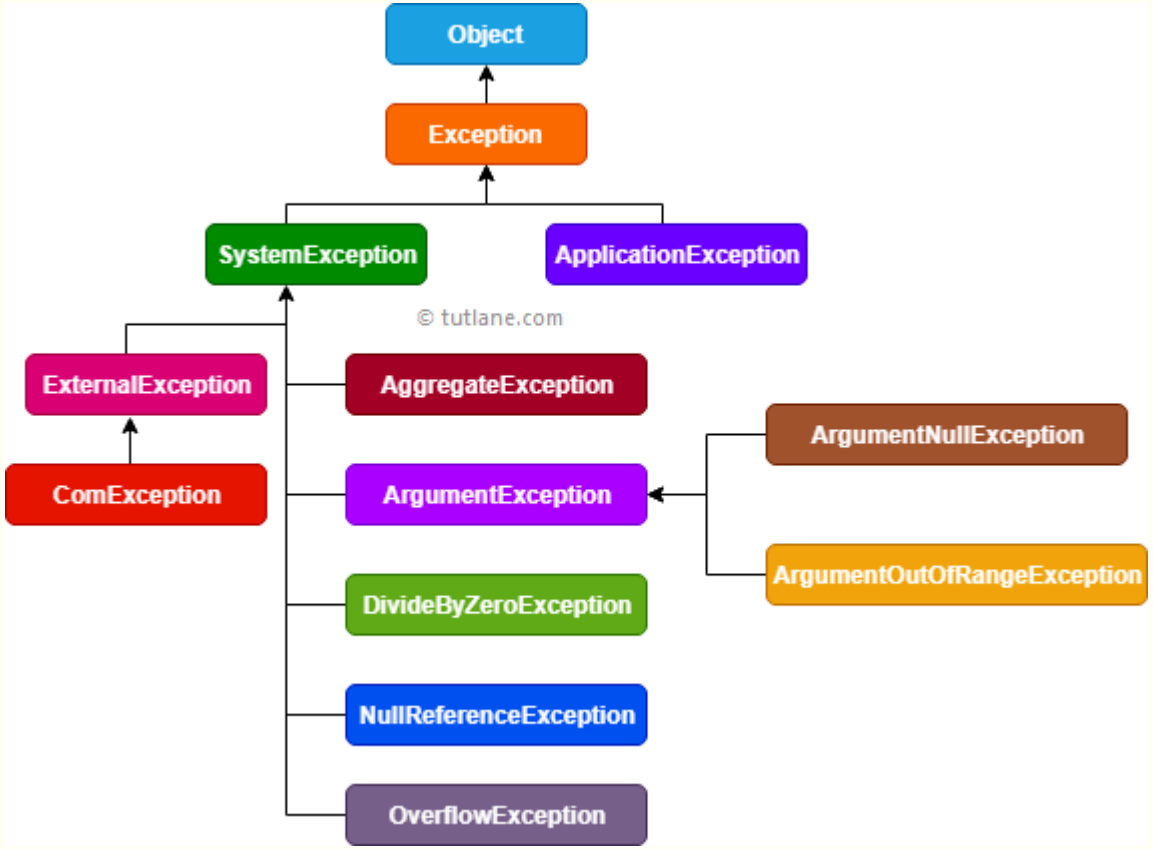
НАСЛІДУВАННЯ НА ПРИКЛАДІ СТРУКТУРОВАНОЇ ОБРОБКИ ВИНЯТКІВ

Питання 4.2.

Поширені терміни

- ***Програмні помилки.*** Помилки, зроблені програмістом.
  - Наприклад, динамічно виділена пам’ять не була вивільнена в C++-коді, що призвело до витоку п’амяті.
- ***Користувацькі помилки.*** Зазвичай виникають через тих, хто запускає додаток.
  - Наприклад, введення неправильно оформленого рядка призводить до помилки, якщо в коді не була передбачена обробка некоректного вводу.
- ***Винятки.*** Аномалії, що виникають під час виконання програми, які складно, а інколи й неможливо, врахувати при розробці додатку.
  - Приклади: спроба підключення до БД, якої більше не існує; відкриття пошкодженого файлу; спроба встановлення зв’язку з машиною, яка на даний момент знаходиться в автономному режимі тощо.
  - У таких випадках програміст чи користувач мало що може зробити з «винятковими» обставинами.

Структурована обробка винятків



- Структурована обробка винятків у .NET — це прийом, призначений для роботи з винятковими ситуаціями, що виникають у ході виконання додатку.
  - Навіть для програмних та користувацьких помилок середовище CLR часто генерує відповідний виняток, який ідентифікує проблему, яка виникла.
  - У бібліотеках базових класів .NET визначено багато різних винятків, таких як FormatException, IndexOutOfRangeException, FileNotFoundException, ArgumentOutOfRangeException и т.д.
- У термінології .NET под “винятком” мають на увазі програмні помилки, некоректний користувацький ввід та помилки часу виконання.

Роль обробки винятків .NET

```
/* Типичный механизм отлавливания ошибок в стиле C. */
#define E_FILENOTFOUND 1000

int UseFileSystem()
{
  // Предполагается, что в этой функции происходит нечто
  // такое, что приводит к возврату следующего значения.
  return E_FILENOTFOUND;
}

void main()
{
  int retVal = UseFileSystem();
  if(retVal == E_FILENOTFOUND)
    printf("Cannot find file...");  // Не удается найти файл
}
```

- Раніше команда розробників могла визначати набір числових констант для представлення відомих збійних ситуацій, а потім використовувати ці константи як вихідні значення методів.
  - константа E_FILENOTFOUND—лише числове значення, яке мало що скаже з приводу вирішення проблеми.
- Також в API-інтерфейсі Windows визначені сотні кодів помилок, які отримуються у вигляді #define і HRESULT, а також багатьох варіацій простих бульових значень (boo1, BOOL, VARIANT_BOOL і т.д.).
  - Багато розробників COM-додатків на C++ використовують невеликий набір стандартних COM-інтерфейсів (наприклад, ISupportErrorInfo, IErrorInfo, ICreateErrorInfo) для повернення значимої інформації щодо помилку клієнту COM.
- Очевидна проблема підходів – відсутність симетрії.
  - У .NET запропонована стандартна методика для генерування та виявлення помилок виконуючого середовища — структурована обробка винятків.

Переваги структурованої обробки

- Розробники мають уніфікований підхід до обробки помилок, спільний для всіх мов, орієнтованих на платформу .NET.
  - Тому спосіб обробки помилок на C# синтаксично подібний аналогічному способу на VB чи C++.
- Винятки представляють собою об’єкти, в яких міститься читабельний опис проблеми, а також детальний знімок стеку викликів на момент первинного виникнення винятку.
  - Кінцевому користувачу можна надати довідкове посилання (URL-адресу), за якою можна отримати подробиці про помилку, а також спеціальні дані, визначені програмістом.
- Програмування зі структурованою обробкою винятків передбачає використання 4 пов’язаних між собою сутностей:
  - тип класу, який представляє деталі винятку;
  - член, здатний генерувати екземпляр класу винятку у викликаючому коді за відповідних обставин;
  - блок коду на викликаючій стороні, який звертається до члену, в якому може виникнути виняток;
  - блок коду на викликаючій стороні, який буде обробляти (або перехоплювати) виняток у випадку його виникнення.

Базовий клас System.Exception

```
public class Éxception : ISerializable, _Exception
{
  // Открытые конструкторы.
  public Exception(string message, Exception innerException);
  public Exception(string message);
  public Exception();
  ...
  // Методы.
  public virtual Exception GetBaseException();
  public virtual void GetObjectData(SerializationInfo info,
    StreamingContext context);
  // Свойства.
  public virtual IDictionary Data { get; }
  public virtual string HelpLink { get; set; }
  public Exception InnerException { get; }
  public virtual string Message { get; }
  public virtual string Source { get; set; }
  public virtual string StackTrace { get; }
  public MethodBase TargetSite { get; }
  ...
}
```

- Всі винятки породжені від базового класу System.Exception, який, у свою чергу, похідний від System.Object.
- Багато властивостей System.Exception за своєю природою є доступними тільки для зчитування.
  - Це пояснюється тим, що стандартні значення для кожного з них зазвичай постачаються породженими типами.
  - Наприклад, стандартне повідомлення типу IndexOutOfRangeException таке: “Index was outside the bounds of the array” (“Індекс вийшов за межі масиву”).

Основні члени типу System.Exception

Свойство System.Exception	Описание
Data	Это свойство, доступное только для чтения, позволяет извлекать коллекцию пар "ключ/значение" (представленную объектом, реализующим IDictionary), которая предоставляет дополнительную определяемую программистом информацию об исключении. По умолчанию эта коллекция пуста
HelpLink	Это свойство позволяет получать или устанавливать URL для доступа к справочному файлу или веб-сайту с подробным описанием ошибки
InnerException	Это свойство, доступное только для чтения, может использоваться для получения информации о предыдущем исключении или исключениях, которые послужили причиной возникновения текущего исключения. Запись предыдущих исключений осуществляется путем их передачи конструктору самого последнего исключения
Message	Это свойство, доступное только для чтения, возвращает текстовое описание заданной ошибки. Само сообщение об ошибке устанавливается как параметр конструктора
Source	Это свойство позволяет получать или устанавливать имя сборки или объекта, который привел к генерации исключения
StackTrace	Это свойство, доступное только для чтения, содержит строку, идентифицирующую последовательность вызовов, которая привела к возникновению исключения. Как нетрудно догадаться, данное свойство очень полезно во время отладки или для сохранения информации об ошибке во внешнем журнале ошибок
TargetSite	Это свойство, доступное только для чтения, возвращает объект MethodBase с описанием многочисленных деталей о методе, который привел к генерации исключения (вызов ToString() будет идентифицировать этот метод по имени)

```
class Car
{
  // Константа для представления максимальной скорости
  public const int MaxSpeed = 100;

  // Свойства автомобиля.
  public int CurrentSpeed {get; set;}
  public string PetName {get; set;}

  // Не вышел ли автомобиль из строя?
  private bool carIsDead;

  // Автомобиль имеет радиоприемник.
  private Radio theMusicBox = new Radio();

  // Конструкторы.
  public Car() {}
  public Car(string name, int speed)
  {
    CurrentSpeed = speed;
    PetName = name;
  }
  public void CrankTunes(bool state)
  {
    // Делегировать запрос внутреннему объекту.
    theMusicBox.TurnOn(state);
  }

  // Проверить, не перегрелся ли автомобиль.
  public void Accelerate(int delta)
  {
    if (carIsDead)
      Console.WriteLine("{0} is out of order...", PetName);
    else
    {
      CurrentSpeed += delta;
      if (CurrentSpeed > MaxSpeed)
      {
        Console.WriteLine("{0} has overheated!", PetName);
        CurrentSpeed = 0;
        carIsDead = true;
      }
      else
        Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
  }
}
```

Простий приклад

- Визначимо два класи (Car (автомобіль) і Radio (радіоприймач)), зв'язавши їх відношенням "має".
  - У класі Radio визначено один метод, що відповідає за ввімкнення/вимкнення приймача:

```
class Radio
{
  public void TurnOn(bool on)
  {
    if(on)
      Console.WriteLine("Jamming...");
    else
      Console.WriteLine("Quiet time...");
  }
}
```

- Клас Car визначено так, що коли користувач перевищує задану максимальну швидкість (за допомогою константного члена MaxSpeed), двигун ламається, приводячи об'єкт Car у неробочий стан (відображається закритим полем carIsDead).
  - Також клас Car має кілька властивостей для представлення поточної швидкості та вказаної користувачем "дружньої назви" автомобіля, різні конструктори.

Продовження коду

- Реалізуємо метод Main(), у якому об’єкт Car буде перевищувати максимальну швидкість 100 в класі Car):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);
  for (int i = 0; i < 10;  i++)
    myCar.Accelerate(10);
  Console.ReadLine();
}
```

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
Zippy has overheated!
Zippy is out of order...
```

Генерація загального винятку

```
// На этот раз в случае превышения пользователем указанного
// в MaxSpeed предела должно генерироваться исключение.
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;
      // Использовать ключевое слово throw для генерации исключения.
      throw new Exception(string.Format("{0} has overheated!", PetName));
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

- Поточна реалізація Accelerate() просто відображає повідомлення про помилку, якщо викликаючий код намагається розігнати автомобіль понад максимальну швидкість.
  - Слід створити й сконфігурувати новий екземпляр класу System.Exception, встановивши значення readonly-властивості Message через конструктор класу.
  - Для відправки об'єкта помилки назад у викликаючий код застосовується ключове слово throw.
- Якщо ви генеруєте виняток, то завжди самостійно вирішуєте, як у точності буде виглядати помилка і як вона повинна викидатись.
  - У прикладі передбачається, що при спробі збільшити швидкість автомобіля, який вже зламався, потрібно згенерувати об'єкт System.Exception для сповіщення про неможливість подальшої роботи методу Accelerate().

Альтернатива та перехоплення винятків

- Метод Accelerate() можна було б реалізувати так, щоб він виконував автоматичне відновлення, не викидаючи винятків.
  - Загалом винятки повинні генеруватись тільки у випадку виникнення більш критичних умов: відсутність потрібного файлу, неможливість підключення до БД тощо.
  - Джерело генерації винятків потребує серйозного осмислення та пошуку суттєвого обґрунтування на етапі проектування.
  - Зараз вважатимемо, що даний випадок – достатньо виправдана причина для викидання винятку.
- Викликаючий код тепер повинен бути готовим обробляти виняткову ситуацію: потрібний блок try/catch.
  - Після перехоплення об’єкта винятку можна звертатись до різних його членів та одержувати інформацію про проблему.
- Що робити з цими деталями далі в основному залежить від вас.
  - Зафіксувати їх у спеціальному файлі звіту, записати в журнал подій Windows, надіслати по електронній пошті системному адміністратору чи показати кінцевому користувачу.

```
// Обработка сгенерированного исключения.
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  // Разогнаться до скорости, превышающей максимальный
  // предел автомобиля, для выдачи исключения.
  try
  {
    for(int i = 0; i < 10; i++)
      myCar.Accelerate(10);
  }
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");              // ошибка
    Console.WriteLine("Method: {0}", e.TargetSite);     // метод
    Console.WriteLine("Message: {0}", e.Message);       // сообщение
    Console.WriteLine("Source: {0}", e.Source);         // источник
  }

  // Ошибка была обработана, продолжается выполнение
  // следующего оператора.
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

- Якщо виняток проявляється, керування переходить до відповідного блоку catch.

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Method: Void Accelerate(Int32)
Message: Zippy has overheated!
Source: SimpleException

***** Out of exception logic *****
```

Конфігурування стану винятку. Властивість TargetSite

```
static void Main(string[] args)
{
...
  // TargetSite actually returns a MethodBase object.
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Member name: {0}", e.TargetSite);
    Console.WriteLine("Class defining member: {0}",
      e.TargetSite.DeclaringType);
    Console.WriteLine("Member type: {0}", e.TargetSite.MemberType);
    Console.WriteLine("Message: {0}", e.Message);
    Console.WriteLine("Source: {0}", e.Source);
  }
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

```
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
```

- Клас Exception має багато додаткових членів (TargetSite, StackTrace, HelpLink и Data), корисних для додаткового уточнення природи проблеми.
- Властивість System.Exception.TargetSite дозволяє з'ясувати різні деталі щодо методу, в якому було згенеровано виняток.
  - Вивід значення TargetSite призводить до відображення вихідного значення, назви та типів параметрів метода, що згенерував виняток.
  - Проте властивість TargetSite повертає не простий рядок, а строго типізований об'єкт System.Reflection.MethodBase.
  - Цей тип дозволяє зібрати різноманітні подробиці, пов'язані з проблемним методом, а також класом, у якому він визначений.

Властивість StackTrace

- Дозволяє визначити послідовність викликів, яка призвела до генерації винятку.
  - Значение этого свойства никогда не устанавливается вручную — это делается автоматически во время создания объекта исключения.
  - Строка, возвращаемая из StackTrace, отражает последовательность вызовов, которая привела к выдаче этого исключения. Обратите внимание, что самый нижний номер в этой строке указывает на место возникновения первого вызова в последовательности, а самый верхний — на место, где точно находится породивший проблему член.
  - Очевидно, что такая информация очень полезна при отладке или ведении журнала для конкретного приложения, поскольку позволяет проследить весь путь к источнику ошибки.

```
catch(Exception e)
{
  ...
  Console.WriteLine("Stack: {0}", e.StackTrace);  // стек
}
```

```
Stack: at SimpleException.Car.Accelerate(Int32 delta)
in c:\MyApps\SimpleException\car.cs:line 65 at SimpleException.Program.Main()
in c:\MyApps\SimpleException\Program.cs:line 21
```

Властивість HelpLink

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;
      // Создать локальную переменную перед генерацией объекта Exception,
      // чтобы можно было обращаться к свойству HelpLink.
      Exception ex =
        new Exception(string.Format("{0} has overheated!", PetName));
      ex.HelpLink = "http://www.CarsRUs.com";
      throw ex;
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);// Вывод текущей
                                                              // скорости
  }
}
```

- для отображения конечному пользователю читабельной информации может применяться свойство System.Exception.Message.
- Кроме того, можно установить свойство HelpLink для указания на конкретный URL или стандартный справочный файл Windows, где содержатся более детальные сведения о проблеме.
  - По умолчанию значением свойства HelpLink является пустая строка.
  - Присваивание этому свойству какого-то более интересного значения должно делаться перед генерацией исключения System.Exception.

```
catch(Exception e)
{
  ...
  Console.WriteLine("Help Link: {0}", e.HelpLink);  // Ссылка для справки
}
```

Властивість Data

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;
      // Создать локальную переменную перед генерацией объекта Exception,
      // чтобы можно было обращаться к свойству HelpLink.
      Exception ex =
        new Exception(string.Format("{0} has overheated!", PetName));
      ex.HelpLink = "http://www.CarsRUs.com";
      // Указать специальные данные, касающиеся ошибки.
      ex.Data.Add("TimeStamp",
        string.Format("The car exploded at {0}", DateTime.Now)); // метка времени
      ex.Data.Add("Cause", "You have a lead foot.");             // причина
      throw ex;
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

- позволяет заполнить объект исключения релевантной вспомогательной информацией (такой как метка времени).
- Свойство Data возвращает объект, реализующий интерфейс по имени IDictionary, который определен в пространстве имен System.Collections.
  - На данный момент важно понять лишь то, что словарные коллекции позволяют создавать наборы значений, извлекаемых по ключу.

Властивість Data

- Для успешного перечисления пар “ключ/значение” нужно сначала указать директиву using для пространства имен System.Collections, поскольку в файле с классом, реализующим метод Main (), будет использоваться тип DictionaryEntry:
  - using System.Collections;
- Затем потребуется обновить логику catch, чтббы обеспечить проверку на предмет равенства null значения свойства Data (null является стандартным значением).
  - После этого можно пользоваться свойствами Key и Value типа DictionaryEntry для вывода специальных данных на консоль:

```
catch (Exception e)
{
  ...
  // По умолчанию поле данных является пустым, поэтому проверить его на null.
  Console.WriteLine("\n-> Custom Data:");
  if (e.Data != null)
  {
    foreach (DictionaryEntry de in e.Data)
    Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
  }
}
```

Програмний вивід

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
Stack: at SimpleException.Car.Accelerate(Int32 delta)
     at SimpleException.Program.Main(String[] args)
Help Link: http://www.CarsRUs.com

-> Custom Data:
-> TimeStamp: The car exploded at 9/12/2015 9:02:12 PM
-> Cause: You have a lead foot.

***** Out of exception logic *****
```

- Свойство Data очень полезно, т.к. позволяет упаковывать специальную информацию об ошибке, не требуя построения нового класса, расширяющего базовый класс Exception.
- Однако разработчики .NET-приложений по-прежнему часто предпочитают создавать строго типизированные классы исключений, которые поддерживают специальные данные с использованием строго типизированных свойств.
  - Этот подход позволяет вызывающему коду перехватывать конкретный производный от Exception тип, а не углубляться в коллекцию данных для получения дополнительных деталей.
  - Чтобы понять, как это работает, необходимо разобраться с отличием между исключениями уровня системы и уровня приложения.

Винятки рівня системи (System.SystemException)

- Исключения, которые генерируются самой платформой .NET, называются системными исключениями.
  - Эти исключения в общем случае рассматриваются как неисправимые фатальные ошибки.
  - Системные исключения порождены прямо от базового класса System.SystemException, который, в свою очередь, порожден от System.Exception (а тот — от класса System.Object):

```
public class SystemException : Exception
{
  // Разнообразные конструкторы.
}
```

- когда тип исключения порожден от System.SystemException, есть возможность определить, что сущностью, сгенерировавшей исключение, является исполняющая среда .NET, а не кодовая база функционирующего приложения.
  - Это можно довольно легко проверить с помощью ключевого слова is:

```
// Действительно так! NullReferenceException является SystemException.
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine("NullReferenceException is-a SystemException? : {0}",
                  nullRefEx is SystemException);
```

Винятки рівня додатку (System.ApplicationException)

- Поскольку все исключения .NET — это типы классов, вполне допускается создавать собственные исключения, специфичные для приложения.
  - Однако из-за того, что базовый класс System.SystemException представляет исключения, генерируемые средой CLR, может сложиться впечатление о том, что специальные исключения тоже должны быть порождены от типа System.Exception.
  - Хоть можно поступать и так, но вместо этого лучше порождать их от System.ApplicationException:

```
public class ApplicationException : Exception
{
  // Разнообразные конструкторы.
}
```

- Как и SystemException, класс ApplicationException не определяет никаких дополнительных членов кроме набора конструкторов.
  - С точки зрения функциональности единственной целью System.ApplicationException является идентификация источника ошибки.
  - При обработке исключения, производного от System.ApplicationException, можно смело предполагать, что исключение было инициировано кодом работающего приложения, а не библиотеками базовых классов .NET либо исполняющей средой .NET.

Побудова спеціальних винятків: спосіб I

- Иногда предпочтительнее построить строго типизированное исключение, которое представляет уникальные детали, связанные с текущей проблемой.
  - предположим, что необходимо создать специальное исключение (по имени CarIsDeadException) для предоставления деталей об ошибке, возникающей из-за увеличения скорости неисправного автомобиля.
  - Первый шаг состоит в порождении нового класса от System.Exception/System.ApplicationException (по соглашению имена всех классов исключений заканчиваются суффиксом Exception; на самом деле это является рекомендуемой практикой в .NET).
- Как правило, все специальные классы исключений должны быть определены как открытые
  - стандартным модификатором доступа для не вложенных типов является internal.
  - Причина в том, что исключения часто передаются за границы сборок, следовательно, они должны быть доступны вызывающей кодовой базе.

Проект CustomException

```
public class CarIsDeadException : ApplicationException
{
  private string messageDetails = String.Empty;
  public DateTime ErrorTimeStamp {get; set;}
  public string CauseOfError {get; set;}

  public CarIsDeadException(){}
  public CarIsDeadException(string message,
    string cause, DateTime time)
  {
    messageDetails = message;
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
  // Переопределение свойства Exception.Message.
  public override string Message
  {
    get
    {
      return string.Format("Car Error Message: {0}", messageDetails);
    }
  }
}
```

- Додамо в проект наші класи Car і Radio, а також новий клас
  - Вместо заполнения словаря данных (через свойство Data), когда генерируется исключение, конструктор позволяет отправителю передавать метку времени и причину возникновения ошибки.
  - Здесь класс CarIsDeadException поддерживает закрытое поле (messageDetails), которое представляет данные, касающиеся текущего исключения;
  - его можно устанавливать с использованием специального конструктора.

```
// Throw the custom CarIsDeadException.
public void Accelerate(int delta)
{
...
  CarIsDeadException ex =
   new CarIsDeadException (string.Format("{0} has overheated!",
PetName),
                          "You have a lead foot", DateTime.Now);
  ex.HelpLink = "http://www.CarsRUs.com";
  throw ex;
...
}
```

Перехоплення винятку

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Отслеживание исключения.
    myCar.Accelerate(50);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
    Console.WriteLine(e.ErrorTimeStamp);
    Console.WriteLine(e.CauseOfError);
  }
  Console.ReadLine();
}
```

- Для перехвата такого входящего исключения теперь можно модифицировать блок catch, чтобы в нем перехватывался специфичный тип CarIsDeadException
  - хотя из-за того, что System.CarIsDeadException “является” System.Exception, также разрешено перехватывать System.Exception
- Коли потрібно створювати власні спеціальні виключення?
  - Обычно необходимость в создании специальных исключений возникает, только если ошибка тесно связана с выдающим ее классом (например, специальный класс для работы с файлами может выдавать набор файловых ошибок, класс Car — ошибки, связанные с автомобилем, объект доступа к данным — ошибки, связанные с отдельной таблицей базы данных, и т.д.).
  - Создание специальных исключений позволяет обеспечить вызывающий код возможностью обрабатывать многочисленные исключения на дескриптивной основе.

Побудова спеціальних винятків: спосіб II

- Текущий тип CarIsDeadException переопределяет виртуальное свойство System.Exception.Message для конфигурирования специального сообщения об ошибке и предлагает два специальных свойства для учета дополнительных порций данных.
  - Тем не менее, в реальности переопределять виртуальное свойство Message не требуется, т.к. можно просто передать входящее сообщение конструктору родительского класса:

```
public class CarIsDeadException : ApplicationException
{
  public DateTime ErrorTimeStamp { get; set; }
  public string CauseOfError { get; set; }
  public CarIsDeadException() { }
  // Передача сообщения конструктору родительского класса.
  public CarIsDeadException(string message, string cause, DateTime time)
    :base(message)
  {
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
}
```

Побудова спеціальних винятків: спосіб II

- На этот раз не объявляется строковая переменная для представления сообщения и не переопределяется свойство Message.
  - Вместо этого соответствующий параметр просто передается конструктору базового класса.
  - При таком решении специальный класс исключения является не более чем уникально именованным классом, производным от System.ApplicationException (при необходимости с дополнительными свойствами), который лишен каких-либо переопределений базовых классов.
- Не удивляйтесь, если многие (а то и все) специальные классы исключений будут следовать такому простому шаблону.
  - Во многих случаях роль специального исключения не обязательно заключается в предоставлении дополнительной функциональности помимо той, что унаследована от базовых классов, а в обеспечении строго именованного типа, который четко идентифицирует природу ошибки;
  - это позволяет клиенту предусмотреть разную логику обработки для разных типов исключений.

Побудова спеціальних винятків: спосіб III

- Если необходимо создать действительно заслуживающий внимания специальный класс исключения, необходимо позаботиться о том, чтобы он соответствовал наилучшим рекомендациям .NET.

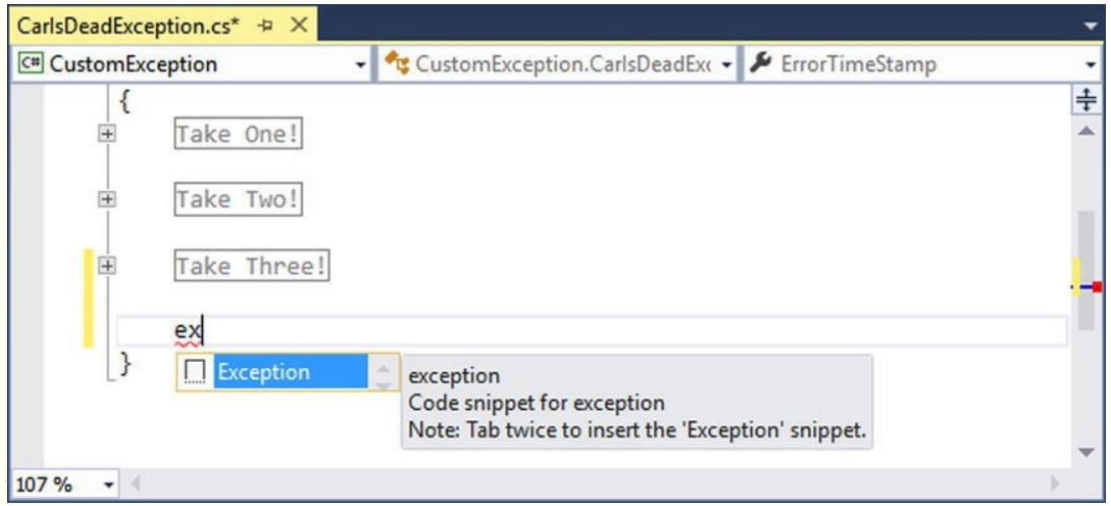
- В частности, это означает, что он должен соответствовать следующим характеристикам:
  - наследоваться от Exception/ApplicationException;
  - быть помеченным атрибутом [System.Seria1izab1e];
  - определять стандартный конструктор;
  - определять конструктор, который устанавливает значение унаследованного свойства Message;
  - определять конструктор для обработки “внутренних исключений”;
  - определять конструктор для поддержки сериализации типа.

Остання версія класу CarIsDeadException

- в Visual Studio предоставляется специальный фрагмент кода под названием Exception: позволяет автоматически генерировать новый класс исключения, отвечающий требованиям рекомендаций .NET.
  - Двічі натиснути Tab

```
[Serializable]
public class CarIsDeadException : ApplicationException
{
  public CarIsDeadException() { }
  public CarIsDeadException(string message) : base( message ) { }
  public CarIsDeadException(string message,
                            System.Exception inner)
    : base( message, inner ) { }
  protected CarIsDeadException(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
    : base( info, context ) { }

  // Любые дополнительные специальные свойства, конструкторы и члены данных...
}
```



Обробка кількох винятків. Проект ProcessMultipleExpceptions

- в реальности часто приходится сталкиваться с ситуациями, когда операторы внутри блока try могут генерировать несколько исключений.
  - добавим в проект файлы Car.cs, Radio.cs и CarIsDeadException.cs из предыдущего примера CustomException
  - Далее изменим метод Accelerate () класса Car так, чтобы он генерировал еще и предопределенное в библиотеках базовых классов исключение ArgumentOutOfRangeException, если передается недопустимый параметр (которым будет считаться любое значение меньше нуля).
  - Обратите внимание, что конструктор этого класса исключения принимает имя проблемного аргумента в первом параметре string, за которым следует сообщение с описанием ошибки.

```
// Проверить аргумент на предмет допустимости перед продолжением.
public void Accelerate(int delta)
{
  if(delta < 0)
    throw new
      // Скорость должна быть больше нуля!
      ArgumentOutOfRangeException("delta", "Speed must be greater than zero!");
  ...
}
```

Тепер логіка в блоці catch може специфічно реагувати на кожний тип винятку:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Arg вызовет исключение выхода за пределы диапазона.
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

- При написании множества блоков catch следует иметь в виду, что когда исключение сгенерировано, оно будет обрабатываться «первым доступным» блоком catch.

Ілюстрація «першого доступного» блоку catch

```
// Этот код не скомпилируется!
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Вызвать исключение выхода за пределы диапазона аргумента.
    myCar.Accelerate(-10);
  }
  catch(Exception e)
  {
    // Обработать все остальные исключения?
    Console.WriteLine(e.Message);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

модифицируем предыдущий код, добавив еще один блок catch, который пытается обработать все исключения кроме CarIsDeadException и ArgumentOutOfRangeException, перехватывая общий тип System.Exception:

- Эта логика обработки исключений приводит к ошибкам на этапе компиляции.
- Проблема в том, что первый блок catch может обрабатывать любые исключения, производные от System.Exception (с учетом отношения “является”), в том числе CarIsDeadException и ArgumentOutOfRangeException.
- Следовательно, два последних блока catch являются недостижимыми!

Емпіричне правило

```
// Этот код скомпилируется без проблем.
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Вызвать исключение выхода за пределы диапазона аргумента.
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  // Это будет перехватывать любые исключения кроме
  // CarIsDeadException и ArgumentOutOfRangeException.
  catch (Exception e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

- необходимо обеспечить такое структурирование блоков catch, при котором самый первый catch должен перехватывать наиболее специфическое исключение
  - производный тип, расположенный позже всех в цепочке наследования типов исключений), а последний catch — наиболее общее исключение (т.е. базовый класс всей цепочки наследования, в данном случае — System.Exception).
  - Таким образом, если необходимо определить блок catch, который будет обрабатывать любые ошибки кроме CarIsDeadException и ArgumentOutOfRangeException, можно написать следующий код:

Загальні оператори catch

```
// A generic catch.
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    myCar.Accelerate(90);
  }
  catch
  {
    Console.WriteLine("Something bad happened...");
  }
  Console.ReadLine();
}
```

- В C# также поддерживается “общий” контекст catch, который не получает явно объект исключения, сгенерированный заданным членом:
  - не самый информативный способ обработки исключений, т.к. нет никакой возможности для получения значащих данных о возникшей ошибке (вроде имени метода, стека вызовов или специального сообщения).
  - Тем не менее, C# разрешает такую конструкцию, поскольку она может оказаться полезной, когда требуется обработать все ошибки в чрезвычайно общей манере.

Повторна генерація винятку

```
// Передача ответственности.
static void Main(string[] args)
{
...
  try
  {
    // Логика увеличения скорости автомобиля...
  }
  catch(CarIsDeadException e)
  {
    // Выполнение любую частичную обработку этой ошибки
    // и передать ответственность далее.
    throw;
  }
  ...
}
```

- Когда исключение перехватывается, внутри блока try разрешено повторно сгенерировать его для передачи вверх по стеку вызовов предшествующему вызывающему коду.
  - Для этого нужно просто воспользоваться ключевым словом throw в блоке catch.
  - Исключение будет передано вверх по цепочке логики вызовов, что может оказаться полезным в ситуации, если блок catch способен обработать текущую ошибку лишь частично
- Следует иметь в виду, что в приведенном выше примере кода конечным получателем исключения CarIsDeadException будет среда CLR, т.к. в методе Main () оно было сгенерировано повторно.
  - По этой причине конечному пользователю будет отображаться системное диалоговое окно с информацией об ошибке.
- Обычно повторная генерация частично обработанного исключения для передачи вызывающему коду производится только в случае, если он имеет возможность обработать входящее исключение более элегантным образом.
- Обратите также внимание на неявную повторную генерацию объекта CarIsDeadException с помощью ключевого слова throw без аргументов. Мы не создаем здесь новый объект исключения, а просто передаем исходный объект исключения (со всей его исходной информацией).
- Это позволяет сохранить контекст первоначального целевого объекта.

Внутрішні винятки

- исключение может быть сгенерировано во время обработки другого исключения.
  - Например, предположим, что осуществляется обработка исключения CarIsDeadException внутри определенного контекста catch, и в ходе этого процесса предпринимается попытка записать данные трассировки стека в файл carErrors.txt на диске C: (для доступа к таким ориентированным на ввод-вывод типам в директиве using должно быть указано пространство имен System.IO):

```
catch(CarIsDeadException e)
{
  // Попытаться открыть файл carErrors.txt на диске C:.
  FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
  ...
}
```

  - Теперь, если указанный файл на диске C: отсутствует, то вызов File.Open() приведет к генерации исключения FileNotFoundException.

Внутрішні винятки

```
catch (CarIsDeadException e)
{
  try
  {
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
  }
  catch (Exception e2)
  {
    // Сгенерировать исключение, которое записывает новое
    // исключение, а также сообщение первого исключения.
    throw new CarIsDeadException(e.Message, e2);
  }
}
```

- Если во время обработки исключения возникает еще одно исключение, то согласно наилучшим практическим рекомендациям, необходимо сохранить новый объект исключения как “внутреннее исключение” в новом объекте того же типа, что и у исходного исключения.
  - Причина, по которой необходимо выделять новый объект для обрабатываемого исключения, связана с тем, что единственным способом документирования внутреннего исключения является параметр конструктора.
- Обратите внимание, что в этом случае конструктору CarIsDeadException во втором параметре передается объект FileNotFoundException.
- После конфигурирования этот объект передается вверх по стеку вызовов следующему вызывающему коду, которым будет метод Main().
- С учетом того, что после Main() нет “следующего вызывающего кода”, который мог бы перехватить исключение, пользователю будет отображаться системное диалоговое окно с сообщением об ошибке.
  - Во многом подобно повторной генерации исключения, запись внутренних исключений обычно полезна только тогда, когда вызывающий код способен обрабатывать данное исключение более элегантно.
  - В этом случае внутри логики catch вызывающего кода с помощью свойства InnerException можно извлечь детали объекта внутреннего исключения.

Блок finally

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  myCar.CrankTunes(true);
  try
  {
    // Логика, связанная с увеличением скорости автомобиля.
  }
  catch(CarIsDeadException e)
  {
    // Обработать CarIsDeadException.
  }
  catch(ArgumentOutOfRangeException e)
  {
    // Обработать ArgumentOutOfRangeException.
  }
  catch(Exception e)
  {
    // Обработать любой другой объект Exception.
  }
  finally
  {
    // Это код будет выполняться всегда, возникало исключение или нет.
    myCar.CrankTunes(false);
  }
  Console.ReadLine();
}
```

- У контексті try/catch можна також визначати необов'язковий блок finally.
  - Він гарантує, що заданный внутри него набор операторов будет выполняться всегда, независимо от того, возникло исключение (любого типа) или нет.
- При отсутствии блока finally в случае возникновения исключения радиоприемник не выключался бы (что может как являться, так и не являться проблемой).
  - В более реалистичном сценарии, когда необходимо освободить объекты, закрыть файл либо отключиться от базы данных (или чего-то подобного), блок finally представляет собой идеальное место для выполнения надлежащей очистки.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: