Récapitulatif Préparation Examen - Développement .NET / C#

Contents

Ce guide est structuré selon le format de l'examen prévu, couvrant la POO, les concepts fondamentaux, ASP.NET Core avec Entity Framework Core, et LINQ.

1 Programmation Orientée Objet (C#) (3 à 4 exercices)

1.1 Concepts Clés

- **Classe vs Objet** : Une classe est un plan (blueprint), un objet est une instance de ce plan.
- **Encapsulation** : Protection des données via les modificateurs d'accès.
  - `public` : Accessible partout.
  - `private` : Accessible uniquement dans la classe.
  - `protected` : Accessible dans la classe et les classes dérivées.
  - `internal` : Accessible dans le même assemblage (projet).
- **Héritage** : Permet à une classe d'hériter des membres d'une autre classe (`: BaseClass`).
- **Polymorphisme** : Capacité d'un objet à prendre plusieurs formes.
  - `virtual` : Méthode pouvant être redéfinie.
  - `override` : Redéfinition d'une méthode virtuelle.
  - `abstract` : Méthode sans implémentation (dans une classe abstraite), doit être implémentée par les enfants.
- **Interface** : Contrat définissant des méthodes sans implémentation. Une classe peut implémenter plusieurs interfaces.

1.2 Exemple Type Examen

```
// D finition d'une interface
public interface IPayment
{
    void ProcessPayment(double amount);
}

// Classe de base
public abstract class Employee
{
    public string Name { get; set; }
    public int Id { get; set; }

    public Employee(int id, string name)
    {
        Id = id;
        Name = name;
    }

    public abstract double CalculateSalary();
```

```
}

// Classe d riv e impl mentant l'h ritage et l'interface
public class Developer : Employee, IPayment
{
    public string Level { get; set; } // Junior, Senior

    public Developer(int id, string name, string level) : base(id, name)
    {
        Level = level;
    }

    // Polymorphisme : Impl mentation de la m thode abstraite
    public override double CalculateSalary()
    {
        return Level == "Senior" ? 5000 : 3000;
    }

    // Impl mentation de l'interface
    public void ProcessPayment(double amount)
    {
        Console.WriteLine($"Paiement de {amount} trait  pour {Name}.");
    }
}
```

2 Questions de Compréhension

Cette section regroupe les questions théoriques potentielles avec leurs réponses détaillées.

2.1 Architecture .NET

1. **Qu'est-ce que le CLR (Common Language Runtime) ?**
   Réponse : C'est le moteur d'exécution de .NET. Il gère :
   - La gestion de la mémoire (Allocation et Garbage Collection).
   - La sécurité des types (Type Safety).
   - La gestion des exceptions.
   - La compilation JIT (Just-In-Time) du code IL vers le code machine.

2. **Qu'est-ce que MSIL (Microsoft Intermediate Language) ou CIL ?**
   Réponse : C'est le langage intermédiaire généré après la compilation du code C# (ou VB.NET). Il est indépendant de la machine et sera compilé en code natif par le JIT lors de l'exécution.

3. **Expliquez le rôle du JIT (Just-In-Time Compiler).**
   Réponse : Le JIT compile le code IL en code machine natif spécifique au processeur de l'ordinateur, juste au moment où le code est appelé, optimisant ainsi les performances.

4. **Qu'est-ce que le Garbage Collector (GC) ?**
   Réponse : C'est un processus automatique du CLR qui libère la mémoire occupée

par les objets qui ne sont plus référencés par l'application, évitant ainsi les fuites de mémoire.

5. **Quelle est la différence entre Code Managé et Code Non Managé ?**
   Réponse :

   - **Managé** : Exécuté par le CLR (ex: C#, Java). Bénéficie du GC, sécurité, etc.
   - **Non Managé** : Exécuté directement par l'OS (ex: C, C++). Le développeur gère la mémoire manuellement.

2.2 Langage C#

1. **Différence entre `Value Type` et `Reference Type` ?**
   Réponse :

   - **Value Type** (int, float, struct) : Stocké dans la **Stack** (Pile). Contient directement la donnée.
   - **Reference Type** (class, string, object) : Stocké dans le **Heap** (Tas). La variable contient une référence (adresse) vers la donnée.

2. **Qu'est-ce que le Boxing et Unboxing ?**
   Réponse :

   - **Boxing** : Conversion implicite d'un type valeur vers un type référence (`object`). (Coûteux en performance).
   - **Unboxing** : Conversion explicite d'un type référence vers un type valeur.

3. **Différence entre `interface` et `abstract class` ?**
   Réponse :

   - **Interface** : Ne contient que des signatures (avant C# 8). Pas de champs. Une classe peut implémenter plusieurs interfaces.
   - **Classe Abstraite** : Peut contenir du code implémenté et des champs. Une classe ne peut hériter que d'une seule classe abstraite. Utilisez l'abstrait pour une relation "est un" (is-a) et l'interface pour une capacité "paut faire" (can-do).

4. **Différence entre `ref` et `out` ?**
   Réponse :

   - **ref** : La variable doit être initialisée *avant* d'être passée.
   - **out** : La variable n'a pas besoin d'être initialisée avant, mais *doit* être assignée dans la méthode appelée.

5. **À quoi servent `async` et `await` ?**
   Réponse : Ils permettent la programmation asynchrone pour ne pas bloquer le thread principal (ex: UI ou thread requête web) lors d'opérations I/O (Datebase, Fichier, Réseau).

2.3 ASP.NET Core

1. **Qu'est-ce que le Middleware ?**
   Réponse : C'est un composant logiciel assemblé dans le pipeline de l'application pour gérer les requêtes et les réponses HTTP. Exemples : Authentification, Logging, Gestion des erreurs, Fichiers statiques.

2. **Qu'est-ce que l'Injection de Dépendance (DI) et ses cycles de vie ?**
   Réponse : C'est une technique pour réaliser l'Inversion de Contrôle (IoC) entre les classes et leurs dépendances.

   - **Transient** : Nouvelle instance à chaque demande. (Léger, sans état).
   - **Scoped** : Une instance par requête HTTP. (Idéal pour DbContext).
   - **Singleton** : Une seule instance pour toute la durée de vie de l'application. (Pour cache, config).

3. **Rôle du fichier `Program.cs` (ou `Startup.cs` anciennement) ?**
   Réponse : C'est le point d'entrée de l'application. Il configure le serveur web (Kestrel), les services (DI), et le pipeline de requêtes (Middlewares).

4. **Qu'est-ce que MVC ?**
   Réponse :

   - **Model** : Représente les données et la logique métier.
   - **View** : Affiche l'interface utilisateur (HTML).
   - **Controller** : Reçoit la requête, manipule le modèle, et sélectionne la vue.

2.4 Entity Framework Core

1. **Qu'est-ce que le Lazy Loading vs Eager Loading ?**
   Réponse :

   - **Lazy Loading** : Les données liées (ex: Produits d'une Catégorie) sont chargées automatiquement *seulement quand on y accède.* (Peut causer le problème N+1 requêtes).
   - **Eager Loading** : Les données liées sont chargées *en même temps* que la requête principale grâce à `.Include()`. (Plus performant généralement).

2. **Qu'est-ce qu'une Migration ?**
   Réponse : C'est un fichier de code généré qui décrit comment mettre à jour le schéma de la base de données pour qu'il corresponde aux classes du modèle C# (Code-First).

3. **Différence entre `DbContext` et `DbSet` ?**
   Réponse :

   - **DbContext** : Représente la session avec la base de données (unité de travail).
   - **DbSet<T>** : Représente une table spécifique (collection d'entités) dans la base.

3 ASP.NET Core et Entity Framework Core (Exercice Pratique)

3.1 Configuration (DbContext)

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(
    options) { }

    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }

    // Fluent API (pour configurations avanc es)
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Relation One-to-Many
        modelBuilder.Entity<Product>()
            .HasOne(p => p.Category)
            .WithMany(c => c.Products)
            .HasForeignKey(p => p.CategoryId);
    }
}
```

3.2 Migrations (Commandes)

- `Add-Migration NomDeLaMigration` : Crée un fichier de migration basé sur les changements du modèle.
- `Update-Database` : Applique les migrations à la base de données réelle.

3.3 Opérations CRUD (Controller)

```
[HttpPost]
public async Task<IActionResult> Create(Product product)
{
    if (ModelState.IsValid)
    {
        _context.Add(product);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(product);
}

// Eager Loading (Include) pour r cup rer les donn es li es
public async Task<IActionResult> Index()
{
    var products = await _context.Products.Include(p => p.Category).
    ToListAsync();
    return View(products);
}
```

4 LINQ / LINQ to Entities

4.1 Syntaxe de Requête vs Méthode

- **Requête** : Ressemble à SQL (`from ... where ... select`).
- **Méthode** : Chaînage de méthodes (`.Where().Select()`).

4.2 Opérateurs Fréquents

1. **Filtrage (Where)** :

```
    var cheapProducts = _context.Products.Where(p => p.Price < 100);

```

2. **Projection (Select)** :

```
    var productNames = _context.Products.Select(p => p.Name);

```

3. **Tri (OrderBy / OrderByDescending)** :

```
    var sorted = _context.Products.OrderBy(p => p.Price);

```

4. **Agrégation (Count, Sum, Average, Max)** :

```
    var total = _context.Products.Count();
    var averagePrice = _context.Products.Average(p => p.Price);

```

5. **Regroupement (GroupBy)** :

```
    var productsByCategory = _context.Products
        .GroupBy(p => p.CategoryId)
        .Select(g => new { Key = g.Key, Count = g.Count() });

```

6. **Jointure (Join)** : Souvent remplacé par `Include` avec EF Core, mais utile pour des jointures explicites.

4.3 Différée vs Immédiate

- **Exécution Différée** : La requête n'est pas exécutée tant qu'on ne l'itère pas (`IQueryable`, `IEnumerable`).
- **Exécution Immédiate** : La requête est exécutée tout de suite (`.ToList()`, `.Count()`, `.FirstOrDefault()`).

4.4 Exemple Combiné

Trouver les noms des produits de la catégorie "Electronique" triés par prix décroissant.

```
var result = _context.Products
    .Where(p => p.Category.Name == "Electronique")
    .OrderByDescending(p => p.Price)
    .Select(p => p.Name)
    .ToList();
```

Darkwing