

Table of Contents (Detailed)

| | |
|---|-----------|
| Chapter 1: Workflow Foundation Conceptual Overview | 1 |
| What Is Workflow? | 3 |
| Windows Workflow Foundation | 4 |
| Workflows | 5 |
| Activities | 6 |
| Standard Activities | 7 |
| Runtime Services | 8 |
| Workflow Business Scenario | 9 |
| High Level Workflow | 10 |
| Details of While Activity | 11 |
| Structure of the Solution | 12 |
| Orders Folder | 13 |
| Manual Step in the Verification | 14 |
| Main Console Display | 15 |
| Issues Folder | 16 |
| Invoices Folder | 17 |
| Learning Microsoft's WF | 18 |
| Windows Workflow Foundation 3 | 19 |
| Orders Workflow in WF 3 | 20 |
| Windows Workflow Foundation 4 | 21 |
| Windows Workflow Foundation 4.5 | 22 |
| Summary | 23 |
| Chapter 2: Getting Started with WF 4.5 | 25 |
| Workflow Structure | 27 |
| Minimal Workflow Program | 28 |
| Sequence Activity | 29 |
| Visual Studio Workflow Projects | 30 |
| Workflow Designer Demo | 31 |
| Variables | 35 |
| Assign Activity | 36 |
| HelloAssign Workflow | 37 |
| C# and Visual Basic Expressions | 38 |
| Arguments | 39 |
| Argument Example | 40 |
| Lab 2A | 41 |
| Control Flow Activities | 42 |
| While Activity | 43 |
| Lab 2B | 44 |
| Summary | 45 |
| Chapter 3: Primitive and Control Flow Activities | 55 |
| Built-In Primitive Activities | 57 |

| | |
|---|------------|
| InvokeMethod (Static) | 58 |
| Workflow (Static InvokeMethod) | 59 |
| Variables in the Workflow | 60 |
| InvokeMethod (Instance) | 61 |
| Workflow (Instance InvokeMethod) | 62 |
| Invoking .NET Framework Library | 63 |
| Input in Workflows | 64 |
| Prompt Example Workflow | 65 |
| Delay Activity | 66 |
| Lab 3A | 67 |
| Control Flow Activities | 68 |
| Parallel | 69 |
| Parallel Activity Example | 70 |
| If | 71 |
| If ... Else | 72 |
| While | 73 |
| DoWhile | 74 |
| Switch<T> | 76 |
| Switch<T> Example | 77 |
| Lab 3B | 80 |
| Summary | 81 |
| Chapter 4: Custom Activities | 95 |
| Why Custom Activities? | 97 |
| Authoring Custom Activities | 98 |
| Arguments in Custom Activities | 99 |
| Activity Class Hierarchy | 100 |
| CodeActivity | 101 |
| CodeActivityContext | 102 |
| CodeActivity<TResult> | 103 |
| CodeActivity Demo | 104 |
| Lab 4A | 109 |
| Composing Existing Activities | 110 |
| Demo: Composing Activities | 111 |
| Lab 4B | 115 |
| Bonus Example | 116 |
| Summary | 117 |
| Chapter 5: Workflow Hosting | 129 |
| WorkflowInvoker | 131 |
| Specifying Input to a Workflow | 132 |
| Using Dictionaries | 133 |
| Output Argument | 134 |
| Invoking a Generic Activity | 135 |
| Result Output Argument | 136 |
| Reusing a Workflow | 137 |

| | |
|---|------------|
| Workflow Timeout..... | 138 |
| Timeout with Idle Time | 139 |
| Long Computation without Idle Time | 141 |
| Invoking the Long Computation | 143 |
| Long Computation with Timeout..... | 144 |
| Lab 5A | 145 |
| Hosting a Workflow in Windows | 146 |
| Windows Host Code | 147 |
| WorkflowInvoker Instance Methods | 148 |
| WorkflowInvoker Asynchronous Methods..... | 149 |
| Asynchronous Demonstration..... | 150 |
| HelloAsync Code (Invoker)..... | 152 |
| Example Workflow | 154 |
| Effect of Sleep..... | 155 |
| WorkflowApplication | 156 |
| WorkflowApplication Demo | 157 |
| Thread Synchronization..... | 161 |
| Arguments..... | 162 |
| WorkflowApplication Async Demo | 163 |
| HelloAsync Code (Application) | 165 |
| WorkflowApplication Delegates | 166 |
| Manual Control of Workflows..... | 167 |
| Stopping Workflow Execution | 168 |
| Workflow Manual Control Example | 169 |
| Enqueue Workflow | 170 |
| Dequeue Workflow | 171 |
| Host Code for Enqueue Workflow | 172 |
| Host Code for Dequeue Workflow | 173 |
| Lab 5B..... | 174 |
| Hosting a Workflow in ASP.NET | 175 |
| ASP.NET Workflow Host | 176 |
| Lab 5C..... | 177 |
| Summary | 178 |
| Chapter 6: Collection and Parallel Activities..... | 187 |
| Collection Activities | 189 |
| Collection Activities Example | 190 |
| Top-Level Workflow | 191 |
| Process Command Activity..... | 192 |
| Add and Show | 193 |
| Remove and Clear..... | 194 |
| ForEach<T> Activity..... | 195 |
| ParallelForEach<T>..... | 196 |
| Factor Workflow Example..... | 197 |
| Using ForEach<T> | 198 |
| Using ParallelForEach<T> | 199 |

| | |
|--|------------|
| Using AsyncCodeActivity | 200 |
| Example Code | 201 |
| Host Code..... | 202 |
| Asynchronous Activities in WF..... | 203 |
| Factor.cs | 204 |
| FactorNumber.cs | 205 |
| Async Coding: BeginExecute | 206 |
| Async Coding: EndExecute | 207 |
| Lab 6 | 208 |
| Summary | 209 |
| Chapter 7: More about Custom Activities | 215 |
| Waiting for Input..... | 217 |
| Bookmarks | 218 |
| NativeActivity | 219 |
| Bookmark Example | 220 |
| GetTwoInt Custom Activity | 221 |
| Host Program | 222 |
| Passing Data on Resume..... | 223 |
| Bookmark Options..... | 224 |
| Bookmarks and Threads | 225 |
| Threads in Host Code | 226 |
| Threads in Workflow | 227 |
| Sample Threading Output | 228 |
| Lab 7 | 229 |
| A Compute Intensive Workflow | 230 |
| EnqueueLoop | 231 |
| DequeueLoop..... | 232 |
| FactorQueueBookmark Solution | 233 |
| FactorConsoleWF | 234 |
| More Experiments..... | 236 |
| Pick Activity | 237 |
| Pick Example | 238 |
| Get and Check Answer | 239 |
| Set Timer..... | 240 |
| Summary | 241 |
| Chapter 8: Flowchart and State Machine | 251 |
| Workflow Modeling Styles..... | 253 |
| Flowchart Activities..... | 254 |
| Flowchart Activity Designer..... | 255 |
| Demo: Absolute Value Flowchart | 256 |
| Auto-Connect..... | 261 |
| Using FlowDecision | 262 |
| Flowchart and Custom Activities | 265 |
| Lab 8A | 266 |

| | |
|---|------------|
| State Machine Workflows in WF 4.5 | 267 |
| State Machine Workflow Modeling | 268 |
| State Machine Workflow Example | 269 |
| Power On Transition | 270 |
| TransitionCommand Trigger | 271 |
| Warming State | 272 |
| Heated Transition..... | 273 |
| Host Program | 274 |
| State Machine Activity Designer..... | 275 |
| Demo: Timer State Machine..... | 276 |
| Timer in the Math Game..... | 281 |
| Do Problem State | 282 |
| TimeUp Transition..... | 283 |
| Time Out State | 284 |
| Complete Transition | 285 |
| Threading Issue..... | 286 |
| Threading Issue Resolved..... | 288 |
| Shared Trigger | 289 |
| Echo Transition..... | 290 |
| Quit Transition..... | 291 |
| Lab 8B..... | 292 |
| Summary | 293 |
| Chapter 9: Persistence..... | 309 |
| Long Running Workflows | 311 |
| Persistence and Bookmarks | 312 |
| Long-Running Workflow Example | 313 |
| Persistent Term Life Example | 314 |
| Workflow Persistence in WF 4.5 | 315 |
| SQL Server Persistence Database..... | 316 |
| Host Code to Enable Persistence | 317 |
| Persistence Demo..... | 318 |
| AutoResetEvent | 322 |
| PersistableIdle Event..... | 324 |
| How to Persist a Workflow..... | 326 |
| Loading a Persisted Workflow..... | 327 |
| Unload and Load Example..... | 328 |
| StartAndUnloadInstance() | 329 |
| LoadAndCompleteInstance()..... | 330 |
| Stopping and Starting the Host | 331 |
| StartWorkflow..... | 332 |
| StartWorkflow..... | 333 |
| Loading After Data Available..... | 334 |
| InitNames..... | 335 |
| GetString..... | 336 |
| String Commands..... | 337 |

| | |
|--|------------|
| Host Program | 339 |
| Lab 9 | 340 |
| Summary | 341 |
| Chapter 10: Workflow Services..... | 347 |
| What Is WCF?..... | 349 |
| WCF Services | 350 |
| WCF = ABC | 351 |
| Address, Binding, Contract..... | 352 |
| Workflow Services..... | 353 |
| Messaging Activities..... | 354 |
| Messaging Activity Templates | 355 |
| Demo – Creating a Workflow Service..... | 356 |
| WCF Test Client | 360 |
| Demo – Workflow Services Client | 362 |
| Multiple Operations | 365 |
| Multiple Operations via Parallel | 366 |
| Hosting a Workflow Service..... | 367 |
| Configuration File..... | 368 |
| WorkflowServiceHost Example | 369 |
| Program.cs | 370 |
| Startup Configuration | 372 |
| Lab 10 | 373 |
| Summary | 374 |
| Chapter 11: Debugging and Error Handling..... | 385 |
| Debugging Workflows..... | 387 |
| Control Flow and Flowchart | 388 |
| Breakpoint Example..... | 389 |
| Breakpoint in XAML..... | 390 |
| Exceptions..... | 391 |
| Exception Demonstration | 392 |
| Account and Bank Classes..... | 393 |
| Use of Dictionary..... | 394 |
| Deposit and Withdraw | 395 |
| Code Activities | 396 |
| Composite Activities..... | 397 |
| Top-Level Workflow | 398 |
| Host Program | 399 |
| Unhandled Exceptions | 400 |
| Using WorkflowApplication..... | 401 |
| Sample Output | 402 |
| TryCatch Activity | 403 |
| TryCatch Demo..... | 404 |
| Transactions | 409 |
| Compensation | 410 |

| | |
|--|------------|
| No Compensation..... | 411 |
| Using Compensation..... | 412 |
| Transfer.xaml..... | 413 |
| Compensable Withdrawal..... | 414 |
| Compensation Token..... | 415 |
| Canceling the Workflow..... | 416 |
| Exceptions and Compensation..... | 417 |
| Compensation Not Performed..... | 418 |
| Lab 11..... | 419 |
| Summary..... | 420 |
| Appendix A: Learning Resources..... | 429 |

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

Chapter 1

Workflow Foundation Conceptual Overview

Workflow Foundation Conceptual Overview

Objectives

After completing this unit you will be able to:

- **Explain what a workflow is and how Windows Workflow Foundation supports workflow applications.**
- **Describe a typical business scenario for workflow and illustrate with a WF application.**
- **Explain the concepts of workflows and activities.**
- **Describe runtime services provided in WF.**
- **Discuss the differences between WF 3 and WF 4/4.5.**

What Is Workflow?

- **In general terms, a workflow can be thought of as a flow of processes or tasks that produce some result.**
- **Workflows are often concerned with documents that flow through various activities and may spawn other documents as they are processed.**
- **Workflows can be manual, with paper documents being transmitted among people working at different desks in an office, each person performing defined tasks according to specified rules.**
- **We are concerned with workflows as software systems that define the flow of work, the activities performed, and the rules that are employed.**
 - Rules can be expressed declaratively or in code.
 - Activities may be entirely automated or may involve human interaction.
 - Workflows may be distributed among multiple computers in diverse locations.
 - Workflows are typically represented in a graphical manner.

Windows Workflow Foundation

- **Windows Workflow Foundation (WF) is a framework that supports creating and running workflow applications on Windows platforms.**
 - WF consists of a programming model, an engine, and tools.
 - The tools include designers for Visual Studio.
- **WF provides a consistent development experience with other .NET 4.x technologies, including WCF and WPF.**
- **The WF API contains support for both C# and Visual Basic, a special workflow compiler, workflow debugging support, and a visual workflow designer.**
- **Workflows can be developed completely in code or created in conjunction with XAML markup.**
- **The WF model and designer are extensible, enabling developers to create custom activities that encapsulate particular workflow functionality.**

Workflows

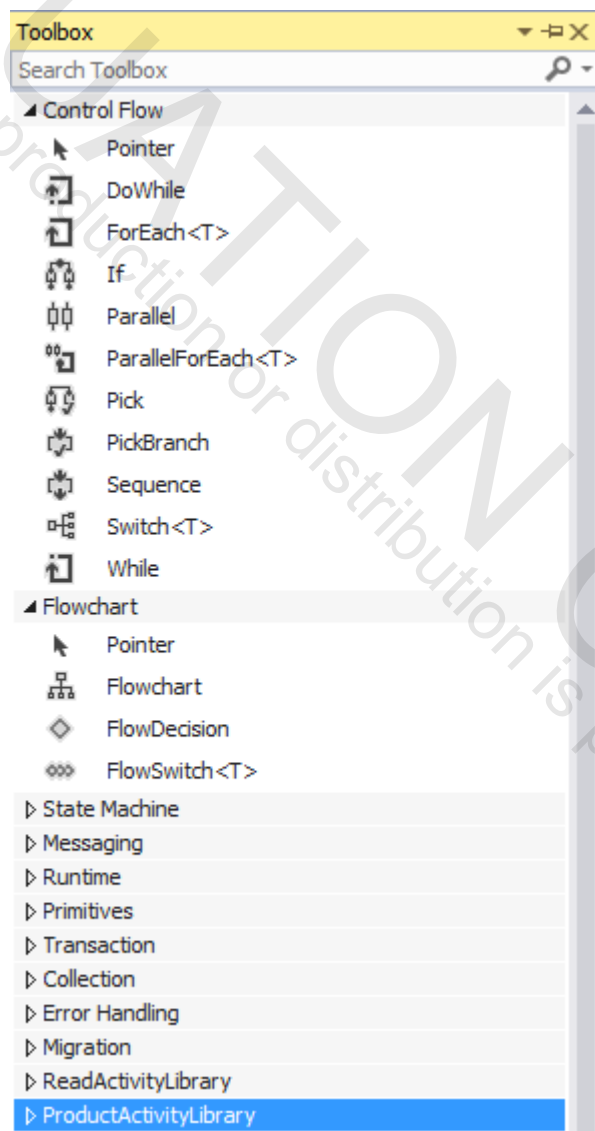
- **A *workflow* is a set of *activities* that are stored in a model describing a business (or other real-world) process.**
- **A workflow describes the order of execution and relationships between units of work.**
 - The units of work may run for a short time or a long time.
 - Activities may be performed by people or the computer.
- **A workflow instance is created and maintained by the *workflow runtime engine*.**
 - There can be several workflow engines within an application domain.
 - Each instance of the engine can support multiple workflow instances.
- **A compiled workflow model can be hosted inside any Windows process, including a console application, a Windows Forms application, a WPF application, a Windows service, an ASP.NET Web application, and a Web service.**

Activities

- **The units of work of a workflow are called *activities*.**
- **When a workflow instance starts, activities are executed in an order as defined by the workflow model.**
 - Both parallel and sequential orders of execution are supported.
 - Conditional and looping behavior of activities is supported.
 - Execution continues until the last activity completes, and the workflow then terminates.
- **Activities can be reused within a workflow and in other workflows.**
- **Activities usually have properties that are configurable.**
- **WF provides many standard activities out-of-the box, and custom activities can be created.**

Standard Activities

- **The Visual Studio Toolbox contains many standard activities that can be dragged onto the surface of the Workflow Designer.**
 - These are arranged in groups, such as Control Flow, Flowchart, State Machine, Messaging, and so on.
 - Custom activities can be provided in activity libraries.



Runtime Services

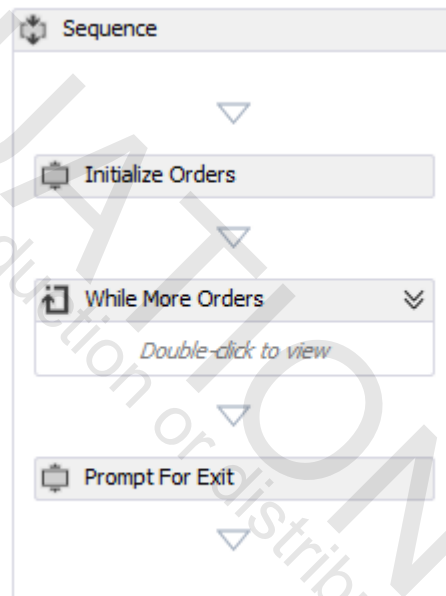
- **WF provides a number of out-of-the box *runtime services* that are available in the workflow engine.**
- ***Persistence services* enable a developer to easily save a WF instance to external storage, such as a database or XML file.**
 - This capability enables workflow applications to maintain state and be long-running, surviving application restarts.
- ***Transaction services* enable you to maintain transactional integrity in workflow applications.**
- ***Tracking services* support monitoring and recording workflow execution.**
- ***Scheduling services* enable you to control how the WF runtime manages threads in your application.**

Workflow Business Scenario

- **To illustrate workflows consider the following business scenario:**
 - As orders are created they are specified in XML files placed in the folder Orders.
 - Each order in the Orders folder is processed, beginning with getting the order information from the XML file.
 - The order is verified, which is done by a person who checks each item in the order for consistency (description furnished by customer matches the description in the vendor database or has a trivial error such as a misspelling). The system also does some verification, making sure that the item ID is found in the database.
 - If the order is valid, it is processed and an invoice is created. The invoice is specified in an XML, stored in the Invoices folder.
 - If the order is not valid, the issue with the order is specified in another XML file, stored in the Issues folder. (A customer service representative can follow up on such orders by emailing or calling the customer.)
- **This scenario is implemented by a workflow application.**
 - See **Chap01\OrderWorkflow**. See the file **Workflow1.xml** in the **OrderWorkflow** project for a diagram of the workflow.

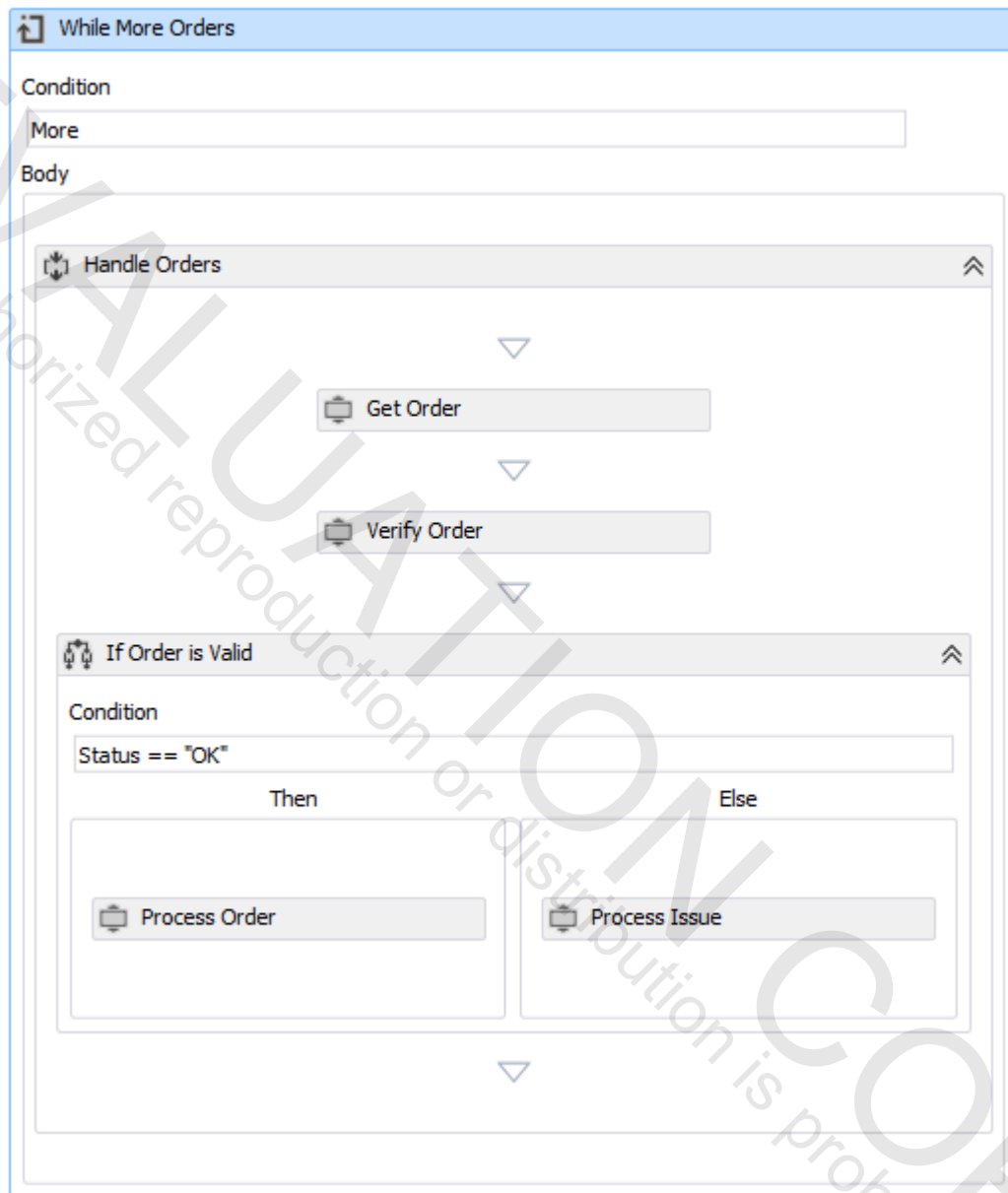
High Level Workflow

- Open up the file *Workflow1.xaml*.
 - The Workflow Designer opens up, showing a diagram of the workflow¹.
 - Examine the workflow at a high level by collapsing the **While More Orders** activity.



¹ You must build the solution before the **Workflow1.xaml** file will load properly in the visual designer.

Details of While Activity



- **The heart of the workflow is a loop that gets and verifies orders.**
 - If order is valid, it is processed.
 - If order is not valid, an issue is processed.

Structure of the Solution

- **The solution consists of three projects:**
 - **OrderLibrary** is an activity library that defines custom activities such as **InitializeOrders**, **GetOrder**, and so on.
 - **OrLib** is an ordinary class library defining the classes that are used in the implementation of the custom activities. This includes code to access a database.
 - **OrderWorkflow** is a console application that contains the workflow itself. It has a graphical representation specified in the XAML file **Workflow1.xaml**.
- **Three folders are provided for XML data files:**
 - The **Orders** folder contains orders to be processed.
 - The **Invoices** folder contains invoices that are created for valid orders.
 - The **Issues** folder contains files describing issues for orders that are not valid.
- **A database of products is provided in the SQL Server database file *Product.mdf* in the *C:\OIC\Data* folder.**
 - The database uses the LocalDB version of SQL Server 2012, which is automatically installed with Visual Studio 2013.
 - The database access code is encapsulated within the **OrLib** class library.

Orders Folder

- The *Orders* folder contains XML files representing orders. The loop goes through all the files in this folder.

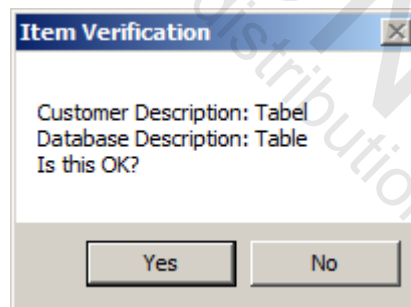
- As an example, consider **1002.xml**. This order is basically valid, but one of the items is slightly questionable.

```
<?xml version="1.0" encoding="utf-8"
standalone="yes"?>
<Order>
  <OrderId>1002</OrderId>
  <Customer>
    <Name>Mary Smith</Name>
    <Email>mary@bar.com</Email>
  </Customer>
  <Item>
    <ItemId>104</ItemId>
    <Description>Sofa</Description>
    <Quantity>1</Quantity>
  </Item>
  <Item>
    <ItemId>102</ItemId>
    <Description>Tabel</Description>
    <Quantity>2</Quantity>
  </Item>
  <Item>
    <ItemId>103</ItemId>
    <Description>Lamp</Description>
    <Quantity>2</Quantity>
  </Item>
</Order>
```

- An automated verification might reject item 102 because of the misspelling of the description.

Manual Step in the Verification

- As is typical in workflow applications, steps in the workflow can be carried out either by the computer or a person.
- As part of the processing of the *VerifyOrder* activity a message box will be displayed for a human to verify each item.
- Build and run the solution.
 - **OrderWorkflow** is the startup project, so this console application will run.
 - As orders are verified, a message box will pop up for each item of each order.
- Here is the message box for item 102 of order 1002.



- The person viewing this will clearly see that this is benign and will click Yes to approve this item.

Main Console Display

- As the workflow is executed, a running report on each item is displayed on a main console.
- Here is the display for order 1002:

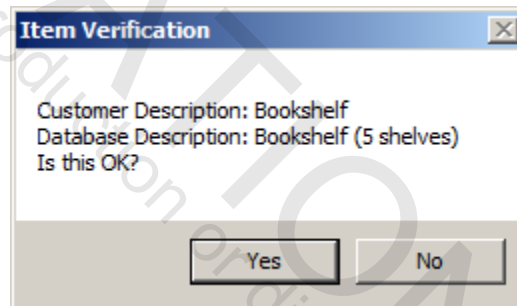
```
C:\OIC\WfCs\Chap01\OrderWorkflow\Orders\1002.xml
OrderId = 1002
Verification message = OK
OrderId = 1002
CustomerName = Mary Smith
CustomerEmail = mary@bar.com
ItemId   Description   Quantity   ...   Price   Extension
104      Sofa          1          ...   $500.00 $500.00
102      Tabel          2          ...   $200.00 $400.00
103      Lamp           2          ...   $50.00  $100.00
Total = $1,000.00
File 1002.xml has been created in folder Invoices
```

- And here is the display for orders 1003 and 1004 for which there were genuine issues.

```
file = 1003.xml
C:\OIC\WfCs\Chap01\OrderWorkflow\Orders\1003.xml
OrderId = 1003
Verification message = Description does not match
Create issue: Description does not match
File 1003.xml has been created in folder Issues
file = 1004.xml
C:\OIC\WfCs\Chap01\OrderWorkflow\Orders\1004.xml
OrderId = 1003
Verification message = Item Not Found
Create issue: Item Not Found
File 1004.xml has been created in folder Issues
Press Enter to exit
```

Issues Folder

- The *Issues* folder will contain an XML file for each order having genuine issues.
- Our example illustrates two kinds of problems.
 - The item is not found in the database, a problem that can be detected automatically, illustrated in order 1004.
 - The item's description indicates a genuine question, such as an ambiguity that should be verified with the customer, illustrated in order 1003.



- The human verifier will reject this item, because the company has several bookshelves having different numbers of shelves, and the customer should be queried to make sure she gets the desired bookshelf.
- Here is the generated issue file **1003.xml**:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Issue>
  <OrderId>1003</OrderId>
  <CustomerName>Bill Jones</CustomerName>
  <CustomerEmail>bill@forest.net</CustomerEmail>
  <Message>Description does not match</Message>
</Issue>
```


Invoices Folder

- **For valid orders an XML file is created from which an invoice can be generated.**
 - The actual invoice that will be sent to the printer is created by a separate subsystem, which could be another workflow, whose input is the files in the **Invoices** folder.
- **As an example, here is the invoice XML file for order 1002.**

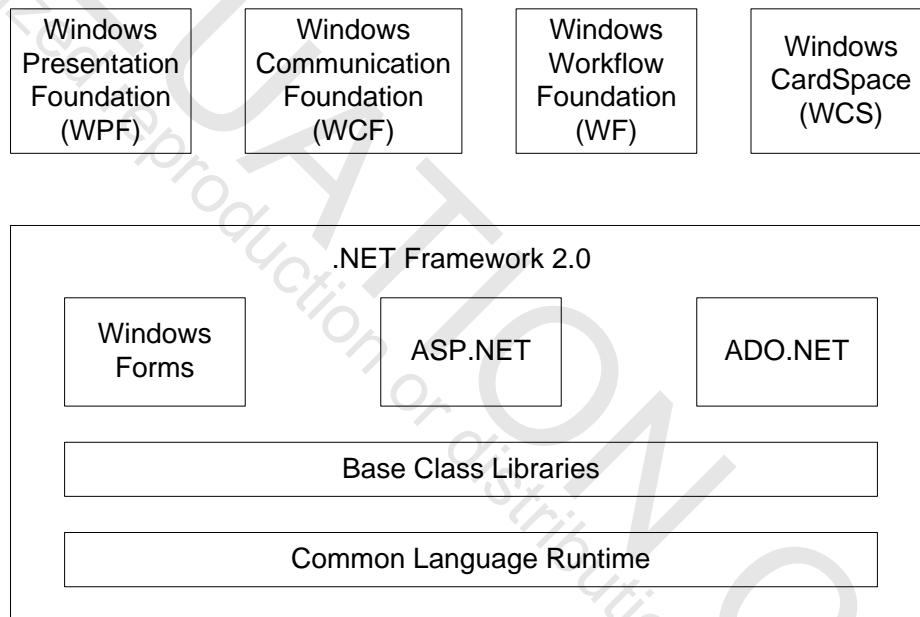
```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Invoice>
  <OrderId>1002</OrderId>
  <CustomerName>Mary Smith</CustomerName>
  <CustomerEmail>mary@bar.com</CustomerEmail>
  <LineItem>
    <ItemId>104</ItemId>
    <Description>Sofa</Description>
    <Quantity>1</Quantity>
    <Price>500.0000</Price>
    <Extension>500.0000</Extension>
  </LineItem>
  <LineItem>
    <ItemId>102</ItemId>
    <Description>Table</Description>
    <Quantity>2</Quantity>
    <Price>200.0000</Price>
    <Extension>400.0000</Extension>
  </LineItem>
  <LineItem>
    <ItemId>103</ItemId>
    <Description>Lamp</Description>
    <Quantity>2</Quantity>
    <Price>50.0000</Price>
    <Extension>100.0000</Extension>
  </LineItem>
  <Total>1000.0000</Total>
</Invoice>
```

Learning Microsoft's WF

- **There are two main challenges in learning to use Windows Workflow Foundation.**
- **The first challenge is to understand the nature of workflow applications, which are quite different in structure from conventional applications.**
 - For this purpose studying a miniature example illustrating an actual business scenario is invaluable.
 - The program **OrderWorkflow** is such an example. (By contrast, a conventional console application is provided in **OrderConsole** in the chapter folder.)
- **The second challenge is to understand the classes and tools in the actual framework.**
 - For this purpose studying small standalone examples, divorced from the complexities of a business situation, will be most helpful.
 - We'll largely follow this second approach in the remainder of the course, beginning with a simple "Hello Workflow" example in Chapter 2.

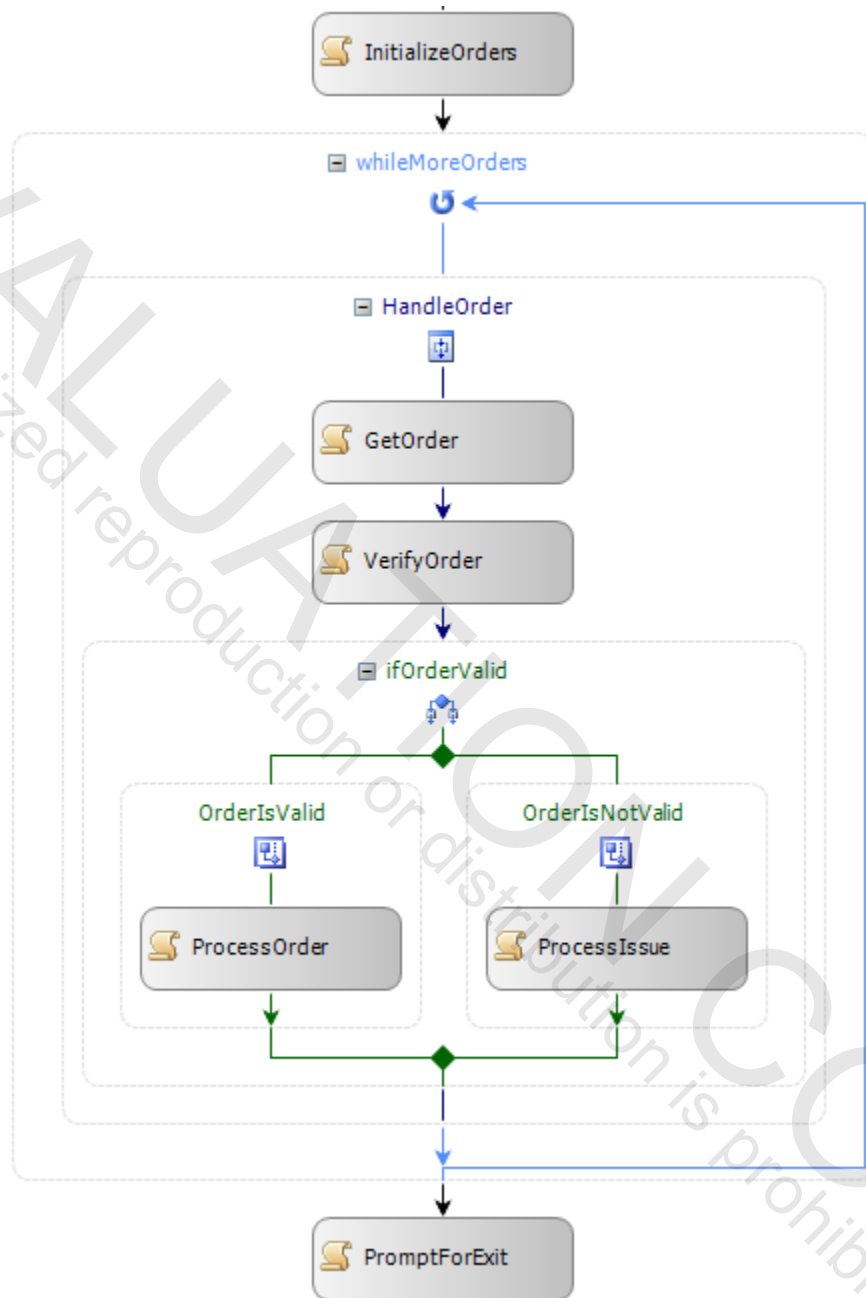
Windows Workflow Foundation 3

- **Windows Workflow Foundation was introduced as part of .NET Framework 3.0 (formerly called WinFX).**
- **.NET Framework 3.0 layers on top of .NET Framework 2.0 and has the components shown in the diagram.**



- **.NET Framework 3.5 added some important new features, notably Language Integrated Query or LINQ, but remained layered on top of .NET 2.0.**
 - WF 3.5 used the same Workflow programming model as WF 3.0.
 - **OrderWF3** illustrates the same order processing workflow implemented using WF 3.5. The graphical representation of the workflow looks different, as can be seen on the next page.

Orders Workflow in WF 3



Windows Workflow Foundation 4

- **.NET 4.0 is a major revision of the .NET Framework, and the WinFX technologies are no longer layered on top of .NET 3.0.**
- **Windows Workflow Foundation in .NET 4.0 is a complete re-architecting of WF.**
 - WF 4 has a completely new set of assemblies: **System.Activities.*** that are used in place of the **System.Workflow.*** assemblies from WF 3.
 - But WF 4 retains the **System.Workflow.*** assemblies, so WF 3 workflows will run unchanged in WF 4 (as illustrated by the **OrderWF3** example).
- **Major changes in WF 4 include:**
 - A new visual designer, built with WPF, supports the ability to work with much larger workflows.
 - Data flows, which were opaque in WF 3, can now be clearly specified as variables and arguments with familiar typing and scoping.
 - A new Flowchart activity is provided.
 - The programming model has been revamped, making Activity a core base type used for both workflows and activities. The model is now fully declarative, expressible in XAML, with no code-beside.
 - Integration with WCF has been improved, with new messaging activities, and fully declarative service definition.

Windows Workflow Foundation 4.5

- **Windows Workflow Foundation in .NET 4.5/4.5.1 and Visual Studio 2013 contains new activities, designer capabilities, and a new workflow development model.**
 - Some of this capability was available in the Service Pack and Platform Update to Visual Studio 2010.
- **A major enhancement for C# programmers is C# expressions.**
 - Prior to .NET 4.5 expressions in workflows had to be written in Visual Basic.
- **.NET 4.5 provides for a state machine development model out-of-the box.**
 - WF 3 supported state machines but WF 4 did not. The state machine development model was provided in the Platform Update to Visual Studio 2010.
- **There are a number of new designer capabilities:**
 - Auto-surround with Sequence: Dragging a second activity into a block expecting a single activity will cause the designer to automatically insert a Sequence.
 - There is an auto-connect feature in flowcharts.
 - Breakpoints can be set on states in state machines.
- **Consult MSDN documentation for other enhancements in WF 4.5.**

Summary

- **A workflow can be thought of as a flow of processes or tasks that produce some result.**
- **Windows Workflow Foundation (WF) is a framework that supports creating and running workflow applications on Windows platforms.**
- **The units of work of a workflow are called *activities*.**
- **Workflows can be defined using a visual designer.**
- **Standard activities are provided in a Toolbox, and custom activities can be created in an activity library, also made available in the Toolbox.**
- **Workflow Foundation 4 is a complete re-architecting of WF.**
- **WF 4.5 contains new activities, designer capabilities, and a new state machine development model. C# expressions are supported in workflows.**

Chapter 9

Persistence

Persistence

Objectives

After completing this unit you will be able to:

- **Explain the need of a persistence mechanism in long running workflows.**
- **Describe the role of bookmarks in persistence.**
- **Describe how to set up a persistence store in SQL Server.**
- **Implement a workflow that persists its state.**
- **Implement a workflow that persists its state, unloads, and then loads its state from the persistence store.**
- **Show how to serialize workflow instance identification so that the host program as well as the workflow can unload and start up again.**
- **Implement a long running workflow that can be unloaded for days or weeks and then resume when needed data is available.**

Long Running Workflows

- **A characteristic of some workflow applications is that they are inherently long running.**
- **For example, some steps in the workflow may require human interaction.**
 - Such a step involving human operation may take days or even weeks.
- **To deal with the requirement of long running workflows, Workflow Foundation provides a *persistence* mechanism.**
- **The state of the workflow may be saved to persistent storage, such as a SQL Server database.**
- **After the state of the workflow has been saved, the workflow may be unloaded.**
- **When needed data becomes available, the workflow may be loaded again and its state restored.**
 - The needed data can be supplied, and the workflow resumes.

Persistence and Bookmarks

- **Conceptually, the persistence scenario is similar to the use of bookmarks.**
 - A workflow may go idle when it needs data.
 - To enable resumption from the point where it left off, the workflow creates a bookmark.
 - When the data becomes available the bookmark may be resumed.
- **But in this scenario the workflow is only idle, not unloaded.**
 - The workflow continues to reside in memory.
- **This scenario is not suitable for truly long-running workflows.**
 - For these we need the persistence mechanism.
- **But the persistence mechanism typically makes use of bookmarks in its implementation.**
 - A bookmark marks a place where a persisted workflow can be resumed.

Long-Running Workflow Example

- **To illustrate a long-running workflow, consider a life insurance workflow application.**
- **Issuing a life insurance policy involves several aspects, including:**
 - The premium must be calculated based on actuarial tables.
 - Risk must be evaluated in an underwriting process.
- **The first step can be completely automated, but the underwriting step involves human interaction and possibly a medical examination.**
- **The workflow can begin with automated steps, but must pause when the underwriting report is needed.**
 - The underwriting process may take days or weeks.
 - During this time the workflow instance should be unloaded, which requires persistence.
- **See *PersistentTermLife\Step2* in the chapter folder.**
 - The instructor will demonstrate this program.
 - Before running the program on your own computer, you will need to set up a persistence store in SQL Server. This procedure is discussed and demonstrated later in the chapter.

Persistent Term Life Example

- **Build and run the program.**

- First some data (age and amount) are entered, which can be done immediately from the application.
- But before insurance can be offered, the underwriting step is needed, which can take days or weeks.
- The program unloads at this point.

Name: John Smith
Age: 40
Amount: 150000
Workflow has unloaded

- Next an underwriting decision is entered when the underwriter's report has become available.

Underwriting Decision
approve, deny, or rating:
approve

- Now the workflow can be reloaded and resumed, and a proposal is created.

Term Life Proposal
Insured: John Smith
Age: 40
Death Benefit: \$150,000.00
Monthly Premium: \$18.30
Workflow has completed
Workflow has unloaded

Workflow Persistence in WF 4.5

- **Workflow persistence is the capture of the state of a workflow instance in a durable storage medium.**
 - This enables unloading workflow instances that are not actively performing work, optimizing memory.
 - It also provides a point of recovery in the event of system failure.
- **The persistence mechanism involves:**
 - Identification of a persistence point
 - Gathering of state data to be saved
 - Delegation of the actual storage of the data to a **persistence provider**.
- **.NET 4.0 provides a persistence provider based on SQL Server, or you can create your own.**
 - The **SqlWorkflowInstanceStore** class enables metadata and state information about a workflow instance to be saved to a SQL Server 2005 or 2008 or 2012 database¹.
- **The examples in this chapter assume you have SQL Server 2012 Express installed.**

¹ At the time of this writing SQL Server 2014 has been released, but we have not yet tested against it. So for this course you should use SQL Server 2012 Express.

SQL Server Persistence Database

- **Before using the persistence API, you must create a database for storing persistent data from workflow instances.**
- **.NET 4.0 provides two SQL scripts to initialize a fresh database for this purpose, which should be run in this order:**
 - `SqlWorkflowInstanceStoreSchema.sql`
 - `SqlWorkflowInstanceStoreLogic.sql`
- **These scripts are located in the windows directory here:**

`\Microsoft.NET\Framework\v4.xxx\SQL\EN`

- **When no longer needed, you can remove the database using SQL Server Management Studio.**
- **Alternatively, the MSDN samples provide two command files (which should be run from a Visual Studio command prompt) for creating and removing the database:**
 - `CreateInstanceStore.cmd`
 - `RemoveInstanceStore.cmd`
- **The course labs provides a copy of these files here:**
 - `\OIC\Data`

Host Code to Enable Persistence

1. Add references to these DLLs:

System.Activities.DurableInstancing.dll

System.Runtime.DurableInstancing.dll

2. Import this namespace: **System.Activities.DurableInstancing**.
3. Provide code such as this to initialize an instance store and assign it to the **InstanceStore** member of **WorkflowApplication**.

```
const string connectionString =  
    "Server=.\SQLEXPRESS;Initial Catalog=  
    SampleInstanceStore;Integrated Security=SSPI";  
SqlWorkflowInstanceStore store =  
    new SqlWorkflowInstanceStore(connectionString);  
wfApp.InstanceStore = store;
```

- The command file **CreateInstanceStore.cmd** created a database whose name is **SampleInstanceStore**.

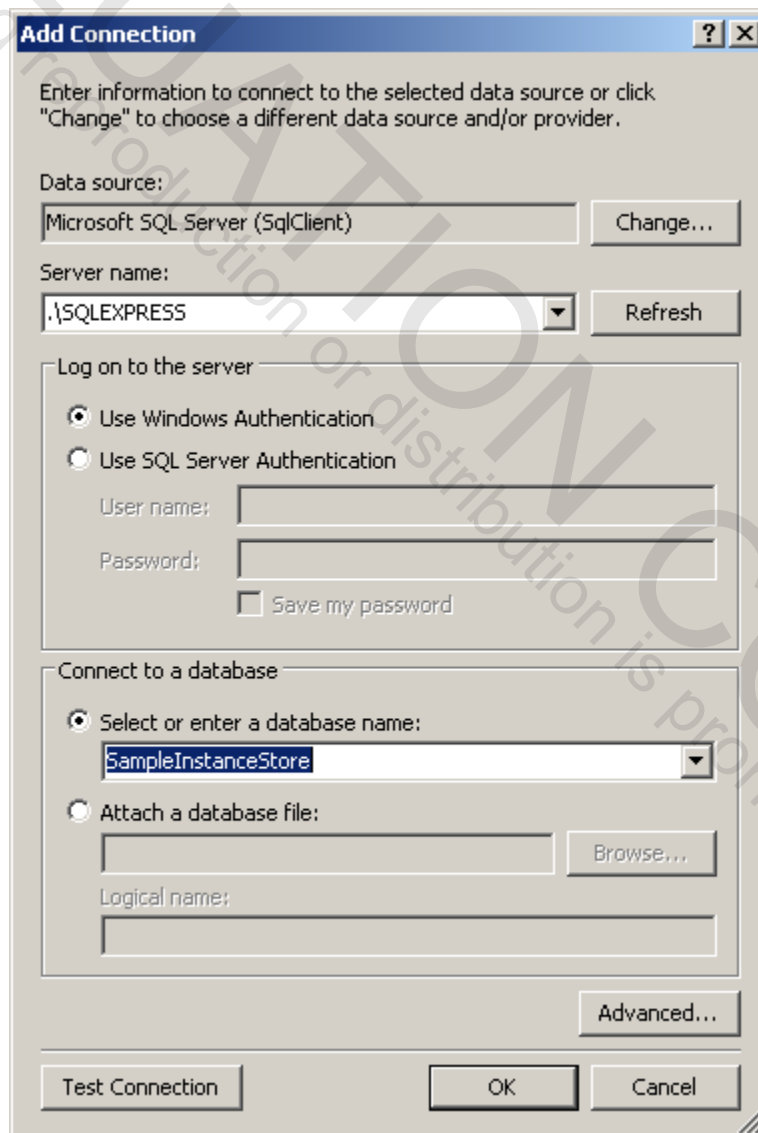
4. Typically you will handle the **PersistableIdle** event of the **WorkflowApplication** object and return either the **Persist** or **Unload** member of the **PersistableIdleAction** enumeration.

```
wfApp.PersistableIdle = delegate(  
    WorkflowApplicationIdleEventArgs e)  
{  
    return PersistableIdleAction.Persist;  
};
```

- **Persist** will persist the workflow, and **Unload** will persist and then unload the workflow.

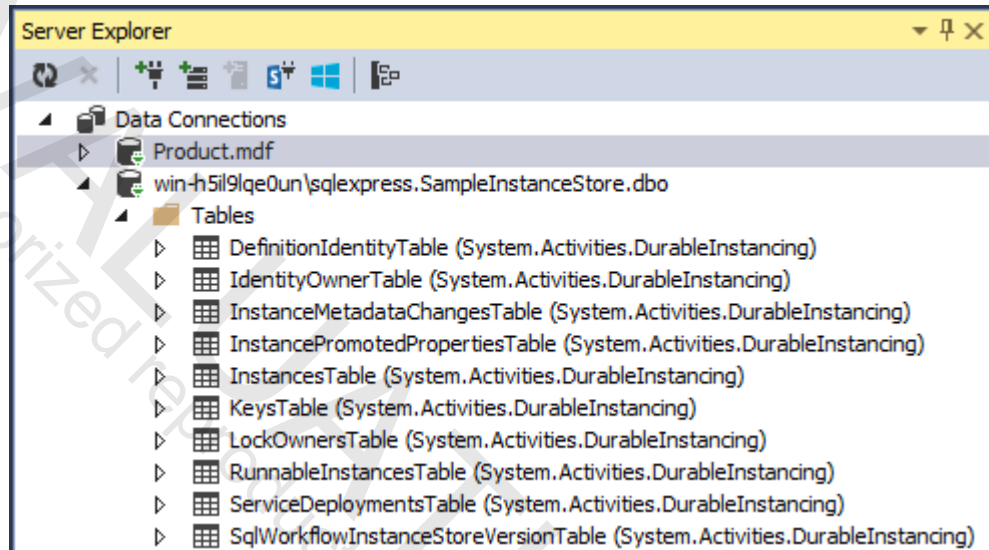
Persistence Demo

- **This demonstration sets up a persistence instance store and persists a simple workflow.**
1. Open up a Visual Studio command prompt and navigate to the folder **C:\OIC\Data**. Run this command file:
`CreateInstanceStore.cmd`
 2. In Server Explorer set up a connection to **SampleInstanceStore**.



Persistence Demo (Cont'd)

- Open up the new database connection in Server Explorer and examine the tables of the database.



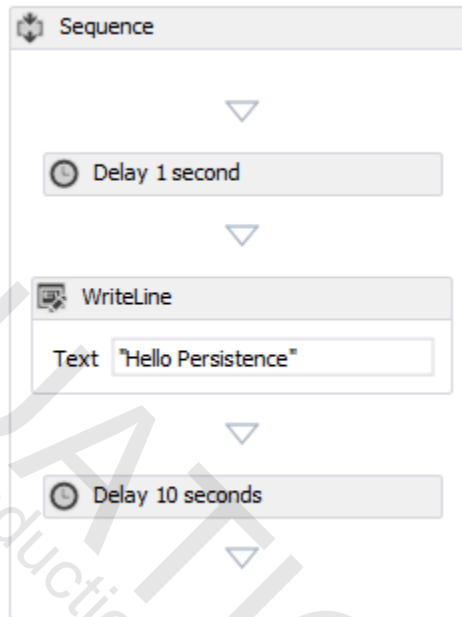
- Right-click over InstancesTable and select Show Table Data.

| System.Activities.Du...InstancesTable [Data] | | | | | | | |
|--|------|-------------------|------------------|--------------------|------------------|--------------------|------------------|
| | Id | SurrogateInsta... | SurrogateLock... | PrimitiveDataPr... | ComplexDataPr... | WriteOnlyPrimit... | WriteOnlyComp... |
| ▶* | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

- No workflow instances are currently stored. Close this window.
- Open up the solution **SimplePersist** in the **Demos** folder. It is backed up in **SimplePersist\Step0** in the chapter folder.

Persistence Demo (Cont'd)

7. There is a simple workflow in **Workflow1.xaml** in a Sequence.



8. Build and run (without debugging). After a brief pause, a message is displayed. After a longer pause, the workflow completes and unloads.

```
Hello Persistence
Workflow has completed
Workflow has unloaded
Press any key to continue . . .
```

9. Examine the host code in **Program.cs**. It uses the **WorkflowApplication** class, because we need it for implementing persistence. We've temporarily commented out some code pertaining to persistence.

Persistence Demo (Cont'd)

```
Workflow1 wf = new Workflow1();
WorkflowApplication wfApp =
    new WorkflowApplication(wf);

AutoResetEvent syncCompleted =
    new AutoResetEvent(false);
AutoResetEvent syncUnloaded =
    new AutoResetEvent(false);

wfApp.Completed = delegate(
    WorkflowApplicationCompletedEventArgs e)
{
    Console.WriteLine("Workflow has completed");
    syncCompleted.Set();
};
wfApp.Unloaded = delegate(
    WorkflowApplicationEventArgs e)
{
    Console.WriteLine("Workflow has unloaded");
    syncUnloaded.Set();
};
wfApp.PersistableIdle = delegate(
    WorkflowApplicationIdleEventArgs e)
{
    Console.WriteLine("Workflow is idle");
    return PersistableIdleAction.None;
};

//const string connectionString =
//    "Server=.\SQLEXPRESS;Initial Catalog=...
//SqlWorkflowInstanceStore store = ...
//wfApp.InstanceStore = store;

wfApp.Run();

syncCompleted.WaitOne();
syncUnloaded.WaitOne();
```

AutoResetEvent

- This code uses a *AutoResetEvent* for the Unloaded event.

```
AutoResetEvent syncUnloaded =  
    new AutoResetEvent(false);
```

- The purpose is to ensure that we will see both messages pertaining to workflow completion and unloading.

- The **AutoResetEvent** is signaled by the event handler.

```
wfApp.Completed = delegate(  
    WorkflowApplicationCompletedEventArgs e)  
{  
    Console.WriteLine("Workflow has completed");  
};  
wfApp.Unloaded = delegate(  
    WorkflowApplicationEventArgs e)  
{  
    Console.WriteLine("Workflow has unloaded");  
    syncUnloaded.Set();  
};
```

- To ensure that both messages are shown, we wait for the Unloaded **AutoResetEvent** to be signaled.

```
syncUnloaded.WaitOne();
```

- If instead we had waited for workflow completion, the host thread may finish before the workflow has been unloaded.

Persistence Demo (Cont'd)

10. Uncomment the instance store setup code and try to build. You will get compile errors.
11. Add references to **System.Activities.DurableInstancing** and to **System.Runtime.DurableInstancing**.
12. Import namespace **System.Activities.DurableInstancing**.
13. Build and run. You should get a clean compile. The output now also shows a message for each time the workflow goes idle from the Delay activities.

Workflow is idle

Hello Persistence

Workflow is idle

Workflow has completed

Workflow has unloaded

Press any key to continue . . .

14. Examine the instance store in Server Explorer. Again, there will be no entries in the **InstancesTable**. We have not yet actually persisted a workflow instance. Close the window showing the table.

PersistableIdle Event

- When persistence is enable, every time the workflow goes idle, both the *Idle* and the *PersistableIdle* events are raised.
 - The event handler should return a member of the **PersistableIdleAction** enumeration. There are three possible values.

| | |
|---------|--|
| None | No action is taken. |
| Persist | The workflow application should persist the workflow. |
| Unload | The workflow application should persist the workflow and then unload the workflow. |

15. The starter code used **None**. Change this to **Persist**.


```
wfApp.PersistableIdle = delegate(  
    WorkflowApplicationIdleEventArgs e)  
{  
    Console.WriteLine("Workflow is idle");  
    return PersistableIdleAction.Persist;  
};
```

16. Build and run. You should see the same output as before. Examine the instance store. It is still empty! Why?

Persistence Demo (Cont'd)

17. The workflow was actually persisted, but we did not see it, only examining the instance store after the workflow completed, when the entry is removed from the instance store. The second Delay of 10 seconds will give you an opportunity to examine the instance store before the workflow has completed. If a window is open showing the instance table, close it now.
18. Run the workflow, and immediately look at **InstancesTable**. Now there has been a workflow instance stored.

| | Id | SurrogateInsta... | SurrogateLock... | PrimitiveDataPr... |
|---|------------------|-------------------|------------------|--------------------|
| ▶ | db2-a99b0b9aeee1 | 2 | 2 | NULL |
| * | NULL | NULL | NULL | NULL |

19. Close this window, and let the application complete. Now look at **InstancesTable** again. It should once again be empty. (Click the Refresh button ) The application at this point is saved in **SimplePersist\Step1**.
20. Modify the code of the **PersistableIdle** handler to return an action of **Unload**.

```
wfApp.PersistableIdle = delegate(
    WorkflowApplicationIdleEventArgs e)
{
    Console.WriteLine("Workflow is idle");
    return PersistableIdleAction.Unload;
};
```

21. Build and run. The workflow immediately unloads during the first Delay, and the workflow does not complete. We don't see any further output.

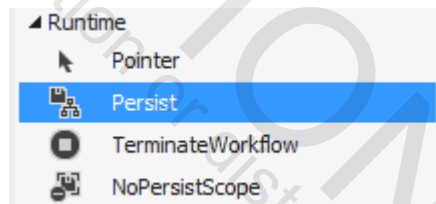
```
Workflow is idle
Workflow has unloaded
```


How to Persist a Workflow

- The most common way to persist a workflow is by specifying a *Persist* or *Unload* action in the handler of the *PersistableIdle* event.
- A second way is to call the *Persist()* or *Unload()* method of the *WorkflowApplication* class.

```
wfApp.Persist();  
wfApp.Unload();
```

- A third way is by using the *Persist* activity in your workflow.
 - The **Persist** activity is in the Runtime group in the Toolbox.



- Another way is implicitly through completion of a transaction activity.

Loading a Persisted Workflow

- **To just persist and unload a workflow by itself is useless.**
- **We also need to be able to reload the workflow and resume its activation.**
- **The basic mechanism is quite simple, relying on an instance store and an application ID.**

- Enable persistence and store app id:

```
store = new SqlWorkflowInstanceStore(  
    connectionString);  
wfApp.InstanceStore = store;  
id = wfApp.Id;
```

- Unload the workflow when it goes idle after creating a bookmark:

```
wfApp.PersistableIdle = delegate(  
    WorkflowApplicationIdleEventArgs e)  
{  
    return PersistableIdleAction.Unload;  
};
```

- Load the workflow:

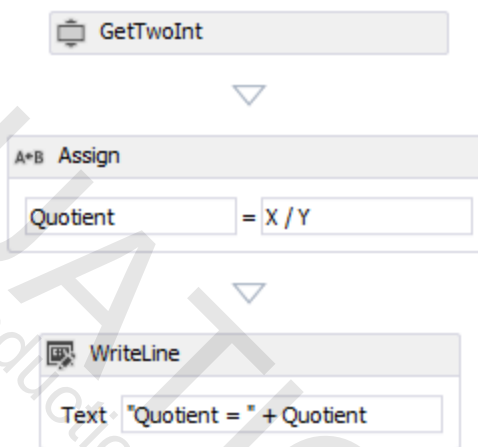
```
wfApp.InstanceStore = store;  
wfApp.Load(id);
```

- Resume the bookmark:

```
wfApp.ResumeBookmark("GetTwoInt", line);
```

Unload and Load Example

- See *BookmarkPersist* in the chapter folder.
 - This program is based on the **SimpleBookmark** example from Chapter 7. The workflow reads two integers and divides them.



- Global variables:

```

static SqlWorkflowInstanceStore store;
const string connectionString = "Server=...
static AutoResetEvent syncUnload =
    new AutoResetEvent(false);
static DivideWorkflow wf = new DivideWorkflow();
  
```

- Main program:

```

static void Main(string[] args)
{
    Guid id = StartAndUnloadInstance();
    Console.WriteLine(
        "Workflow has been persisted. Press any key to
        continue.");
    Console.ReadKey();
    LoadAndCompleteInstance(id);
}
  
```

StartAndUnloadInstance()

```
static Guid StartAndUnloadInstance()
{
    Guid id;
    // Create WorkflowApplication
    WorkflowApplication wfApp =
        new WorkflowApplication(wf);

    // Define lifecycle event handlers
    wfApp.PersistableIdle = delegate(
        WorkflowApplicationIdleEventArgs e)
    {
        return PersistableIdleAction.Unload;
    };
    wfApp.Unloaded = delegate(
        WorkflowApplicationEventArgs e)
    {
        Console.WriteLine("Workflow has unloaded");
        syncUnload.Set();
    };

    // Enable persistence and store app id
    store = new SqlWorkflowInstanceStore(
        connectionString);
    wfApp.InstanceStore = store;
    id = wfApp.Id;

    // Start the workflow
    wfApp.Run();

    // Wait for workflow unloaded
    syncUnload.WaitOne();
    return id;
}
```

LoadAndCompleteInstance()

```
static void LoadAndCompleteInstance(Guid id)
{
    // Create WorkflowApplication
    WorkflowApplication wfApp =
        new WorkflowApplication(wf);

    // Define lifecycle event handlers
    wfApp.Completed = delegate(
        WorkflowApplicationCompletedEventArgs e)
    {
        Console.WriteLine("Workflow has completed");
    };
    wfApp.Unloaded = delegate(
        WorkflowApplicationEventArgs e)
    {
        Console.WriteLine("Workflow has unloaded");
        syncUnload.Set();
    };

    // Load the application
    wfApp.InstanceStore = store;
    wfApp.Load(id);

    // Collect user input and resume the bookmark.
    Console.WriteLine(
        "Enter two integers separated by space");
    string line = Console.ReadLine();
    wfApp.ResumeBookmark("GetTwoInt", line);

    // Wait for workflow unloaded
    syncUnload.WaitOne();
}
```

Stopping and Starting the Host

- In our examples so far the host program was running continuously.
- The persistence mechanism makes it entirely possible to reload a workflow after the host has stopped.
 - The instance store can be recreated from the connection string, and the application ID can be serialized.
 - The example **SerializePersist** illustrates this scenario.
- The solution has three projects:
 - **ActivityLibrary1** contains the custom native activity **GetTwoInt**.
 - **StartWorkflow** contains a copy of the workflow and host code to start the workflow and unload it when **GetTwoInt** asks for input, creating a bookmark.
 - **CompleteWorkflow** contains a copy of the workflow and host code to load and complete the workflow. It supplies the data needed for **GetTwoInt** and resumes the bookmark.
- Run this example as follows:
 - First, build and run the solution without debugging. This will run **StartWorkflow**, because that is the startup project. (If for any reason it is not the startup project, make it so!)
 - Next, make **CompleteWorkflow** the startup project. Build and run the solution without debugging.

StartWorkflow

- *StartWorkflow* calls only *StartAndUnloadInstance()*.
 - The code is almost the same as the corresponding method in the **BookmarkPersist** example.
 - The new feature is the method **SavePersistInfo(Guid id)**, which serializes the application ID.

```
static void StartAndUnloadInstance()
{
    SqlWorkflowInstanceStore store;
    Guid id;
    ...

    // Enable persistence and store app id
    store = new SqlWorkflowInstanceStore(
        connectionString);
    wfApp.InstanceStore = store;
    id = wfApp.Id;

    // Save persistence info
    SavePersistInfo(id);
    ...
}

static void SavePersistInfo(Guid id)
{
    FileInfo f = new FileInfo(
        @"C:\OIC\Data\id.bin");
    Stream s = f.Open(FileMode.Create);
    BinaryFormatter fmt = new BinaryFormatter();
    fmt.Serialize(s, id);
    s.Close();
}
```

StartWorkflow

- ***CompleteWorkflow*** calls ***LoadAndCompleteInstance()***.
 - The code is almost the same as the corresponding method in the **BookmarkPersist** example.
 - The new feature is the method **LoadPersistInfo()**, which deserializes the application ID.

```
static Guid LoadPersistInfo()
{
    FileInfo f = new FileInfo(
        @"C:\OIC\Data\id.bin");
    Stream s = f.Open(FileMode.Open);
    BinaryFormatter fmt = new BinaryFormatter();
    Guid id = (Guid)fmt.Deserialize(s);
    s.Close();
    return id;
}

static void LoadAndCompleteInstance()
{
    SqlWorkflowInstanceStore store;
    Guid id;

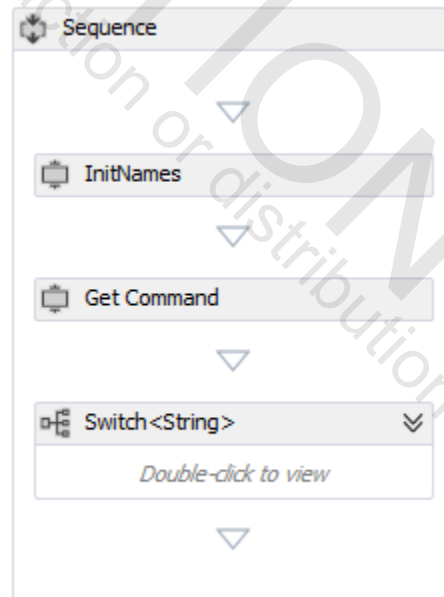
    // Load persistence info
    id = LoadPersistInfo();
    store = new SqlWorkflowInstanceStore(
        connectionString);
    ...

    // Load the application
    wfApp.InstanceStore = store;
    wfApp.Load(id);

    // Collect user input and resume the bookmark.
    ...
}
```


Loading After Data Available

- Our examples so far illustrated getting the needed data after the workflow has been reloaded.
- A better approach is to only reload the workflow *after* the needed data has been obtained.
- Our next example illustrates this scenario.
 - See **PersistentCollection** in the chapter folder.
 - This example also illustrates persisting data as part of the workflow state that is saved, in this case a collection of strings.

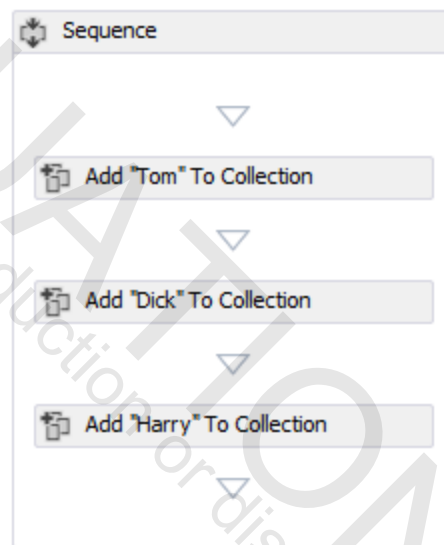


- Variables **Names** and **Cmd** are defined.

| Name | Variable type | Scope | Default |
|-----------------|---------------|----------|-----------------------|
| Names | List<String> | Sequence | new List<string>() |
| Cmd | String | Sequence | Enter a C# expression |
| Create Variable | | | |

InitNames

- ***InitNames*** is a composite activity with the argument ***aNames*** that is bound to the variable ***Names***.
 - It is a composite activity that uses the collection activity **AddToCollection<String>** to initialize a collection **List<String>** of names.



- Each invocation of **AddToCollection<String>** adds a different hardcoded name.

| | |
|--------------|-------------------------|
| Collection | aNames |
| DisplayName | Add "Tom" To Collection |
| Item | "Tom" |
| TypeArgument | String |

GetString

- ***GetString*** (used with display name “Get Command” in *Workflow1.xaml*) is a native activity that reads a string from the console.

- It creates a bookmark and waits for the string to be supplied by the host when it resumes the bookmark.

```
public sealed class GetString : NativeActivity
{
    public static AutoResetEvent
        syncNeedData = new AutoResetEvent(false);

    public OutArgument<string> aStr { get; set; }

    protected override void Execute(
        NativeActivityContext context)
    {
        context.CreateBookmark("GetString", Resumed);
        syncNeedData.Set();
    }

    // The callback method
    private void Resumed(
        NativeActivityContext context,
        Bookmark bookmark, object value)
    {
        string str = value as string;
        aStr.Set(context, str);
    }

    protected override bool CanInduceIdle
    {
        get { return true; }
    }
}
```

String Commands

- The workflow will unload to allow user to enter a command. The legal commands are “show” and “concat”.
- When the workflow resumes, a Switch<String> activity will process the command.

Switch<String>

Expression

Default

WriteLine

Text

Case show

ShowNames

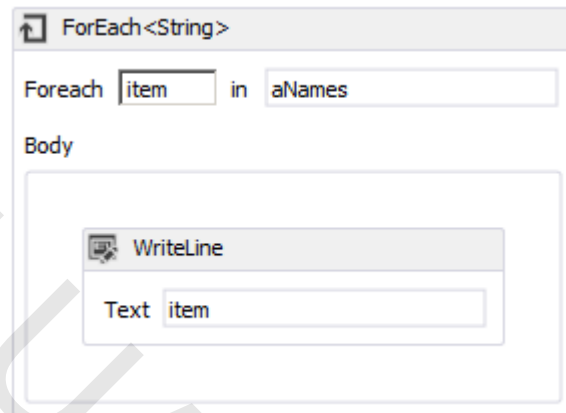
Case concat

ConcatenateNames

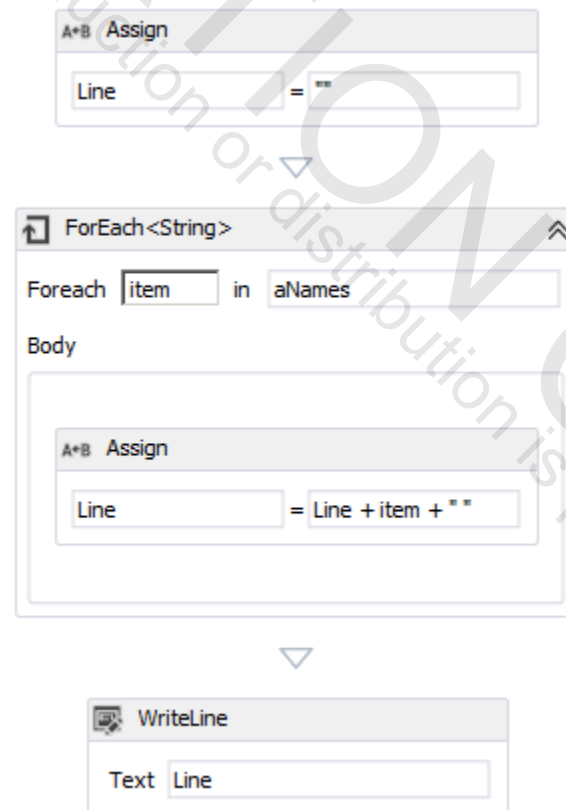
[Add new case](#)

String Commands (Cont'd)

- *ShowNames.xaml* is a composite activity.



- *ConcatenateNames.xaml* is also a composite activity.



Host Program

- **The host program is very similar to the host for the *BookmarkPersist* example.**

- Global variables:

```
static SqlWorkflowInstanceStore store;
const string connectionString = "Server=...
static AutoResetEvent syncUnload =
    new AutoResetEvent(false);
static Workflow1 wf = new Workflow1();
```

- Main program:

```
static void Main(string[] args)
{
    Guid id = StartAndUnloadInstance();
    Console.Write("show or concat: ");
    string cmd = Console.ReadLine();
    LoadAndCompleteInstance(id, cmd);
}
```

- **The key difference is that the workflow remains unloaded until after the user has supplied data.**

- **Here is a sample run:**

```
Workflow has unloaded
show or concat: concat
Tom Dick Harry
Workflow has completed
Workflow has unloaded
Press any key to continue . . .
```

Lab 9

Persistence in a Business Scenario

In this lab you will you will implement a life insurance workflow application that prepares a term life insurance proposal based upon an application and an underwriter's report, which evaluates the applicant's insurability. A base premium can be calculated immediately from the applicant's age and amount of insurance requested. After performing this part of the process, the workflow will unload until the user enters the underwriter's report. This will cause the workflow to be loaded again and complete the proposal or statement that insurance has been denied.

Detailed instructions are contained in the Lab 9 write-up at the end of the chapter.

Suggested time: 60 minutes

Summary

- To deal with the requirement of long running workflows, Workflow Foundation provides a persistence mechanism.
- You can set up a persistence store in SQL Server.
- A common way to persist a workflow is by specifying a *Persist* or *Unload* action in the handler of the *PersistableIdle* event.
- You can serialize workflow instance identification so that the host program as well as the workflow can unload and start up again.
- You can implement a long running workflow that can be unloaded for days or weeks and then resume when needed data is available.

Lab 9

Persistence in a Business Scenario

Introduction

In this lab you will implement a life insurance workflow application that prepares a term life insurance proposal based upon an application and an underwriter's report, which evaluates the applicant's insurability. A base premium can be calculated immediately from the applicant's age and amount of insurance requested. After performing this part of the process, the workflow will unload until the user enters the underwriter's report. This will cause the workflow to be loaded again and complete the proposal or statement that insurance has been denied.

Suggested Time: 60 minutes

Root Directory: OIC\WfCs

Directories:

- Labs\Lab9\PersistentTermLife** (do your work here)
- Chap09\PersistentCollection** (model host persistence code)
- Chap09\PersistentTermLife\Step0** (backup of starter code)
- Chap09\PersistentTermLife\Step1** (solution to Part 1)
- Chap09\PersistentTermLife\Step2** (solution to Part 2)

Part 1. Add Bookmark and Underwriter's Report to the Workflow

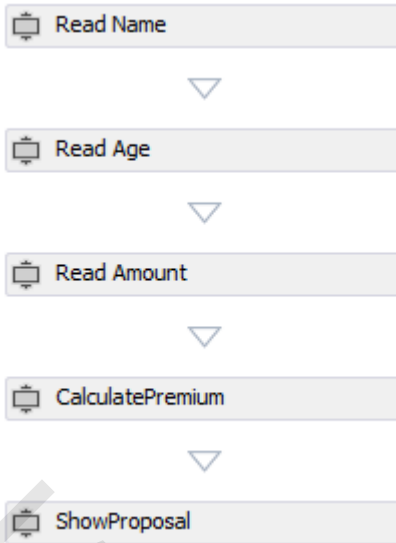
You are provided with starter code that prepares a proposal based on the applicant's name, age and amount of insurance requested. Modify the application to query the user for an underwriter's report that will be one of "approve", "rating" or "deny". In the first case the premium will be the base premium already calculated. In the second case insurance will be offered at a higher than normal premium, which will be double the base premium. In the third case insurance will be denied. Implement the query for the underwriter's report as a native activity that creates a bookmark. The activity will resume when the host program supplies a string.

Optional detailed instructions:

1. Build and run the starter program. Here is a sample run:

```
Name: John Smith
Age: 40
Amount: 150000
Term Life Proposal
Insured: John Smith
Age: 40
Death Benefit: $150,000.00
Monthly Premium: $18.30
```

- Examine the starter workflow, which is a simple sequence of custom code activities.



- Examine **CalculatePremium**:

```

public sealed class CalculatePremium : CodeActivity<decimal>
{
    public InArgument<int> aAge { get; set; }
    public InArgument<int> aAmount { get; set; }

    protected override decimal Execute(CodeActivityContext context)
    {
        int age = context.GetValue(this.aAge);
        decimal amount = (decimal)context.GetValue(this.aAmount);
        return Rate.MonthlyPremium(age, amount);
    }
}

```

- This calculation in turn is based on a **Rate** class, which calculates a monthly premium based on a table giving rates for different 5-year age intervals between 20 and 69.
- ShowProposal** is a simple activity which merely displays a proposal based upon its input arguments.
- Implement a custom native activity **GetString**, which will read a string from the console and return the value in an output argument **aStr**. This activity makes use of the bookmark “GetString”. A public static **AutoResetEvent** will be used by the host for synchronizing with when the activity needs data.

```

public sealed class GetString : NativeActivity
{
    public static AutoResetEvent syncNeedData =
        new AutoResetEvent(false);

    public OutArgument<string> aStr { get; set; }
}

```

```

protected override void Execute(NativeActivityContext context)
{
    context.CreateBookmark("GetString", Resumed);
    syncNeedData.Set();
}

// The callback method
private void Resumed(NativeActivityContext context,
    Bookmark bookmark, object value)
{
    string str = value as string;
    aStr.Set(context, str);
}

protected override bool CanInduceIdle
{
    get { return true; }
}
}

```

7. Implement standard host code to run the workflow in a **WorkflowApplication**. The host's thread should wait for **syncNeedData** to be signaled and then prompt the user for an underwriter's decision, one of "approve", "deny" or "rating". The bookmark should then be resumed, passing the string entered by the user.

```

class Program
{
    static WorkflowApplication wfApp;

    static void Main(string[] args)
    {
        // Create the Workflow and the WorkflowApplication
        Workflow1 wf = new Workflow1();
        wfApp = new WorkflowApplication(wf);

        // Initialize AutoResetEvent for thread synchronization
        AutoResetEvent sync = new AutoResetEvent(false);

        // Handle the completed event to signal the host
        // that the workflow has completed
        wfApp.Completed = delegate(
            WorkflowApplicationCompletedEventArgs e)
        {
            sync.Set();
        };

        // Start the workflow
        wfApp.Run();

        // Obtain the input the workflow is expecting and resume bookmark
        GetString.syncNeedData.WaitOne();
        Console.WriteLine("Underwriting Decision");
        Console.WriteLine("approve, deny, or rating:");
        string str = Console.ReadLine();
        wfApp.ResumeBookmark("GetString", str);
    }
}

```

```

        // Wait for workflow completion
        sync.WaitOne();
    }
}

```

8. Modify **Workflow1.xaml** to get the underwriter's report using **GetString** and then switch on the string that is returned, with separate cases for **approve**, **rating** and **deny**.



9. Implement the three cases. For **approve**, just invoke **ShowProposal**. For **rating**, provide a **Sequence** that will double **Premium** and then invoke **ShowProposal**. For **deny**, just write a message that insurance has been denied.
10. Build and run, trying out the three cases. You are at Step 1.

Part 2. Implement Persistence in the Workflow

Modify the host program to implement persistence. After calculating the base premium the workflow instance should unload, persisting the workflow state to the instance store.

The use is prompted for the underwriter's report. Then the workflow is loaded, the bookmark resumed with the data, and the workflow completes.

Optional detailed instructions:

1. Add references in the workflow project to **System.Activities.DurableInstancing** and to **System.Runtime.DurableInstancing**.
2. Import the namespace **System.Activities.DurableInstancing**.
3. There is a pattern to host code to run the workflow in the manner we want, with user input sandwiched between methods to start/unload and load/complete the workflow. We can use almost the exact same code as in the **PersistentCollection** example. Open this solution.
4. Copy the contents of the **Program** class in **PersistentCollection** into the **Program** class of the host program you are working with. Don't overwrite what is there, as you will want to use a tiny segment of code prompting the user for the underwriter report.
5. Observe the sandwich pattern of **Main()**. Replace the prompt for the collection example with the prompt for the life insurance application you are working on. For clarity, rename the string **str** to **underwriting**.

```
static void Main(string[] args)
{
    Guid id = StartAndUnloadInstance();
    Console.WriteLine("Underwriting Decision");
    Console.WriteLine("approve, deny, or rating:");
    string underwriting = Console.ReadLine();
    LoadAndCompleteInstance(id, underwriting);
}
```

6. In **LoadAndCompleteInstance()** replace occurrences of **cmd** by **underwriting**.

```
static void LoadAndCompleteInstance(Guid id, string underwriting)
{
    ...

    // Resume the bookmark with data already gathered
    wfApp.ResumeBookmark("GetString", underwriting);

    ...;
}
```

7. Now you can remove the old contents of the **Program** class.
8. Build and run. The program should behave like the Step 2 version, but now the workflow has been persisted and unloaded.
9. Finally, you may wish to examine the instance store at various places before, during and after program execution.