Variable Declaration

In C#, you declare a variable by specifying its type followed by the variable name. Here are some examples:

```
int number;           // Declares an integer variable named number
double price;         // Declares a double variable named price
string name;          // Declares a string variable named name
bool isActive;        // Declares a boolean variable named isActive
```

Assigning Values to Variables

You can assign values to variables at the time of declaration or afterwards:

```
int number = 10;      // Declares an integer variable and assigns the value 10
double price = 19.99; // Declares a double variable and assigns the value 19.99
string name = "John"; // Declares a string variable and assigns the value "John"
bool isActive = true; // Declares a boolean variable and assigns the value true

// Assigning values after declaration
number = 20;
name = "Jane";
```

Printing Variables

To print variables to the console, you can use `Console.WriteLine`:

```
Console.WriteLine(number);  // Prints the value of number
Console.WriteLine(price);   // Prints the value of price
Console.WriteLine(name);    // Prints the value of name
Console.WriteLine(isActive);// Prints the value of isActive
```

Getting Input from the User

To get input from the user, you can use `Console.ReadLine` and then convert the input to the desired type:

```
Console.WriteLine("Enter a number:");
number = int.Parse(Console.ReadLine());  // Reads input and converts to an integer

Console.WriteLine("Enter a price:");
price = double.Parse(Console.ReadLine()); // Reads input and converts to a double

Console.WriteLine("Enter your name:");
name = Console.ReadLine();                // Reads input as a string

Console.WriteLine("Are you active (true/false):");
```

```
isActive = bool.Parse(Console.ReadLine()); // Reads input and converts to a boolean
```

Finding Data Types

In C#, you can use the `GetType` method to find out the type of a variable:

```
Console.WriteLine(number.GetType());  // Prints System.Int32
Console.WriteLine(price.GetType());   // Prints System.Double
Console.WriteLine(name.GetType());    // Prints System.String
Console.WriteLine(isActive.GetType());// Prints System.Boolean
```

Finding Memory Addresses

In C#, finding the memory address of a variable directly is not straightforward due to the managed nature of the language and its runtime environment. However, you can use pointers in unsafe context to get the address of a variable:

```
unsafe
{
    int num = 5;
    int* p = &num;
    Console.WriteLine((IntPtr)p);  // Prints the memory address of num
}
```

To compile code with unsafe context, you need to enable "Allow unsafe code" in your project properties and use the `unsafe` keyword. Here’s how you can set it up:

1. Right-click on your project in Visual Studio.
2. Select Properties.
3. Go to the Build tab.
4. Check the "Allow unsafe code" checkbox.

Then you can use the `unsafe` keyword in your code:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int num = 5;
            int* p = &num;
            Console.WriteLine((IntPtr)p);  // Prints the memory address of num
        }
    }
}
```

Complete Example

Here's a complete example incorporating all the concepts mentioned above:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Variable declaration and initialization
        int number = 10;
        double price = 19.99;
        string name = "John";
        bool isActive = true;

        // Printing variables
        Console.WriteLine("Initial Values:");
        Console.WriteLine($"Number: {number}");
        Console.WriteLine($"Price: {price}");
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"Is Active: {isActive}");

        // Getting input from the user
        Console.WriteLine("\nEnter a number:");
        number = int.Parse(Console.ReadLine());

        Console.WriteLine("Enter a price:");
        price = double.Parse(Console.ReadLine());

        Console.WriteLine("Enter your name:");
        name = Console.ReadLine();

        Console.WriteLine("Are you active (true/false):");
        isActive = bool.Parse(Console.ReadLine());

        // Printing updated values
        Console.WriteLine("\nUpdated Values:");
        Console.WriteLine($"Number: {number}");
        Console.WriteLine($"Price: {price}");
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"Is Active: {isActive}");

        // Finding data types
        Console.WriteLine("\nData Types:");
        Console.WriteLine(number.GetType());
        Console.WriteLine(price.GetType());
        Console.WriteLine(name.GetType());
        Console.WriteLine(isActive.GetType());

        // Finding memory addresses (unsafe context)
```

```
        unsafe
        {
            int* p = &number;
            Console.WriteLine($"\nMemory Address of number: {(IntPtr)p}");
        }
    }
}
```

1. **Arithmetic Operators**
2. **Assignment Operators**
3. **Comparison Operators**
4. **Logical Operators**
5. **Bitwise Operators**
6. **Miscellaneous Operators**

Let's go through each category with examples.

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations:

- **Addition (+)**
- **Subtraction (-)**
- **Multiplication (*)**
- **Division (/)**
- **Modulus (%)**

```
int a = 10;
int b = 3;

int sum = a + b;        // 13
int difference = a - b; // 7
int product = a * b;    // 30
int quotient = a / b;   // 3 (integer division)
int remainder = a % b;  // 1 (remainder of 10 / 3)
```

2. Assignment Operators

Assignment operators are used to assign values to variables. They include:

- **Simple Assignment (=)**
- **Addition Assignment (+=)**
- **Subtraction Assignment (-=)**
- **Multiplication Assignment (*=)**
- **Division Assignment (/=)**
- **Modulus Assignment (%=)**

```
int a = 5;
a += 3;  // Equivalent to a = a + 3; now a is 8
a -= 2;  // Equivalent to a = a - 2; now a is 6
a *= 4;  // Equivalent to a = a * 4; now a is 24
a /= 3;  // Equivalent to a = a / 3; now a is 8
a %= 5;  // Equivalent to a = a % 5; now a is 3
```

3. Comparison Operators

Comparison operators are used to compare two values. They return a boolean value (`true` or `false`):

- **Equal to (==)**
- **Not equal to (!=)**
- **Greater than (>)**
- **Less than (<)**
- **Greater than or equal to (>=)**
- **Less than or equal to (<=)**

```
int a = 5;
int b = 10;

bool isEqual = (a == b);         // false
bool isNotEqual = (a != b);      // true
bool isGreaterThan = (a > b);    // false
bool isLessThan = (a < b);       // true
bool isGreaterOrEqual = (a >= b); // false
bool isLessOrEqual = (a <= b);    // true
```

4. Logical Operators

Logical operators are used to combine conditional statements:

- **Logical AND (&&)**
- **Logical OR (||)**
- **Logical NOT (!)**

```
bool a = true;
bool b = false;

bool andResult = a && b;  // false
bool orResult = a || b;   // true
bool notResult = !a;      // false
```

5. Bitwise Operators

Bitwise operators are used to perform bit-level operations:

- **AND (&)**
- **OR (|)**
- **XOR (^)**
- **NOT (~)**
- **Left Shift (<<)**
- **Right Shift (>>)**

```
int a = 5;  // 0101 in binary
int b = 3;  // 0011 in binary

int andResult = a & b;  // 0001 (1 in decimal)
int orResult = a | b;   // 0111 (7 in decimal)
int xorResult = a ^ b;  // 0110 (6 in decimal)
int notResult = ~a;     // 11111111111111111111111111111010 (inverting all bits of 5)
int leftShift = a << 1; // 1010 (10 in decimal, shifts bits to the left)
int rightShift = a >> 1;// 0010 (2 in decimal, shifts bits to the right)
```

6. Miscellaneous Operators

These include:

- **Conditional (Ternary) Operator (?:)**
- **Null-Coalescing Operator (??)**
- **Null-Coalescing Assignment Operator (??=)**

```
// Ternary Operator
int a = 5;
string result = (a > 3) ? "Greater than 3" : "Less than or equal to 3"; // "Greater than 3"

// Null-Coalescing Operator
string str = null;
string resultStr = str ?? "Default value"; // "Default value"

// Null-Coalescing Assignment Operator
string str2 = null;
str2 ??= "Default assigned";
Console.WriteLine(str2); // "Default assigned"
```

7. Type Testing and Conversion Operators

- `is`
- `as`

```
object obj = "Hello, World!";
bool isString = obj is string; // true

object obj2 = "Hello, World!";
string str3 = obj2 as string; // "Hello, World!" (or null if obj2 wasn't a string)
```

8. Increment and Decrement Operators

- **Increment (++)**
- **Decrement (--)**

```
int a = 5;
a++; // a is now 6
a--; // a is now 5 again

int b = 5;
int preIncrement = ++b; // preIncrement is 6, b is 6
int postIncrement = b++; // postIncrement is 6, b is now 7
```

1. `if` Statement:

The `if` statement is used for conditional execution. It executes a block of code if a specified condition is true.

Real-world Example: Imagine you're building a program to determine whether a person is eligible to vote based on their age.

```
int age = 18;

if (age >= 18)
{
    Console.WriteLine("You are eligible to vote.");
}
```

2. `if-else` Statement:

The `if-else` statement executes one block of code if the condition is true and another if it's false.

Real-world Example: Creating a program to check if a number is even or odd.

```
int number = 7;

if (number % 2 == 0)
{
    Console.WriteLine("The number is even.");
```

```
}
else
{
    Console.WriteLine("The number is odd.");
}
```

3. `else if` Statement:

The `else if` statement allows you to check multiple conditions.

`Real-world Example:` Determining a person's weight category based on their BMI.

```
double weight = 70;
double height = 1.75;
double bmi = weight / (height * height);

if (bmi < 18.5)
{
    Console.WriteLine("Underweight");
}
else if (bmi >= 18.5 && bmi < 24.9)
{
    Console.WriteLine("Normal weight");
}
else if (bmi >= 24.9 && bmi < 29.9)
{
    Console.WriteLine("Overweight");
}
else
{
    Console.WriteLine("Obese");
}
```

4. `while` Loop:

The `while` loop executes a block of code repeatedly as long as a specified condition is true.

`Real-world Example:` Counting down to a rocket launch.

```
int countdown = 10;

while (countdown > 0)
{
    Console.WriteLine($"T-{countdown} seconds");
    countdown--;
}

Console.WriteLine("Blast off!");
```

5. `for` Loop:

The `for` loop executes a block of code a specified number of times.

`Real-world Example:` Calculating compound interest.

```
double principal = 1000;
double rate = 0.05;
int years = 5;

for (int i = 1; i <= years; i++)
{
    principal += principal * rate;
}

Console.WriteLine($"After {years} years, the principal amount is {principal:C}");
```

6. `foreach` Loop:

The `foreach` loop iterates over a collection of items.

`Real-world Example:` Printing the elements of an array.

```
int[] numbers = { 1, 2, 3, 4, 5 };

foreach (int num in numbers)
{
    Console.WriteLine(num);
}
```

7. `do-while` Loop:

The `do-while` loop is similar to the `while` loop, but it executes the block of code at least once before checking the condition.

`Real-world Example:` Taking input from the user until valid input is received.

```
int userInput;

do
{
    Console.WriteLine("Enter a positive number:");
} while (!int.TryParse(Console.ReadLine(), out userInput) || userInput <= 0);

Console.WriteLine($"You entered: {userInput}");
```

`1. Function Declaration and Invocation:`

```
using System;

public class Program
{
    // Function declaration
    static void Greet(string name)
    {
        Console.WriteLine($"Hello, {name}!");
    }

    public static void Main(string[] args)
    {
        // Function invocation
        Greet("John");
    }
}
```

In this example, we declare a function Greet that takes a string parameter name and prints a greeting message. We then invoke this function in the Main method with the argument "John".

2. Return Values:

```
using System;

public class Program
{
    // Function with return value
    static int Add(int a, int b)
    {
        return a + b;
    }

    public static void Main(string[] args)
    {
        int sum = Add(5, 3);
        Console.WriteLine("Sum: " + sum);
    }
}
```

Here, Add function returns the sum of two integers. We store the result in a variable sum and print it.

3. Optional Parameters:

```
using System;
```

```
public class Program
{
    // Function with optional parameter
    static void Greet(string name, string greeting = "Hello")
    {
        Console.WriteLine($"{greeting}, {name}!");
    }

    public static void Main(string[] args)
    {
        Greet("John");
        Greet("Emily", "Hi");
    }
}
```

In this example, the Greet function has an optional parameter greeting with a default value of "Hello". When invoked without providing greeting, it defaults to "Hello". You can also provide a custom greeting as shown in the second invocation.

4. Named Arguments:

```
using System;

public class Program
{
    // Function with named arguments
    static void DisplayInfo(string name, int age)
    {
        Console.WriteLine($"Name: {name}, Age: {age}");
    }

    public static void Main(string[] args)
    {
        DisplayInfo(name: "John", age: 30);
        DisplayInfo(age: 25, name: "Emily");
    }
}
```

Named arguments allow you to specify the parameters by name, which can improve the clarity of the code, especially when dealing with functions with many parameters.

5. Recursion:

```
using System;

public class Program
{
```

```
    // Recursive function
    static int Factorial(int n)
    {
        if (n == 0)
            return 1;
        else
            return n * Factorial(n - 1);
    }

    public static void Main(string[] args)
    {
        int result = Factorial(5);
        Console.WriteLine("Factorial of 5: " + result);
    }
}
```

The Factorial function calculates the factorial of a given number using recursion.

6. Pass by Value vs. Pass by Reference:

```
using System;

public class Program
{
    // Function demonstrating pass by value
    static void IncrementValue(int num)
    {
        num++;
        Console.WriteLine("Inside IncrementValue function: " + num);
    }

    // Function demonstrating pass by reference
    static void IncrementRef(ref int num)
    {
        num++;
        Console.WriteLine("Inside IncrementRef function: " + num);
    }

    public static void Main(string[] args)
    {
        int value = 5;

        // Pass by value
        IncrementValue(value);
        Console.WriteLine("After IncrementValue function: " + value);

        // Pass by reference
        IncrementRef(ref value);
```

```
        Console.WriteLine("After IncrementRef function: " + value);
    }
}
```

This code demonstrates the difference between passing arguments by value and by reference.

7. Lambda Expressions:

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        // Lambda expression to square a number
        Func<int, int> square = x => x * x;
        Console.WriteLine("Square of 5: " + square(5));

        // Lambda expression with multiple parameters
        Func<int, int, int> add = (x, y) => x + y;
        Console.WriteLine("Sum of 3 and 4: " + add(3, 4));
    }
}
```

Lambda expressions are anonymous functions that can be used to create delegates or expression tree types.

8. Delegates:

```
using System;

public class Program
{
    delegate void Operation(int x, int y);

    static void Add(int a, int b)
    {
        Console.WriteLine("Sum: " + (a + b));
    }

    static void Multiply(int a, int b)
    {
        Console.WriteLine("Product: " + (a * b));
    }

    public static void Main(string[] args)
```

```
    {
        Operation op = Add;
        op += Multiply;

        op(3, 4); // Both Add and Multiply methods will be invoked
    }}
```

Let's imagine we're building a simple banking system. In this system, we need to represent bank accounts. A bank account has attributes like an account number, the name of the account holder, and the balance. We can use a class to define the structure of a bank account, and then create objects (instances) of that class for individual bank accounts.

Here's how you might define a `BankAccount` class in C#:

```
using System;

public class BankAccount
{
    // Attributes or fields
    private string accountNumber;
    private string accountHolderName;
    private decimal balance;

    // Constructor
    public BankAccount(string accountNumber, string accountHolderName, decimal initialBalance)
    {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.balance = initialBalance;
    }

    // Methods
    public void Deposit(decimal amount)
    {
        balance += amount;
        Console.WriteLine($"{amount:C} deposited. New balance: {balance:C}");
    }

    public void Withdraw(decimal amount)
    {
        if (balance >= amount)
        {
            balance -= amount;
            Console.WriteLine($"{amount:C} withdrawn. New balance: {balance:C}");
        }
        else
        {
```

```
            Console.WriteLine("Insufficient funds.");
        }
    }

    public void PrintAccountInfo()
    {
        Console.WriteLine($"Account Number: {accountNumber}");
        Console.WriteLine($"Account Holder: {accountHolderName}");
        Console.WriteLine($"Balance: {balance:C}");
    }
}
```

Now, let's break down what we have here:

- We've defined a class called `BankAccount`.
- Inside this class, we have attributes (`accountNumber`, `accountHolderName`, and `balance`), which represent the state of a bank account.
- We have a constructor that allows us to create a `BankAccount` object with initial values for these attributes.
- We have methods (`Deposit`, `Withdraw`, and `PrintAccountInfo`) that allow us to perform actions on a `BankAccount` object.

Now, let's create objects of this class and use them:

```
class Program
{
    static void Main(string[] args)
    {
        // Creating a BankAccount object
        BankAccount account1 = new BankAccount("123456", "John Doe", 1000);

        // Performing operations on the account
        account1.Deposit(500);
        account1.Withdraw(200);

        // Printing account information
        account1.PrintAccountInfo();
    }
}
```

In this example, we created a `BankAccount` object named `account1` with an account number "123456", holder name "John Doe", and initial balance of $1000. Then, we deposited $500 into the account, withdrew $200 from it, and finally printed its information.

Certainly! Object-oriented programming (OOP) is a paradigm that revolves around the concept of objects, which can contain data in the form of fields (attributes or properties) and code in the form of methods (functions). There are several key concepts in OOP, and I'll illustrate each of them using banking scenarios:

1. **Encapsulation**: Encapsulation is the bundling of data and methods that operate on the data into a single unit, called a class. The data is hidden from the outside world and can only be accessed through the methods provided by the class. This helps in data protection and prevents accidental modification of data.

```
public class BankAccount
{
    private string accountNumber;
    private string accountHolderName;
    private decimal balance;

    public BankAccount(string accountNumber, string accountHolderName, decimal
initialBalance)
    {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.balance = initialBalance;
    }

    // Methods for accessing and modifying data are encapsulated within the class
    public void Deposit(decimal amount)
    {
        balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (balance >= amount)
        {
            balance -= amount;
        }
        else
        {
            Console.WriteLine("Insufficient funds.");
        }
    }
}
```

2. **Abstraction**: Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object. In the banking scenario, a user doesn't need to know how the bank account operations are implemented internally; they only need to know how to use them.

```
public class BankAccount
{
    // Attributes and methods are abstracted away
    // Users only interact with the public interface of the class
```

```
}
```

3. **Inheritance**: Inheritance is a mechanism wherein a new class is derived from an existing class. The new class inherits the attributes and methods of the existing class and can also have its own additional attributes and methods. Inheritance promotes code reusability and establishes a relationship between classes.

```
public class SavingsAccount : BankAccount
{
    public decimal InterestRate { get; }

    public SavingsAccount(string accountNumber, string accountHolderName, decimal initialBalance, decimal interestRate)
        : base(accountNumber, accountHolderName, initialBalance)
    {
        this.InterestRate = interestRate;
    }

    public void CalculateInterest()
    {
        decimal interest = balance * InterestRate / 100;
        Deposit(interest);
    }
}
```

4. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables methods to be overridden in derived classes to provide specific implementations. In the banking scenario, different types of accounts may have different behaviors for certain operations.

```
public class CurrentAccount : BankAccount
{
    public decimal OverdraftLimit { get; }

    public CurrentAccount(string accountNumber, string accountHolderName, decimal initialBalance, decimal overdraftLimit)
        : base(accountNumber, accountHolderName, initialBalance)
    {
        this.OverdraftLimit = overdraftLimit;
    }

    public override void Withdraw(decimal amount)
    {
        if (balance + OverdraftLimit >= amount)
        {
            balance -= amount;
        }
        else
```

```
        {
            Console.WriteLine("Withdrawal amount exceeds overdraft limit.");
        }
    }
}
```

Certainly! In C#, strings are a sequence of characters represented by the `string` data type. There are several built-in methods in C# that allow you to manipulate strings. Let's go through some of the commonly used string functions with examples:

1. **`Length`**: Returns the length of the string.

   ```
   string str = "Hello";
   int length = str.Length; // length will be 5
   ```

2. **`Substring`**: Retrieves a substring from the original string.

   ```
   string str = "Hello World";
   string substr = str.Substring(6); // substr will be "World"
   ```

3. **`IndexOf`**: Finds the index of the first occurrence of a specified substring.

   ```
   string str = "Hello World";
   int index = str.IndexOf("World"); // index will be 6
   ```

4. **`Replace`**: Replaces all occurrences of a specified substring with another substring.

   ```
   string str = "Hello World";
   string newStr = str.Replace("World", "Universe"); // newStr will be "Hello Universe"
   ```

5. **`ToUpper`** / **`ToLower`**: Converts the string to uppercase or lowercase.

   ```
   string str = "Hello";
   string upper = str.ToUpper(); // upper will be "HELLO"
   string lower = str.ToLower(); // lower will be "hello"
   ```

6. **`Trim`**: Removes leading and trailing white spaces from the string.

```
string str = "   Hello   ";
string trimmed = str.Trim(); // trimmed will be "Hello"
```

7. **Split**: Splits a string into substrings based on a specified delimiter.

```
string str = "apple,banana,orange";
string[] fruits = str.Split(','); // fruits will be ["apple", "banana", "orange"]
```

Now, let's combine some of these functions to demonstrate a real-time example:

Suppose we have a CSV (Comma-Separated Values) file containing student records with names and scores. We want to read this file, process it, and output the names of students who scored above a certain threshold.

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string filePath = "students.csv";
        int passingScore = 70;

        // Read all lines from the file
        string[] lines = File.ReadAllLines(filePath);

        Console.WriteLine("Students who passed:");

        // Process each line
        foreach (string line in lines)
        {
            // Split each line into name and score
            string[] parts = line.Split(',');
            string name = parts[0];
            int score = int.Parse(parts[1]);

            // Check if score is above passingScore
            if (score > passingScore)
            {
                Console.WriteLine(name);
            }
        }
    }
}
```

Sure! Arrays are fundamental data structures in C# that allow you to store multiple values of the same type in a single variable. Let's go through the concepts of arrays with explanations and real-world examples:

1. **Declaration and Initialization**: Arrays in C# are declared using square brackets `[]` after the data type.

   ```
   // Declaration
   int[] numbers;

   // Initialization
   numbers = new int[5]; // Creates an array of 5 integers
   ```

   Real-world example: Suppose you want to store the scores of students in a class. You can declare an array of integers to store these scores.

2. **Accessing Elements**: Elements of an array are accessed using their index, which starts from 0.

   ```
   int[] numbers = {10, 20, 30, 40, 50};
   int firstNumber = numbers[0]; // Accesses the first element (10)
   ```

   Real-world example: If you have an array storing temperatures recorded each hour in a day, you can access the temperature at a specific hour by its index.

3. **Length Property**: Arrays have a `Length` property that gives the total number of elements in the array.

   ```
   int[] numbers = {10, 20, 30, 40, 50};
   int length = numbers.Length; // length will be 5
   ```

   Real-world example: You can use the `Length` property to iterate over all elements of an array in a loop.

4. **Initialization with Values**: Arrays can be initialized with values at the time of declaration.

   ```
   int[] numbers = {10, 20, 30, 40, 50};
   ```

   Real-world example: When you know the initial values you want to store in an array (e.g., days of the week), you can initialize it with those values directly.

5. **Iterating through an Array**: You can use loops like `for` or `foreach` to iterate through all elements of an array.

```
int[] numbers = {10, 20, 30, 40, 50};
foreach (int num in numbers)
{
    Console.WriteLine(num);
}
```

Real-world example: If you have an array storing product prices, you might want to iterate through it to display all prices to a user.

6. **Array of Objects**: Arrays can hold objects of any type, including custom objects.

```
// Custom class
class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
}

// Array of custom objects
Student[] students = new Student[3];
students[0] = new Student { Name = "Alice", Age = 20 };
students[1] = new Student { Name = "Bob", Age = 22 };
students[2] = new Student { Name = "Charlie", Age = 21 };
```

Certainly! In C#, collections are used to store and manipulate groups of objects. They provide various data structures that offer different functionalities and performance characteristics. Let's explore the concepts of collections in C# with explanations and real-world examples:

1. **List<T>**: List is one of the most commonly used collections in C#. It represents a dynamic array that can grow or shrink in size.

```
List<string> names = new List<string>();
names.Add("Alice");
names.Add("Bob");
names.Add("Charlie");

foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Real-world example: If you're developing an application that manages a list of tasks, you can use a `List<Task>` to store and manipulate these tasks efficiently.

2. **`Dictionary<TKey, TValue>`**: Dictionary stores key-value pairs where each key is unique.

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("Alice", 25);
ages.Add("Bob", 30);
ages.Add("Charlie", 28);

int bobAge = ages["Bob"];
Console.WriteLine($"Bob's age is {bobAge}");
```

Real-world example: In an address book application, you can use a `Dictionary<string, string>` to store names and corresponding phone numbers.

3. **`HashSet<T>`**: HashSet represents a collection of unique elements.

```
HashSet<string> uniqueNames = new HashSet<string>();
uniqueNames.Add("Alice");
uniqueNames.Add("Bob");
uniqueNames.Add("Alice"); // This will not be added again

foreach (string name in uniqueNames)
{
    Console.WriteLine(name);
}
```

Real-world example: When you need to store a list of unique email addresses in an email client application, you can use a `HashSet<string>` to ensure no duplicates are present.

4. **`Queue<T>`**: Queue represents a first-in, first-out (FIFO) collection of objects.

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("Task 1");
queue.Enqueue("Task 2");
queue.Enqueue("Task 3");

string nextTask = queue.Dequeue();
```

```
Console.WriteLine($"Next task to be processed: {nextTask}");
```

Real-world example: In a ticketing system, you can use a `Queue<Ticket>` to manage incoming support tickets in the order they were received.

5. **`Stack<T>`**: Stack represents a last-in, first-out (LIFO) collection of objects.

```
Stack<string> stack = new Stack<string>();
stack.Push("Page 1");
stack.Push("Page 2");
stack.Push("Page 3");

string currentPage = stack.Pop();
Console.WriteLine($"Current page: {currentPage}");
```

Absolutely! Collections in C# come with a plethora of methods and functions to manipulate the data they contain. Let's explore each collection type along with some of its common methods and functions, demonstrated with real-world examples:

1. **`List<T>`**:

   - `Add(T item)`: Adds an item to the end of the list.
   - `Remove(T item)`: Removes the first occurrence of a specific item from the list.
   - `Contains(T item)`: Determines whether the list contains a specific value.

```
List<string> names = new List<string>();
names.Add("Alice");
names.Add("Bob");
names.Add("Charlie");

if (names.Contains("Alice"))
{
    names.Remove("Alice");
}
```

Real-world example: In an online shopping application, you can use a list to store items in the user's shopping cart. You can then add, remove, or check for the presence of items using these methods.

2. **`Dictionary<TKey, TValue>`**:

- `Add(TKey key, TValue value)`: Adds the specified key and value to the dictionary.
- `Remove(TKey key)`: Removes the value with the specified key from the dictionary.
- `ContainsKey(TKey key)`: Determines whether the dictionary contains the specified key.

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("Alice", 25);
ages.Add("Bob", 30);
ages.Add("Charlie", 28);

if (ages.ContainsKey("Alice"))
{
    ages.Remove("Alice");
}
```

Real-world example: In an employee management system, you can use a dictionary to store employee IDs as keys and their corresponding details as values. You can then add, remove, or check for the presence of employees using these methods.

3. **`HashSet<T>`**:

- `Add(T item)`: Adds the specified item to the set.
- `Remove(T item)`: Removes the specified item from the set.
- `Contains(T item)`: Determines whether the set contains the specified item.

```
HashSet<string> uniqueNames = new HashSet<string>();
uniqueNames.Add("Alice");
uniqueNames.Add("Bob");
uniqueNames.Add("Alice"); // This will not be added again

if (uniqueNames.Contains("Alice"))
{
    uniqueNames.Remove("Alice");
}
```

Real-world example: In a social media platform, you can use a hash set to store unique user IDs. You can then add new users, remove deactivated users, or check for the existence of users using these methods.

4. **`Queue<T>`**:

- `Enqueue(T item)`: Adds an item to the end of the queue.
- `Dequeue()`: Removes and returns the object at the beginning of the queue.
- `Peek()`: Returns the object at the beginning of the queue without removing it.

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("Task 1");
queue.Enqueue("Task 2");
queue.Enqueue("Task 3");

string nextTask = queue.Dequeue();
```

Real-world example: In a printer queue application, you can use a queue to manage print jobs. You can enqueue new print jobs, dequeue completed jobs, and peek at the next job to be processed.

5. **`Stack<T>`**:

- `Push(T item)`: Adds an item to the top of the stack.
- `Pop()`: Removes and returns the object at the top of the stack.
- `Peek()`: Returns the object at the top of the stack without removing it.

```
Stack<string> stack = new Stack<string>();
stack.Push("Page 1");
stack.Push("Page 2");
stack.Push("Page 3");

string currentPage = stack.Pop();
```

Banking Project:

Certainly! Let's modify the previous banking project to include the functionality you described. We'll start with a while loop that presents options for creating a new account, accessing an existing account, or exiting the program. When creating a new account, the user can choose between a savings account and a current account. Existing accounts can perform deposit, withdrawal, and balance check operations. We'll use collections to store the accounts.

Here's the modified code:

```
using System;
using System.Collections.Generic;

class BankAccount
{
    public string AccountNumber { get; }
    public string OwnerName { get; }
    public decimal Balance { get; private set; }

    public BankAccount(string accountNumber, string ownerName, decimal initialBalance)
    {
        AccountNumber = accountNumber;
        OwnerName = ownerName;
        Balance = initialBalance;
    }

    public void Deposit(decimal amount)
    {
        Balance += amount;
        Console.WriteLine($"Deposited {amount:C}. Current balance: {Balance:C}");
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= Balance)
        {
            Balance -= amount;
            Console.WriteLine($"Withdrawn {amount:C}. Current balance: {Balance:C}");
        }
        else
        {
            Console.WriteLine("Insufficient funds.");
        }
    }

    public void CheckBalance()
    {
        Console.WriteLine($"Account balance: {Balance:C}");
    }
}

class Program
{
    static Dictionary<string, BankAccount> accounts = new Dictionary<string,
BankAccount>();

    static void Main()
    {
        bool exit = false;

        do
```

```
        {
            Console.WriteLine("Welcome to MyBank!");
            Console.WriteLine("1. Create Account");
            Console.WriteLine("2. Already have Account");
            Console.WriteLine("3. Exit");
            Console.Write("Enter your choice: ");

            string choice = Console.ReadLine();

            switch (choice)
            {
                case "1":
                    CreateAccount();
                    break;
                case "2":
                    AccessAccount();
                    break;
                case "3":
                    exit = true;
                    Console.WriteLine("Thank you for banking with us!");
                    break;
                default:
                    Console.WriteLine("Invalid choice. Please try again.");
                    break;
            }

            Console.WriteLine();
        } while (!exit);
    }

    static void CreateAccount()
    {
        Console.WriteLine("Select account type:");
        Console.WriteLine("1. Savings Account");
        Console.WriteLine("2. Current Account");
        Console.Write("Enter your choice: ");
        string accountTypeChoice = Console.ReadLine();

        Console.Write("Enter account number (10 digits): ");
        string accountNumber = Console.ReadLine();

        if (accountNumber.Length != 10)
        {
            Console.WriteLine("Account number should be 10 digits long.");
            return;
        }

        if (accounts.ContainsKey(accountNumber))
        {
            Console.WriteLine("Account number already exists.");
            return;
```

```
        }

        Console.Write("Enter owner's name: ");
        string ownerName = Console.ReadLine();

        Console.Write("Enter initial balance: ");
        decimal initialBalance;
        while (!decimal.TryParse(Console.ReadLine(), out initialBalance))
        {
            Console.WriteLine("Invalid input. Please enter a valid amount.");
        }

        BankAccount account;
        if (accountTypeChoice == "1")
        {
            account = new BankAccount(accountNumber, ownerName, initialBalance);
        }
        else if (accountTypeChoice == "2")
        {
            account = new BankAccount(accountNumber, ownerName, initialBalance);
        }
        else
        {
            Console.WriteLine("Invalid account type.");
            return;
        }

        accounts.Add(accountNumber, account);

        Console.WriteLine("Account created successfully!");
    }

    static void AccessAccount()
    {
        Console.Write("Enter account number: ");
        string accountNumber = Console.ReadLine();

        if (!accounts.ContainsKey(accountNumber))
        {
            Console.WriteLine("Account number does not exist.");
            return;
        }

        BankAccount account = accounts[accountNumber];

        bool exit = false;
        do
        {
            Console.WriteLine("1. Deposit");
            Console.WriteLine("2. Withdraw");
            Console.WriteLine("3. Check Balance");
```

```
            Console.WriteLine("4. Exit");
            Console.Write("Enter your choice: ");

            string choice = Console.ReadLine();

            switch (choice)
            {
                case "1":
                    Console.Write("Enter deposit amount: ");
                    decimal depositAmount;
                    while (!decimal.TryParse(Console.ReadLine(), out depositAmount))
                    {
                        Console.WriteLine("Invalid input. Please enter a valid amount.");
                    }
                    account.Deposit(depositAmount);
                    break;
                case "2":
                    Console.Write("Enter withdrawal amount: ");
                    decimal withdrawalAmount;
                    while (!decimal.TryParse(Console.ReadLine(), out withdrawalAmount))
                    {
                        Console.WriteLine("Invalid input. Please enter a valid amount.");
                    }
                    account.Withdraw(withdrawalAmount);
                    break;
                case "3":
                    account.CheckBalance();
                    break;
                case "4":
                    exit = true;
                    break;
                default:
                    Console.WriteLine("Invalid choice. Please try again.");
                    break;
            }

            Console.WriteLine();
        } while (!exit);
    }
}
```

In this modified version:

- When creating an account, the user can choose between a savings account and a current account.
- The program checks the validity of the account number (must be 10 digits) and ensures that no duplicate accounts are created.
- Accessing an existing account allows users to deposit funds, withdraw funds, or check their balance.

- Operations like deposit, withdrawal, and balance check are performed on the selected account.

Certainly! Exception handling is a mechanism in C# (and many other programming languages) that allows you to gracefully handle unexpected or exceptional situations that may occur during the execution of your code. Let's explore how exceptions work and how to handle them, along with real-world examples:

Understanding Exceptions:

In C#, exceptions are objects that represent errors that occur during the execution of a program. When an exceptional situation occurs, an exception object is created and thrown.

Here's a basic structure of an exception:

```
try
{
    // Code that may cause an exception
}
catch (ExceptionType ex)
{
    // Code to handle the exception
}
```

- The `try` block contains the code that may throw an exception.
- If an exception occurs within the `try` block, the runtime searches for a matching `catch` block.
- If a matching `catch` block is found, the code inside the `catch` block is executed to handle the exception.
- If no matching `catch` block is found, the exception is propagated up the call stack until it is caught or the program terminates.

Handling Exceptions:

Let's consider some common scenarios and how to handle them using exceptions:

1. **Divide by Zero Exception**:

```
try
{
    int result = 10 / 0; // This will throw DivideByZeroException
}
catch (DivideByZeroException ex)
{
```

```
    Console.WriteLine("Error: Cannot divide by zero.");
}
```

2. **File I/O Exception**:

```
try
{
    // Attempt to read a file
    string content = File.ReadAllText("nonexistentfile.txt");
    Console.WriteLine(content);
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("Error: File not found.");
}
catch (IOException ex)
{
    Console.WriteLine("Error reading file: " + ex.Message);
}
```

3. **Database Connection Exception**:

```
try
{
    // Attempt to connect to a database
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        // Execute database operations
    }
}
catch (SqlException ex)
{
    Console.WriteLine("Error connecting to database: " + ex.Message);
}
```

Real-World Examples:

- **Online Shopping Application**: When processing a customer's order, if an item is out of stock or the payment transaction fails, you can handle these exceptional situations using try-catch blocks.
- **Flight Reservation System**: If a user tries to book a flight for an invalid date or destination, you can handle these exceptions gracefully by providing informative error messages.

- **Healthcare Management System**: When retrieving patient records from a database, if there's a connection issue or the requested data is not available, you can handle these exceptions to ensure smooth operation of the system.

Certainly! File I/O (Input/Output) operations in C# are commonly used for reading from and writing to files. Let's cover the basic concepts of file I/O, including reading, writing, and logging, with a real-time project example:

Reading from a File:

You can read from a file using classes like `StreamReader`. Here's how to read from a file and log the operation:

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string filePath = "example.txt";

        try
        {
            // Start time
            DateTime startTime = DateTime.Now;

            // Read from file
            using (StreamReader reader = new StreamReader(filePath))
            {
                string content = reader.ReadToEnd();
                Console.WriteLine("File content:");
                Console.WriteLine(content);
            }

            // End time
            DateTime endTime = DateTime.Now;

            // Calculate and log the time taken
            TimeSpan duration = endTime - startTime;
            Log($"Read operation completed in {duration.TotalMilliseconds}
milliseconds.");
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("File not found.");
        }
        catch (IOException ex)
```

```
        {
            Console.WriteLine($"Error reading file: {ex.Message}");
        }
    }

    static void Log(string message)
    {
        // Log message with timestamp
        string logMessage = $"{DateTime.Now} - {message}";
        Console.WriteLine(logMessage);

        // Write to log file
        File.AppendAllText("log.txt", logMessage + Environment.NewLine);
    }
}
```

Writing to a File:

Similarly, you can write to a file using classes like `StreamWriter`. Here's how to write to a file and log the operation:

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string filePath = "example.txt";

        try
        {
            // Start time
            DateTime startTime = DateTime.Now;

            // Write to file
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                writer.WriteLine("Hello, world!");
            }

            // End time
            DateTime endTime = DateTime.Now;

            // Calculate and log the time taken
            TimeSpan duration = endTime - startTime;
            Log($"Write operation completed in {duration.TotalMilliseconds}
milliseconds.");
        }
```

```
        catch (IOException ex)
        {
            Console.WriteLine($"Error writing to file: {ex.Message}");
        }
    }

    static void Log(string message)
    {
        // Log message with timestamp
        string logMessage = $"{DateTime.Now} - {message}";
        Console.WriteLine(logMessage);

        // Write to log file
        File.AppendAllText("log.txt", logMessage + Environment.NewLine);
    }
}
```

Real-Time Project Example: Logging System:

In a real-time project, you could create a logging system that records various events and operations, along with their timestamps, to a log file. This logging system can help in troubleshooting and monitoring the application.

For example, in a web server application, you can log each HTTP request received, along with its details such as client IP address, request method, requested URL, and timestamp. This log can be invaluable for analyzing traffic patterns, identifying errors, and ensuring the smooth operation of the server.

Basic Concepts:

1. **Public:** Members declared as public are accessible from any other class or assembly. They have the widest scope.

```
public class MyClass
{
    public int PublicVar = 10;
    public void PublicMethod()
    {
        Console.WriteLine("This is a public method.");
    }
}
```

2. **Private:** Members declared as private are accessible only within the same class. They have the narrowest scope.

```
public class MyClass
{
    private int PrivateVar = 20;
    private void PrivateMethod()
    {
        Console.WriteLine("This is a private method.");
    }
}
```

3. **Protected:** Members declared as protected are accessible within the same class and its derived classes.

```
public class MyBaseClass
{
    protected int ProtectedVar = 30;
}

public class MyDerivedClass : MyBaseClass
{
    public void AccessProtectedVar()
    {
        Console.WriteLine("Protected variable from base class: " + ProtectedVar);
    }
}
```

4. **Internal:** Members declared as internal are accessible within the same assembly.

```
internal class MyInternalClass
{
    internal void InternalMethod()
    {
        Console.WriteLine("This is an internal method.");
    }
}
```

Advanced Concepts:

1. **Protected Internal:** Members declared as protected internal are accessible within the same assembly and from derived classes.

```
public class MyBaseClass
{
    protected internal int ProtectedInternalVar = 40;
}

public class MyDerivedClass : MyBaseClass
{
    public void AccessProtectedInternalVar()
    {
        Console.WriteLine("Protected internal variable from base class: " +
ProtectedInternalVar);
    }
}
```

2. **Private Protected:** Members declared as private protected are accessible within the same assembly and from derived classes, but only if the derived class is within the same assembly.

```
public class MyBaseClass
{
    private protected int PrivateProtectedVar = 50;
}

public class MyDerivedClass : MyBaseClass
{
    public void AccessPrivateProtectedVar()
    {
        Console.WriteLine("Private protected variable from base class: " +
PrivateProtectedVar);
    }
}
```

Real-time examples can vary depending on the context of your application, but these access modifiers help you control the visibility and accessibility of your code, ensuring better encapsulation and maintainability. Feel free to ask if you have any questions or need further clarification on any of these concepts!