Lors de la conception d'un projet, votre programme commence de façon minimale, tout le code source étant regroupé dans un seul fichier. Puis le projet grandit avec une structure interne constitué de différents éléments ayant des fonctions dédiées. Nous aboutissons généralement à plusieurs fichiers sources, formant la structure globale du projet.

Lorsque vous commencerez à écrire des gros programmes, organiser votre code va devenir important car vous ne pourrez plus garder en tête l'intégralité de votre programme. En regroupant des fonctionnalités qui ont des points communs et en les séparant des autres, vous clarifiez l'endroit où trouver le code spécifique afin de pouvoir le relire ou le modifier.

***Rust** possède de nombreuses fonctionnalités qui vous permettent de gérer l'organisation de votre code, grâce à ce que la communauté **Rust** appelle le système de **modules**. Ce système définit quels sont les éléments qui sont accessibles depuis l'extérieur de la bibliothèque (notion de **privé** ou **public**), ainsi que leur **portée**. Ces fonctionnalités comprennent :*

- ***Les chemins :** des espaces de nom dédiés à vos structures, vos énumérations, vos fonctions, vos modules, etc.*
- ***Les modules :** utilisés avec le mot-clé **use**, il vous permettent de contrôler l'organisation, la portée et la visibilité de vos chemins.*
- ***Les caisses (crates) :** une arborescence de modules qui fournit une bibliothèque ou un exécutable.*
- ***Les paquets** : une fonctionnalité de **Cargo** qui vous permet de compiler, tester, et partager des caisses.*

Définitions fondamentales des caisses et des modules

Un projet **Rust** correspond à une **caisse**, sorte de paquetage. Une **caisse** réunit tout le code source d'une librairie de fonctions ou d'un programme exécutable, ainsi que tous les tests, exemples, outils, éléments de configuration, etc.

*Pour certains projets, vous pouvez avoir besoin de librairies tierces pour les affichages graphiques, les calculs parallèles, les traitements d'images, etc. Chacune de ces librairies est alors délivrée sous forme de **caisse**.*

En plus de regrouper des fonctionnalités, les **modules** vous permettent d'encapsuler les détails de l'implémentation d'une opération : vous pouvez écrire du code puis l'utiliser comme une abstraction à travers l'interface de programmation publique (**API**) du code sans se soucier de connaître les détails de son implémentation. La façon dont vous écrivez votre code définit quelles parties sont **publiques** et donc utilisables par un autre code, et quelles parties sont des détails d'implémentation **privés** dont vous vous réservez le droit de modifier.

*Un concept qui lui est associé est la **portée** : le contexte dans lequel le code est écrit avec un jeu de noms qui sont définis comme "dans la portée". Quand ils lisent, écrivent et compilent du code, les développeurs et les compilateurs ont besoin de savoir ce que tel nom désigne à tel endroit, et s'il s'agit d'une variable, d'une fonction, d'une structure, d'une énumération, d'un module, d'une constante, etc. Vous pouvez créer des **portées** et décider quels noms lui sont associées. Vous ne pouvez pas avoir deux entités avec le même nom dans la même portée ; cependant, des outils existent pour résoudre tous ces conflits de nom.*

Les paquets et les caisses

La première partie du système de **modules** que nous allons aborder concerne les **paquets** et les **caisses**. Une **caisse** est un produit fini, sous forme d'exécutable (application) ou d'une bibliothèque. Pour la compiler, **Rust** part d'un fichier source, la racine de la caisse, à partir duquel est alors créé le **module** racine de votre caisse (nous verrons les modules plus en détail dans la section suivante). Un **paquet** se compose d'une ou plusieurs **caisses** qui fournissent un ensemble de fonctionnalités. Un paquet contient un fichier « ***Cargo.toml*** » qui décrit comment construire ces différentes **caisses**.

*Il existe plusieurs règles qui déterminent ce qu'un **paquet** peut contenir. Il doit contenir une seule **caisse** de bibliothèque, ou aucune. Il peut contenir autant de **caisses** binaires (programmes) que vous le souhaitez, mais il doit contenir au moins une caisse (que ce soit une bibliothèque ou un binaire). Découvrons ce qui se passe quand nous créons un paquet. D'abord, nous utilisons la commande **cargo new** :*

Création d'un nouveau projet

```
$ cargo new mon-projet
    Created binary (application) `mon-projet` package
$ ls mon-projet
Cargo.toml
src
$ ls mon-projet/src
main.rs
```

*Lorsque nous saisissons la commande, **Cargo** crée un fichier **Cargo.toml,** qui définit un paquet. Si nous regardons le contenu de **Cargo.toml**, le fichier **src/main.rs** n'est pas mentionné car **Cargo** obéit à une convention selon laquelle **src/main.rs** est la racine de la **caisse** binaire portant le même nom que le paquet.*

*De la même façon, **Cargo** sait que si le dossier du paquet contient **src/lib.rs**, alors le paquet contient une **caisse** de bibliothèque qui porte le même nom que le **paquet**, et que **src/lib.rs** est sa racine. **Cargo** transmet les fichiers de la **caisse** racine à **rustc** pour compiler la bibliothèque ou le binaire.*

*Dans notre cas, nous avons un paquet qui contient uniquement **src/main.rs**, ce qui veut dire qu'il contient uniquement une **caisse** binaire qui s'appelle **mon-projet**. Si un paquet contient **src/main.rs** et **src/lib.rs**, nous avons alors deux **caisses** : une bibliothèque et une binaire, chacune avec le même nom que le **paquet**. Un **paquet** peut avoir plusieurs **caisses** binaires en ajoutant des fichiers dans le répertoire **src/bin** : chaque fichier sera une **caisse** séparée.*

*Une **caisse** regroupe plusieurs fonctionnalités associées dans une portée afin qu'elles soient faciles à partager entre plusieurs projets. Lorsque vous générez une **caisse**, il est possible par la suite de l'utiliser dans un nouveau projet en l'important, dans la même portée de notre projet. Toutes les fonctionnalités fournies par la **caisse** introduite sont accessibles via le nom de la **caisse** importée.*

Définir des modules pour gérer la portée et la visibilité

Un **module Rust** correspond à un espace de noms. Il s'agit d'un conteneur réunissant des fonctions, des types, des constantes et autres éléments du langage, constituant le programme principal ou une librairie. Alors qu'une **caisse** sert à partager du code entre plusieurs projets, un **module** sert à organiser le code dans les limites du même projet.

*Les **modules** nous permettent de regrouper le code d'une caisse pour une meilleure lisibilité et pour faciliter de réutilisation. Les **modules** permettent aussi de gérer la visibilité des éléments, qui précise si un élément peut être utilisé à l'extérieur du **module** (c'est **public**) ou s'il est un constituant **interne** et n'est pas disponible pour une utilisation externe (c'est **privé**).*

Programmation modulaire

```
mod math {
    pub mod fonctions {
        // fonctions publiques
        pub fn paire(x: u32) -> bool { !impaire(x) }
        pub fn factoriel(n: u64) -> u64 { ...  }
        pub fn diviseurs(mut nombre: u32) -> Vec<u32> { ... }
        pub fn hypothenuse(a: f32, b: f32) -> f32 { ... }

        // fonctions privées
        fn impaire(x: u32) -> bool { x%2!=0 }
        fn estPremier(nombre: u32) -> (bool, u32) { ... }
    }

    pub mod structures {
        // structures publiques
        pub struct Complexe {
            pub reel: f32,
            pub imaginaire: f32
        }

        impl Complexe {
            pub fn affiche(&self) { ... }
            pub fn module(&self) -> f32 { ... }
        }
    }
}

fn main() {
    use math::fonctions::{paire, factoriel, diviseurs};
    use math::structures::Complexe;

    i.affiche();
    p.affiche();
}
```

*Nous définissons un **module** en commençant avec le mot-clé **mod** et nous précisons ensuite le nom du **module** (dans notre cas, **math**) et nous ajoutons des accolades autour du corps du **module**. Dans les **modules**, nous pouvons avoir d'autres **modules**, comme dans notre cas avec les modules **fonctions** et **structures**. Les **modules** peuvent aussi contenir des définitions pour d'autres éléments, comme des structures, des énumérations, des constantes, des traits, ou des fonctions.*

*Grâce aux **modules**, nous pouvons regrouper ensemble des définitions qui sont liées et donner un nom à ce lien. Les développeurs qui utiliseront ce code pourront plus facilement trouver les définitions dont ils ont besoin car ils peuvent parcourir le code en fonction des groupes plutôt que d'avoir à lire toutes les définitions. Les développeurs qui veulent rajouter des nouvelles fonctionnalités à ce code sauront maintenant où placer le code tout en gardant le programme organisé.*

Pour l'instant l'ensemble de ces modules est placé dans le fichier principal. Le contenu global de ce fichier constitue lui-même un module qui s'appelle crate qui se trouve donc à la racine de l'arborescence de nos modules. Ce qui veut dire que l'ensemble de ce système constitue une caisse dont voici l'architecture :

```
crate
└── math
    ├── fonctions
    │   ├── paire
    │   ├── factoriel
    │   ├── diviseurs
    │   ├── impaire
    │   └── estPremier
    └── structures
        ├── énumération Complexe
        └── méthodes de Complexe
```

*Cette arborescence montre comment les **modules** sont imbriqués entre eux (par exemple, **fonctions** est imbriqué dans **math**). L'arborescence montre aussi que certains **modules** sont les frères d'autres **modules**, ce qui veut dire qu'ils sont définis dans le même **module** (**fonctions** et **structures** sont définis dans **math**).*

Désigner un élément dans l'arborescence de modules

Pour indiquer à **Rust** où trouver un élément dans l'arborescence de **modules**, nous utilisons un chemin à l'instar des chemins que nous utilisons lorsque nous naviguons dans un système de fichiers. Si nous désirons appeler une fonction, nous avons besoin de connaître son chemin dans la structure des **modules**.

*Les modules ne servent pas uniquement à organiser votre code. Ils définissent aussi les limites de **visibilité** de **Rust** : le code externe n'est pas autorisé à connaître, à appeler ou à se fier à des éléments internes au **module**. Donc, si vous voulez rendre un élément **privé** comme une **fonction** ou une **structure**, vous devez le placer dans un **module**.*

*La **visibilité** en **Rust** fait en sorte que tous les éléments (fonctions, méthodes, structures, énumérations, modules et constantes) sont **privés par défaut**. Les éléments dans un module parent ne peuvent pas utiliser les éléments privés dans les modules enfants, mais les éléments dans les modules enfants peuvent utiliser les éléments dans les modules parents. C'est parce que les modules enfants englobent et cachent les détails de leur implémentation, mais les modules enfants peuvent voir dans quel contexte ils sont définis.*

***Rust** a décidé de faire fonctionner le système de modules de façon à ce que les détails d'implémentation interne soit cachés par défaut. Ainsi, vous savez quelles parties du code interne vous pouvez changer sans casser le code externe. Vous pouvez exposer des parties internes des modules enfants en utilisant le mot-clé **pub** afin de les rendre publiques.*

Exposer des chemins avec le mot-clé pub

Ajouter le mot-clé **pub** devant **mod fonctions** et **mod structures** rend public les **modules** concernés. Avec cette spécification, nous pouvons accéder à fonctions et à structures. Mais sauf avis contraire leurs contenus reste **privé** ; rendre le **module public** ne rend pas son contenu **public**. Le mot-clé **pub** sur un module permet uniquement au code de ses parents d'y faire référence.

*Dans un **module**, vous pouvez ensuite sélectionner les **fonctions** et les autres éléments qui pourront être accessibles à l'extérieur, en utilisant également le mot-clé **pub**. Par contre, vous pouvez conserver certaines fonctions ou autres éléments comme **privés** (sans désignation particulière) parce qu'elles ne sont utiles qu'au sein du **module** pour des traitements en coulisse et pour résoudre l'implémentation des fonctions publiques. C'est le cas de la fonction **privée estPremier()** qui sert uniquement pour la fonction **publique diviseurs()**.*

Modules et fonctions publiques

```
mod math {
   pub mod fonctions {
      // fonctions publiques
      pub fn paire(x: u32) -> bool { !impaire(x) }
      pub fn factoriel(n: u64) -> u64 {
         match n {
            0 | 1 => 1,
            _ => n*factoriel(n-1)
         }
      }
      pub fn diviseurs(mut nombre: u32) -> Vec<u32> {
         let mut n = nombre;
         let mut liste = Vec::new();
         loop {
            match estPremier(n) {
               (true, _) => { if n!=nombre { liste.push(n) } return liste },
               (false, d) => { liste.push(d); n /= d }
            }
         }
      }
      pub fn hypothenuse(a: f32, b: f32) -> f32 {
            (a*a + b*b).sqrt()
       }

      // fonctions privées
      fn impaire(x: u32) -> bool { x%2!=0 }
      fn estPremier(nombre: u32) -> (bool, u32) {
         let limite = (nombre as f32).sqrt() as u32;
         for n in 2..=limite {
            if nombre%n==0 { return (false, n) }
         }
         (true, nombre)
      }
   }
   …

}
```

Rendre publiques des structures et des énumérations

Nous pouvons aussi utiliser **pub** pour déclarer des structures et des énumérations publiquement, mais il y a d'autres points à prendre en compte. Si nous utilisons **pub** avant la définition d'une structure, nous rendons la **structure publique**, mais les **champs de la structure restent privés**. Nous pouvons rendre chaque champ public ou non au cas par cas.

Structure publique avec ses champs publics

```
mod math {
...
   pub mod structures {
      // structures publiques
      pub struct Complexe {
         pub reel: f32,
         pub imaginaire: f32
      }

      impl Complexe {
         pub fn affiche(&self) {
            println!("(réel={}, imaginaire={})", self.reel, self.imaginaire)
         }
         pub fn module(&self) -> f32 {
            super::fonctions::hypothenuse(self.reel, self.imaginaire)
         }
      }
   }
}

fn main() {
   use math::structures::Complexe;

   let i = Complexe {reel: 0., imaginaire: 1.};
   let mut p = Complexe {reel: 1., imaginaire: 1.};
   i.affiche();
   p.imaginaire = 2.;
   p.affiche();
}
```

Résultat

```
(réel=0, imaginaire=1)
(réel=1, imaginaire=2)
2.236068
```

*Avec la structure **Complexe**, rendre les **attributs publics** peut paraître judicieux. Dans la cadre d'un développement objet et pour respecter le principe d'encapsulation, nous pourrions éviter que les attributs soient directement accessibles et les laisser **privés**, sans le mot-clé **pub** à chacun des champs. Dans cette démarche ce sont les méthodes qui font évoluer l'état de l'objet. Dans notre exemple, il faut alors prévoir un constructeur adapté.*

Structure publique avec champs privés

```
mod math {
...
   pub mod structures {
      // structures publiques
      pub struct Complexe {
         reel: f32,
         imaginaire: f32
      }

      impl Complexe {
         pub fn new(reel: f32, imaginaire: f32) -> Complexe {  Complexe { reel, imaginaire } }
         pub fn affiche(&self) {
            println!("(réel={}, imaginaire={})", self.reel, self.imaginaire)
         }
         pub fn module(&self) -> f32 {
            super::fonctions::hypothenuse(self.reel, self.imaginaire)
         }
      }
   }
}

fn main() {
   use math::structures::Complexe;

   // let i = Complexe {reel: 0., imaginaire: 1.};        // plus possible
   // let mut p = Complexe {reel: 1., imaginaire: 1.};    // plus possible
   let i = Complexe::new(0., 1.);
   let mut p = Complexe::new(1., 1.);

   i.affiche();
   // p.imaginaire = 2.;   // plus possible
   p.affiche();
   println!("{}", p.module());
}
```

Résultat

```
(réel=0, imaginaire=1)
(réel=1, imaginaire=1)
1.4142135
```

Chemin d'accès et imports

Pour accéder aux éléments d'un **module**, vous utilisez l'opérateur « **::** ». Depuis n'importe quel endroit du code source, vous pouvez solliciter une fonction d'une librairie standard en utilisant le chemin d'accès absolu :

Chemin d'accès absolu

```
if s1 > s2 {
   ::std::mem::swap(&mut s1, & mut s2);
}
```

*Le nom de la fonction est précédé de son chemin d'accès absolu, **::std::mem::swap**, puisqu'il commence par le double signe deux-points. La partie **::std** correspond au module de plus haut niveau de la librairie standard. La mention **::std::mem** désigne un **sous-module** de cette librairie et **::std::mem::swap** désigne la fonction à accès publique de ce **sous-module**.*

*Vous pourriez utiliser des références absolues partout, et écrire ainsi **::std::f64::consts::PI** et **::std::collections::HashMap::new** à chaque fois que vous avez besoin de calculer un cercle ou utiliser un dictionnaire, mais cela va vite devenir fatigant à écrire et à relire. La solution consiste à déclarer l'import de certaines fonctions dans les modules dans lesquels vous les utilisez :*

Chemin d'accès absolu

```
use std::mem;

if s1 > s2 {
   mem::swap(&mut s1, & mut s2);
}
```

*La déclaration **use** a pour effet que le nom **mem** devient un alias local **::std::mem** pour tout le bloc ou **module** dans lequel se trouve cette déclaration. Dans les déclarations **use**, les chemins sont automatiquement des chemins absolus. Il n'est donc pas nécessaire d'ajouter le préfixe **::**.*

*Nous pouvons écrire une déclaration très précise, **std::mem::swap** afin de connaître directement la fonction **swap**, mais certains considère que cette façon d'écrire n'est généralement pas conseillé.*

Chemin d'accès absolu

```
use std::mem::swap;

if s1 > s2 {
   swap(&mut s1, & mut s2);
}
```

Si vous décidez de décrire les fonctions à utiliser dans votre importation, plusieurs noms peuvent être cités dans la même déclaration d'importation :

Importation des fonctions définies dans les modules

```
use math::fonctions::{paire, factoriel, diviseurs};   // importe les trois fonctions d'un coup
use math::fonctions::*;                               // importe toutes les fonctions

use math::fonctions::paire;                           // écriture
use math::fonctions::factoriel;                       // équivalente
use math::fonctions::diviseurs;                       // à la première ligne
```

*Sachez qu'un **module** n'hérite pas automatiquement des noms déclarés pour les modules parents. Chaque **module** commence avec un espace de nom vide et doit donc importer les noms dont il a besoin.*

*Le mot-clé **super** prend un sens particulier dans les imports : il devient un **alias** du module parent. De même, **self** est un **alias** pour le **module** en cours lui-même. Même si les chemins dans les imports sont absolus par défaut, vous pouvez faire un import d'un chemin relatif au moyen de **self** et de **super**.*

Génération des modules et utilisation avec les importations respectives

```
mod math {
   pub mod fonctions {
      // fonctions publiques
      pub fn paire(x: u32) -> bool { !impaire(x) }
      pub fn factoriel(n: u64) -> u64 {
         match n {
            0 | 1 => 1,
            _ => n*factoriel(n-1)
         }
      }
```

```
        pub fn diviseurs(mut nombre: u32) -> Vec<u32> {
            let mut n = nombre;
            let mut liste = Vec::new();
            loop {
                match estPremier(n) {
                    (true, _) => { if n!=nombre { liste.push(n) } return liste },
                    (false, d) => { liste.push(d); n /= d }
                }
            }
        }
        pub fn hypothenuse(a: f32, b: f32) -> f32 { (a*a + b*b).sqrt()  }

        // fonctions privées
        fn impaire(x: u32) -> bool { x%2!=0 }
        fn estPremier(nombre: u32) -> (bool, u32) {
            let limite = (nombre as f32).sqrt() as u32;
            for n in 2..=limite {
                if nombre%n==0 { return (false, n) }
            }
            (true, nombre)
        }
    }

    pub mod structures {
        // structures publiques
        pub struct Complexe {
            reel: f32,
            imaginaire: f32
        }

        impl Complexe {
            pub fn new(reel: f32, imaginaire: f32) -> Complexe {  Complexe { reel, imaginaire }   }
            pub fn affiche(&self) { println!("(réel={}, imaginaire={})", self.reel, self.imaginaire)  }
            pub fn module(&self) -> f32 { super::fonctions::hypothenuse(self.reel, self.imaginaire) }
        }
    }
}

fn main() {
    use math::fonctions::{paire, factoriel, diviseurs};
    use math::structures::Complexe;

    let i = Complexe::new(0., 1.);
    let p = Complexe::new(1., 1.);

    i.affiche();
    p.affiche();
    println!("Module de p : {}", p.module());
    println!("Parité de 5 : {}", paire(5));
    println!("5!={}", factoriel(5));
    println!("Diviseurs de 17 : {:?}", diviseurs(17));
    println!("Diviseurs de 24 : {:?}", diviseurs(24));
}
```

Résultat

```
(réel=0, imaginaire=1)
(réel=1, imaginaire=1)
Module de p : 1.4142135
Parité de 5 : false
5!=120
Diviseurs de 17 : []
Diviseurs de 24 : [2, 2, 2, 3]
```

*Un **sous-module** peut accéder aux éléments **privés** de ses parents, mais il faut pour cela qu'il les importe explicitement. La mention **use super ::* ;** ne permet d'accéder qu'aux éléments qui sont marqués **pub** (éléments **publics**).*

Réexporter des éléments avec pub use

Lorsque nous importons un élément dans la portée avec le mot-clé **use**, son nom dans la nouvelle portée est **privé**. Pour permettre au code appelant d'utiliser ce nom comme s'il était défini dans cette portée, nous pouvons associer **pub** et **use**. Cette technique est appelée **réexporter** car nous importons un élément dans la portée, mais nous rendons aussi cet élément disponible aux portées des autres.

Réexportation avec pub use

```
mod math {
    pub mod fonctions {
```

```
        // fonctions publiques
        pub fn paire(x: u32) -> bool { !impaire(x) }
        pub fn factoriel(n: u64) -> u64 { … }
        pub fn diviseurs(mut nombre: u32) -> Vec<u32> { …  }
        pub fn hypothenuse(a: f32, b: f32) -> f32 { … }

        // fonctions privées
        fn impaire(x: u32) -> bool { x%2!=0 }
        fn estPremier(nombre: u32) -> (bool, u32) { … }
    }

    pub mod structures {
        // structures publiques
        pub struct Complexe {
            reel: f32,
            imaginaire: f32
        }

        impl Complexe {
            pub fn new(reel: f32, imaginaire: f32) -> Complexe { … }
            pub fn affiche(&self) { … }
            pub fn module(&self) -> f32 { … }
        }
    }
}

pub use math::structures::Complexe;
pub use math::fonctions::*;

fn main() {
//    use math::fonctions::{paire, factoriel, diviseurs};
//    use math::structures::Complexe;

    let i = Complexe::new(0., 1.);
    let p = Complexe::new(1., 1.);

    i.affiche();
    p.affiche();
    println!("Module de p : {}", p.module());
    println!("Parité de 5 : {}", paire(5));
    println!("5!={}", factoriel(5));
    println!("Diviseurs de 17 : {:?}", diviseurs(17));
    println!("Diviseurs de 24 : {:?}", diviseurs(24));
}
```

Réexporter *est utile quand la structure interne de votre code est différente de la façon dont les développeurs qui utilisent votre code se la représentent.*

Fichiers distincts pour les modules

Jusqu'à présent, tous les exemples dans cette étude ont défini plusieurs **modules** dans un seul fichier. Quand les **modules** vont grossir, vous allez probablement vouloir déplacer leurs définitions dans des fichiers sources séparés pour faciliter le parcours de votre code. Un fichier séparé devient lui-même un nouveau **module** qui porte alors le nom du fichier.

Dans les versions précédentes, nous avions ajouté le corps du ***module math*** *entre paires d'accolades. Dans cette nouvelle approche, nous devons le déclarer dans le fichier principal « **main.rs** » qu'il existe un fichier séparé « **math.rs** » grâce à la syntaxe* ***mod math ;***

Avec cette déclaration, nous informons le compilateur ***Rust*** *que le contenu du module* ***math*** *se trouve dans un fichier distinct portant le nom de « **math.rs** ». Ce fichier réunit alors le contenu du* ***module****. Aucune instruction de déclaration du* ***module*** *n'est maintenant plus nécessaire dans ce fichier. C'est un fichier source normal.*

La différence entre ce ***module*** *séparé et la première version intégrée est le déport du code. Les règles concernant ce qui est accessible de l'extérieur et ce qui ne l'est pas sont les mêmes.*

math.rs

```
pub mod fonctions {
  // fonctions publiques
  pub fn paire(x: u32) -> bool { !impaire(x) }
  pub fn factoriel(n: u64) -> u64 {
    match n {
      0 | 1 => 1,
      _ => n*factoriel(n-1)
    }
  }
  pub fn diviseurs(mut nombre: u32) -> Vec<u32> {
    let mut n = nombre;
    let mut liste = Vec::new();
```

```
    loop {
      match estPremier(n) {
        (true, _) => { if n!=nombre { liste.push(n) } return liste },
        (false, d) => { liste.push(d); n /= d }
      }
    }
  }
  pub fn hypothenuse(a: f32, b: f32) -> f32 { (a*a + b*b).sqrt()  }

  // fonctions privées
  fn impaire(x: u32) -> bool { x%2!=0 }
  fn estPremier(nombre: u32) -> (bool, u32) {
    let limite = (nombre as f32).sqrt() as u32;
    for n in 2..=limite {
      if nombre%n==0 { return (false, n) }
    }
    (true, nombre)
  }
}

pub mod structures {
  // structures publiques
  pub struct Complexe {
    reel: f32,
    imaginaire: f32
  }

  impl Complexe {
    pub fn new(reel: f32, imaginaire: f32) -> Complexe {  Complexe { reel, imaginaire } }
    pub fn affiche(&self) {  println!("(réel={}, imaginaire={})", self.reel, self.imaginaire)  }
    pub fn module(&self) -> f32 {   super::fonctions::hypothenuse(self.reel, self.imaginaire) }
  }
}
```

main.rs

```
mod math;

use math::structures::Complexe;
use math::fonctions::{paire, factoriel, diviseurs};

fn main() {

    let i = Complexe::new(0., 1.);
    let p = Complexe::new(1., 1.);

    i.affiche();
    p.affiche();
    println!("Module de p : {}", p.module());
    println!("Parité de 5 : {}", paire(5));
    println!("5!={}", factoriel(5));
    println!("Diviseurs de 17 : {:?}", diviseurs(17));
    println!("Diviseurs de 24 : {:?}", diviseurs(24));
}
```

*Nous pouvons séparer les **sous-modules** dans des **fichiers séparés** tout en conservant l'architecture initiale. Pour cela, nous devons placer ces fichiers séparés dans un répertoire qui porte le nom du **module** principal, ici **math**. Du coup, ces fichiers intégrés dans ce répertoire spécifique sont bien considérés comme des **sous-modules.***

*Malgré tout, pour que cela fonctionne correctement, nous devons rajouter le fichier « **mod.rs** » dans ce répertoire, dont le nom doit être exactement celui-ci puisqu'il spécifie bien que le répertoire est un **module**. Dans ce fichier, vous précisez juste la publication des **sous-modules**.*

*À chaque répertoire que vous créez, vous devez rajouter impérativement à l'intérieur ce fichier supplémentaire « **mod.rs** » qui précise bien que le répertoire est un **module** et il spécifiera systématiquement la liste des **sous-modules**.*

```
├── main.rs
└── math
    ├── mod.rs
    ├── fonctions.rs
    └── structures.rs
```

structures.rs

```
pub struct Complexe {
  reel: f32,
  imaginaire: f32
}
```

```rust
impl Complexe {
  pub fn new(reel: f32, imaginaire: f32) -> Complexe { Complexe { reel, imaginaire }  }

  pub fn affiche(&self) {
    println!("(réel={}, imaginaire={})", self.reel, self.imaginaire)
  }

  pub fn module(&self) -> f32 {
    super::fonctions::hypothenuse(self.reel, self.imaginaire)
  }
}
```

fonctions.rs

```rust
// fonctions publiques
pub fn paire(x: u32) -> bool {
  !impaire(x)
}

pub fn factoriel(n: u64) -> u64 {
  match n {
    0 | 1 => 1,
    _ => n*factoriel(n-1)
  }
}

pub fn diviseurs(nombre: u32) -> Vec<u32> {
  let mut n = nombre;
  let mut liste = Vec::new();
  loop {
    match est_premier(n) {
      (true, _) if n!=nombre  => { liste.push(n); return liste },
      (true, _)               => return liste,
      (false, d)              => { liste.push(d); n /= d }
    }
  }
}

pub fn hypothenuse(a: f32, b: f32) -> f32 {  (a*a + b*b).sqrt() }

// fonctions privées
fn impaire(x: u32) -> bool {
  x%2!=0
}

fn est_premier(nombre: u32) -> (bool, u32) {
  for n in 2..=(nombre as f32).sqrt() as u32 {
    if nombre%n==0 { return (false, n) }
  }
  (true, nombre)
}
```

mod.rs

```rust
pub mod fonctions;
pub mod structures;

pub use structures::Complexe;
pub use fonctions::*;
```

main.rs

```rust
mod math;
use  math::*;

fn main() {
    let i = Complexe::new(0., 1.);
    let p = Complexe::new(1., 1.);

    i.affiche();
    p.affiche();
    println!("Module de p : {}", p.module());
    println!("Parité de 5 : {}", paire(5));
    println!("5!={}", factoriel(5));
    println!("Diviseurs de 17 : {:?}", diviseurs(17));
    println!("Diviseurs de 24 : {:?}", diviseurs(24));
}
```