CSC108

Debugging and Exceptions

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Use the debugger to examine executing code
2. Throw and catch exceptions to respond to errors

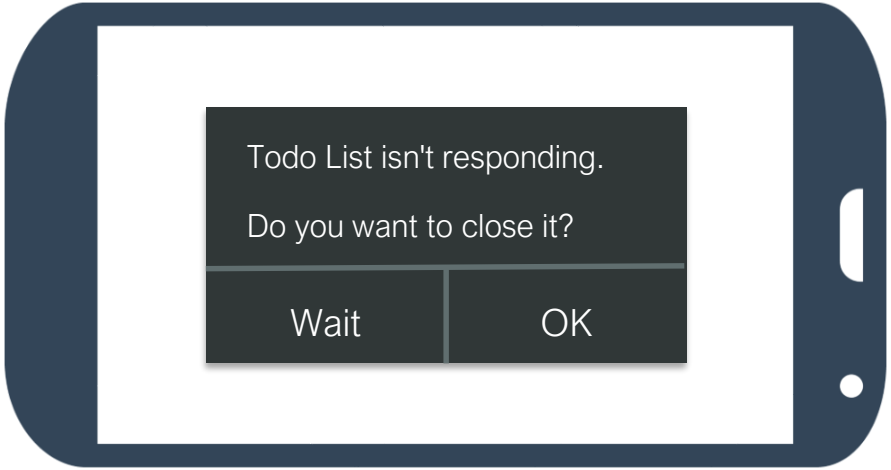
Use the debugger to examine executing code

Tasks

1. Use break points to stop execution
2. Examine values at break points
3. Use Watch Statements
4. Examine the call stack

Motivation

- Users of your application will be frustrated if your application stops responding, crashes or loses their data



What is a “bug”?

- A “bug” is a flaw in an application that causes it to produce unexpected results, behave in unexpected ways or crash

The Debugging Process

- Use the debugging process to correct programs by locating and eliminating programmer mistakes (referred to as "bugs") that stop your program from running as intended

Why Debugging is important

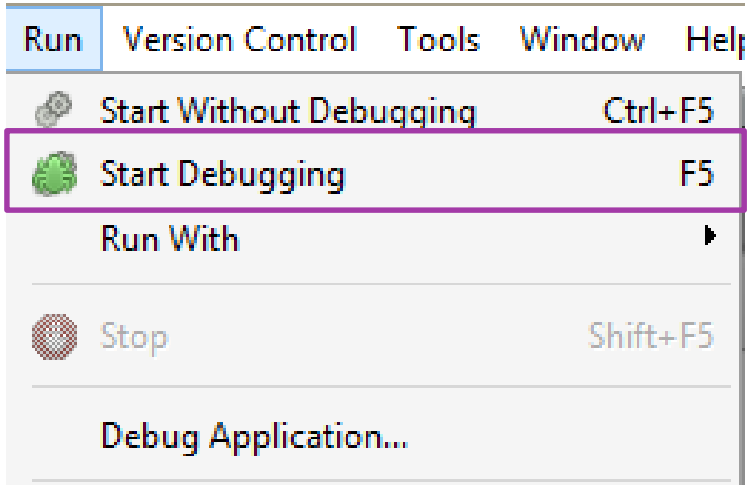
- C# syntax errors are detected by the compiler and are easy to fix, but some bugs, such as logic problems, only occur when the program is running

- These "run-time" bugs are often hard to reproduce and sometimes are even found by the customer

Using the debugger

- The IDE includes an integrated *debugger* which is used to isolate and identify problems that occur at runtime

- Includes a broad set of capabilities to help narrow down where a bug is happening and then isolate it to a specific line of code



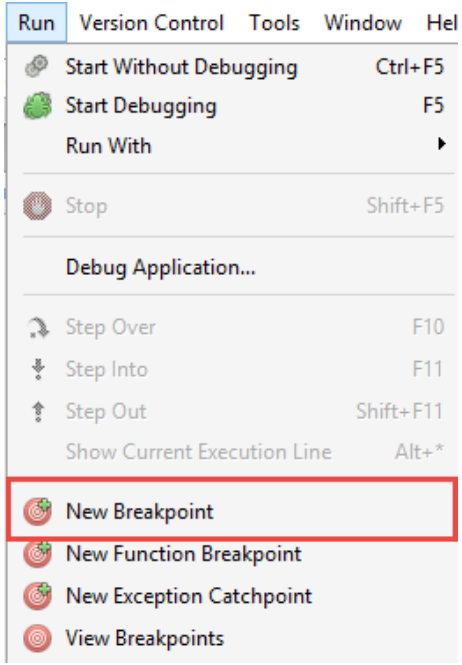
Use breakpoints to stop your program

- Breakpoints signal the debugger to run your program and stop at a specific location, referred to as a *breakpoint*

```
18      {
19          protected override void OnCreate (Bundle bundle)
20          {
21              base.OnCreate (bundle);
22
23              Xamarin.Forms.Forms.Init (this, bundle);
24
25              SetPage (App.GetMainPage ());
26          }
```

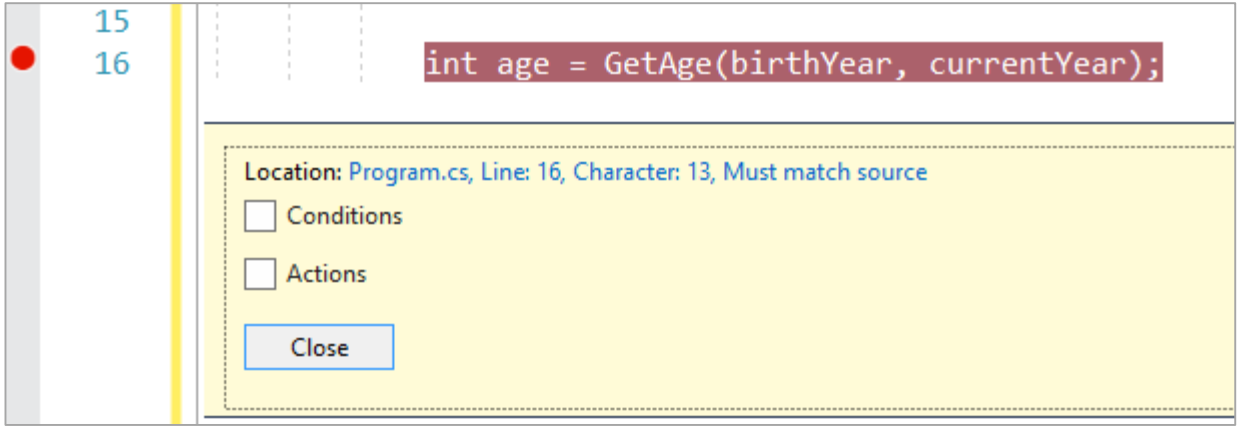
Set a breakpoint by clicking on the gutter of the code window next to the line code you want to stop on

Or in the menu



Creating a Breakpoint

- Visual Studio contains dialogs that allow you to control the behavior of a breakpoint



The locals window

- The locals window shows you all the objects and variables in scope

Look inside collections

Examine local variables

Locals | Breakpoints

Name	Value
emp	{Debugging.Employee}
employees	Count=3
[0]	{Debugging.Employee}
Email	"jesse@falafel.com"
FullName	"Jesse Liberty"
Phone	"339-234-8675"
Non-public members	
[1]	{Debugging.Employee}
[2]	{Debugging.Employee}
Raw View	
isJesse	true

The watch window

- Keep an eye on the value of an object or variable

Watch | Locals | Breakpoints

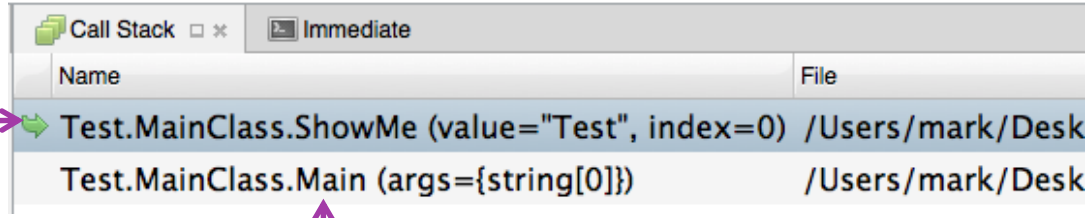
Name	Value	Type
emp.FullName	"Jesse Liberty"	string
emp	{Debugging.Employe	Debugging.Employee
Email	"jesse@falafel.com	string
FullName	"Jesse Liberty"	string
Phone	"339-234-8675"	string
Non-public members		
Click here to add a new watch		

Add new watches by clicking in window or dragging into window

The call stack window

- Use the call stack window to see which method called the current method

Arrow indicates current location



This is the function that called the current one

Along with the functions and procedures, in the call stack window, you can view module name, line numbers, parameter names, types, and values

Group Exercise

Set breakpoints and inspect values

Flash Quiz

Xamarin University

Flash Quiz

① What can you use to stop your program and examine values?

a) Watch window

b) Breakpoint

c) Exceptions

Flash Quiz

① What can you use to stop your program and examine values?

a) Watch window

b) Breakpoint

c) Exceptions

Flash Quiz

② What can you use to evaluate variables and expressions and keep the results?

a) Watch window

b) Breakpoint

c) Exceptions

Flash Quiz

② What can you use to evaluate variables and expressions and keep the results?

a) Watch window

b) Breakpoint

c) Exceptions

Flash Quiz

③ How does the IDE assist you with the debugging process?

a) Detect and correct errors

b) Design better controls

c) Detect exceptions

Flash Quiz

③ How does the IDE assist you with the debugging process?

a) Detect and correct errors

b) Design better controls

c) Detect exceptions

Summary

1. Use break points to stop execution
2. Examine values at break points
3. Use Watch Statements
4. Examine the call stack

Throw and catch exceptions to respond to errors

Xamarin University

Tasks

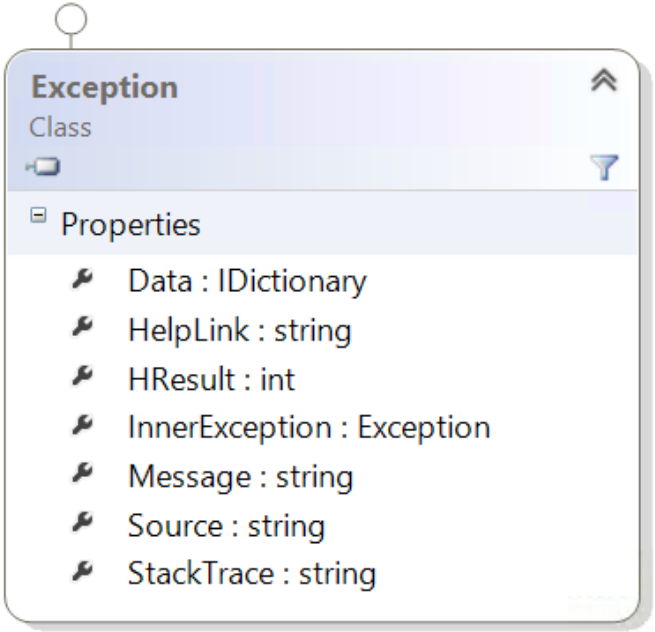
1. Distinguish between Exceptions and Errors
2. Use try / catch statement around potential exceptions

Importance of Exception Handling

- Exceptional conditions in your code often cause the program to terminate prematurely (called "crashes")
- When this happens, C# creates an **`Exception`** object to represent the failure
- C# uses the **`try`**, **`catch`** and **`finally`** keywords to keep these exceptions from ending the program

What is an Exception?

- `Exception` classes are used to represent a runtime exception
  - derived classes provide specialization
- Properties contain information about the error and where it occurred



Common Exception Types

- .NET includes many predefined exception types for common situations that will occur in your code or when your program executes

`ArgumentNullException`	`DivideByZeroException`
`ArgumentOutOfRangeException`	`NullReferenceException`
`IndexOutOfRangeException`	`StackOverflowException`
`InvalidCastException`	`OutOfMemoryException`

There are many more, and you can create your own custom exception types as well

Why do we need exceptions?

- Recall: we use `int.Parse` to transform a string into an integer

```
int result = int.Parse("One");
```

What should the method return?
0? 1? -1?
Is there a valid value for this input?

Why do we need exceptions?

- Recall: we use `int.Parse` to transform a string into an integer

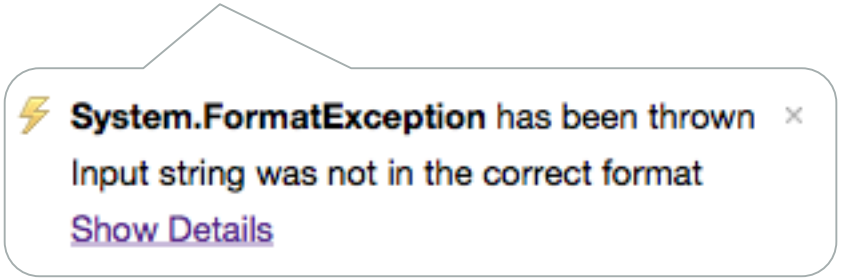
```
int result = int.Parse("One");
```

Instead of returning a value, this code reports an exception indicating the method could not perform the requested operation

Unhandled exceptions

- Exceptions thrown in a method that are *not* caught cause the program to terminate!

```
int result = int.Parse("One");
```



Catching exceptions

- Exceptions **must be caught** in order to *handle* the exceptional case

```
try {
    int result = int.Parse("One");
    // Use result
}
catch (FormatException e) {
   Console.WriteLine("Number text was incorrect.");
}
```

Try block surrounds the code which *might* throw an exception

Catching exceptions

- Exceptions **must be caught** in order to *handle* the exceptional case

```
try {
    int result = int.Parse("One");
    // Use result
}
catch (FormatException e) {
   Console.WriteLine("Number text was incorrect.");
}
```

Catch block indicates the type of exception it can handle

Catching exceptions

- Exceptions **must be caught** in order to *handle* the exceptional case

```
try {
    int result = int.Parse("One");
    // Use result
}
catch (FormatException e) {
   Console.WriteLine("Number text was incorrect.");
}
```

The Catch block executes *only if* code in the try block throws an exception

Catching different exception types

- Can add multiple **catch** handlers to process different exception types

```
try { ... }

catch (ArgumentNullException e) { ... }

catch (DivideByZeroException) { ... }

catch (Exception e) { ... }
```

they are evaluated **in-order**, so make sure to always put the least-specific exception type **last**

Catching different exception types

- Can add multiple `catch` handlers to process different exception types

```
try { ... }

catch (ArgumentNullException e) { ... }

catch (DivideByZeroException) { ... }

catch (Exception e) { ... }
```

no parameter name present

Performing required cleanup

- One additional keyword can be used with **try** to ensure a block of code is executed even if an exception happens – this is the **`finally`** block

```
try {
    result = value / divisor;
}
catch {
   // Shown on any error – like catch (Exception)
   Console.WriteLine("An error occurred during division.");
   result = int.MinValue;
}
finally {
   // This is always shown
   Console.WriteLine("Thanks for playing!");
}
```

Always at the end, and always executed no matter what

Reporting errors

- Exceptions are the *preferred* way to report unexpected or invalid conditions in your code that prevent the method from functioning

- Do *not* use exceptions for normal program flow or for errors which happen frequently such as common input errors

Throwing exceptions

- Exceptions are *thrown* inside a method to describe a problem that would prevent the method from completing successfully

```
public string SayHello (string name)
{
   if (string.IsNullOrEmpty(name))
      throw new ArgumentNullException("name");
   ...
}
```

Method throws an **exception object** with the error details, in this case, the parameter name that is invalid – the method immediately exits

Flash Quiz

Xamarin University

Flash Quiz

① What are the exception handling keywords?

a) Try
b) Catch
c) Throw
d) All the above

Flash Quiz

① What are the exception handling keywords?

a) Try

b) Catch

c) Throw

d) All the above

Flash Quiz

② At what point does the catch block execute?

a) When a try block is encountered
b) Only when the try block throws an exception
c) After the finally block is encountered
d) None of the above

Flash Quiz

② At what point does the catch block execute?

a) When a try block is encountered

b) Only when the try block throws an exception

c) After the finally block is encountered

d) None of the above

Individual Exercise

Throw, catch, and handle an exception

Xamarin University

Summary

1. Distinguish between Exceptions and Errors
2. Use try / catch statement around potential exceptions

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft