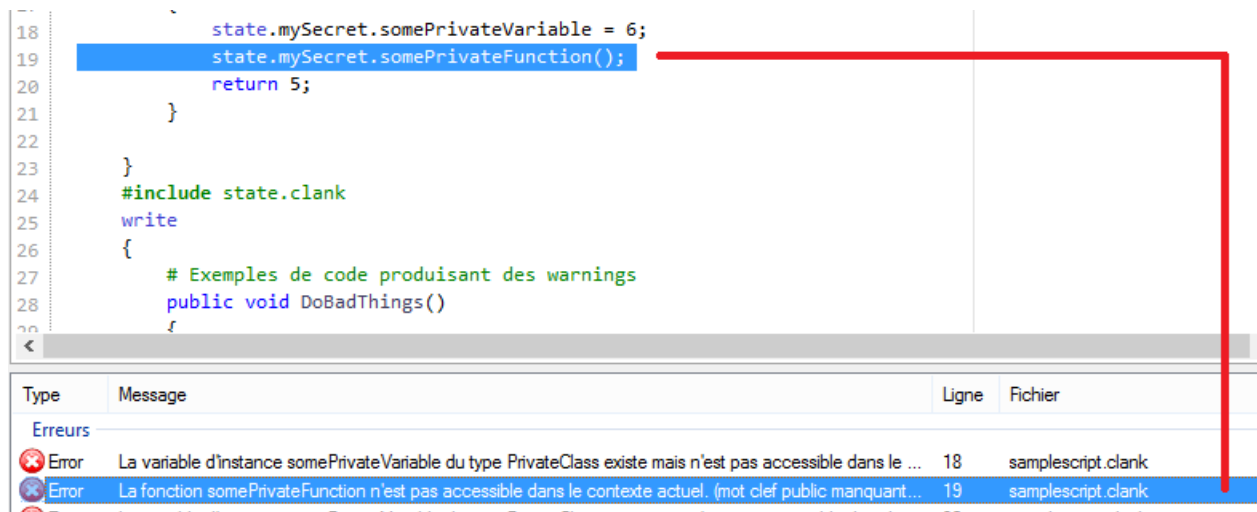
*Note : pour ouvrir un projet existant, CTRL+SHIFT+O (ou Fichier->Ouvrir un projet).*


Choisissez ensuite un des fichiers ajoutés dans le panneau latéral, et faites clic droit -> Définir comme cible de compilation.



Cela permet de dire au compilateur que c'est ce fichier qu'il faut compiler.

Lorsque vous faites des modifications sur le fichier à compiler, ou un des fichiers utilisés dans ce fichier (via `#include`), l'IDE va compiler le code et afficher les erreurs dans la console d'erreur située en bas de la fenêtre. Pour visualiser une erreur dans le code, double-cliquez sur cette erreur :



Pour lancer la compilation (et donc créer les fichiers de code), il suffit de cliquer sur le bouton , ou d'appuyer sur F7 (ou Génération->Générer).

## 6. Détails de l'implémentation actuelle du système communiquant.

L'implémentation de génération de code client et serveur actuelle est basée sur un système d'échange de données textuelles simple.

Les données sont échangées via des flux TCP, et encodées en **UTF-8 sans BOM**. Le protocole de communication est basé sur le fait que le récepteur des données connaît à l'avance le type des données reçues et leur signification.

Lorsque le client ou le serveur veut envoyer un objet par le réseau, il sérialise un à un ses membres. Tous les types marqués sérialisables dans Clank peuvent être sérialisés par ce moyen.

### 6.1. Représentation des types primitifs

- Int : représenté sous forme de texte, suivi d'un caractère de fin de ligne.
- Float : représenté sous forme textuelle, au format culturel 'en-us' (le séparateur décimal est un point), et suivi d'un caractère de fin de ligne.
- Bool : représenté comme un int ayant pour valeur 0 ou 1.
- String : représenté sous forme textuelle, suivi d'un caractère de fin de ligne. / !\ Les caractères de fin de lignes ne sont pas autorisés dans les strings !
- Types énumérés : leur valeur est encodée comme un int.