Module 5

Creating a Class Hierarchy by Using Inheritance

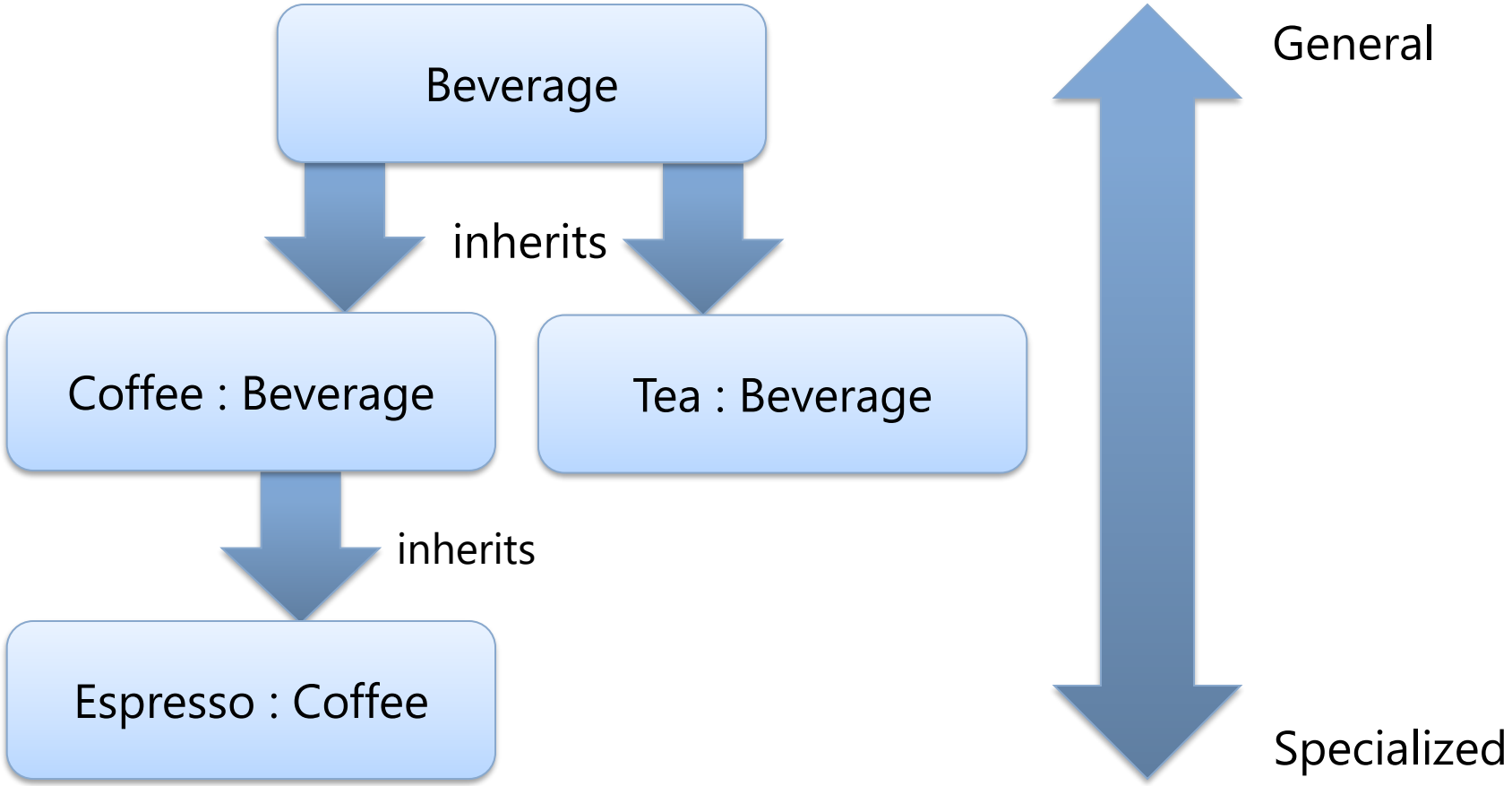
Module Overview

- Creating Class Hierarchies
- Extending Existing Code

Lesson 1: Creating Class Hierarchies

- What Is Inheritance?
- Controlling Access to Inherited Members
- Calling Base Class Constructors and Members
- Virtual Members
- Overriding v Hiding
- Controlling Inheritance

What Is Inheritance?



The diagram shows a class hierarchy where a class named **Espresso** inherits from a class named **Coffee**, which in turn inherits from a class named **Beverage**. The inherited classes are increasingly specialized instances of the base class.

What Is Inheritance?

- Add the name of the base class to the declaration

```
public class Coffee : Beverage
```

- The class **Coffee** now contains all members of **Beverage**

- What is the benefit of inheritance?
  - Code reuse
  - Modelling the real world
  - Simplicity

Controlling Access to Inherited Members

- Access modifier recap
  - What access modifiers do we know already?

- Use the **protected** access modifier to make members available to derived types

```
protected int servingTemperature;
```

- protected internal

Calling Base Class Constructors and Members

- To call a base class constructor from a derived class, add the base constructor to your constructor declaration

```
public Coffee(string name, bool isFairTrade, int temp)
   : base(name, isFairTrade, servingTemp)
```

  - Pass parameter names to the base constructor as arguments
  - Do not use the base keyword within the constructor body
- To call base class methods from a derived class, use the base keyword like an instance variable

```
base.GetServingTemperature();
```

Virtual Members

- Use the **virtual** keyword to create members that you can override in derived classes

```
public virtual int GetServingTemperature() { … }
```

- `virtual` in C# v `Overridable` in VB.NET

- Use the **abstract** keyword to create a virtual method without a body

```
public abstract int GetServingTemperature();
```

Overriding v Hiding

- Polymorphism
- How do we override?

```
override public int GetServingTemperature()
```

- What is hiding?
  - An alternative to overriding
  - Both methods are preserved in the derived class
- **abstract** members must be overriden

Mandatory Inheritance

- Use the **abstract** keyword to create a base class that cannot be instantiated

```
public abstract class Beverage
```

- It's required if a class has an abstract member

- To create an instance
  - Create a class that derives from the abstract class
  - Implement any abstract members
  - Instantiate the derived class

Preventing Inheritance

- Use the **sealed** keyword to create a class that cannot be inherited

```
public sealed class Tea : Beverage
```

- To prevent classes further down the class hierarchy from overriding your override methods, use the **sealed** keyword

```
sealed public override int GetServingTemperature()
```

Demonstration: Calling Base Class Constructors

In this demonstration, you will learn how to:

- Create a derived class constructor that calls a default base class constructor implicitly
- Create a derived class constructor that calls a specific base class constructor explicitly
- Observe the order of constructor execution as a derived class is instantiated

Lesson 2: Extending Existing Code

- Inheriting from Existing Code
- Creating Custom Exceptions
- Throwing and Catching Custom Exceptions
- Inheriting from Generic Types
- Creating Extension Methods
- Demonstration: Refactoring Common Functionality into the User Class Lab

Inheriting from Existing Code

- Inheriting from a library
  - Our own organisation's code
  - .NET Framework
  - 3rd party library
- Inherit from shared classes to:
  - Reduce development time
  - Standardize functionality
- Inherit from any type that is not **sealed** or **static**

Creating Custom Exceptions

To create a custom exception type:

1. Inherit from the **System.Exception** class
2. Implement three standard constructors:
   - base()
   - base(string message)
   - base(string message, Exception inner)
3. Add additional members if required

Throwing and Catching Custom Exceptions

- Use the **throw** keyword to throw a custom exception

```
throw new LoyaltyCardNotFoundException();
```

- Use a try/catch block to catch the exception

```
try
{
   // Perform the operation that could cause the exception.
}
catch(LoyaltyCardNotFoundException ex)
{
   // Use the exception variable, ex, to get more information.
}
```

Inheriting from Generic Types

For each base type parameter, you must either:

- Provide a type argument in your class declaration

```
public class CustomList : List<int>
```

- Include a matching type parameter in your class declaration

```
public class CustomList<T> : List<T>
```

Creating Extension Methods

- Create a static method in a static class
- Use the first parameter to indicate the type you want to extend
- Precede the first parameter with the **this** keyword

```
public static bool ContainsNumbers(this string s) {...}
```

- Call the method like a regular instance method

```
string text = "Text with numb3r5 ";
if(text.ContainsNumbers())
{
   // Do something.
}
```

Demonstration: Refactoring Common Functionality into the User Class Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module

Lab: Refactoring Common Functionality into the User Class

- Exercise 1: Creating and Inheriting from the User Base Class
- Exercise 2: Implementing Password Complexity by Using an Abstract Method
- Exercise 3: Creating the ClassFullException Custom Exception

Estimated Time: 60 minutes

Lab Scenario

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

Module Review and Takeaways

- Review Questions