C# Concepts and Constraints

Overview

This document covers various C# programming concepts, including generics, value vs. reference-based equality, constraints, and hash code evaluation.

Constructor Behavior

- Starting from C# 6, constructors implicitly assign default values to variables.

Generics

- Generics can be defined at the class or method level.
- Multiple generic types can be defined, e.g., `<T1, T2, T3>`.

Syntax Sugar in C# 12

- `[10, 20, 30, 40, 50]` is shorthand for `new int[] { 10, 20, 30, 40, 50 }`.

Equality in Structs and Classes

- Structs do not have a default implementation for `==` and `!=`, whereas classes do.
- **Reference-based equality:** Two class instances with the same values are not considered equal unless explicitly overridden.
- **Value-based equality:** Can be implemented for structs and classes by overriding `.Equals()`.

Example: Value-Based Equality in Structs

```
internal struct Point {
    int X;
    int Y;
    // Other members
}

Point p1 = new Point(10, 20);
Point p2 = new Point(10, 30);
Console.WriteLine(p1.Equals(p2)); // false
```

Example: Reference-Based Equality in Classes

```
internal class Employee {
    int Id;
    string Name;
    double Salary;
}
```

```
Employee e1 = new Employee(1, "Ahmed", 1000);
Employee e2 = new Employee(1, "Ahmed", 1000);
Console.WriteLine(e1.Equals(e2)); // false (reference-based equality)
```

- In classes, `==` and `!=` are reference-based by default.
- To enforce value-based equality, override `.Equals()`.

Generic Constraints

- **Primary Constraints:**
  - `class` → T must be a class.
  - `struct` → T must be a struct.
  - `notnull` → T must be non-nullable (C# 8.0).
  - `default` → Default constraint.
  - `unmanaged` → Unmanaged constraint.
  - `Enum` → T must be an enum (C# 7.3) i can consider it special primary constraint as Enum is a class which all enums inherit from it
- **Special Primary Constarint (User-Defined Class (Except Sealed))** → T must be a specific class or its derived types (excluding sealed classes).
- **Secondary Constraints:**
  - Interfaces such as `IComparable<T>`.
- **Constructor Constraint:**
  - `new()` → T must have an accessible parameterless constructor.
  - Cannot use `new()` with a struct constraint.

Example of a Generic Class with Constraints

```
internal class Helper<T, T2, T3>
    where T : class, IComparable<T>, new()
    where T2 : class
    where T3 : Point
{
    // Implementation
}
```

Hash Code Evaluation

Modern Approach (C# 7.3+, .NET Core 2.1+)

```
return HashCode.Combine(this.Id, this.Name, this.Salary);
```

True Way (Performance Optimized)

```
int hashValue = 11;
hashValue = (hashValue * 7) + Id.GetHashCode();
hashValue = (hashValue * 7) + (Name?.GetHashCode() ?? default(int));
hashValue = (hashValue * 7) + Salary.GetHashCode();
return hashValue;
```

Faster But Incorrect Approaches

```
return this.Id.GetHashCode() ^ (this.Name?.GetHashCode() ?? default(int)) ^
this.Salary.GetHashCode();
```

```
return this.Id.GetHashCode() + (this.Name?.GetHashCode() ?? default(int)) +
this.Salary.GetHashCode();
```

This document provides an overview of C# concepts related to generics, equality, constraints, and hash codes, ensuring better understanding and implementation practices.