

Hoàn thiện dự án: exception, try-catch, Settings

[Hướng dẫn tự học lập trình C# toàn tập](#) > [Hoàn thiện dự án: exception, try-catch, Settings](#)

Trong loạt bài từ đầu đến giờ, chúng ta đã lần lượt hoàn thiện tất cả chức năng của ứng dụng theo phân tích. Tuy nhiên, trước khi đưa ứng dụng đến được người dùng cuối, chúng ta cần bổ sung thêm một số tính năng, vốn không liên quan trực tiếp đến việc phân tích nghiệp vụ.

Các chức năng mới này mang tính kỹ thuật hơn là nghiệp vụ, bao gồm bắt và xử lý ngoại lệ (Exception, giúp ứng dụng ổn định hơn), sử dụng file cấu hình (Settings, giúp người dùng thay đổi cấu hình của ứng dụng).

NỘI DUNG CỦA BÀI [Ấn]

1. Xử lý ngoại lệ (Exception Handling)
 - 1.1. Ngoại lệ và chế độ Debug
 - 1.2. Xử lý ngoại lệ ở chế độ Release
2. Thực hành 1: bổ sung chức năng bắt và xử lý lỗi
 - 2.1. Bước 1. Điều chỉnh phương thức Main
 - 2.2. Bước 2. Dịch và chạy chương trình với các lệnh lỗi
3. Vấn đề cấu hình của ứng dụng
4. Thực hành 2: bổ sung chức năng thiết lập cấu hình
 - 4.1. Bước 1. Tạo file settings
 - 4.2. Bước 2. Nhập các giá trị vào bảng thông tin cấu hình
 - 4.3. Bước 3. Tạo class Config trong file mã nguồn Config.cs trực thuộc project
 - 4.4. Bước 4. Thay đổi code của các class data access
 - 4.5. Bước 5. Thay đổi code của phương thức Main
 - 4.6. Bước 6. Thay đổi code của phương thức ConfigRouter
 - 4.7. Bước 7. Xây dựng thêm lớp ConfigController trong file ConfigController.cs trong thư mục Controllers
 - 4.8. Bước 8. Thay đổi phương thức ConfigRouter
 - 4.9. Bước 9. Dịch và chạy thử chương trình với các chức năng mới
5. Kết luận

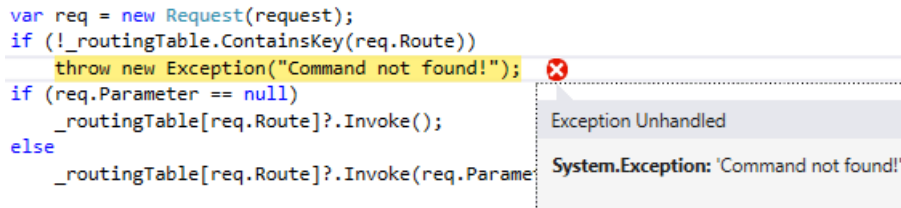
Xử lý ngoại lệ (Exception Handling)

Chúng ta đã nhắc đến khái niệm ngoại lệ (exception) và xem xét cách thức đơn giản nhất để phát thông báo ngoại lệ bằng lệnh `throw` và lớp `Exception`. Exception là một cơ chế rất mạnh trong .NET giúp phát hiện lỗi logic trong chương trình ở giai đoạn Runtime.

Ngoại lệ và chế độ Debug

Khi chạy chương trình ở chế độ *debug*, nếu phát sinh ngoại lệ, Visual Studio sẽ mở file mã nguồn ở đúng vị trí lỗi cùng với thông báo cụ thể. Qua đó, chúng ta có thể xác định nguồn gốc của lỗi và đưa ra cách giải quyết.

Hình dưới đây minh họa tình huống lỗi khi người dùng nhập vào một lệnh chưa tồn tại.



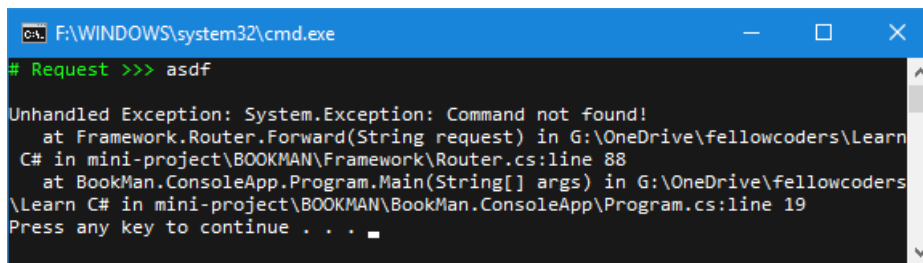
Ngoại lệ ở chế độ chạy debug

Đây là cơ chế bắt và xử lý lỗi ở chế độ Debug. Chương trình chúng ta viết từ đầu dự án đến giờ đều dịch và chạy ở chế độ Debug.

Xử lý ngoại lệ ở chế độ Release

Một chương trình trước khi đem triển khai cho người dùng cuối phải được dịch ở chế độ Release. Chương trình được dịch ở chế độ này sẽ không chạy được ở chế độ Debug nữa.

Nếu chương trình chạy ở chế độ Release mà gặp lỗi, thông báo lỗi sẽ được hiển thị như dưới đây.



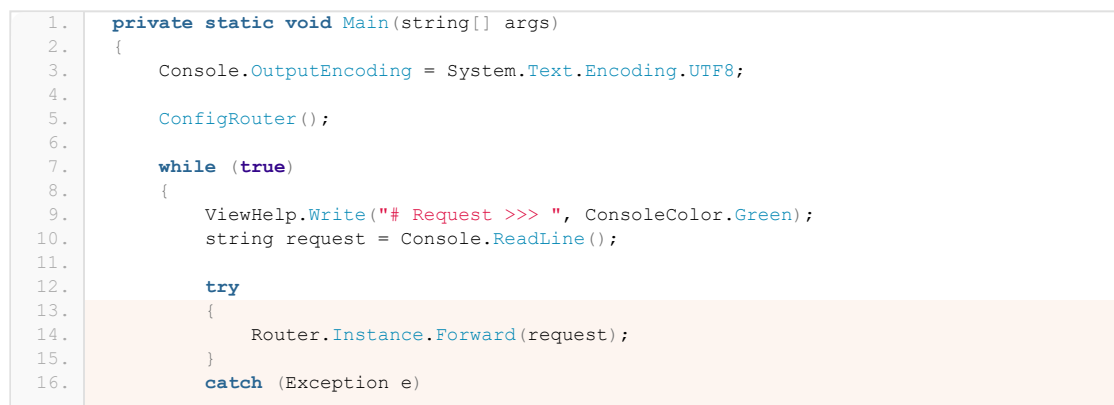
Đây là cơ chế bắt và xử lý lỗi mặc định của .NET framework đối với ứng dụng console.

Như chúng ta thấy, cơ chế bắt và xử lý lỗi mặc định của .NET framework tương đối không thân thiện với người dùng.

.NET cũng cung cấp cho các chương trình tính năng *bắt và xử lý ngoại lệ* (Exception Handling) để tự mình xác định hoạt động của chương trình khi xảy ra lỗi (ngoại lệ), tránh phải sử dụng cơ chế bắt và xử lý lỗi mặc định.

Thực hành 1: bổ sung chức năng bắt và xử lý lỗi

Bước 1. Điều chỉnh phương thức Main



```

17.     {
18.         ViewHelp.WriteLine(e.Message, ConsoleColor.Red);
19.     }
20.     finally
21.     {
22.         Console.WriteLine();
23.     }
24. }
25. }

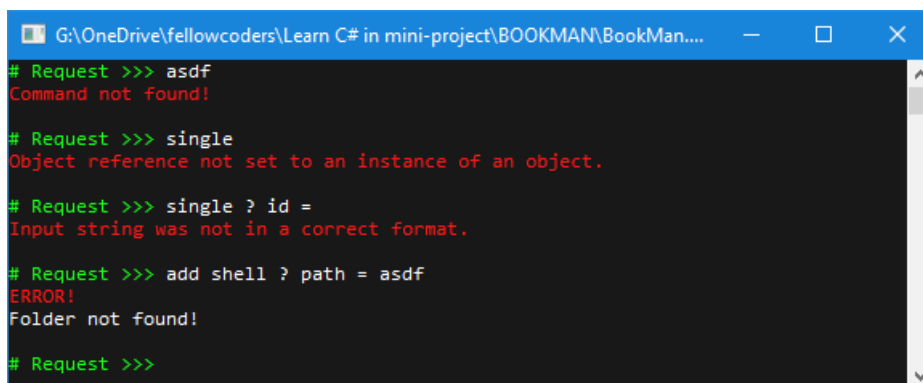
```

Khối `try` được đặt để kiểm soát “cửa ngõ” của chương trình: lời gọi phương thức `Forward`. Lời gọi phương thức này là khởi đầu của bất kỳ hoạt động nào của ứng dụng khi người dùng nhập truy vấn.

Khối `catch` được viết để bắt tất cả các loại lỗi (vì bắt lỗi thuộc loại `Exception`) và đưa vào biến `e`. Thông báo lỗi cụ thể được truy xuất qua thuộc tính `Message` của object `Exception` và viết ra mới màu đỏ.

Trong bất kỳ tình huống nào (dù thực hiện lệnh `Forward` có lỗi hay không) sẽ luôn in thêm một dòng trống sau khi thực hiện (khối `finally`).

Bước 2. Dịch và chạy chương trình với các lệnh lỗi



```

G:\OneDrive\fellowcoders\Learn C# in mini-project\BOOKMAN\BookMan...
# Request >>> asdf
Command not found!

# Request >>> single
Object reference not set to an instance of an object.

# Request >>> single ? id =
Input string was not in a correct format.

# Request >>> add shell ? path = asdf
ERROR!
Folder not found!

# Request >>>

```

Chạy chương trình với các lệnh lỗi

Chúng ta có thể thấy cơ chế bắt lỗi mặc định của .NET đã không còn kích hoạt nữa. Thay vào đó, cơ chế bắt lỗi riêng của chương trình đã hoạt động. Các thông báo cũng đơn giản nhẹ nhàng hơn. Có những thông báo xuất phát từ lỗi do chúng ta tự phát ra, cũng có một số lỗi được phát ra từ các phương thức chuẩn của .NET.

Chúng ta cũng để ý rằng, dù chạy ở chế độ nào (Debug hay Release), cơ chế bắt lỗi riêng cũng đều hoạt động. Điều này cũng có nghĩa là chúng ta đã vứt bỏ ưu thế của chế độ Debug: không xác định được vị trí gây lỗi, cũng không theo dõi được stack khi bị lỗi. Việc bắt lỗi như vậy không thích hợp ở giai đoạn phát triển ứng dụng. Cũng vì lý do này mà nội dung bắt và xử lý lỗi của chúng ta xem xét ở bài cuối cùng của dự án.

Vấn đề cấu hình của ứng dụng

Khi một chương trình đã được dịch, đóng gói và triển khai cho người dùng cuối, chúng ta không thể dễ dàng thay đổi nó được nữa vì liên quan đến nhiều khâu. Ví dụ, nếu chúng ta quyết định sử dụng file nhị phân để lưu trữ dữ liệu của chương trình thì sau khi triển khai,

nếu muốn chuyển sang dùng file json, chúng ta lại phải thực hiện trọn vẹn quy trình sửa mã nguồn => dịch => đóng gói => triển khai phiên bản mới.

Trong nhiều tình huống chúng ta đã dự trù sẵn những thay đổi cho chương trình mà người dùng đầu cuối có thể thực hiện. Ví dụ, chúng ta muốn cho phép người dùng cuối thay đổi màu sắc và văn bản của con trỏ nhắc lệnh. Chúng ta cũng muốn cho người dùng cuối lựa chọn loại file dữ liệu để lưu trữ (file nhị phân, xml hay json).

Một tình huống khác đó là thiết lập của ứng dụng ở giai đoạn phát triển không thể sử dụng ở giai đoạn triển khai. Ví dụ kết nối cơ sở dữ liệu ở giai đoạn phát triển ứng dụng hoàn toàn khác với kết nối ở giai đoạn triển khai. Khi đó, thông tin về chuỗi kết nối không thể code thẳng trong ứng dụng mà phải đặt ở file cấu hình để khi cài đặt chương trình người dùng sẽ trực tiếp thay đổi.

Rất nhiều tình huống tương tự dẫn tới nhu cầu lưu trữ và truy xuất thông tin cấu hình cho chương trình.

Trong phần thực hành này chúng ta sẽ vận dụng cơ chế lưu trữ và truy xuất thông tin cấu hình của .NET để xây dựng tính năng cấu hình. Tính năng này cho phép người dùng cuối thực hiện các thao tác sau:

- Thay đổi màu sắc và văn bản của dấu nhắc lệnh: Hiện nay chúng ta đang thiết lập cứng văn bản của dấu nhắc lệnh là `"# Request >>>"` với màu Green. Chúng ta muốn người dùng có thể tùy ý chọn văn bản và màu sắc của dấu nhắc lệnh.
- Thay đổi cơ chế và nơi lưu trữ dữ liệu: Hiện nay chúng ta đã xây dựng ba cơ chế lưu trữ dữ liệu khác nhau: binary, json, xml. Các cơ chế này lưu dữ liệu vào vào các file tương ứng là `data.bin`, `data.json`, `data.xml`. Chúng ta muốn người dùng có thể tùy chọn cơ chế lưu trữ và file dữ liệu.

Thực hành 2: bổ sung chức năng thiết lập cấu hình

Bước 1. Tạo file settings

Click đúp vào nút Properties của BookMan.ConsoleApp. Trong cửa sổ chọn mục Settings.

Vì project này chưa có file settings nào, phần nội dung bên tay phải đang trống. Click vào đường link sẽ tạo ra file settings đầu tiên của project. Mặc định file này có tên gọi Settings.settings và nằm trong nút Properties.

Visual Studio tạo ra một giao diện đồ họa để nhập các thiết lập (setting). Cũng có thể mở giao diện này bằng cách click đúp vào nút Settings.settings.

Bước 2. Nhập các giá trị vào bảng thông tin cấu hình

Nhập giá trị cho bảng settings

Đây là bảng thông tin đặc biệt, trong đó dữ liệu từ bảng sẽ được Visual Studio sử dụng để tự động sinh ra một class hỗ trợ truy xuất thông tin cầu hình. Class này có tên là Settings nằm trong không gian tên con Properties. Như trong project này, tên đầy đủ của lớp Settings là BookMan.ConsoleApp.Properties.Settings. Ở tất cả các file mã nguồn của project chúng ta đều có thể sử dụng lớp Settings này.

Mỗi setting chứa 4 thông tin:

Name: tên của setting. Thông tin Name sẽ được sử dụng để tạo ra một property tương ứng của class Settings. Settings là một class được Visual Studio sinh ra tự động dựa trên dữ liệu của bảng này, trong đó mỗi setting sẽ là một property của class. Vì vậy, Name phải tuân thủ theo quy tắc đặt tên biến và quy ước đặt tên property mà chúng ta đã học.

Type: kiểu dữ liệu của setting. Như trên đã nói, mỗi setting sẽ trở thành một property của class Settings, giá trị của *Type* sẽ là kiểu dữ liệu của property.

Lưu ý với setting PromptColor, kiểu dữ liệu System.ConsoleColor bình thường sẽ không xuất hiện trong danh sách lựa chọn. Chúng ta cần tự mình chỉ định vị trí chứa kiểu này: Trong

combo box của cột *Type* chọn *Browse*; trong hộp thoại *Select a Type* mở nhánh *mscorlib* => *System* => *ConsoleColor*. Khi đó, trong ô *Value* sẽ có thể mở combobox để lựa chọn một trong 16 màu của *ConsoleColor*.

Value: giá trị mặc định của setting. Giá trị này cũng tương đương với giá trị gán ban đầu cho mỗi property. Mặc dù trong ô nhập dữ liệu chúng ta chỉ nhập được chuỗi ký tự, giá trị này thực sự được chuyển đổi về kiểu tương ứng của thuộc tính. Vì lý do này, chỉ những kiểu có khả năng serialize (tuần tự hóa) về xml mới có thể được sử dụng.

Scope: quyết định phạm vi sử dụng của setting. *Scope* có thể nhận một trong hai giá trị: *User* hoặc *Application*.

Application scope quyết định rằng đây là một thuộc tính chỉ đọc (read-only). Chúng ta không thể thay đổi giá trị của setting trong khi chương trình hoạt động. *Scope* này sử dụng đối với các setting cấu hình một lần khi triển khai hệ thống và sau đó không (hoặc hiếm khi) thay đổi nữa.

User scope sử dụng cho những setting cần thay đổi, ngay cả khi chương trình hoạt động (và có hiệu lực ngay lập tức). Trong project này chúng ta đặt cả 4 setting trong user scope.

Khi lưu bảng cấu hình lại, Visual Studio sẽ tự động lưu thông tin vào file *App.config* như sau:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <configuration>
3.   <configSections>
4.
5.     <sectionGroup name="userSettings" type="System.Configuration.UserSettingsGroup, Syst
6.     <section name="BookMan.ConsoleApp.Properties.Settings" type="System.Configuration.
7.   </sectionGroup>
8. </configSections>
9. <startup>
10.   <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
11. </startup>
12.
13. <userSettings>
14.   <BookMan.ConsoleApp.Properties.Settings>
15.     <setting name="DataAccess" serializeAs="String">
16.       <value>binary</value>
17.     </setting>
18.     <setting name="PromptText" serializeAs="String">
19.       <value># Command >>></value>
20.     </setting>
21.     <setting name="PromptColor" serializeAs="String">
22.       <value>Green</value>
23.     </setting>
24.     <setting name="DataFile" serializeAs="String">
25.       <value>data.bin</value>
26.     </setting>
27.   </BookMan.ConsoleApp.Properties.Settings>
28. </userSettings>
29. </configuration>
```

Khi dịch ra ứng dụng, nội dung của file *App.config* sẽ được copy vào một file được đặt tên theo tên ứng dụng với phần mở rộng *.config*. Đối với ứng dụng này, file cấu hình khi triển khai là *BookMan.ConsoleApp.exe.config*.

Song song với lưu dữ liệu vào file *App.config*, Visual Studio cũng tự sinh ra code cho class Settings:

Không nên thay đổi nội dung của file này vì nếu như dữ liệu thay đổi, Visual Studio sẽ tự động sinh lại code cho class này. Khi đó những thay đổi của người dùng sẽ mất. Vì đây là một partial class, nếu thực sự muốn bổ sung code có thể tạo thêm file mã nguồn nữa ghép nối với class này.

Bước 3. Tạo class Config trong file mã nguồn Config.cs trực thuộc project

```
1. using System;
2. namespace BookMan.ConsoleApp
3. {
4.     using DataServices;
5.
6.     internal class Config
7.     {
8.         private static Config _instance;
9.         public static Config Instance = _instance ?? (_instance = new Config());
10.        private Config() { }
11.
12.        private Properties.Settings _s = Properties.Settings.Default;
13.
14.        public void Reload() => _s.Reload();
15.
16.        public IDataAccess IDataAccess
17.        {
18.            get {
19.                var da = _s.DataAccess;
20.                switch (da.ToLower())
21.                {
22.                    case "binary": return new BinaryDataAccess();
23.                    case "json": return new JsonDataAccess();
24.                    case "xml": return new XmlDataAccess();
25.                    default: return new BinaryDataAccess();
26.                }
27.            }
28.        }
29.
30.        public string DataAccess
31.        {
32.            get => _s.DataAccess;
33.            set {
34.                _s.DataAccess = value;
35.                _s.Save();
36.            }
37.        }
38.
39.        public string PromptText
40.        {
41.            get => _s.PromptText;
42.            set {
43.                _s.PromptText = value;
44.                _s.Save();
45.            }
46.        }
47.    }
48. }
```



```

45.         }
46.     }
47.
48.     public ConsoleColor PromptColor
49.     {
50.         get => _s.PromptColor;
51.         set {
52.             _s.PromptColor = value;
53.             _s.Save();
54.         }
55.     }
56.
57.     public string DataFile
58.     {
59.         get => _s.DataFile;
60.         set {
61.             _s.DataFile = value;
62.             _s.Save();
63.         }
64.     }
65. }
66. }

```

Bước 4. Thay đổi code của các class data access

```

1. // lớp BinaryDataAccess
2. public class BinaryDataAccess : IDataAccess
3. {
4.     public List<Book> Books { get; set; } = new List<Book>();
5.     private readonly string _file = Config.Instance.DataFile; // "data.dat";
6.     ...
7.
8. // lớp JsonDataAccess
9. public class JsonDataAccess : IDataAccess
10. {
11.     public List<Book> Books { get; set; } = new List<Book>();
12.     private readonly string _file = Config.Instance.DataFile; // "data.json";
13.     ...
14.
15. // lớp XmlDataAccess
16. public class XmlDataAccess : IDataAccess
17. {
18.     public List<Book> Books { get; set; } = new List<Book>();
19.     private readonly string _file = Config.Instance.DataFile; // "data.xml";
20.     ...

```

Bước 5. Thay đổi code của phương thức Main

```

1. private static void Main(string[] args)
2. {
3.     Console.OutputEncoding = System.Text.Encoding.UTF8;
4.     var text = Config.Instance.PromptText;
5.     var color = Config.Instance.PromptColor;
6.     ConfigRouter();
7.
8.     while (true)
9.     {
10.         ViewHelp.Write(text, color);
11.         string request = Console.ReadLine();
12.
13.         try
14.         {
15.             Router.Instance.Forward(request);
16.         }
17.         catch (Exception e)
18.         {
19.             ViewHelp.WriteLine(e.Message, ConsoleColor.Red);
20.         }
21.         finally
22.         {
23.             Console.WriteLine();
24.         }
25.     }
26. }

```

Bước 6. Thay đổi code của phương thức ConfigRouter

```
1. private static void ConfigRouter()  
2. {  
3.     IDataAccess context = Config.Instance.IDataAccess; //new BinaryDataAccess();  
4.     BookController controller = new BookController(context);  
5.     ShellController shell = new ShellController(context);  
6.     ...
```

Bước 7. Xây dựng thêm lớp ConfigController trong file ConfigController.cs trong thư mục Controllers

```
1. using System;  
2.  
3. namespace BookMan.ConsoleApp.Controllers  
4. {  
5.     using Framework;  
6.  
7.     internal class ConfigController : ControllerBase  
8.     {  
9.         private Config _c = Config.Instance;  
10.        public void ConfigPromptText(string text)  
11.        {  
12.            _c.PromptText = text;  
13.            Success("The command prompt will change next time");  
14.        }  
15.  
16.        public void ConfigPromptColor(string text)  
17.        {  
18.            if (Enum.TryParse(text, true, out ConsoleColor color))  
19.            {  
20.                _c.PromptColor = color;  
21.                Success("The command prompt color will change nex time");  
22.            }  
23.        }  
24.  
25.        public void CurrentDataAccess()  
26.        {  
27.            var da = _c.DataAccess;  
28.            var file = _c.DataFile;  
29.            Inform($"Current data access engine: {da}\nCurrent data file: {file}");  
30.        }  
31.  
32.        public void ConfigDataAccess(string da, string file)  
33.        {  
34.            _c.DataAccess = da;  
35.            _c.DataFile = file;  
36.            Success("The changes will be available next time");  
37.        }  
38.    }  
39. }
```

Bước 8. Thay đổi phương thức ConfigRouter

Bổ sung khai báo object của kiểu ConfigController:

```
1. private static void ConfigRouter()  
2. {  
3.     IDataAccess context = Config.Instance.IDataAccess;  
4.  
5.     BookController controller = new BookController(context);  
6.     ShellController shell = new ShellController(context);  
7.     ConfigController config = new ConfigController();  
8.  
9.     Router r = Router.Instance;
```

Bổ sung thêm các route sau:

```
1. r.Register(route: "config prompt text",  
2.     action: p => config.ConfigPromptText(p["text"]),  
3.     help: "[config prompt text ? text = <value>]");
```

```
4. r.Register(route: "config prompt color",
5.     action: p => config.ConfigPromptColor(p["color"]),
6.     help: "[config prompt color ? color = <value>]");
7. r.Register(route: "current data access",
8.     action: p => config.CurrentDataAccess(),
9.     help: "[current data access]");
10. r.Register(route: "config data access",
11.     action: p => config.ConfigDataAccess(p["da"], p["file"]),
12.     help: "[config data access ? da = <value:json, binary, xml> & file = <value>]");
```

Bước 9. Dịch và chạy thử chương trình với các chức năng mới

*Chạy
chức
năng
cấu
hình*

Sau khi thực hiện các lệnh trên, đóng và chạy lại chương trình để thấy các thiết lập mới đã có hiệu lực.

Như vậy, chúng ta đã cung cấp cho người dùng cuối khả năng thay đổi các cấu hình cơ bản: màu sắc và văn bản của dấu nhắc lệnh; cơ chế lưu trữ dữ liệu và file dữ liệu.

Kết luận

Trong bài học này chúng ta đã xem xét hai vấn đề: Exception và Settings. Đây là những kỹ thuật bổ sung giúp ứng dụng hoạt động ổn định và uyển chuyển hơn. Chúng ta cũng đã vận dụng hai kỹ thuật này vào ứng dụng.

Trong bài tiếp theo và là bài học cuối cùng của chuỗi bài giảng này, chúng ta sẽ học cách triển khai ứng dụng.

- + Nếu bạn thấy site hữu ích, trước khi rời đi hãy **giúp đỡ** site bằng một hành động nhỏ để site có thể phát triển và phục vụ bạn tốt hơn.
 - + Nếu bạn thấy bài viết hữu ích, hãy giúp **chia sẻ** tới mọi người.
 - + Nếu có thắc mắc hoặc cần trao đổi thêm, mời bạn viết trong phần **thảo luận** cuối trang.
- Cảm ơn bạn!