Creating custom types

Creating and Using Enums

- Create variables with a fixed set of possible values

```
enum Day { Sunday, Monday, Tuesday, Wednesday, … };
```

- Set instance to the member you want to use

```
Day favoriteDay = Day.Friday;
```

- Set enum variables by name or by value

```
Day day1 = Day.Friday;
// is equivalent to
Day day1 = (Day)4;
```

Creating and Using Structs

- Use structs to create simple custom types:
  - Represent related data items as a single logical entity
  - Add fields, properties, methods, and events

- Use the **struct** keyword to create a struct

```
public struct Coffee { ... }
```

- Use the **new** keyword to instantiate a struct

```
Coffee coffee1 = new Coffee();
```

Initializing Structs

- Use constructors to initialize a struct

```
public struct Coffee
{
   public Coffee(int strength, string bean, string origin)
   { ... }
}
```

- Provide arguments when you instantiate the struct

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
```

- Add multiple constructors with different combinations of parameters

Creating Properties

- Properties use get and set accessors to control access to private fields

```
private int strength;
public int Strength
{
   get { return strength; }
   set { strength = value; }
}
```

- Properties enable you to:
  - Control access to private fields
  - Change accessor implementations without affecting clients
  - Data-bind controls to property values

What Is a Method?

- Methods encapsulate operations that protect data
- .NET applications contain a **Main** entry point method
- .NET provides many methods in the base class library

Creating Methods

- Methods comprise two elements:
  - Method specification (return type, name, parameters)
  - Method body
- Passing parameters:

```
void StartService(int upTime, bool shutdownAutomatically)
{
    // Perform some processing here.
}
```

- Use the **return** keyword to return a value from the method

```
string GetServiceName()
{
    return "FourthCoffee.SalesService";
}
```

Invoking Methods

To call a method specify:

- Method name
- Any arguments to satisfy parameters

```
var upTime = 2000;
var shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

// StartService method.
void StartService(int upTime, bool shutdownAutomatically)
{
   // Perform some processing here.
}
```

Debugging Methods

- Visual Studio provides debug tools that enable you to step through code
- When debugging methods you can:
  - Step into the method
  - Step over the method
  - Step out of the method

Demonstration: Creating, Invoking, and Debugging Methods

In this demonstration, you will create a method, invoke the method, and then debug the method

Creating Overloaded Methods

- Overloaded methods share the same method name
- Overloaded methods have a unique signature

```
void StopService()
{
   ...
}

void StopService(string serviceName)
{
   ...
}

void StopService(int serviceId)
{
   ...
}
```

Creating Methods that Use Optional Parameters

- Define all mandatory parameters first

```
void StopService(
   bool forceStop,
   string serviceName = null,
   int serviceId =1)
{
   ...
}
```

- Satisfy parameters in sequence

```
var forceStop = true;
StopService(forceStop);

// OR

var forceStop = true;
var serviceName = "FourthCoffee.SalesService";
StopService(forceStop, serviceName);
```

Calling a Method by Using Named Arguments

- Specify parameters by name
- Supply arguments in a sequence that differs from the method's signature
- Supply the parameter name and corresponding value separated by a colon

```
StopService(true, serviceID: 1);
```

Creating Methods that Use Output Parameters

- Use the **out** keyword to define an output parameter

```
bool IsServiceOnline(string serviceName, out string statusMessage)
{
    ...
}
```

- Provide a variable for the corresponding argument when you call the method

```
var statusMessage = string.Empty;
var isServiceOnline = IsServiceOnline(
   "FourthCoffee.SalesService",
    out statusMessage);
```