Notes and Coding Solutions:

Week 1

Understanding a Porgramme/Software

A Software is a computer program, designed to perform a well-defined function. A program is a sequence of instructions written to solve a particular problem.

There are two types of software −

- System Software - a collection of programs designed to operate, control, and extend the processing capabilities of the computer itself.
    - Generally prepared by the computer manufacturers
    - Examples include Operating systems such as Microsoft Windows, Mac (for Apple devices), and Linux for desktop computers, laptops and tablets:
- Application Software - also referred to as 'apps' aid users in performing their tasks. Examples include
    - **MS Excel**: It is spreadsheet software that you can use for presenting and analyzing data.
    - **Photoshop**: It is a photo editing application software by Adobe. You can use it to visually enhance, catalogue and share your pictures.
    - **Skype:** It is an online communication app that you can use for video chat, voice calling and instant messaging.

How is the program developed?

- One of the fundamental procedures of developing software in a step by step manner is by following the Software Development Life Cycle (SDLC) or sometimes called the Software Development Process.
- The Software Development Life Cycle (SDLC) is a ***process*** used by the software industry to ***design, develop and test*** high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.
- SDLC acts as a ***framework*** that holds some specific tasks, shown in Figure 2 to be achieved at every phase during the software development progression.
- The main purpose is to have a standard that defines all the tasks required for developing and maintaining software

The .NET Framework:

- .NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications.
- C# is are programming language designed for creating a variety of applications that run on the .NET Framework. These languages are powerful, type-safe, and object-oriented
- It includes an interactive development environment, visual designers for building Windows and Web applications, a compiler, and a debugger.

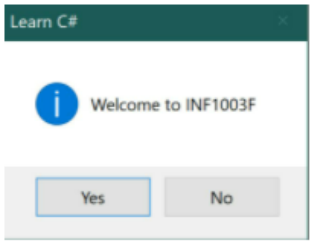
Getting to understand the environment:

- We will open a new project - learn to start and properly save a program!
    - Toolbox - displays controls that you can add to Visual Studio projects
    - To open Toolbox, choose View > Toolbox from the menu bar, or press Ctrl+Alt+X.
    - You can drag and drop different *controls* onto the surface of the *designer* you are using, and resize and position the controls.

      Toolbox appears in conjunction with designer views, such as the designer view of a XAML file or a *Windows Forms App project*. Toolbox displays only those controls that can be used in the current designer. You can search within Toolbox to further filter the items that appear.

Display Output using a Message box:

- A MessageBox is a predefined dialogue box that displays application-related information to the user.
- Message boxes are also used to request information from the user.
- They can contain text, buttons, and symbols that inform and instruct the user.
- You can go to the lecture videos to see ***Example 2: Show the message dialogue box presented below***




```
MessageBox.Show("Welcome to INF1003F","Learn C#", MessageBoxButtons.YesNo, MessageBoxIcon.Information);
```

- String Text - to show a message on a button click event handler = ***"Welcome to INF1003F"***
- Title/caption of the messagebox = ***"Learn C#"***
- Messagebox buttons to be displayed = Yes/No
- MessageboxIcon to be displayed = The ***i*** icon

Concatenation:

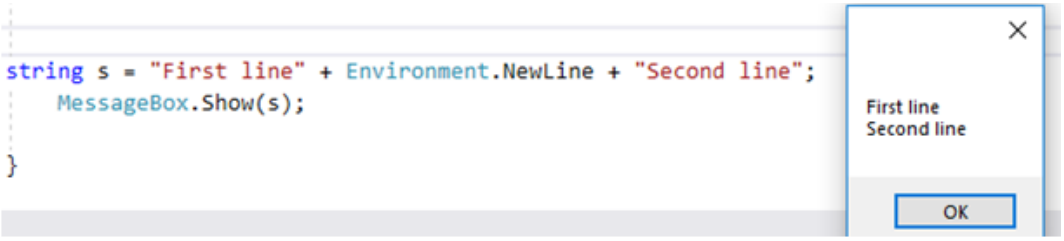
- *Concatenation* is the process of appending one string to the end of another string.
- You concatenate strings by using the + operator.
- For string literals and string constants, concatenation occurs at compile time; no run-time concatenation occurs. For string variables, concatenation occurs only at run time.

- To concatenate string variables, you can use the + or += operators, string interpolation or
  the String.Format, String.Concat, String.Join or StringBuilder.Append methods.
- The + operator is easy to use and makes for intuitive code. Even if you use several + operators in one statement, the string content is copied only once. The following code shows examples of using the + and += operators to concatenate strings:

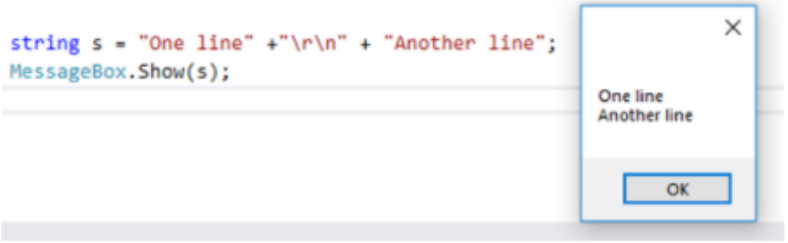
```
string str = "Hello " + userName + ". Today is " + dateString
+ ".";
```

Line Breaks:

- **Environment.NewLine.** A newline on Windows equals *\r\n*.
- To add newlines in a C# string, there are a couple options—and the best option may depend on your program.
- Example: create a string with a line break in the middle of it, which will form a 2-line string.

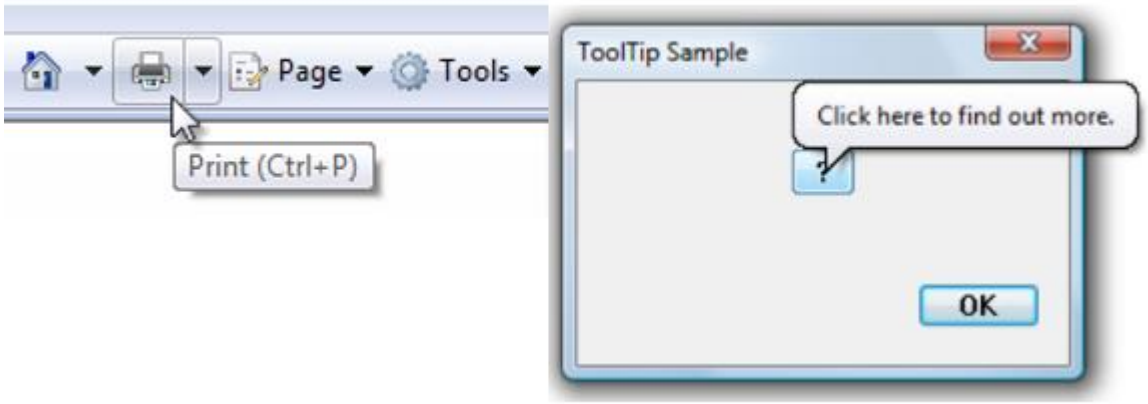


- The same code can be written in the following manner



Tooltips:

- A tooltip is a small pop-up window that labels the unlabeled control being pointed to, such as unlabeled toolbar controls or command buttons. It can contain a single object of any type (such as a string, an image, or a panel).
- You use a ToolTip control to provide information to the user.
- The content of a ToolTip control can vary from a simple text string to more complex content such as a StackPanel that has embedded text and images.
- You can use a ToolTip on multiple elements
- The properties of the ToolTip class are used to define the position and behavior of the tooltip.
- for more see msdn
- ***See Example*** in the Lecture videos ***to understand how tooltip work.***



How to add navigation rules:

Key board access keys

- As users become familiar with an application, they are more likely to use access keys to speed up common operations.
- This tendency is even more common among assistive technologyusers.
- Access keys are useful in this complex process, especially when a group has a lot of controls.
- Access keys are keys or key combinations used for accessibility to interact with all controls or menu items using the keyboard.
- Shortcut keys are keys or key combinations used by advanced users to perform frequently used commands for efficiency.
- Windows indicates access keys by underlining the access key assignment.

- *you can assign access keys for common controls by placing an ampersand (&) before the letter assigned to the access key.*
- To view your program, press ALT and the letter used as an access key

See Example in the Lecture videos ***to understand how to implement keyboard access keys.***

Change the colour of the control:

- We can change the font colour of the control such as a button as well as the background colour.
- A Button is an essential part of an application, or software, or webpage. It allows the user to interact with the application or software. In Windows form, you are allowed to set the background color of the button with the help of BackColor property. This property is provided by Button class and helps programmers to create more good looking buttons in their forms.
- Follow *Example* in the Lecture videos to see how to change the control's colour.

Week 2

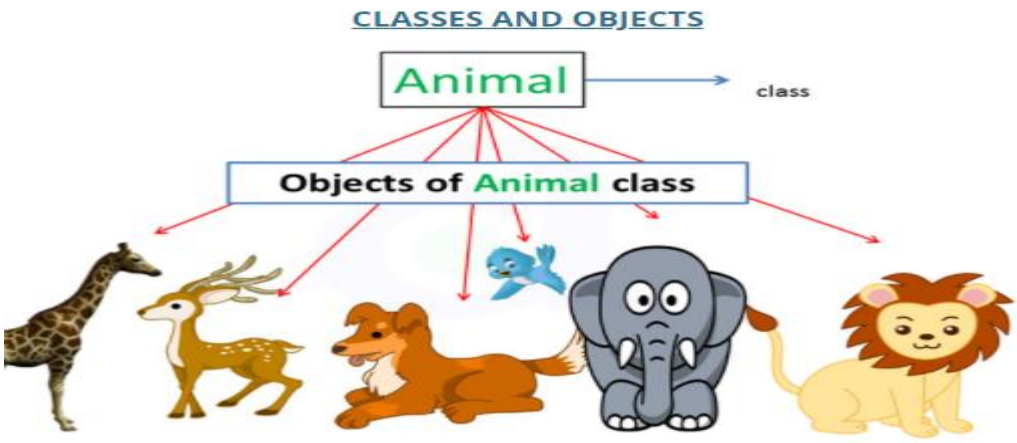
Class and Object Concepts:

An *object* is a self-contained unit like a form or control that combines code and data. Read here about Object-Oriented Programming (C#)

A *class* is the code that defines the characteristics of an object. You can think of a class as a template for an object.

An object is an *instance* of a class, and the process of creating an object from a class is called *instantiation*.

More than one object instance can be created from a single class.



Property, Method and Event Concepts:

- *Properties* define the characteristics of an object and the data associated with an object.
- *Methods* are the operations that an object can perform.
- *Events* are signals sent by an object to the application telling it that something has happened that can be responded to.
- Properties, methods, and events can be referred to as *members* of an object.
- If you instantiate two or more instances of the same class, all of the objects have the same properties, methods, and events. However, the values assigned to the properties can vary from one instance to another.

Objects and Forms:

·When you use the Form Designer, Visual Studio automatically generates C# code that creates a new class based on the Form class. Then, when you run the project, a form object is instantiated from the new class.

·When you add a control to a form, Visual Studio automatically generates C# code in the class for the form that instantiates a control object from the appropriate class and sets the control's default properties.

·When you move and size a control, Visual Studio automatically sets the properties that specify the location and size of the control

Member and Property:

1.To display a list of the available members for a class or an object, type the class or object name followed by a period (called a *dot operator*, or just *dot*).

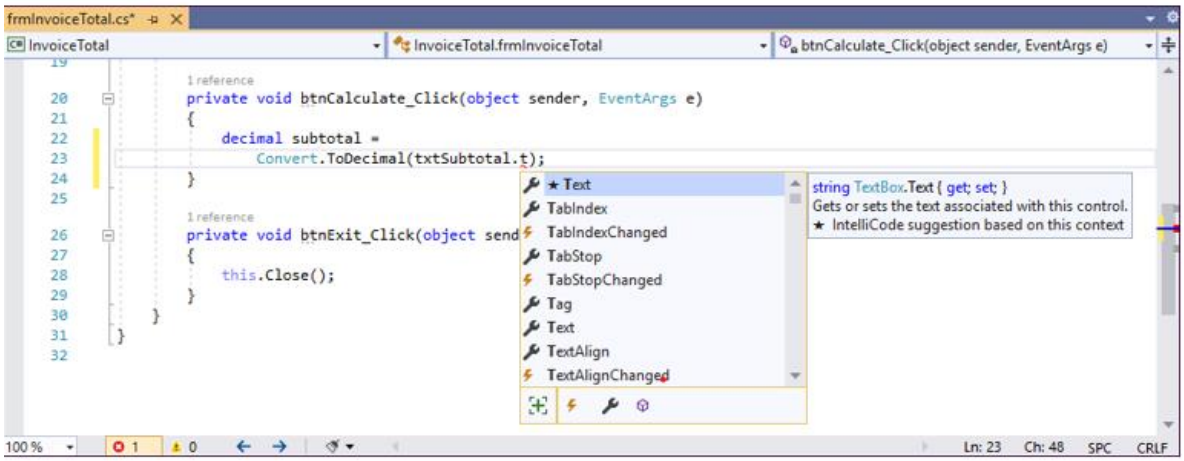
2.Type one or more letters of the member name, and the Code Editor will select the first entry in the list that matches those letters. Or, you can scroll down the list to select the member you want.

3.Press the Tab or Enter key to insert the member into your code.

Example Syntax referring to a member of a class or object

```
ClassName.MemberName

objectName.MemberName
```

A member List in code Editor window



Statement that refer to properties

`txtTotal.Text = "10";`	Assigns a string holding the number 10 to the Text property of the text box named txtTotal.
`txtTotal.ReadOnly = true;`	Assigns the true value to the ReadOnly property of the text box named txtTotal so the user can't change its contents.

Method and Events:

Note that names of methods are always followed by parenthesis

`txtMonthlyInvestment.Focus();`	Uses the Focus() method to move the focus to the text box named txtMonthlyInvestment.
`this.Close();`	Uses the Close() method to close the form that contains the statement. In this example, this is a keyword that is used to refer to the current instance of the class.

Events:

·Windows Forms applications work by responding to events that occur on objects.

·To indicate how an application should respond to an event, you code an *event handler*, which is a special type of method that handles the event.

·To connect the event handler to the event, Visual Studio automatically generates a statement that wires the event to the event handler. This is known as *event wiring*.

·An event can be an action that's initiated by the user like the Click event, or it can be an action initiated by program code like the Closed event.

Statements that refers to Events

`btnExit.Click`	Refers to the Click event of a button named btnExit.

Wiring: The application determines what method to execute

```
this.btnExit.Click +=
    new System.EventHandler(this.btnExit_Click);
```

Response: The method for the Click event of the Exit button is executed

```
private void btnExit_Click(object
sender, System.EventArgs e)
{
  this.Close();
}
```

How to comment:

·*Comments* are used to help document what a program does and what the code within it does.

·To code a *single-line comment*, type // before the comment. You can use this technique to add a comment on its own line or to add a comment at the end of a line.

·To code a *delimited comment*, type /* at the start of the comment and */ at the end. You can also code asterisks to identify the lines in the comment, but that isn't necessary.

Data Variables:

- are memory/storage locations that hold data that can be changed during project execution.
- Every variable has a type that determines what values can be stored in the variable
- Used in programming for storing values, calculating and for reusability

Int i = 4;

Declaring a Variable:

- What you need: *identifier*, *data type* and possibly an *operator* to assign a *value*
- *Identifier:* When you declare a variable, the compiler reserves an area of memory and assigns it a name, called an *identifier.*

- **Syntax**

```
type variableName;
variableName = value;
```

Example

```
int counter;
counter = 1;
```

```
int studentCount;        // number of students in the cl
int numberOfExams;       // number of exams
int coursesEnrolled;     // number of courses enrolled
```

How to declare and initialize a variable in one statement Syntax

```
type variableName = value;
```

Rules for creating an identifier

- Combination of alphabetic characters (a-z and A-Z), numeric digits (0-9), and the underscore
- First character in the name may not be numeric
- No embedded spaces – concatenate (append) words together
- Keywords cannot be used
- Use the case of the character to your advantage
- Be descriptive with meaningful names
- Follow naming conventions / general naming conventions
- Naming convention

  Start the names of variables with a lowercase letter, and capitalize the first letter of each word after the first word. This is known as ***camel notation***

- Therefore we can see that the *syntax* of declaring a variable is as follows: DataType VariableName;

Data types

- The type of data that a variable declaration specifies and defines what type of information will be stored in the allocated memory space.
- Each time a variable is declared, C# requires that the variable data type is specified. This is because C# is a strongly typed language. Therefore prior to using a variable, the compiler must be aware of it….aware of what type of data is expected (msdn)

- A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

Description	Identifier	Data type	Data
Grade point average	`gradePointAverage`	`double`	3.99
Current age	`age`	`int`	19
Full name	`studentName`	`string`	Elizabeth Hill
Final grade in a course	`courseGrade`	`char`	A

Operators:

- You can store a value inside a variable when you declare it. *This is called initializing the variable.*
- Operators are used to perform operations on variables and values.
- The main operator used frequently is the Assignment operator.
    - The assignment operator = assigns the value of its right-hand operand to a variable, a property, or an indexer element given by its left-hand operand ([msdn]).
    - The result of an assignment expression is the value assigned to the left-hand operand.
    - The type of the right-hand operand must be the same as the type of the left-hand operand or implicitly convertible to it.
    - To store a value in a variable you must use an =. Here is an example of how to store the value 5 inside an integer variable
    - Example

```
int x = 100 + 50;  // we use the + operator to add together two values
```

```
//Although the +operator is often used to add together two values, like in the example above,
//it can also be used to add together a variable and a value, or a variable and another variable:
int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 = sum1 + 250;      // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

Arithmetic Operators:

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

Constants:

- Constants are fields whose values are set at compile time and can never be changed. Use constants to provide meaningful names instead of numeric literals ("magic numbers") for special values.
- Memory locations that hold data that *cannot be changed during project execution*
- declared using the keyword CONST and are given a name, a data type, and a value
- Syntax: `const type identifier = expression;`
- Example:
    - public const double Pi = 3.14159; public const int SpeedOfLight = 300000; // km per sec.
    - Multiple constants of the same type can be declared at the same time, for example:

```
class Calendar2
   {
      public const int Months = 12, Weeks = 52, Days = 365;
   }
```

Casting:

- Type casting is when you *assign* a value of one data type to another type.
- In C#, there are two types of casting:
    - **Implicit Casting** (automatically) - converting a smaller type to a larger type size `char` -> `int` -> `long` -> `float` -> `double`
        - This is done automatically when passing a smaller size type to a larger size type....i.e For built-in numeric types, an implicit

conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off (msdn).

- Example:

```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double

Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9
```

- **Explicit Casting** (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char

- Example

```
double myDouble = 9.78;
int myInt = (int) myDouble;    // Manual casting: double to int

Console.WriteLine(myDouble);   // Outputs 9.78
Console.WriteLine(myInt);      // Outputs 9
```

Exercise 1:

Write a program that accepts two numbers from the user; and then calculates the product of those numbers

Solution:

Under the calculate button, code:

```
{
            int Fnumber = int.Parse(FirstNumText.Text);
            int  Snumber = int.Parse(secondNumText.Text);
            int product = Fnumber * Snumber;

            MessageBox.Show("The product of the first number : " + Fnumber.ToString()
+ Environment.NewLine +
                            " and the second number: " + Snumber.ToString() +
                            " is: " + product.ToString(), "Product of two numbers",
MessageBoxButtons.OK, MessageBoxIcon.Information);

   }
```

Exercise 2:

Write a program that calculates the area of a circle and produces the following output - check the GUI requirements

Solution: under the get area button code,

```
{

            double radius = double.Parse(radiusText.Text);
            double areaCircle = pi * radius * radius;

            MessageBox.Show("A circle with the radius: " + radius.ToString() +
                Environment.NewLine + "has the area: " + areaCircle.ToString(), "Area
of a circle", MessageBoxButtons.OK, MessageBoxIcon.Information);

   }
```

Formatting Output:

- You can format data by adding dollar signs, percent symbols and/or commas to separate digits
- You can suppress leading zeros
- You can pad a value with special characters, by placing characters to the left or right of the significant digits
- Use format specifiers.

Character	Description	Examples	Output
C or c	Currency	`Console.Write("{0:c}", 26666.7888);`	$26,666.79
C or c	Currency	`Console.Write("{0:c}", -2);`	($2.00)
C or c	Currency	`Console.Write("{0:c}", 38.8);`	$38.80
F or f	Fixed point	`Console.Write("{0:F4}", 50);`	50.0000
F or f	Fixed point	`Console.Write("{0:F0} {1}", 50, 22);`	50 22

Week 3

Why make decisions:

- In management, decision making refers to ***making choices among alternative courses of action***—which may also include inaction. While it can be argued that management *is* decision making, half of the decisions made by managers within organizations fail. Therefore, increasing effectiveness in decision making is an important part of maximizing your effectiveness at work.
- In programming, decisions are an essential part of executing a specific block of code based on the fulfillment of the condition.

Boolean Expressions:

When you code an expression that evaluates to a true or false value, that expression can be called a **Boolean expression.** Because you use **Boolean expressions** within the control structures you code, you need to learn how to code Boolean expressions before you learn how to code control structures.

RELATIONAL OPERATORS

We use relational operators to code a Boolean expression. These operators let you compare two operands, as illustrated by the examples below. An operand can be any expression, including a variable, a literal, an arithmetic expression, or a keyword such as null, true, or false.

To compare two operands for equality, make sure you use two equal signs. If you use a single equals sign, the compiler will interpret it as an assignment statement, and your code won't compile.

When comparing strings, you can only use the equality and inequality operators. If you compare two numeric operands with different data types, C# will cast the less precise operand to the type of the more precise operand.

Operator	Name
==	Equality
!=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Example

```
firstName == "Frank"      // equal to a string literal
txtYears.Text == ""       // equal to an empty string
```

```
message == null          // equal to a null value
discountPercent == 2.3   // equal to a numeric literal
isValid == false         // equal to the false value
code == productCode      // equal to another variable

lastName != "Jones"      // not equal to a string literal

years > 0                // greater than a numeric literal
i < months               // less than a variable
subtotal >= 500          // greater than or equal to
                         // a literal value
quantity <= reorderPoint  // less than or equal to a variable
```

Logical Operators:

Operator	Name	Description
&&	Conditional-And	Returns a true value if both expressions are true. This operator only evaluates the second expression if necessary.
||	Conditional-Or	Returns a true value if either expression is true. This operator only evaluates the second expression if necessary.
&	And	Returns a true value if both expressions are true. This operator always evaluates both expressions.
|	Or	Returns a true value if either expression is true. This operator always evaluates both expressions.
!	Not	Reverses the value of the expression.

We use the logical operators to code a compound Boolean expression that consists of two or more Boolean expressions.

For example

```
subtotal >= 250 && subtotal < 500

timeInService <= 4 || timeInService >= 12
```

The first compound expression above uses the && operator. As a result, it evaluates to true if both the expression before the && operator and the expression after the && operator evaluate to true. Conversely, the second compound expression uses the || operator. As a result, it evaluates to true if either the expression before the || operator or the expression after the || operator evaluates to true. When you use the && and || operators, the second expression is only evaluated if necessary.

Since the && and || operators only evaluate the second expression if necessary, they're sometimes referred to as short-circuit operators. These operators are slightly more efficient than the & and | operators.

By default, Not operations are performed first, followed by And operations, and then Or operations. These operations are performed after arithmetic operations and relational operations. You can use parentheses to change the sequence in which the operations will be performed or to clarify the sequence of operations.

ADDITIONAL EXAMPLES

Conditional AND operator &&

- If one of the operands is false, the AND operator will evaluate it to false i.e *Returns true both operands are true, otherwise returns false*.
- Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then (A && B) is false.

Expression1	Expression2	operand1 && operand2
true	true	true
true	false	false
false	true	false
false	false	false

Conditional Logical OR operator ||

- If one of the operand is true, the OR operator will evaluate it to true. i.e *Returns true when any one operand is true*.
- Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then ***(A || B) is true***.
- See example in Table 5.5

Table 5-5 Conditional logical OR ||

Expression1	Expression2	operand1 || operand2
true	true	true
true	false	true
false	true	true
false	false	false

Logical Negation (Not)

- Inverts the value of a boolean
- The unary prefix `!` operator computes logical negation of its operand.
- That is, it produces `true`, if the operand evaluates to `false`, and `false`, if the operand evaluates to `true`

```
bool passed = false;

!passed  // output: True
```

!true // output: False

- In short, the ! operator ***returns false if result is true and returns true if result is false. See examples in Table 5-6***

Table 5-6 Logical NOT !

Expression1	! operand1
true	false
false	true

Exercise 1

Write a program that accepts two numbers and then uses logical and conditional operators on the entered numbers to determine the output

Solution:

Code under the compare button:

```
{
            decimal firstNumber = Convert.ToDecimal(txtFirstNumber.Text);
            decimal secondNumber = Convert.ToDecimal(txtSecondNum.Text);

            bool result1, result2;









            result1 = firstNumber > secondNumber;
            result2 = secondNumber > firstNumber;

            MessageBox.Show("Is the first number greater than the second number?" +
result1 +
          Environment.NewLine + "Is the second number greater than the first number?"
+ result2);
}
```

IF Statements:

Now that you know how to code Boolean expressions, you're ready to learn how to code conditional statements and expressions. That includes the if-else and switch statements, the switch expression, and expressions that use the conditional operator. In this section, you'll learn how to code the if statement statement. if statement is the C# implementation of a control structure known as the **selection structure** because it lets you select different actions based on the results of Boolean expression.

The syntax of the if-else statement

```
if (booleanExpression) { statements }
```

```
[else if (booleanExpression) { statements }] ...

[else { statements }]
```

Note that : In the syntax above, the brackets [ ] indicate that a clause is optional, and the ellipsis (...) indicates that the preceding element can be repeated as many times as needed. In other words, this syntax shows that you can code an if clause with or without else if clauses or an else clause. It also shows that you can code as many else if clauses as you need. Let us see how to code different if statements.

The *if statement*

- An if statement consists of a *boolean expression* which is evaluated to a *boolean value*.
- If the value is true then ***if block is executed*** otherwise ***next statement(s) would be executed***.
- When the expression evaluates to false, the statement following expression is skipped or bypassed
- No special statement(s) is included for the false result
- No semicolon placed at end of an expression

For Example:

With a single statement

```
if (numberGrade > 88)
     letterGrade = "A";
```

With a block of statements

```
if (numberGrade > 88)
  {
 letterGrade = "A";
  status = "Excellent Work!";
  }
```

The if-else statement

- Either the true statement(s) executed or the false statement(s), but not both
- If the "if" condition is not true, the program control goes to the "else" condition.

```
if (condition)
```

```
{// if condition is true
}
else
{
// if the condition becomes false
}
```

An if statement with an else clause

```
if (subtotal >= 100)
 discountPercent = .2m;
else
 discountPercent = .1m;
```

The Nested if statement

- In this type of conditional statement, one if statement will be nested or inside another if or else statement.
- Here multiple conditions have to be evaluated as true only then the nested block associated with multiple "if conditions" will be executed

```
if (customerType == "R")
    {
   if (subtotal >= 100)       // begin nested if
       discountPercent = .2m;
   else
       discountPercent = .1m;          // end nested if
    }
else    // customerType isn't "R"
 discountPercent = .4m;
```

-
  - Acceptable to write an if within an if
  - When block is completed, all remaining conditional expressions are skipped or bypassed
  - Difference: With a nested if…else, the statement may be another if statement
  - No restrictions on the depth of nesting

- Limitation comes in the form of whether you and others can read and follow your code

Exercise 2:

Write a program that allows the user to input two values. Determine the largest of the two values (**Please Do not use Math.Max/Min(a,b) option**). Then print the largest and its square root.

Solution:

Code under the calculate button:

```
int firstNumber = int.Parse(txtFirstNum.Text);
            int secondNumber = int.Parse(txtSecondNum.Text);

            int largestOne;

            if (firstNumber>secondNumber)
            {
                largestOne = firstNumber;
            }

            else
            {
                largestOne = secondNumber;
            }

             MessageBox.Show("The largest value entered was:" + largestOne +
           Environment.NewLine + " Its square root is:" +
Math.Round(Math.Sqrt(largestOne), 2));
```

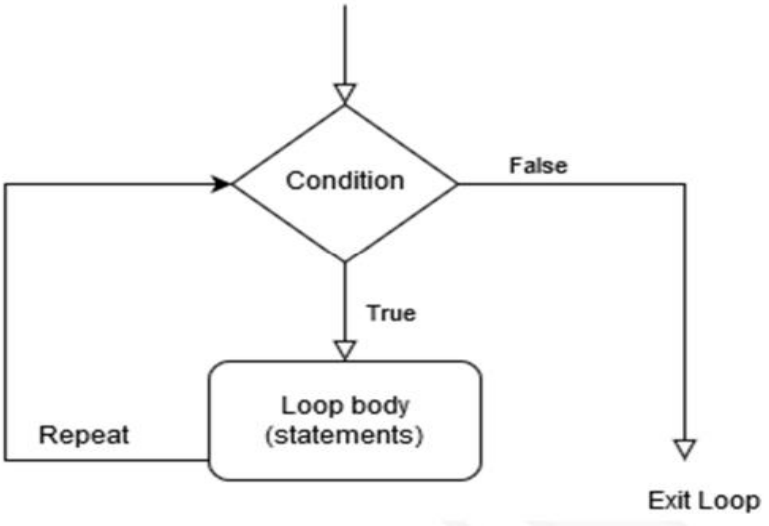
Week 4: Control Structures

Loops

- Loops are regarded as an essential technique when writing software
- Looping = the ability to repeat a block of code for a specified number of times.
- Loops = repeated execution of a sequence of operations/ fragment of source code *until a given condition is true (exists).*
- a loop is controlled by a *boolean expression* that determines how many times the statement will be executed
- Loops that never end are called infinite loops

The syntax of the while statement – Pretest

```
while (booleanExpression)
{
   statements
}
```



The syntax of the while statement

```
while (booleanExpression)
{
    statements
}
```

A while loop that adds the numbers 1 through 4

```
int i = 1, sum = 0;
while (i < 5)
{
    sum += i;
    i++;
}
```

A while loop that calculates a future value

```
int i = 1;
while (i <= months)
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
    i++;
}
```

- The While loop
- Used for iterative purposes... to repeat a certain set of statements for a particular number of times
- Syntax:

While(Boolean Condition)

{

//code block / loop body//

}

- the loop condition is any expression that returns a *Boolean result – true or false*. The condition determines how long the loop body will be repeated. This Boolean expression is calculated and if it is true the sequence of operations in the body of the loop is executed. Point to remember: *The boolean statement is validated before the execution of the code*.

- the loop body: is the programming code executed at each iteration of the loop, i.e. whenever the input condition is true. The body of the while loop

may not be executed even once, if in the beginning, the condition of the cycle returns false.

- If the condition of the cycle is never broken the loop will be executed indefinitely and result in an infinite loop.

- Example: *A while loop that adds the numbers 1 through 4*

```
int i = 1, sum = 0;
    while (i < 5)
    {
       sum += i;
       i++;
    }
```

Side note: The variable i (known as the *loop control variable*) is used in three ways: it is initialized, tested, and updated.

- The do-while
- Performs the same role as a while loop except that it *executes the code block at least once*.
- Syntax

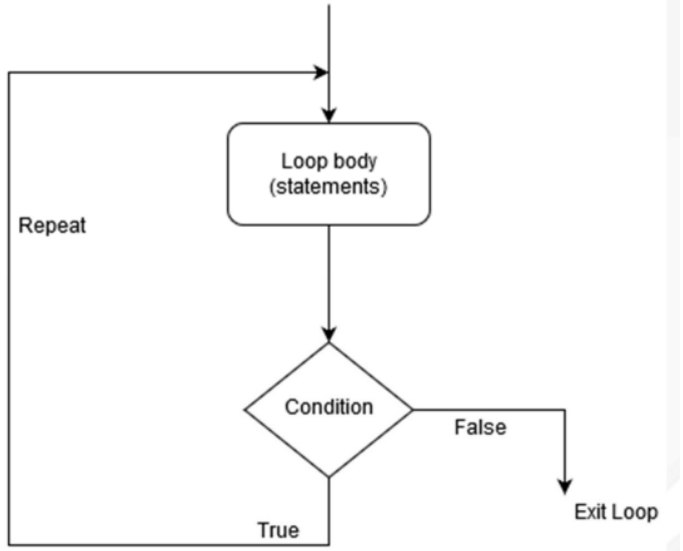
```
do
    {
      statements
    }
    while (booleanExpression);
```

- The do while loop stops execution when a boolean condition, the while(condition) specified at the end of the block, evaluates to false.
- As a result, it executes the code block at least once....because the conditional expression appears at the end of the loop. If the condition is true, the flow of control jumps back up to the do statement and the statement(s) in the loop execute again.

- Tests the condition at the end of the loop body and executes the loop once irrespective of whether the condition is true or not.
- You can use break or return to exit from the do while loop.

The syntax of the do-while statement -posttest

```
do
{
   statements
}
while (booleanExpression);
```



The syntax of the do-while statement

```
do
{
    statements
}
while (booleanExpression);
```

A do-while loop that calculates a future value

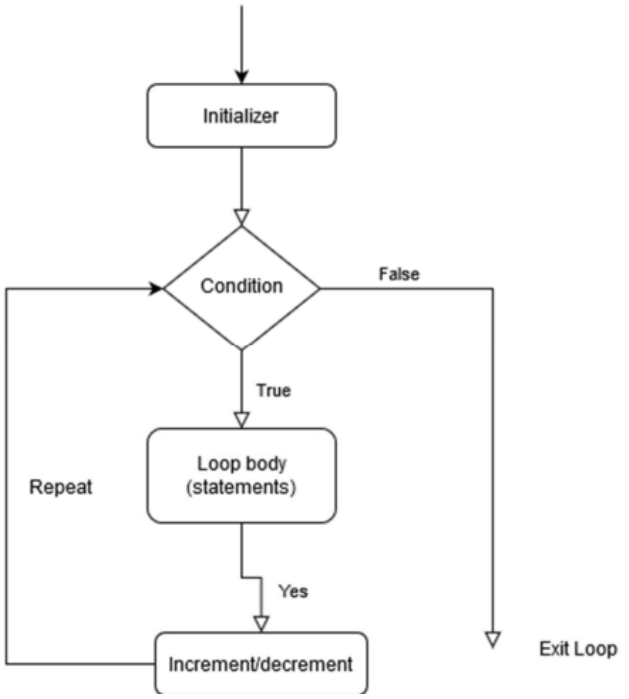
```
int i = 1;
do
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
    i++;
}
while (i <= months);
```

While vs Do While Loops

- When you use a while statement, the condition is tested before the while loop is executed. When you use a do-while statement, the condition is tested after the do-while loop is executed.
- A while or do-while loop executes the block of statements within its braces as long as its Boolean expression is true.
- If a loop requires more than one statement, you must enclose the statements in braces. Then, any variables or constants that are declared in the block have block scope. If a loop requires just one statement, you don't have to enclose the statement in braces.
- If the conditional expression never becomes false, the loop never ends. Then, the program goes into an infinite loop that you can cancel by using the Break All or Stop Debugging commands from the Debug toolbar.

The syntax of the for statement

```
for (initializationExpression; booleanExpression;
     incrementExpression)
{
   statements
}
```



- Introducing the loop
    - The for keyword indicates a loop in C# and is used to start off the 'for loop' statement that executes a block of statements repeatedly until the *specified condition returns false*.
    - Has the following three statements, separated by a semicolon:
        - Initializer: The initializer section is used to initialize a variable that will be local to a for loop and cannot be accessed outside loop. It can also be zero or more assignment statements, method call, increment, or decrement expression e.g., ++i or i++, and await expression.
        - Condition: The condition is a boolean expression that will return either true or false. If an expression evaluates to true, then it will execute the loop again; otherwise, the loop is exited.
        - Iterator: The iterator defines the incremental or decremental of the loop variable.
- Syntax

```
for (initializer; condition; iterator)
    {
      //code block
    }
```

- Example: *Use a for loop to write a program that will compute sum of first n natural numbers*

```
private void btnForLoop_Click(object sender, EventArgs e)
        {
            int sum = 0;
            int n = int.Parse(txtNum.Text);

            for (int i = 1; i <= n; i++)
            {
                sum += i;
            }
      MessageBox.Show("Sum of first " + n.ToString() +
                      " natural numbers is " + sum.ToString(),
                      "Summation of Natural numbers",MessageBoxButtons.OK,
                       MessageBoxIcon.Information);
        }
```

>>If the input is 5; when we run the program, the output will be: 15

The syntax of the for statement

```
for (initializationExpression; booleanExpression;
     incrementExpression)
{
   statements
}
```

A for loop that stores the numbers 0 through 4 in a string

With a single statement

```
string numbers = null;
for (int i = 0; i < 5; i++)
    numbers += i + " ";
```

With a block of statements

```
string numbers = null;
for (int i = 0; i < 5; i++)
{
    numbers += i;
    numbers += " ";
}
```

A for loop that adds the numbers 8, 6, 4, and 2

```
int sum = 0;
for (int j = 8; j > 0; j-=2)
{
    sum += j;
}
```

A for loop that calculates a future value

```
for (int i = 1; i <= months; i++)
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
}
```

How to work with Nested Loops

Any statement can be included within the body of a loop, which means another loop can be nested inside an outer loop. When this occurs, the inner nested loop is totally completed before the outside loop is tested a second time.

Example

```
int inner;
 string result = " ";
 for (int outer = 0; outer < 3; outer++)
        {
           for (inner = 10; inner > 5; inner--)
           {
              result += string.Format("Outer: {0} \tInner: {1} \n\n", outer,
                     inner);
           }
        } MessageBox.Show(result);
     }
```

How to work with Nested Loops

```
string result = "";

for (int outer = 0; outer < 3; outer++)
{

    int inner = 10;

    while (inner>5)
    {


        result += string.Format("Outer : {0} \t Inner: {1} \n\n", outer, inner);

        inner--; }
}
MessageBox.Show(result);


}
```

Here is how it works: The for statement using the variable inner is the executable statement for the outermost for loop. After the innermost for statement is completed, control transfers back to the update portion of the outside for loop. Here, the variable outer is updated. Another evaluation of outer occurs to determine whether outer is less than 3. Because it is less, control transfers back into the nested innermost for loop. Here, the entire for statement is executed again. Notice the identifier inner is redeclared and reinitialized to 10. The sequence of evaluating the conditional expression using the variable inner, executing the body of the loop and updating inner, which is the control variable, continues until the innermost conditional expression again evaluates to false. At that point, outer, the loop control variable in the outermost loop, is updated and the outermost conditional expression is evaluated again. This cycle continues until the outermost for loop expression evaluates to false.

The program will print the result 15 times. As shown from the output, after the variable outer is initialized to zero and evaluated to determine whether outer is less than 3, the innermost loop is executed five times. The variable inner takes on the values of 10, 9, 8, 7, and 6. When inner is updated to 5, the second for statement, in the innermost loop, evaluates to false (inner > 5). That loop is now completed.

JUMP statements

- **Break**
  - –You use a break statement to jump out of a loop.
- **Continue**
  - –You use a continue statement to jump to the start of a loop

A loop with a break statement

```
string message = null;
int i = 1;
while (i <= months)
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
    if (futureValue > 100000)
    {
        message = "Future value is too large.";
        break;
    }
    i++;
}
```

A break statement is placed inside the body of a loop to provide an immediate exit, when the break statement is reached, control immediately transfers to the statement on the outside of the loop body. The conditional expression is never evaluated again.

While loop

Example: The following example shows how the continue statement can be used to bypass sections of code and begin the next iteration of a loop. This code prints the power of 3 of odd numbers. The code used modulus %, to determine if a number's remainder when divided by 2, equals zero, and if this is true, then the number is even. if (counter%2==0) evaluates to true, the continue statement is executed, which causes the loop body to halt at that location. The remaining statements in the loop are skipped and control transfers back to increment the counter. The variable result is not appended, for that iteration through the loop.

Note: To avoid creating an infinite loop when working with continue statements; make sure your counter can be reached and your loop can eventually terminate. For example the while loop on the right result into an infinite loop.

e loop that produces results te While Loop

```
int counter = 0;

string result = "\tNumber Power of 3 \n";

while (counter < 10)
 {
  counter ++;
 if (counter%2==0)
 continue;
 result += "\t" + counter + "\t"
          + Math.Pow(counter, 3) + "\n";
 }MessageBox.Show(result, "Power of ODD numbers");
```

```
int counter = 1;

string result = "\tNumber Power of 3 \n";

 while (counter < 11)
 {
 if (counter%2==0)
 continue;
 result += "\t" + counter + "\t"
         + Math.Pow(counter, 3) + "\n";
 counter++;
 } MessageBox.Show(result, "Power of ODD numbers");
```

A loop with a continue statement

```
string numbers = null;
for (int i = 1; i < 6; i++)
{
    numbers += i;
    numbers += "\n";
    if (i < 4)
       continue;
    numbers += "Big\n";
}
```

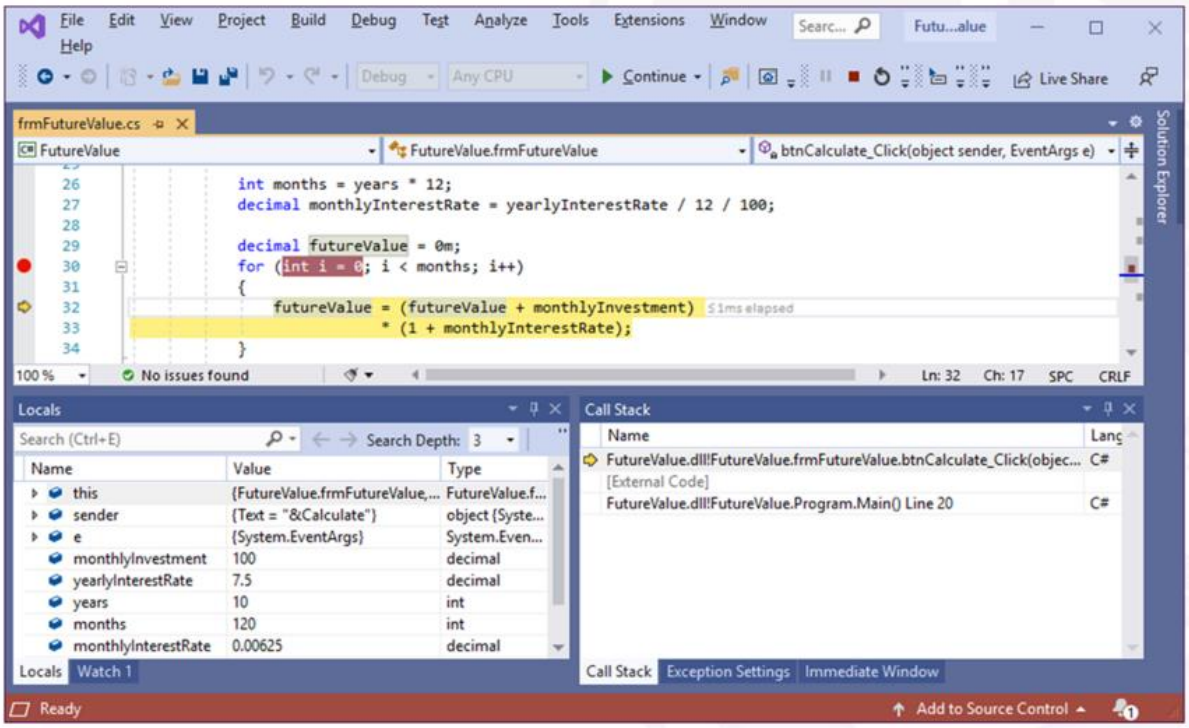
The result of this loop

```
1
2
3
4
Big
5
Big
```

A continue statement, jumps to the start of the loop, skipping any code in between. The continue statement, like the break, is a form of jump statement. When a continue is reached, a new iteration of the nearest enclosing while, do. . .while, for, or foreach statement is started. Physically, it does not matter

where the execution is. It could be at the top or bottom of the loop body. When continue is reached and executed, the rest of that iteration of the loop body is not performed. continue immediately transfers control straight to the conditional expression for evaluation.

A for loop with a breakpoint and an execution point



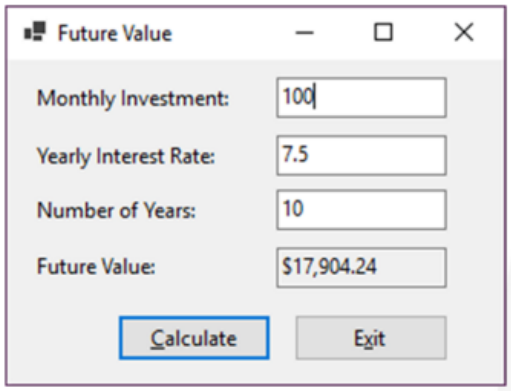
How to set and clear breakpoints

- To set a breakpoint, click in the *margin indicator bar* to the left of a statement. Or, press the F9 key to set a breakpoint at the cursor insertion point. Then, a red dot will mark the breakpoint.
- To remove a breakpoint, use any of the techniques for setting a breakpoint. To remove all breakpoints at once, use the Delete All Breakpoints command in the Debug menu.

How to work in break mode

- In break mode, a yellow arrowhead marks the current *execution point*, which points to the next statement that will be executed.
- To *step through* your code one statement at a time, press the F11 key or click the Step Into button on the Debug toolbar.
- To continue normal processing until the next breakpoint is reached, press the F5 key.

The Future Value form



The property settings for the form

Default name	Property	Setting
Form1	Text	Future Value
	AcceptButton	btnCalculate
	CancelButton	btnExit
	StartPosition	CenterScreen

The property settings for the controls

Default name	Property	Setting
label1	Text	Monthly Investment:
label2	Text	Yearly Interest Rate:
label3	Text	Number of Years:
label4	Text	Future Value:
textBox1	Name	txtMonthlyInvestment
textBox2	Name	txtInterestRate
textBox3	Name	txtYears
textBox4	Name ReadOnly TabStop	txtFutureValue True False
button1	Name Text	btnCalculate &Calculate
button2	Name Text	btnExit E&xit

The code for the event handlers in the Future Value application

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    decimal monthlyInvestment =
        Convert.ToDecimal(txtMonthlyInvestment.Text);
    decimal yearlyInterestRate = Convert.ToDecimal(txtInterestRate.Text);
    int years = Convert.ToInt32(txtYears.Text);

    int months = years * 12;
    decimal monthlyInterestRate = yearlyInterestRate / 12 / 100;

    decimal futureValue = 0m;
    for (int i = 0; i < months; i++)
    {
        futureValue = (futureValue + monthlyInvestment)
                    * (1 + monthlyInterestRate);
    }

    txtFutureValue.Text = futureValue.ToString("c");
    txtMonthlyInvestment.Focus();
}

private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

DECIDING WHICH LOOP TO USE

The decision regarding which type of loop to use is sometimes a personal choice; however, there are some considerations you might want to acknowledge and allow to overrule your personal biases.

Do While Statement

Remember that the body of the do. . .while is always executed at least once. Thus, you should avoid using this form if there is the possibility that the loop body should not be executed when certain types of data are input into your program. Conversely, if you know your loop will always be executed at least once, the do. . .while is a good option.

For Statement

If a numeric variable is being changed by a consistent amount with each iteration through the loop, the compactness of the for statement might work best. The initialization, conditional expression, and update can all be located on the same line. Although the format allows for multiple entries for all three of these entries, a good rule of thumb is that the for statement initialization, condition, and an

update should be able to be placed on one line. Remember, readability is always an important consideration.

While Statement

The while statement can be used to write any type of loop. It can implement counter-controlled loops just as the for statement can. So it is always a good option. It offers the advantage of being a pretest type. With all of the loops, you want to ensure that you understand and design the condition that is used to end the loop as well as how the condition is updated or changed.

ADVANCED LOOP STATEMENT SUGGESTIONS

As with selection statements, avoid writing compound loop conditional expressions Instead of writing <= or >=, add or subtract 1 from the endpoint (i.e., while (a >=100) is the same as while (a > 99), if a is defined as an integer variable). Additional thought will normally help you to eliminate this type of compound statement. When displaying error messages, in addition to telling what is wrong, the message should also tell what the user should do to solve the problem.

Week 5 – Methods

- A **method** is a *block of code* that contains *a series of statements* which executes or only runs to perform an action, when it is called.
- The method is called by the program - A program causes the statements to be executed by calling the method and specifying any required method arguments
- Methods allow for reusability: To reuse code - define the code once, and use it many times.

Characteristics and Syntax of a Method

- A method allows one to pass *data*, known as *parameters*
- Methods are declared in a class by specifying:
    - An optional *access level*, such as public or private. The default is private. This determines the ***visibility*** of a variable or a method from another class.
    - The *return value*, or void if the method has none. A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
    - The *method name*.
    - Any method *parameters*. Method parameters are enclosed in parentheses and are separated by commas.
        - The ***parameters are used to pass and receive data from a method***.
        - The parameter list refers to the *type, order, and number of the parameters of a method*.
        - Parameters are optional; that is, a method may contain no parameters. Empty parentheses indicate that the method requires no parameters.
    - Method *body* – This contains the *set of instructions* needed to complete the *required activity*.
- For more see the following notes
- Syntax:

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

Examples

```
[public|private] returnType MethodName([parameterList])
{
   statements
}
```

A method with no parameters and no return type

```
private void DisableButtons()
{
    btnCalculate.Enabled = false;
    btnExit.Enabled = false;
}
```

- An example of a method with one parameter

```
private decimal GetDiscountPercent(decimal subtotal)
{
    decimal discountPercent = 0m;
    if (subtotal >= 500)
        discountPercent = .2m;
    else
        discountPercent = .1m;
    return discountPercent;
}
```

A method with one parameter that returns a decimal value

```
private decimal GetDiscountPercent(decimal
subtotal)
{
    decimal discountPercent = 0m;
    if (subtotal >= 500)
        discountPercent = .2m;
    else
        discountPercent = .1m;
    return discountPercent;
}
```

```
1 reference
private string calculateLetterGrade(decimal numberGrade)
{
    string letterGrade;
    if (numberGrade >= 88)
    {
        letterGrade = "A";
    }
    else if (numberGrade >= 80 && numberGrade <= 87)
    {
        letterGrade = "B";
    }
    else if (numberGrade >= 68 && numberGrade <= 79)
    {
        letterGrade = "C";
    }
    else if (numberGrade >= 60 && numberGrade <= 67)
    {
        letterGrade = "D";
    }
    else
    {
        letterGrade = "F";
    }

    return letterGrade;
}
```

A method with three parameters that returns a decimal value

```
1 reference
private decimal Calculate(decimal operand1, string operator1, decimal operand2)
{
    decimal result = 0;
    if (operator1 == "+")
        result = operand1 + operand2;
    else if (operator1 == "-")
        result = operand1 - operand2;
    else if (operator1 == "*")
        result = operand1 * operand2;
    else if (operator1 == "/")
        result = operand1 / operand2;
    return result;
}
```

How to call Methods in your program – Invocation

- Call a method using the *name of the method* and parentheses.
- Arguments are listed within the parentheses and are separated by commas.

- The method definition specifies the names and types of any parameters that are required.
- When a caller invokes the method, it provides concrete values, ***called arguments,*** for each parameter.
- The *arguments must be compatible with the parameter type*, but the argument name, if one is used in the calling code, does not have to be the same as the parameter named defined in the method.
- In the following example, the Square method includes a single parameter of type int named *i*. The first method call passes the Square method a variable of type int named *num*; the second, a numeric constant; and the third, an expression.
- Example:

defining a method

```
int Square(int length)
 {

   return length * length;
 }
```

Invoking the method

```
private void btnSArea_Click(object sender, EventArgs e)
    {
      int sLength = int.Parse(squareLength.Text);
      int SquareArea = Square(sLength);
      MessageBox.Show("The area is :" + SquareArea.ToString(), "Area of a square",
              MessageBoxButtons.OK, MessageBoxIcon.Information);
    }
```

==========

```
[this.]MethodName([argumentList])
```

A statement that calls a method that has no parameters

```
this.DisableButtons();
```

A statement that passes one argument

```
decimal discountPercent =
this.GetDiscountPercent(subtotal);
```

A statement that passes three arguments

```
decimal futureValue = CalculateFutureValue(
    monthlyInvestment, monthlyInterestRate, months);
```

The syntax for an optional parameter

```
type parameterName = defaultvalue
```

The GetFutureValue() method with two optional parameters

```
private decimal GetFutureValue(decimal monthlyInvestment,
decimal monthlyInterestRate = 0.05m, int months = 12)
{
    decimal futureValue = 0m;
    for (int i = 0; i < months; i++)
    {
        futureValue = (futureValue + monthlyInvestment) *
                      (1 + monthlyInterestRate);
    }
    return futureValue;
}
```

A statement that passes arguments for all three parameters to the function

```
decimal futureValue =
    this.GetFutureValue(monthlyInvestment,
    monthlyInterestRate, months);
```

A statement that omits the argument for the third parameter

```
decimal futureValue =
    this.GetFutureValue(monthlyInvestment,
    monthlyInterestRate);
```

Two statements that pass the arguments for two parameters by name

```
decimal futureValue =
    this.GetFutureValue(monthlyInvestment:monthlyInvestment,
                        months:months);
decimal futureValue =
    this.GetFutureValue(months:months,
                        monthlyInvestment:monthlyInvestment);
```

A statement that passes one argument by position and one by name

```
decimal futureValue =
    this.GetFutureValue(monthlyInvestment, months:months);
```

A method that executes a single statement

```
private void DisplayErrorMessage(string
message, string caption)
{
    MessageBox.Show(message, caption);
}
```

The same method with an expression body

```
private void DisplayErrorMessage(string
message, string caption)
    => MessageBox.Show(message, caption);
```

A method that returns the value of an expression

```
1 reference
private decimal Calculate(decimal operand1, string operator1, decimal operand2)
{
    return operator1 switch
    {
        "+" => operand1 + operand2,
        "-" => operand1 - operand2,
        "*" => operand1 * operand2,
        "/" => operand1 / operand2,
        _ => 0,
    };

}
```

The same method with an expression body

```
1 reference
private decimal Calculate(decimal operand1, string operator1, decimal operand2) =>
   operator1 switch
    {
        "+" => operand1 + operand2,
        "-" => operand1 - operand2,
        "*" => operand1 * operand2,
        "/" => operand1 / operand2,
        _ =>0,            };
```

How to use refactoring to create a new method from existing code:

1. Select the code you want to create a new method from, press Ctrl + period (.), and select Extract Method from the Quick Actions menu

that's displayed. This adds a method named NewMethod() to the class, replaces the selected code with a statement that calls the method, and displays the Rename dialog box for the method name.

2. Enter the name that you want to use for the new method and click the Apply button.

How to write an event to an existing event handler:

1. Select the control in the Form Designer.
2. If necessary, click the Events button in the Properties window to display a list of the events for the control.
3. Click to the right of the event you want to handle and display the drop-down list.
4. Select the event handler you want to use for that event.

Useful Methods to know:

- Substring method: You call the Substring() method to extract a substring from a string that begins at a specified character position and ends at the end of the string. The starting character position is a zero-based; in other words, the first character in the string is at index 0, not index 1. See example

- ToLower() method: Returns a copy of this string converted to lowercase.
- ToUpper() method: Returns a copy of this string converted to uppercase. See examples of these methods here

- Trim() method: Removes all leading and trailing white-space characters from the current string. See example

Practical exercises

Exercise 1: Write a function *FindMax* that takes two integer values and returns the larger of the two. You can find the code here.

Exercise 2: Assuming the two values entered in workshop 2 are the length and width of a rectangle. Write a method to calculate the area of the rectangle and Perimeter. See the code here.

Exercise 3: Write a method to calculate Letter Grade -> the user enter a number grade and then displays the letter grade when the user clicks the Calculate button. See the code here.

Exercise 4: Design and code a project to calculate the amount due and provide a summary of rentals(see instructions here). All movies rent for R25.80 and all customers receive a 10 per cent discount.

> The form should contain input for the *member number* and the *number of movies* rented.
>
> Inside a group box, display the *rental amount*, the 10 per cent *discount*, and the *amount due* in read-only textboxes.
>
> Inside a second group box, display the *number of customers* served and the *total rental income (after discount)*.
>
> Include buttons for Calculate, Clear, and Exit
>
> The Clear button clears the information for the current rental but does not clear the summary information.
>
> Disable both input controls when *total rental income* exceeds R500.

- See the program without using a method here;
- and the same program when using a method here

Exception handling

- An exception is a problem that arises during the execution of a program.
- Exceptions in the application must be handled to prevent crashing of the program and unexpected result.
- A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- The exception handling mechanism uses three main keywords: **try**, **catch**, and **finally**
- **try** − A try block identifies a block of code for which particular exceptions is activated.
    - Any suspected code that may raise exceptions should be put inside a try{ } block.
    - During the execution, if an exception occurs, the flow of the control jumps to the first matching catch block.
    - It is followed by one or more catch blocks.
    - There may be multiple catch blocks after the try block.

- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem.
    - The catch keyword indicates the catching of an exception.
    - The catch block handles the exception that has occured in an appropriate manner using exception handling mechanisms.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown.
    - executes some statements irrespective of whether an exception has occurred or not.
    - If there is no exception in the try block, then control directly transfers to finally block and skips the catch block.
    - For example, if you open a file, it must be closed whether an exception is raised or not.
- Syntax

```
try

{
  // statements causing exception
} catch( ExceptionName e1 )

{
  // error handling code
} catch( ExceptionName e2 )

{
  // error handling code
} catch( ExceptionName eN )

{
  // error handling code
} finally

{
  // statements to be executed
}
```

Types of Exceptions

- C# exceptions are represented by classes.
- The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class.

- The Exception class is the base class of all exceptions in the .NET Framework.
- Many derived classes rely on the inherited behavior of the members of the Exception class
- See the various types of exceptions in the table below

How to Handle Exceptions and Validate Data

Objectives (part 1)

Applied

1. Given a form that uses text boxes to accept data from the user, write code that catches any exceptions that might occur.

2. Given a form that uses text boxes to accept data and the validation specifications for that data, write code that validates the user entries.

3. Use dialog boxes as needed within your applications.

Knowledge

1. Explain what an exception is and what it means for an exception to be thrown and handled.

2. Describe the Exception hierarchy and name two of its subclasses.

3. Describe the use of try-catch statements to catch specific exceptions as well as all exceptions.

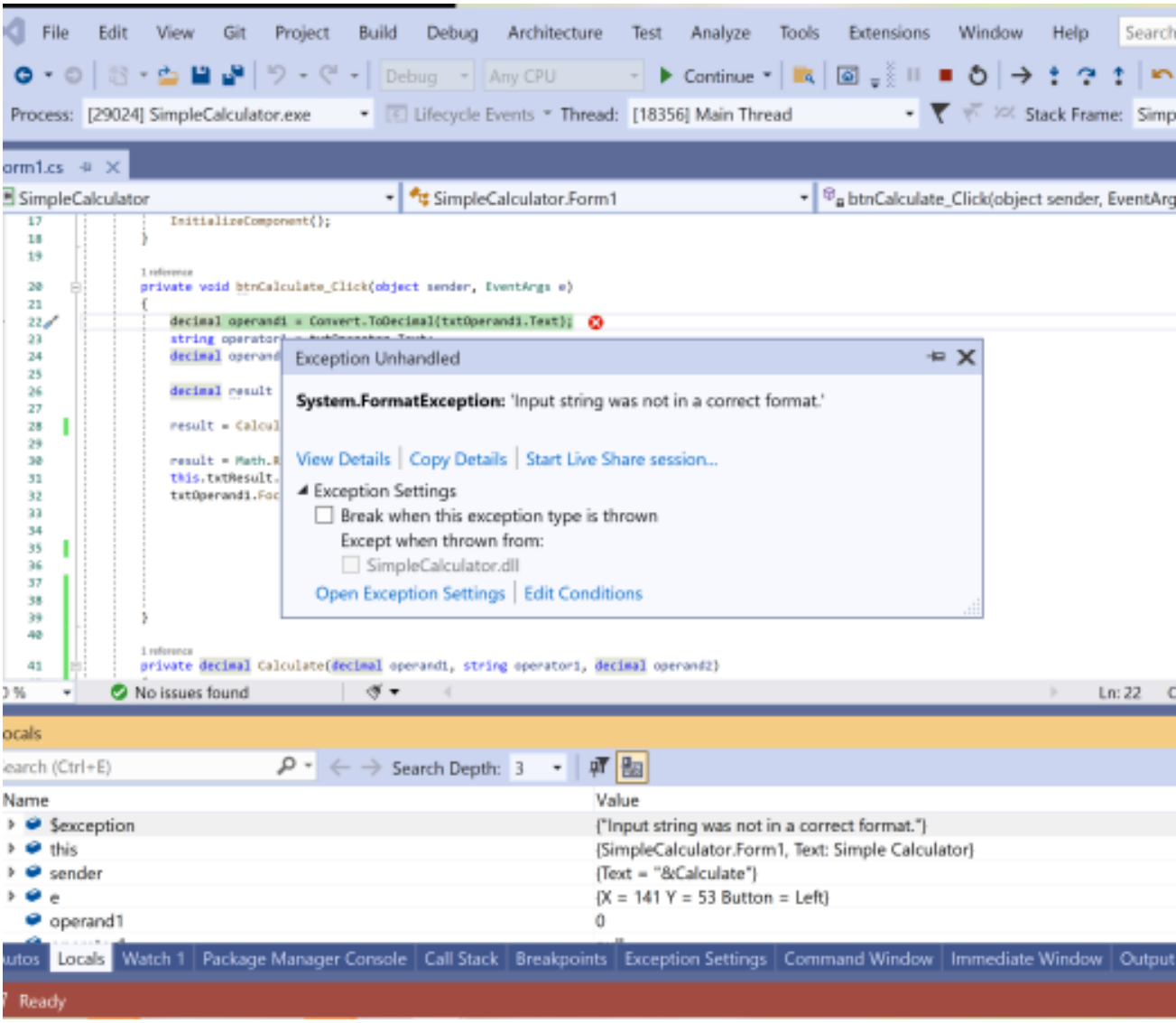
4. Describe the use of the properties and methods of an exception object.

5. Describe the use of throw statements.

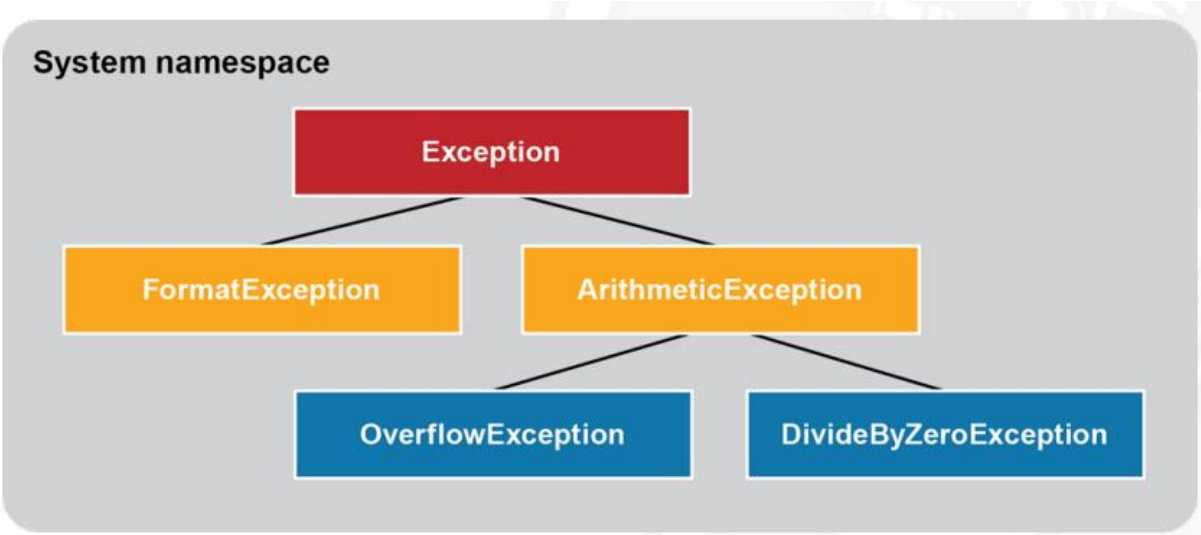
6. Describe the three types of data validation that you're most likely to perform on a user entry.

7. Describe two ways that you can use generic validation methods in a method that validates all of the user entries for a form

Dialog for an unhandled exception



The Exception Hierarchy for five common exceptions



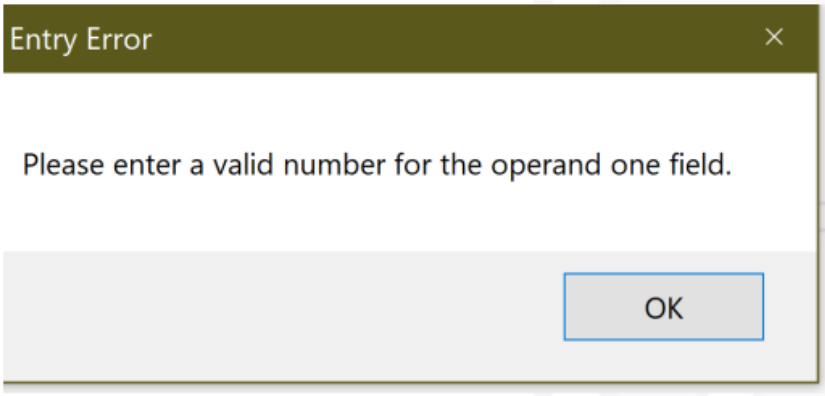
Methods that might throw exceptions

Class	Method	Exception
`Convert`	`ToDecimal(string)`	`FormatException` `OverflowException`
`Convert`	`ToInt32(string)`	`FormatException` `OverflowException`
`Decimal`	`Parse(string)`	`FormatException` `OverflowException`
`DateTime`	`Parse(string)`	`FormatException`

The syntax to display a dialog box with an OK button

```
MessageBox.Show(text[, caption]);
```

A dialog box with an OK button



The statement that displays this dialog box

```
MessageBox.Show(
    "Please enter a valid number for the operand one field.",
    "Entry Error");
```

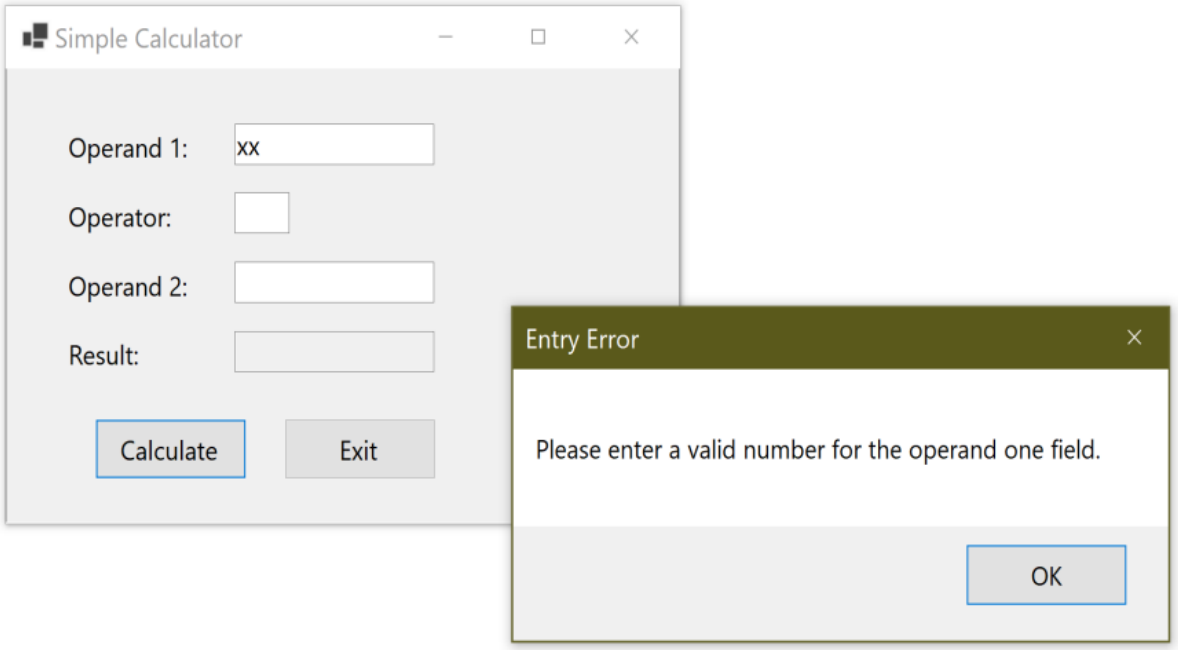
The syntax for a simple try-catch statement

```
try { statements }
catch { statements }
```

A try-catch statement

```
try
{
    decimal subtotal = Convert.ToDecimal(txtSubtotal.Text);
    decimal discountPercent = .2m;
    decimal discountAmount = subtotal * discountPercent;
    decimal invoiceTotal = subtotal - discountAmount;
}
catch
{
    MessageBox.Show(
        "Please enter a valid number for the Subtotal field.",
        "Entry Error");
}
```

The dialog box that's displayed for an value



The syntax for a try-catch statement that accesses the exception

```
try { statements }
catch(ExceptionClass exceptionName) { statements }
```

Two common properties for all exceptions

Property	Description
`Message`	Gets a message that briefly describes the current exception.
`StackTrace`	Gets a string that lists the methods that were called before the exception occurred.

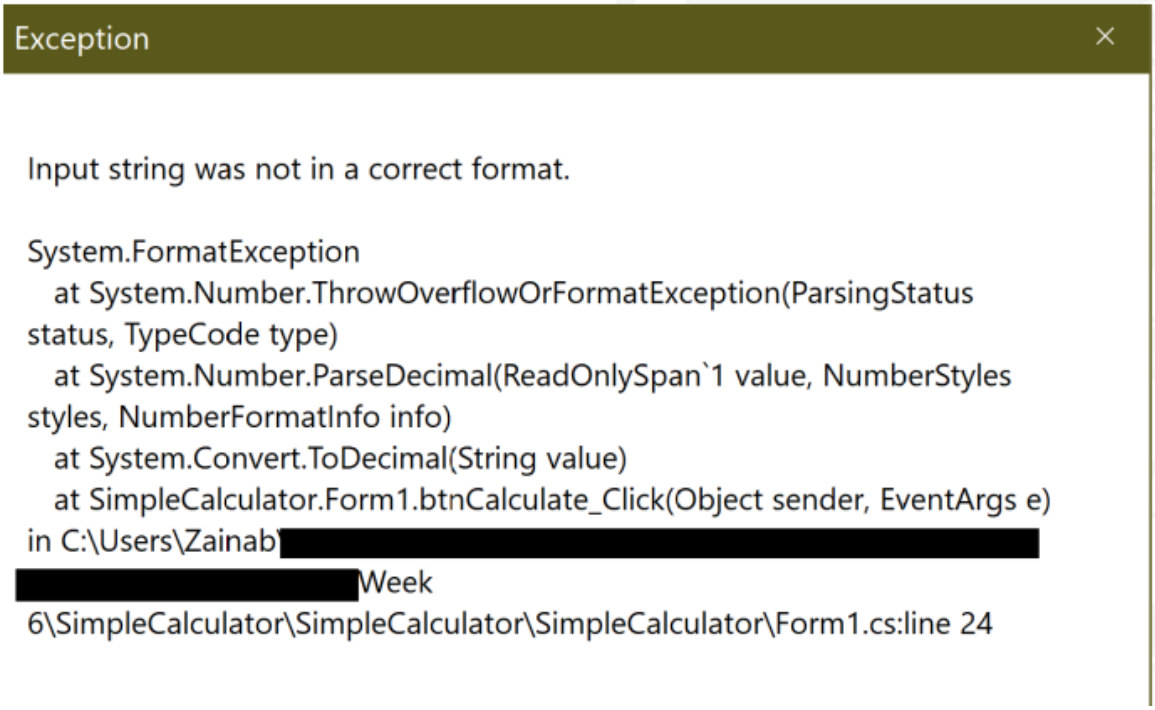
A common method for all exceptions

Method	Description
`GetType()`	Gets the type of the current exception.

A try-catch statement that accesses the exception

```
try
{
    decimal subtotal =
        Convert.ToDecimal(txtSubtotal.Text);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message + "\n\n" +
        ex.GetType().ToString() + "\n" +
        ex.StackTrace, "Exception");
}
```

The dialog box that's displayed for the exception



The syntax for a try-catch statement that catches specific types of exceptions

```
try { statements }
[catch(MostSpecificException
[exceptionName]) { statements }]...
[catch(NextMostSpecificException
[exceptionName]) { statements }]...
[catch([LeastSpecificException
[exceptionName]]) { statements }]
[finally { statements }]
```

The Syntax for a try-catch statement that catches specific types of exceptions

```
try
        {
            decimal operand1 = Convert.ToDecimal(txtOperand1.Text);
            string operator1 = txtOperator.Text;
            decimal operand2 = Convert.ToDecimal(txtOperand2.Text);
            decimal result = Calculate(operand1, operator1, operand2);

            result = Math.Round(result, 4);
            this.txtResult.Text = result.ToString();
            txtOperand1.Focus();
        }
        catch (FormatException)
        {
            MessageBox.Show(
                "Invalid numeric format. Please check all entries.",
                "Entry Error");
        }
        catch (OverflowException)
        {
            MessageBox.Show(
                "Overflow error. Please enter smaller values.",
                "Entry Error");
        }
        catch (DivideByZeroException)
        {
            MessageBox.Show(
                "Divide-by-zero error. Please enter a non-zero value for operand 2.",
                "Entry Error");
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message + "\n\n" +
            ex.GetType().ToString() + "\n" +
            ex.StackTrace, "Exception");
```

The syntax for throwing a new exception

```
throw new ExceptionClass([message]);
```

The syntax for throwing an existing exception

```
throw exceptionName;
```

When to throw an exception

- When a method encounters a situation where it isn't able to complete its task.
- When you want to generate an exception to test an exception handler.
- When you want to catch the exception, perform some processing, and then throw the exception again.

A method that throws an exception when an exceptional condition occurs

```
private decimal CalculateFutureValue(
decimal monthlyInvestment,
decimal monthlyInterestRate, int months)
{
    if (monthlyInvestment <= 0)
        throw new Exception(
            "Monthly Investment must be greater than 0.");
    if (monthlyInterestRate <= 0)
        throw new Exception(
            "Interest Rate must be greater than 0.");
    ...
}
```

Code that throws an exception for testing

```
try
{
    decimal subtotal =
        Convert.ToDecimal(txtSubtotal.Text);
    throw new Exception(
        "An unknown exception occurred.");
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message + "\n\n"
        + ex.GetType().ToString() + "\n"
        + ex.StackTrace, "Exception");
}
```

Code that rethrows an exception

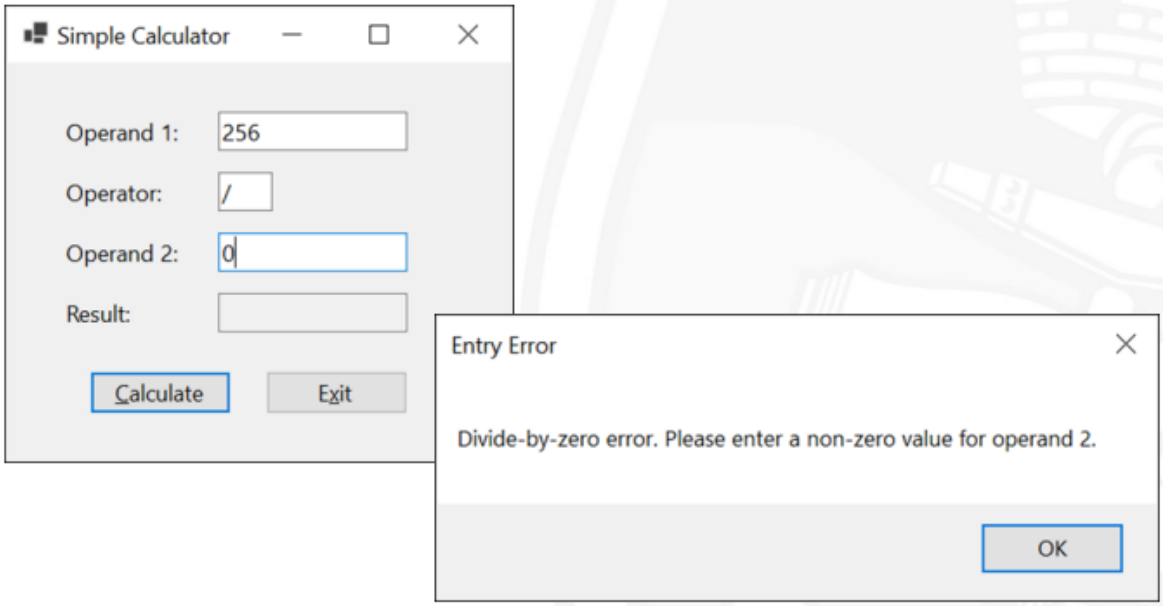
```
try
{
    Convert.ToDecimal(txtSubtotal.Text);
}
catch (FormatException fe)
{
    txtSubtotal.Focus();
    throw fe;
}
```

The code for the Simple Calculator with exception handling

```
private void btnCalculate_Click(object sender, EventArgs e)
        {
            try
            {
                decimal operand1 = Convert.ToDecimal(txtOperand1.Text);
                string operator1 = txtOperator.Text;
                decimal operand2 = Convert.ToDecimal(txtOperand2.Text);
                decimal result = Calculate(operand1, operator1, operand2);

                result = Math.Round(result, 4);
                this.txtResult.Text = result.ToString();
                txtOperand1.Focus();
            }
            catch (FormatException)
            {
                MessageBox.Show(
                    "Invalid numeric format. Please check all entries.",
                    "Entry Error");
            }
            catch (OverflowException)
            {
                MessageBox.Show(
                    "Overflow error. Please enter smaller values.",
                    "Entry Error");
            }
            catch (DivideByZeroException)
            {
                MessageBox.Show(
                    "Divide-by-zero error. Please enter a non-zero value for operand 2.",
                    "Entry Error");
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message + "\n\n" +
                ex.GetType().ToString() + "\n" +
                ex.StackTrace, "Exception");
            }
}
```

```
private decimal Calculate(decimal operand1, string operator1,
            decimal operand2)
        {
            decimal result = 0;
            if (operator1 == "+")
                result = operand1 + operand2;
            else if (operator1 == "-")
                result = operand1 - operand2;
            else if (operator1 == "*")
                result = operand1 * operand2;
            else if (operator1 == "/")
                result = operand1 / operand2;
            return result;
        }

        private void btnExit_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void ClearResult(object sender, EventArgs e)
        {
            this.txtResult.Text = "";
        }
```

Add exception handling to the simple calculator



Add exception handling to the Simple Calculator form.

Data Validation

- to make sure that a required entry has been made,
- to make sure that an entry has a valid numeric format,
- to make sure that an entry is within a valid range (known as range checking).

A method that checks for a required field

```
public bool IsPresent(TextBox textBox, string name)
        {
            if (textBox.Text == "")
            {
                MessageBox.Show(name + " is a required field.",
"Entry Error");
                textBox.Focus();
                return false;
            }
            return true;
        }
```

A method that checks for a valid numeric format

```
public bool IsDecimal(TextBox textBox, string name)
        {
            decimal number = 0m;
            if (Decimal.TryParse(textBox.Text, out number))
            {
                return true;
            }
            else
            {
                MessageBox.Show(name + " must be a decimal
number.", "Entry Error");
                textBox.Focus();
                return false;
            }
```

A method that checks for a valid numeric range

```
public bool IsWithinRange(TextBox textBox, string name,
            decimal min, decimal max)
        {
            decimal number = Convert.ToDecimal(textBox.Text);
            if (number < min || number > max)
            {
                MessageBox.Show(name + " must be between " + min
+
                    " and " + max + ".", "Entry Error");
                textBox.Focus();
                return false;
            }
            return true;
        }
```

```
public bool IsOperator(TextBox textBox, string name)
        {
            if (textBox.Text == "+" ||
                textBox.Text == "-" ||
                textBox.Text == "*" ||
                textBox.Text == "/")
            {
                return true;
            }
            else
            {
                MessageBox.Show(name + " is not valid.",
"Entry Error");
                textBox.Focus();
                return false;
            }
        }
public bool IsValidOperation(TextBox divisor, string
operation)
        {
            decimal value =
Decimal.Parse(divisor.Text);
            if (value == 0 && operation == "/")
            {
                MessageBox.Show("You may not divide by
zero.", "Entry Error");
                divisor.Focus();
                return false;
            }
            else
            {
                return true;
            }
        }
```

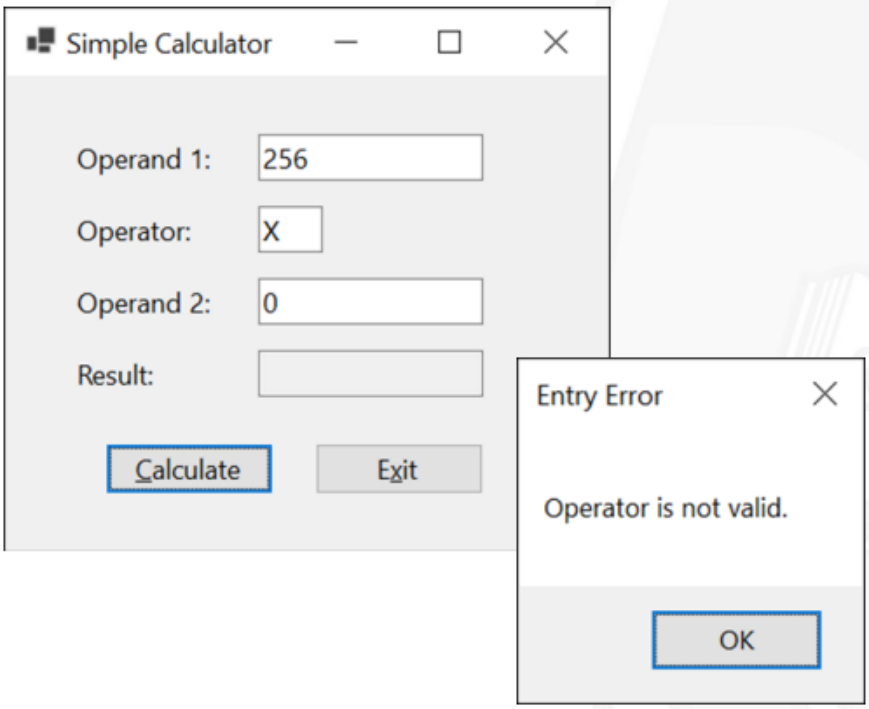
Code that validates multiple entries

```
public bool IsValidData()
        {
            return
                // Validate the Operand1 text box
                IsPresent(txtOperand1, "Operand 1") &&
                IsDecimal(txtOperand1, "Operand 1") &&
                IsWithinRange(txtOperand1, "Operand 1", 0, 1000000) &&

                // Validate the Operator text box
                IsPresent(txtOperator, "Operator") &&
                IsOperator(txtOperator, "Operator") &&

                // Validate the Operand2 text box
                IsPresent(txtOperand2, "Operand 2") &&
                IsDecimal(txtOperand2, "Operand 2") &&
                IsWithinRange(txtOperand2, "Operand 2", 0, 1000000) &&
                IsValidOperation(txtOperand2, txtOperator.Text);
        }
```

Add data validation to the simple calculator

```
  private void btnCalculate_Click(object sender, EventArgs e)
        {
            try
            {
                if (IsValidData())
                {
                    decimal operand1 = Convert.ToDecimal(txtOperand1.Text);
                    string operator1 = txtOperator.Text;
                    decimal operand2 = Convert.ToDecimal(txtOperand2.Text);
                    decimal result = Calculate(operand1, operator1, operand2);

                    result = Math.Round(result, 4);
                    this.txtResult.Text = result.ToString();
                    txtOperand1.Focus();
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message + "\n\n" +
                ex.GetType().ToString() + "\n" +
                ex.StackTrace, "Exception");
            }

        }
```

Add data validation to the simple calculator



Add data validation to the Simple Calculator form.

Week 7

Arrays

One Dimensional Array

WHY USE AN ARRAY?

The .NET Framework includes various built-in data types, such as int, decimal, string, and Boolean. These data types can be identified as simple data types because they consist of a single, simple value such as a number, text value, or a true or false setting

In term one, we learned how to declare and use variables using simple data types in C#. A variable can hold only one value and are declared as int x = 9; string animal = “zebra” or decimal m = 1.9. Suppose now, you want to store multiple values of the same data type. Take time to think how would do you do it?

Okay let us say, you want to store all possible Lion’s prey in a hunt; the preys are such as “ Pigs, Springbok, buffaloes, zebras, kudu, rabbits, hippos, and wildebeests”.

You will have to declare multiple string variables as

string prey1 = “Pigs”;

string prey2 = “Springbok”

string prey3 = “buffaloes”

string prey4 = “Kudu”

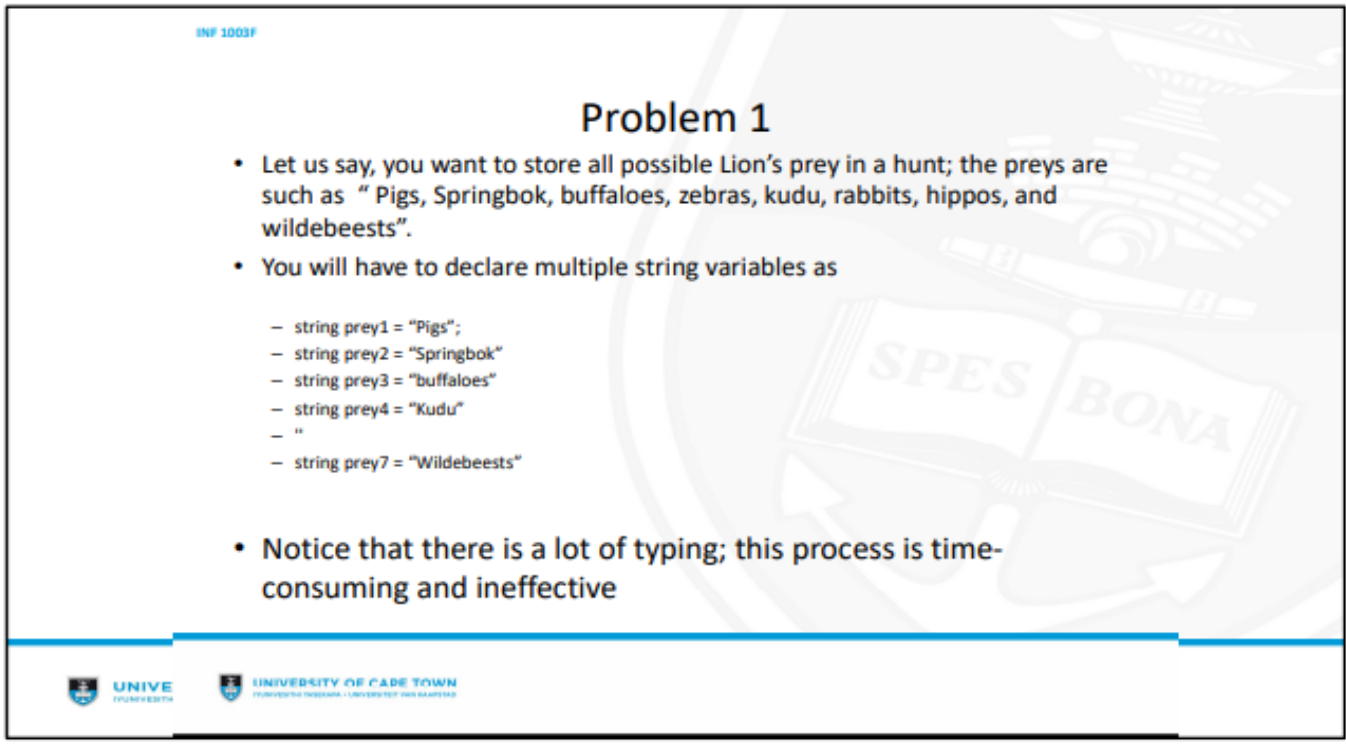
"

string prey7 = “Wildebeests”

Notice that there is a lot of typing; this process is time-consuming and ineffective. The simplest way to implement our collection is to use an array to hold the items. C# and most programming languages provide a way to store a collection of values of the same type in an Array. So instead of declaring a variable for every element, you can just declare one variable, that is responsible for storing the elements of a collection.

This week, we consider a C# fundamental construct known as the *array*. An array stores a sequence of values that are all of the same type. We want not just to store values but also to be able to quickly access each individual value. The

method that we use to refer to individual elements in an array is to number and then *index* them. Watch the video below to learn more about Arrays.



Why use an Array?

The .NET Framework includes various built-in data types, such as int, decimal, string, and Boolean. These data types can be identified as simple data types because they consist of a single, simple value such as a number, text value, or a true or false setting

In term one, we learned how to declare and use variables using simple data types in C#. A variable can hold only one value and are declared as int x = 9; string animal = "zebra" or decimal m = 1.9. Suppose now, you want to store multiple values of the same data type. Take time to think how would do you do it?

Okay let us say, you want to store all possible Lion's prey in a hunt; the preys are such as " Pigs, Springbok, buffaloes, zebras, kudu, rabbits, hippos, and wildebeests". You will have to declare multiple string variables as seen here. Notice that there is a lot of typing; this process is time-consuming and ineffective.

INF 1003F

Problem 2

What if you want to store class scores

You will have to declare multiple int variables as

- int score1 = 40
- int score2 = 70
- int score3 = 45
- int score4 = 45

- int numbers105=

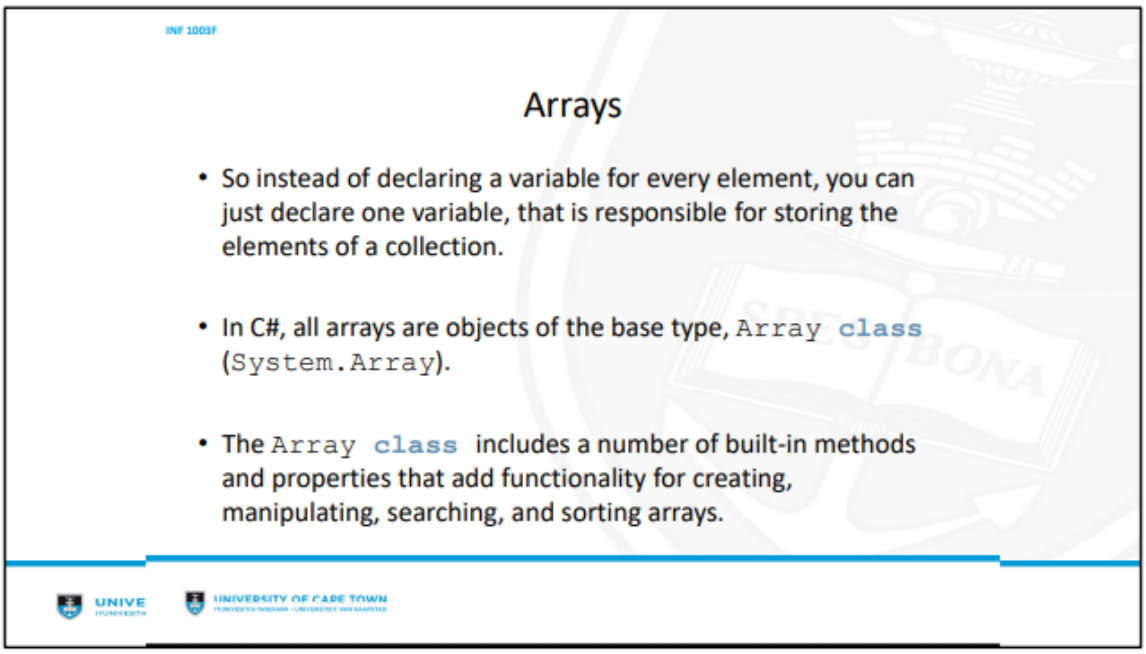
A lot of typing; this process is time-consuming and ineffective

UNIVE

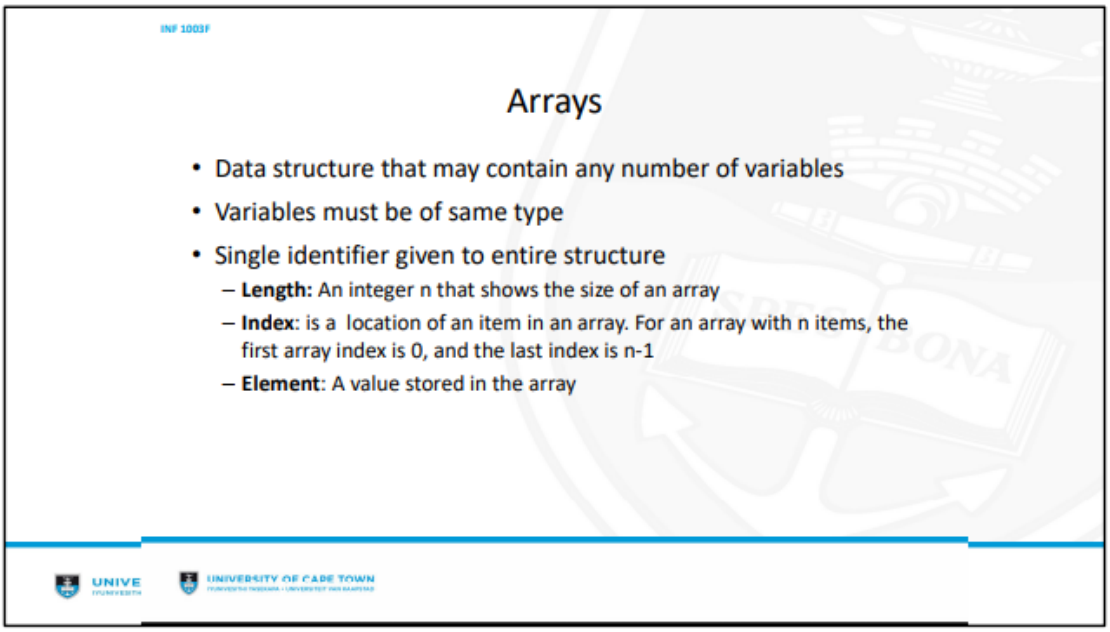
UNIVERSITY OF CAPE TOWN

Again here there is a lot of typing; this process is time-consuming and ineffective. The simplest way to implement our collection is to use an array to hold the items. C# and most programming languages provide a way to store a collection of values of the same type in an Array.

So instead of declaring a variable for every element, you can just declare one variable, that is responsible for storing the elements of a collection.



Arrays and collections are objects that act as containers. As you develop C# applications, you'll find many uses for arrays and collections.

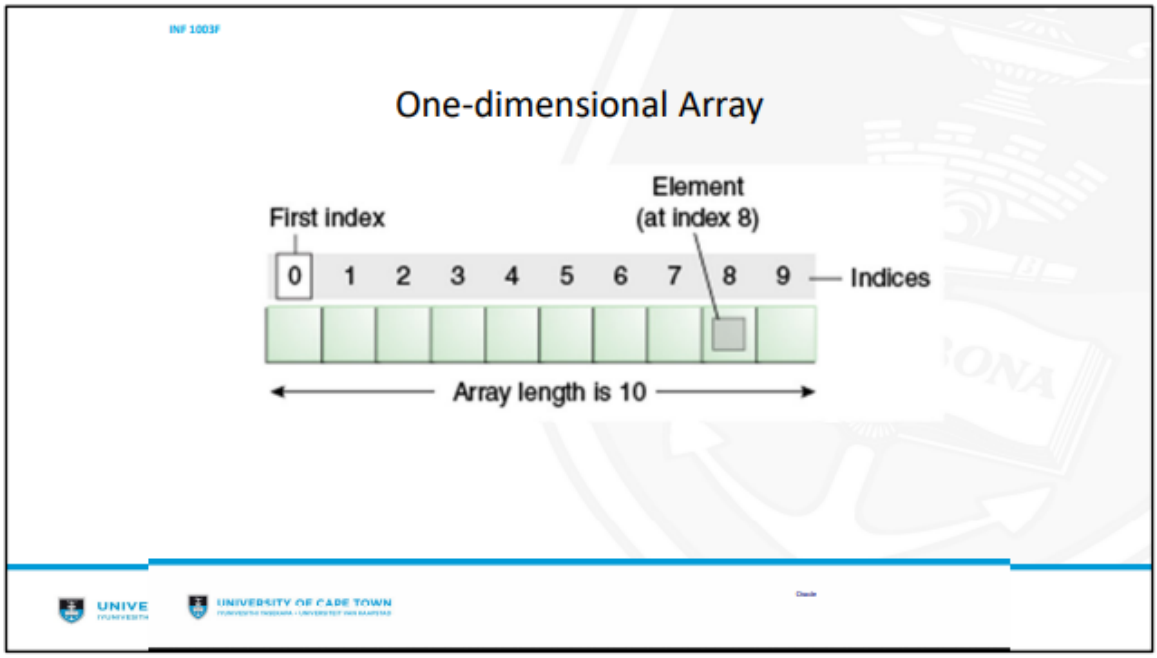


An **array** is a data structure that may contain any number of variables. In the C# language, the variables must be of the same type. A single identifier, or name, is given to the entire structure.

All data values placed in an array must be of the same data or base **type**. The type can be one of the predefined types like int or string, or some other .NET class. The type can also be a class that you create in C#.

The integral value is used to specify the number of elements. This is called the **length** or **size of the array**.

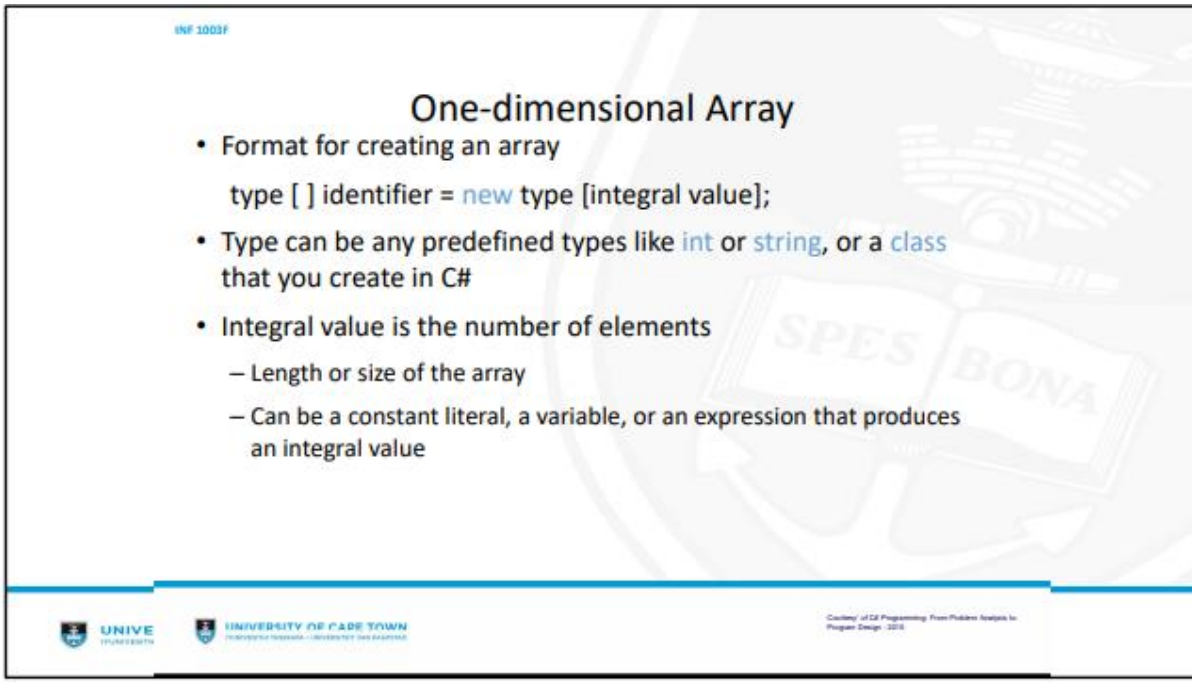
The individual variables in the array are called the elements of the array and are accessed through an index. The index, also called the subscript of the array, references the location of the variable relative to the first element in the array. Elements in an array are sometimes referred to as indexed or subscripted variables.



You can use an array to store a set of related data. Since a one-dimensional array is the simplest type of array, in week 7 you'll start by learning how to create and use one-dimensional arrays

Array elements are normally stored in contiguous, side-by-side, memory locations. the index of the first element is always 0.

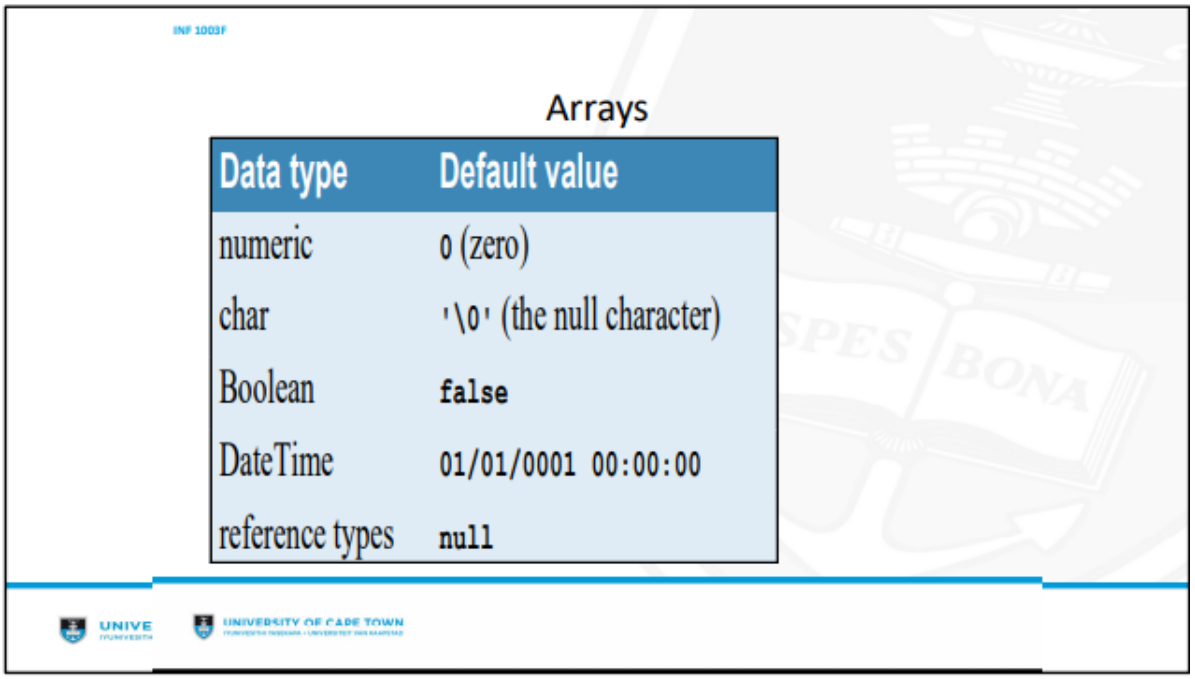
The index of the second element is 1 and is referenced using the identifier and a 1 enclosed in square brackets. The index of the last element in the array is always n – 1, where *n* represents the number of elements in the array



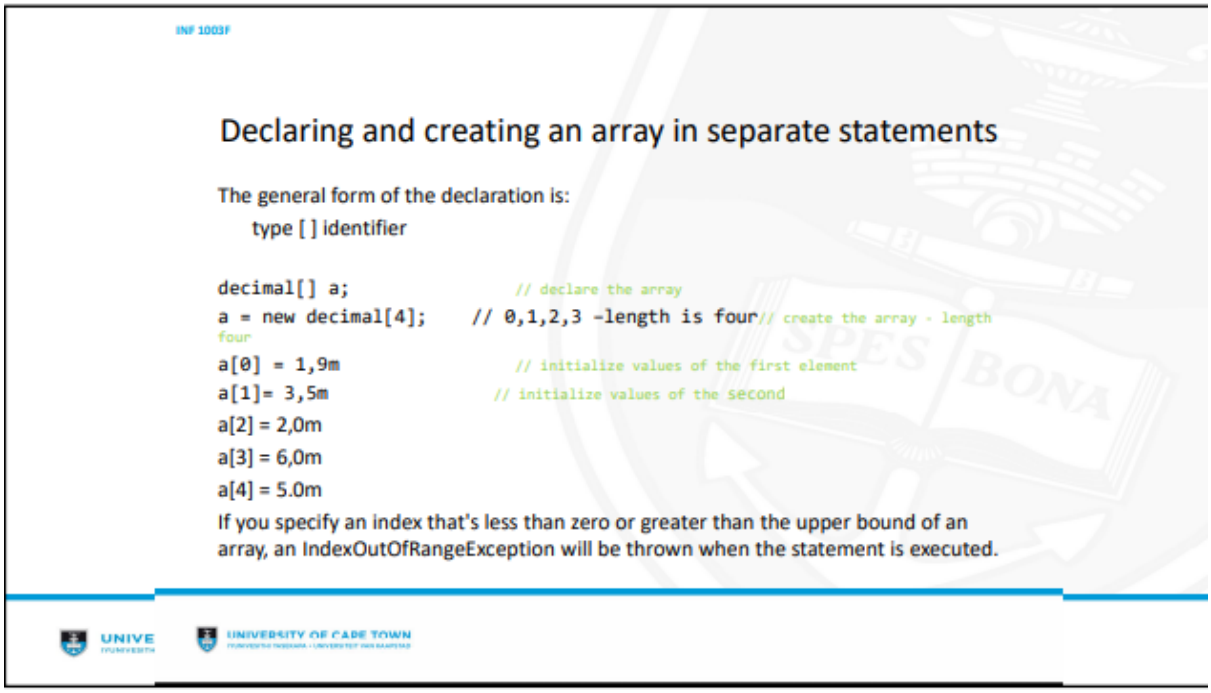
The format for creating an array is type [ ] identifier = new type [integral value];

Notice that you also use the new keyword to create an array. That's because arrays are created from the Array class that's provided by the .NET Framework. So when you create an array, you're creating an instance of this class. Recall how you created an instance of a Random class? You create an array in much the same way you instantiate an object of a user-defined class—by using the new keyword at the time of declaration. You can also specify during declaration the number of individual elements for the array – Integral value – length.

This integral value can be a constant literal, a variable, or an expression that produces an integral value. Because the integral value is indicating the actual number of elements for which space will be allocated, the value must be positive. It must represent an integer, a whole number, or an expression that maps to a whole number.



When you create an array, each element in the array is initialized to a default value. This value depends on the data type of the array as indicated by the table. If the array contains value types, each element is set to the default value shown above. If the array contains reference types, each element is set to null.



As the syntax at the top of this figure indicates, you refer to an element in an array by coding the array name followed by its index in brackets. The index that you specify must be from 0, which refers to the first element in the array, to the upper bound of the array, which is one less than the length of the array. To understand how this works.

This example declares an array of decimals that contains four elements. Then, it uses indexes 0 through three to assign values to those four elements. Here, the length of the array is 4, and the upper bound of the array is 3. Note that if you use an index that's greater than the upper bound, an IndexOutOfRangeException will be thrown.

INF 1003F

Declaring and creating an array in a single statement

An array of integers

- `Int [] num = new int [3] // declare and create the array`
- `num[0] = 3; // initialize values of the first element`
- `num[1] = 5; // initialize values of the second element`
- `num[2] = 6; // initialize values of the third element`

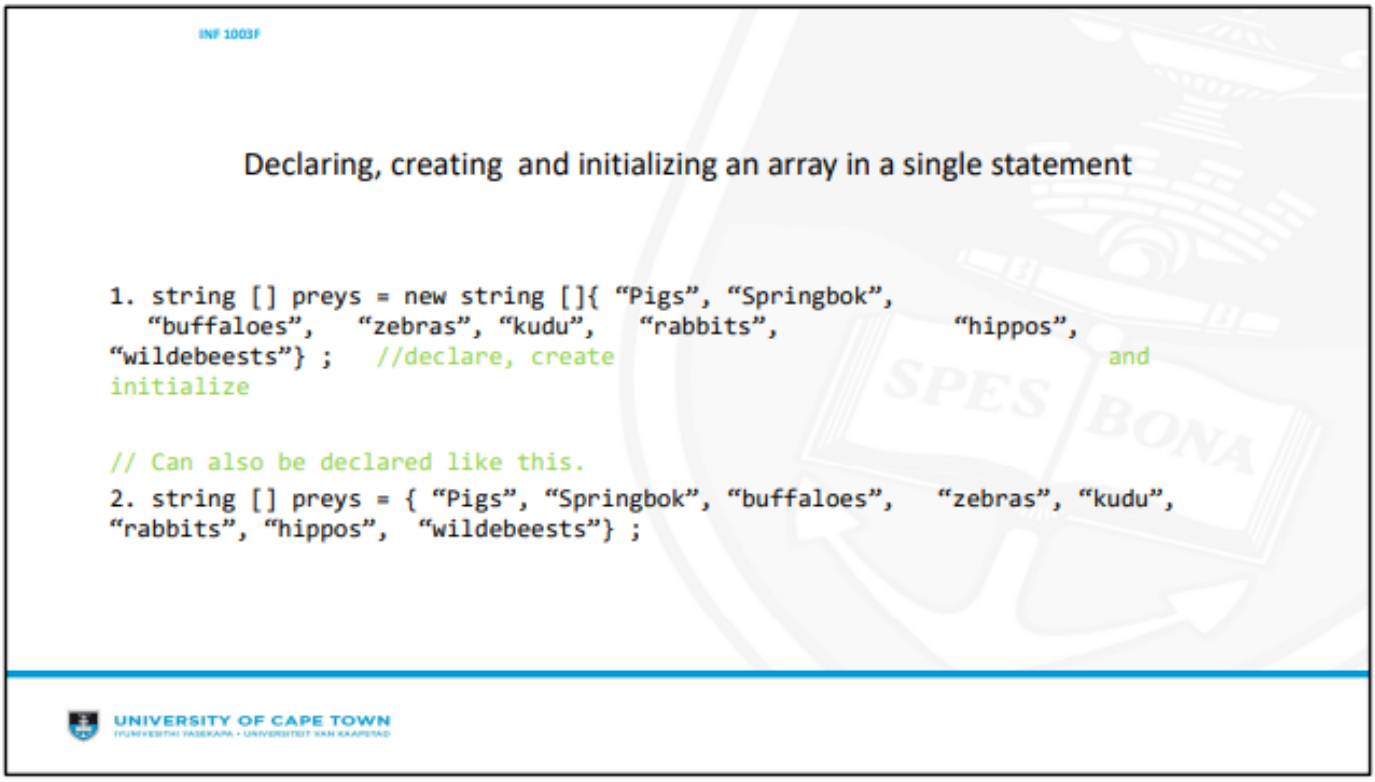
This example declares an array of int that contains three elements. Then, it uses indexes 0 through three to assign values to those four elements

INF 1003F

Declaring and creating an array in a single statement

- An array of strings
- `string[] suits = new string[4] // declare and create the array`
- `suits[0] = "Clubs"; // initialize values of the first element`
- `suits[1]= "Diamonds"; // initialize values of the second element`
- `suits[2] = "Hearts"; // initialize values of the third element`
- `suits[3] = "Spades"; // initialize values of the fourth element`

This example declares an array of strings that contains four elements. Then, it uses indexes 0 through three to assign values to those three elements

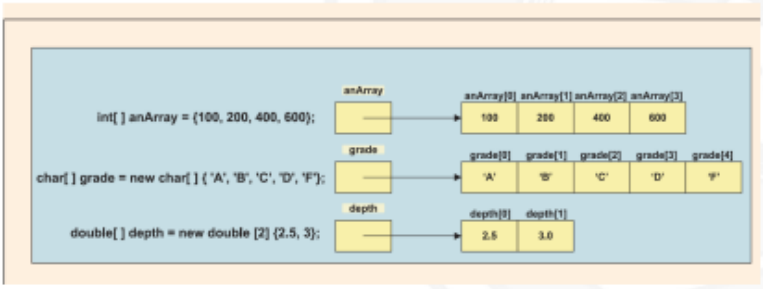


The syntax and examples 1 and 2 show how to declare an array and assign values to its elements in a single statement. Here, you start the array declaration as before, but you code a list of values in braces after the declaration. Then, C# automatically sets the length of the array to the number of elements within the braces. If you include the data type and length specification, however, they must match the list of values you provide.

For the second example notice that you can omit the new keyword and the type and length specification. If you chose this option array elements must match the data type.

Declaring, creating and initializing an array in a single statement

```
int [ ] anArray = {100, 200, 400, 600};
char [ ] grade = new char[ ] {'A', 'B', 'C', 'D', 'F'};
double [ ] depth = new double[2] {2.5, 3};
```



UNIVERSITY OF CAPE TOWN

The first statement creates an array of four elements. The initial values are 100, 200, 400, and 600. Notice that the keyword new is not included here, and the type is not repeated.
The second statement shows that the type of the value used for initialisation must be compatible with the declaration type. Because char is the specified declaration type, all initialisation values are enclosed in single quotes. Five elements are created using the grade identifier. No length specifier is required. But notice that this differs from the declaration of anArray in that the new keyword is used and the type is specified with opening and closing square brackets.
The last statement creates an array with a length of two. The value 3 is assignment compatible with the double
type.
The image below shows the memory contents after the initialisation.

Infer the type of an array from its values

- var grades = new [] {95,89,91,98};

- The Var keyword : Local variables can be declared without giving an explicit type.
- The var keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialisation statement.

You can also create an array without specifying a data type. To do that, you use the var keyword.
Then, the data type is inferred from the values that are assigned to the array.
When you use this technique, you must not code square brackets following the var keyword, you must code the new keyword followed by square brackets, and you must not specify a length

Accessing Array

- `Int [] numbers = new int [3] // declare and create the array`
- `numbers[0] = 3;`
- `numbers[1] = 5;`
- `numbers[2] = 6;`

```
Int total = numbers[0] + numbers[1] +
numers[2]
= 3+5+6
=14
```

- What if there were 25 umbers? Adding all 25 umbers to total would take a lot of typing.?

UNIVERSITY OF CAPE TOWN

17

To access an array element, you must specify which element is to be accessed by placing an index inside square brackets following the identifier.
This is because the array is now referenced by a single name. To retrieve individual elements, an index or subscript is required. Arrays are zero-based structures; thus, the index of the first element is always 0. The index of the second element is 1 and is referenced using the identifier and a 1 enclosed in square brackets. The index of the last element in the array is always n – 1, where n represents the number of elements in the array.
The index references the physical location of the element relative to the beginning element.
To place a value in the first element of numbers, the following assignment statement could be made Notice that the last element was referenced by numbers[2]. There is no numbers[3]. Fourteen elements are referenced by numbers. The first one is numbers[0]. The last valid index is always the length of the array minus one.

Using For with Array

ArrayName.Length – Property

```
for (int i = 0; i < ArrayName.Length; i++)
```

- Access the array
- Input value into an aray

  ArrayName[i]= value;
- Manipulate the array

Using For with Array

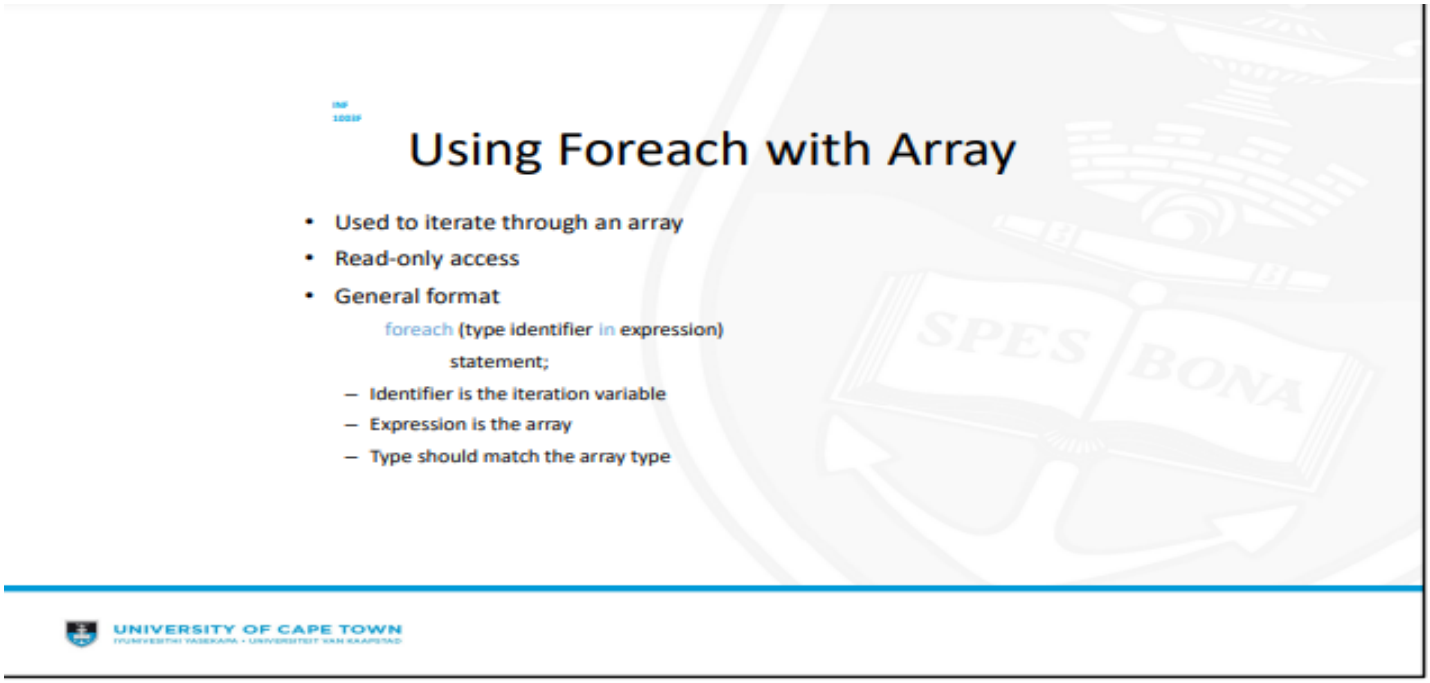
- **For Loop:**
- This example shows a better way to sum the values.
- A counter-controlled loop with a variable for the index is used.

```
for (int i = 0; i < numbers.Length; i++)
    numbers[i] = i;
for (int i = 0; i < numbers.Length; i++)
   { total += numbers[i];
```


C# always performs bounds checking on array indexes. One of the special properties in the Array class is Length. It returns an int representing the total number of elements in an array.

In this example, Length would return 3 . The loop control variable, i, is evaluated before each iteration through the loop to determine whether it is less than 3 (numbers.Length). FOR loop can be used to change the contents of any of the elements in an array BY ACCESSING THE ARRAY LOCATION.

Also Notice that the conditional expression that was used with the for statement used less than (<) instead of less than or equal to (<=). Because Length returns a number representing the size of the array, the last valid index should always be one less than Length. Therefore, as soon as the index is equal to Length, the loop should terminate.



FOR EACH

The foreach statement is used to iterate or move through a collection, such as a list and an array. The foreach loop offers another option for traversing through a list without having to increment indexes or counter-controlled variables like is needed when you use a for statement loop.. However, it can be used for read-only access to the elements. Unlike for loop The foreach loop cannot be used to change the contents of any of the elements in an array. You can use it to sum the values.

For each syntax:

foreach (type identifier in collection) statement; The collection in foreach is the name of the collection, such as list items etc. Type is the kind of values found in the collection, example int.

The identifier is a user-supplied name used as an iteration control variable to reference the individual entries in the collection.

Like other forms of looping structures, after the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the foreach block.

Accessing Array

- For each loop
- Example

```
foreach (int values in numbers)
 {           total += values;     }

string values = “ ”;
string[] color = { "red", "green", "blue" };
          foreach (string val in color)
              values += val + “\n”;
              valesLabel.Text= values;
```


As I mentioned previously . The foreach loop offers another option for traversing through an array without having to increment indexes or counter-controlled variables like is needed when you use a **for** statement loop.. However, it can be used for read-only access to the elements. Unlike for loop The **foreach loop** cannot be used to change the contents of any of the elements in an array. You can use it to sum the values.

Examples

- How to get the length of an array

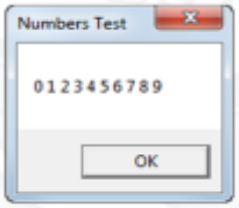
```
arrayName.Length
```

Code that puts the numbers 0 through 9 into an array

```
int[] numbers = new int[10];
for (int i = 0; i < numbers.Length; i++)
numbers[i] = i;
```

Display the numbers

```
string results = "";
for (int i = 0; i < numbers.Length; i++)
    results += numbers[i] + " ";
MessageBox.Show(results, "Numbers Test");
```



Try this examples in Visual Studio.

The first example starts by presenting the Length property of an array, which gets the number of elements in the array. You'll use this property frequently as you work with arrays.
The second example uses a for loop to put the numbers 0 through 9 into an array. Here, the first statement declares an array named numbers that contains 10 int values. Then, the for loop assigns the value of the loop's counter to each of the elements in the array.

The third example uses a for loop to display the elements of the numbers array in a message box. Here, the first statement declares a string variable. Then, the for loop accesses each element of the numbers array and appends the number and a space to the string variable. When the for loop finishes, the next statement displays the string that contains the numbers in a message box.

Examples

Code that computes the average of an array of totals

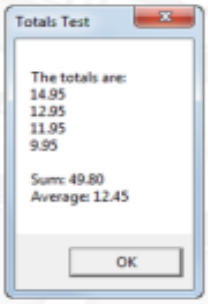
```
decimal[] totals = {14.95m, 12.95m, 11.95m, 9.95m};
decimal sum =  totals[0] + totals[1] + totals[2] + totals[3];
decimal average = sum/4;
```

Code that prints the average

```
decimal sum = 0.0m;
for (int i = 0; i < totals.Length; i++)
sum += totals[i];
decimal average = sum/totals.Length;
```

Code that displays the totals array

```
string totalsString = "";
for (int i = 0; i < totals.Length; i++)
    totalsString += totals[i] + "\n";
MessageBox.Show("The totals are:\n" +
    totalsString + "\n" +
    "Sum: " + sum + "\n" +
    "Average: " + average, "Totals Test");
```



Try this examples in Visual Studio.

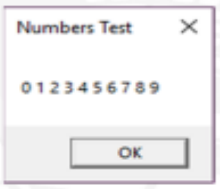
The first example shows how to get the values from an array and calculate the average value of the elements. Here, the first statement declares a totals array that contains four decimal values. Then, the second statement sets the sum variable equal to the sum of the four elements in the array. Finally, the third statement computes the average value of these elements by dividing the sum by four.
The fourth example shows how to use a loop to calculate the average value of the elements in the totals array. Here, the first statement declares the sum variable. Then, the for loop gets the value of each element of the array and adds that value to the current value of the sum variable. After the loop, the next statement uses the sum variable and the Length property of the array to calculate the average. If you compare this code to the code in the previous examples, you'll see that they both accomplish the same task. However, the code in this example will work equally well whether the totals array contains 4 or 400 values. As a result, it usually makes sense to use loops when working with arrays.

The last example shows how to display the elements of the totals array, along with the sum and average for the array, in a message box. Like the third from the previous slide example, this one uses a for loop to format the values in the array as a string. Then, it displays the string and the sum and average values in a message box.

Examples

Code that displays the numbers array

```
string numbersString = "";
foreach (int number in numbers)
numbersString += number + " ";
MessageBox.Show(numbersString, "Numbers Test");
```



Code that computes the average of the totals array

```
decimal sum = 0.0m;
foreach (decimal total in totals)
    sum += total;
decimal average = sum/totals.Length;
```


When a foreach loop is executed, its statements are executed once for each element in the array.

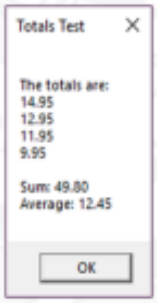
For instance, in this first example shows how you can use a foreach loop to display the elements in the numbers array that you saw in the previous slides. Similarly, the second example computes the average of the totals array, and the third example displays the elements in the totals array. If you compare these examples to the for loop examples in the previous slide, you'll see that foreach loops are less complicated. In particular, when you use a foreach loop, you don't need to use a counter variable, and you don't need to use an index to get an element from the array. As a result, it's often easier to use a foreach loop than a for loop.

Please note that there are still times when you'll need to use a for loop to work with an array. For example, if you only want to access some of the elements in an array, you'll need to use a for loop. You'll also need to use a for loop if you want to use a counter variable to assign values to the elements of the array as we saw in the for loop example.

Examples

Code that uses a foreach statement to display the totals array

```
string totalsString = "";
foreach (decimal total in totals)
    totalsString += total + "\n";
MessageBox.Show("The totals are:\n" +
    totalsString + "\n" +
    "Sum: " + sum + "\n" +
    "Average: " + average,
    "Totals Test");
```



This example shows how to display the elements of the totals array, along with the sum and average for the array, in a message box. Like the examples from the previous slide with for loop, this one uses a foreach loop to format the values in the array as a string. Then, it displays the string and the sum and average values in a message box.

Taking user input in an Array

- How do you take array's length from the user?
  - Int length = convert.ToInt32(textboxname.Text)

Taking user input in an Array

- What about array values from user input?

  – type [ ] identifier = new type [integral value];
    - So will be Int[] numbers = new int [length];

  Numbers[i] = convert.ToInt32(textboxname.Text)

ListBox class

- **ListBox.ObjectCollection:**
  – This collection class holds all the elements of the ListBox control.
- **ListBox.SelectedObjectCollection:**
  – This collection class holds the collection of selected items in the ListBox control.
- **ListBox.SelectedIndexCollection:**
  – This collection class holds the collection of selected indices, these elements are a subset of the indices of the ListBox.ObjectCollection and this specifically selected indexes in the ListBox control.

Common members of list box and combo box controls

Property	Description
`SelectedIndex`	The index of the selected item. Items are numbered from 0. If no item is selected, this property has a value of -1.
`SelectedItem`	The object of the selected item.
`Text`	The text value for the selected item.
`Sorted`	If set to true, the items in the list are sorted alphabetically in ascending order.
`Items`	Provides access to the collection of items.
`DropDownStyle`	Determines whether the user can enter text in the text portion that's at the top of a combo box. If this property is set to DropDownList, the user must select an item from the list. If this property is set to DropDown, the user can enter data in the text box portion of the combo box.
`SelectionMode`	Determines whether the user can select more than one item from a list box. If this property is set to One, the user can only select one item. If it's set to MultiSimple or MultiExtended, the user can select multiple items.
Event	**Description**
`SelectedIndexChanged`	Occurs when the user selects a different item from the list.
`TextChanged`	Occurs when the user enters a value into the text box portion of a combo box.

The image shows the properties, methods, and events that you're likely to use as you work with combo boxes and list boxes.

To get the index of the item that the user selects, for example, you use the SelectedIndex property. To get the selected item itself, you use the SelectedItem property. And to get a string that represents the selected item, you use the Text property. You'll see coding examples that use some of these properties in the demonstration.

One property that applies only to a combo box is the DropDownStyle property. The default is DropDown, which means that the user can either click on the drop-down arrow at the right side of the combo box to display the drop-down list or enter a value directly into the text portion of the combo box. Note that if the user enters a value, that value doesn't have to appear in the list. If you want to restrict user entries to just the values in the list, you can set the DropDownStyle property to DropDownList. Then, the user can only select a value from the list or enter the first letter of a value in the list to select it.

One property that applies only to a list box is the SelectionMode property. The default is One, which means that the user can only select one item from the list box. However, you can let the user select multiple items by setting this property to MultiSimple or MultiExtended. If you set it to MultiSimple, the user can only select multiple entries by

clicking on them. If you set it to MultiExtended, the user can hold down the Ctrl and Shift keys to select nonadjacent and adjacent items. This works just as it does for any standard Windows application. By the way, you can also set this property to None, in which case the user can't select an entry. You might use this setting if you just want to display items.

If you allow the user to select multiple items from a list box, you can use the SelectedIndices property to return a collection of the selected indexes, and you can use the SelectedItems property to return a collection of selected items. Or, you can use the SelectedIndex and SelectedItem properties to return the first index or item in the collection of selected items. When you work with the items in a list box or combo box, you should realize that you're actually working with the items in a collection. To refer to this collection, you use the Items property of the control. Then, you can use an index to refer to any item in the collection. Or, you can use properties and methods that .NET provides for working with collections.

The most common event for working with combo boxes and list boxes is the SelectedIndexChanged event.
For a combo box, you can also use the TextChanged event to detect when the user enters a value into the text portion of the control. Keep in mind, though, that this event will occur each time a single character is added, changed, or deleted.

Common members of the Items collection

Indexer	Description
`[index]`	Gets or sets the item at the specified index in the list.
Property	**Description**
`Count`	Gets the number of items in the list.
Method	**Description**
`Add(object)`	Adds the specified item to the list.
`Insert(index, object)`	Inserts an item into the list at the specified index.
`Remove(object)`	Removes the specified item from the list.
`RemoveAt(index)`	Removes the item at the specified index from the list.
`Clear()`	Removes all items from the list.

UNIVERSITY OF CAPE TOWN

Using a Listbox control and for each Loop

The foreach statement is used to iterate or move through a collection, such as an array. You will be working with arrays next week. An array is a data structure that allows multiple values to be stored under a single identifier. Values are usually accessed in the array using the identifier and an index representing the location of the element relative to the beginning of the first element. The foreach loop

offers another option for traversing through the array without having to increment indexes or counter-controlled variables like is needed when you use a **for** statement loop.

For each syntax:

for each (type identifier in collection)

Statement;

The **collection** in foreach is the name of the collection, such as the array. Type is the kind of values found in the collection, example int. The identifier is a user-supplied name used as an iteration control variable to reference the individual entries in the collection.

Like other forms of looping structures, after the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the foreach block.

```
1 reference
private void lstbxOlympics_SelectedIndexChanged(object sender, EventArgs e)
{
    string result = "";

    foreach(string sports in lstbxOlympics.SelectedItems)

    {
        result += sports + " ";
    }
    txtDisplay.Text = result;
}
```

Multidimensional Arrays

- Multidimensional : arrays with more than one dimension
  - Two-dimensional array int [,]
  - Three-dimensional array int [, ,]
  - Array of arrays – Jagged Arrays
- There is no limit for arrays dimensions
  - two-dimensional array-rectangular arrays

Multidimensional Arrays Declaration and Allocation

- Each dimension is represented by a comma in square brackets
- One dimensional array => int []
  - type [ ] identifier = new type [integral value]
- Two-dimensional array int [,]
  - type [,] identifier = new type [integral rows, integral columns]
- Three-dimensional array int [, ,]
  - type [, ,] identifier = new type [integral planes, integral row, integral column];

Multidimensional Arrays Declaration and Allocation

- Declaration
  - Int [,] twoDiArray
  - Float [,] floatMatrix
  - String [, ,] threeDiArray

- We allocate memory by using key word new
  - Two-dimensional

```
Int [ , ] twoDiArray = new int [3, 4];
Float[ , ]  floatMatrix = new float [ 8, 2 ];
```

  - Three –dimensional arrays

```
String [ , , ] threeDiArray = new string [5,  5, 5];
```

Rectangular array

- Two-dimensional array int [,]
  - Matrices
  - Table => row x column
- All elements must be of the same data type
  - type [,] identifier = new type [integral value, integral value]
- The integral values are rows **m** and the columns **n**
- If a two-dimensional array has a size of **m** by **n**, there are exactly **m*n** elements

---

A Rectangular Array Initialization

```
int[,] arrayName = {
            {1, 3, 6, 2},
            {8, 5, 9, 1},
            {4, 7, 3, 0} };
```

- The table of size is 3x4 (3 row, 4 cols)
- 12 elements

	0	1	2	3
0	1	3	6	2
1	8	5	9	1
2	4	7	3	0

Accessing a Rectangular Array

Each dimension in a multidimensional array starts at index 0.

- Rectangular Array
- arrayName[row, col]
  - arrayName[0, 0] arrayName[0, 1] arrayName[0, 2] arrayName[0, 3]
  - arrayName[1, 0] arrayName[1, 1] arrayName[1, 2] arrayName[1, 3]
  - arrayName[2, 0] arrayName[2, 1] arrayName[2, 2] arrayName[2, 3]

- nDimensionalArray[index1, …, indexN]

Length of a Rectangular Array

- **arrayName.GetLength(0)** – returns number of rows
- **arrayName.GetLength(1)** – returns number of columns
- **arrayName.Rank** – returns the number of dimensions of the array.
- **Methods** GetUpperBound( ) and GetLowerBound( ) return the upper or lower bounds of the specified dimension
- **arrayName.Length** – returns the length of all dimensions

Displaying a Rectangular Array

```
string result = "";

for (int row = 0; row < arrayName.GetLength(0); row++)
  {
     for (int col = 0; col < arrayName.GetLength(1); col++)
      result += arrayName[row, col] + " ";
     result += "\n";
     }MessageBox.Show(result);
```

D

Examples on VULA

- **Demonstrates usage of a two-dimensional array to calculate the average number of calories intake per day, per meal type, and per meal.**

```
Int[,] calories = {  {900, 750, 1020},
                     {300, 1000, 2700},
                   {500, 700, 2100},
                   {400, 900, 1780},
                   {600, 1200, 1100},
                    {575, 1150, 1900},
                    {600, 1020, 1700} };

string[] mealTime = { "Breakfast", "Lunch", "Dinner" };
```

Jagged Arrays

- Also known as 'arrays of arrays'
- Each row contains an array of its own
- Rows have unequal lengths

Declaration:

```
type[][] arrayName = new type[rowCount][];
```

Code that creates a jagged array with three rows of different lengths

```
int[][] numbers = new int[3][]; // the number of rows is 3
numbers[0] = new int[3]; // the number of columns for row 1
numbers[1] = new int[4]; // the number of columns for row 2
numbers[2] = new int[2]; // the number of columns for row 3
```

The syntax for referring to an element of a jagged array

```
arrayName[rowIndex][columnIndex]
```

Code that assigns values to the numbers array

```
numbers[0][0] = 1;  numbers[1][0] = 4;  numbers[2][0] = 8;
numbers[0][1] = 2;  numbers[1][1] = 5;  numbers[2][1] = 9;
numbers[0][2] = 3;  numbers[1][2] = 6;
                    numbers[1][3] = 7;
```

Code that creates the numbers array with one statement

```
int[][] numbers = { new int [] {1, 2, 3},
                    new int [] {4, 5, 6, 7},
                    new int [] {8, 9} };
```

Code that creates a jagged array of strings

```
string[][] titles = {
   new string [3] {"War and Peace", "Wuthering Heights", "1984"},
   new string [4] {"Casablanca", "Wizard of Oz", "Star Wars", "Birdy"},
   new string [2] {"Blue Suede Shoes", "Yellow Submarine"} };
```

Another way to create a jagged array of strings

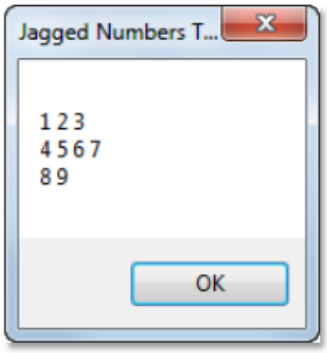
```
var titles = new[]
    {(new[] {"War and Peace", "Wuthering Heights", "1984"}),
     (new[] {"Casablanca", "Wizard of Oz", "Star Wars", "Birdy"}),
     (new[] {"Blue Suede Shoes", "Yellow Submarine"})};
```

The elements of the jagged array can be **one-dimensional** and **multi-dimensional arrays.**

```
int[][,] numbers = new int[2][,];
```

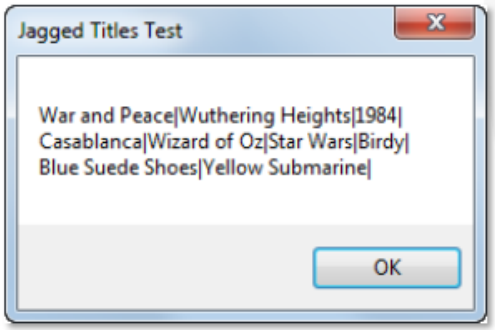
Code that displays the numbers

```
string numbersString = "";
for (int i = 0; i < numbers.GetLength(0); i++)
{
    for (int j = 0; j < numbers[i].Length; j++)
        numbersString += numbers[i][j] + " ";
    numbersString += "\n";
}
MessageBox.Show(numbersString, "Jagged Numbers Test");
```



Code that displays the titles array

```
string titlesString = "";
for (int i = 0; i < titles.GetLength(0); i++)
{
    for (int j = 0; j < titles[i].Length; j++)
        titlesString += titles[i][j] + "|";

    titlesString += "\n";
}
MessageBox.Show(titlesString, "Jagged Titles Test");
```



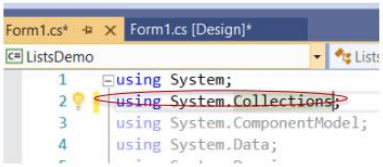
Arrays vs Collections

– A collection is an object that can hold one or more elements.

- Similarities
  - Both can store multiple elements, which can be value types or reference types
- Differences
  - Arrays are fixed in size. Collections are variable in size.
  - An array is a feature of the C# language. Collections are classes in the .NET Framework.
  - Collection classes provide methods to perform operations that arrays don't provide.
    - **Collections are Mutable**

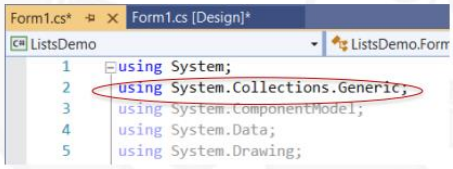
Collections

- Untyped collections (.NET 1x)
  - System.Collections namespace
  - Can store any type of object in the collection.
  - `ArrayList numbers = new ArrayList();`
  - `Data type errors are discovered at runtime`
  - `Requires casting`



Typed Collections (.NET 2.0 – Current .NET version)

- System.Collections.Generic namespace
- Can only store the specified data type.
- specify the data type in angle brackets (<>) after the name of the collection class.
- `List<int> numbers = new List<int>();`
- `Check type at compile time`
- `No casting`

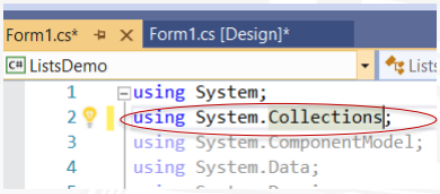


Commonly used collection classes

.NET 2.0 to 4.6	.NET 1.x	Description
`List<T>`	`ArrayList`	Uses an index to access each element. Is very efficient for accessing elements sequentially. Can be inefficient when inserting elements into the middle of a list.
`SortedList<K, V>`	`SortedList`	Uses a key to access a value, which can be any type of object. Can be inefficient for accessing elements sequentially. Is very efficient for inserting elements into the middle of a list.

.NET 2.0 to 4.6	.NET 1.x	Description
`Queue<T>`	`Queue`	Uses methods to add and remove elements.
`Stack<T>`	`Stack`	Uses methods to add and remove elements.

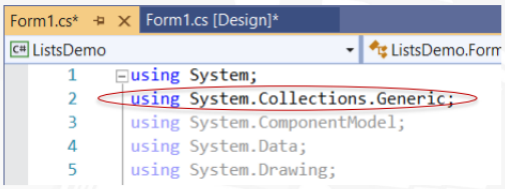
Untyped Collections



- **Using Directives**
  - `using System.Collections;`
- **Code that uses an untyped collection**

```
ArrayList numbers = new ArrayList();
numbers.Add(3);
numbers.Add(7);
numbers.Add("Test");    // will compile - causes runtime error
int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
    int number = (int)numbers[i];  // cast is required
    sum += number;
}
```

Typed collections



- **Using Directives**

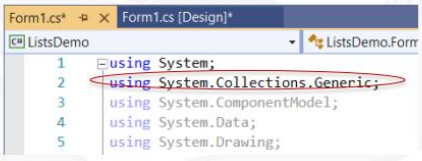
```
using System.Collections.Generic;
```

- **Code that uses a type collection**

```
List<int> numbers = new List<int>();
numbers.Add(3);
numbers.Add(7);
//numbers.Add("Test");     // won't compile
int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
    int number = numbers[i];  // no cast needed
    sum += number;
}
```

LIST<T>

- A Collection that automatically adjusts its capacity to accommodate new elements
- You can specify a different capacity when you create a list.
  - `List<string> lastNames = new List<string>(3);`
- The default capacity of a list is O elements
- When the number of elements in a list exceeds its capacity, the capacity is automatically doubled.
- The List<> class is in the System.Collections.Generic namespace
  - `List<string> titles = new List<string>();`
  - `List<decimal> prices = new List<decimal>()`



Common properties and methods of the List<> class

Indexer	Description
`[index]`	Gets or sets the element at the specified index. The index for the first item in a list is 0.
Property	**Description**
`Capacity`	Gets or sets the number of elements a list can hold.
`Count`	Gets the number of elements in a list.
Method	**Description**
`Add(object)`	Adds an element to the end of a list and returns the element's index.
`Clear()`	Removes all elements from a list and sets its Count property to zero.

Method	Description
`Contains(object)`	Returns a Boolean value that indicates if a list contains the specified object.
`Insert(index, object)`	Inserts an element into a list at the specified index.
`Remove(object)`	Removes the first occurrence of the specified object from a list.
`RemoveAt(index)`	Removes the element at the specified index of a list.
`BinarySearch(object)`	Searches a list for a specified object and returns the index for that object.
`Sort()`	Sorts the elements in a list into ascending order.

Code that causes the size of a list of names to be doubled

```
List<string> lastNames = new List<string>(3);
lastNames.Add("Petre");
lastNames.Add("Lungo");
lastNames.Add("Mtebe");
lastNames.Add("Taylor");      //Capacity is doubled to 6 elements
lastNames.Add("Spera");
lastNames.Add("Nandi");
lastNames.Add("Sarafina");    //Capacity is doubled to 12 elements
```

How to work with List properties and Methods

- Retrieving values from a list
  - `listName[index]`
- Coding a list using collection initializer
  - `List<decimal> salesTotals = new List<decimal> { 3275.68m, 4398.55m, 5289.75m, 1933.98m };`
- Code that retrieves the first value from the list
  - `decimal sales1 = salesTotals[0];  // sales1 = 3275.68`

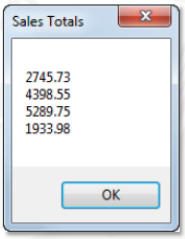
How to work with List properties and Methods

- Code that inserts and removes an element from the list

```
salesTotals.Insert(0, 2745.73m);     // insert a new first element
sales1 = salesTotals[0];             // sales1 = 2745.73
decimal sales2 = salesTotals[1];     // sales2 = 3275.68
salesTotals.RemoveAt(1);             // remove the second element
sales2 = salesTotals[1];             // sales2 = 4398.55
```

- Code that displays the list

```
string salesTotalsString = "";
foreach (decimal d in salesTotals)
     salesTotalsString += d.ToString() + "\n";
MessageBox.Show(salesTotalsString, "Sales Totals");
```



How to work with List properties and Methods

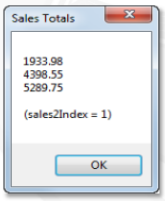
- Code that checks for an element in the list and removes it if it exists

```
decimal x = 2745.73m;
if (salesTotals.Contains(x))
    salesTotals.Remove(x);
```

- Code that sorts and searches the list

```
salesTotals.Sort();
int sales2Index = salesTotals.BinarySearch(sales2);
```

- A message box that displays the results



String Class

- Stores a collection of Unicode characters
  - String, string
  - System.String
- Immutable series of characters
  - Once you give a string a value it can not be modified
  - Methods returns a new string with modifications

String Class

- Reference type
  - Normally equality operators, == and !=, compare the object's references, but operators function differently with string than with other reference objects
    - Equality operators are defined to compare the contents or values
    - Lexicographical comparisons
    - Not defined for string (> ,< ,>=,<=)
- Includes large number of predefined methods

String Class

- Can process variables of string type as a group of characters
  - Can also access individual characters in string using an index with [ ]
    - First character is indexed by zero
  - Predefined String Class Methods
    - More than [ ] and + concatenation
    - Functional and Flexible

String Class

```
string sValue = "C# Programming";
object sObj;
string s = "C#";
```

Methods and properties	Description	Example

`Clone( )`	Returns a reference to this instance of **`string`**.	`sObj = sValue.Clone( );`
		`label1.Text =sObj;`
		`//Displays`
		`C# Programming`

`Compare( )`	Overloaded. Class method. Compares two strings.	`if(string.Compare(sValue, s) > 0)`
		`label1.Text ="sValue is " +`
		`"lexicographically greater.";`
		`//Displays`
		`sValue is lexicographically greater.`

`Concat( )`	Overloaded. Class method. Concatenates one or more string(s).	`string ns = string.Concat(sValue, s); label1.Text = ns;`
		`//Displays`
		`C# ProgrammingC#`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example

`Copy( )`	Class method. Creates a new copy of a **`string`** with the same values as the source **`string`**.	`s = string. Copy(sValue);`
		`label1.Text = s;`
		`//Displays`
		`C# Programming`

`EndsWith( )`	Determines whether the end of this instance matches the specified **`string`**.	`bool result =`
		`sValue.EndsWith("#");`
		`label1.Text = result;`
		`//Displays`
		`False`

`Equals( )`	Overloaded. Determines whether two strings have the same value.	`bool result = sValue.Equals(s);`
		`label1.Text = result;`
		`//Displays`
		`False`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example

`Format( )`	Overloaded. Class method. Replaces each format specification in a **`string`** with the textual equivalent of the corresponding object's value.	`double nu = 123.45678;`
		`string nn =`
		`string.Format("{0:F2}", nu);`
		`label1.Text =(nn);`
		`//Displays`
		`123.46`

`IndexOf( )`	Overloaded. Returns the index of the first occurrence of a **`string`**, or one or more characters, within this instance.	`label1.Text`
		`=(sValue.IndexOf("#"));`
		`//Displays`
		`1`

`Insert( )`	Inserts a specified instance of a **`string`** at a specified index position.	`s = sValue.Insert(3, ".NET ");`
		`label1.Text = s;`
		`//Displays`
		`C# .NET Programming`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example
`LastIndexOf ( )`	Overloaded. Returns the index of the last occurrence of a specified character or **`string`**.	`label1.Text = sValue.LastIndexOf("P"); //Displays 3`
`Length – Property`	Gets the number of characters.	`label1.Text = sValue.Length; //Displays 14`
`PadLeft( )`	Overloaded. Right-aligns the characters in the **`string`**, padding on the left with spaces or a specified character.	`s = sValue.PadLeft(20, '#'); label1.Text = s ; //Displays ######C# Programming label1.Text =("", PadLeft(10, '-'); //Displays ----------`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example
`PadRight( )`	Overloaded. Left-aligns the characters in the **`string`**, padding on the right with spaces or a specified character.	`s = sValue.PadRight(20, '#'); label1.Text = s ; //Displays C# Programming######`
`Remove( )`	Deletes a specified number of characters beginning at a specified position.	`s = sValue.Remove(3, 8); label1.Text = s; //Displays C# ing`
`Replace( )`	Overloaded. Replaces all occurrences of a character or **`string`** with another character or **`string`**.	`s = sValue.Replace("gram", "GRAM"); label1.Text = s ; //Displays C# ProGRAMming`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example

`Split( )`	Overloaded. Identifies the substrings in the **`string`** that are delimited by one or more characters specified in an array, then places the substrings into a **`string`** array.	`string [ ] sn = sValue.Split(' ');`
		`foreach (string i in sn)`
		`label1.Text = i + "\n";`
		`//Displays`
		`C#`
		`Programming`

`StartsWith( )`	Determines whether the beginning of the **`string`** matches the specified **`string`**.	`label1.Text =sValue.StartsWith("C#");`
		`//Displays`
		`True`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example

`Substring( )`	Overloaded. Retrieves a substring from the **`string`**.	`label1.Text = sValue.Substring(3, 7);`
		`//Displays`
		`Program`

`ToCharArray( )`	Copies the characters in the **`string`** to a character array.	`char[ ] cArray = sValue.ToCharArray(0, 2);`
		`label1.Text = cArray ;`
		`//Displays`
		`C#`

`ToLower( )`	Overloaded. Returns a copy of the **`string`** in lowercase.	`label1.Text = sValue.ToLower( );`
		`//Displays`
		`C# programming`

- **Table :** Members of the string class

String Class

Methods and properties	Description	Example

`ToString( )`	Overloaded.
	Converts the value of the instance into a **`string`**.
		`int x = 234;`
		`s = x.ToString( );`
		`label1.Text = s;`
		`//Displays`
		`234`

`ToUpper( )`	Overloaded. Returns a copy of the **`string`** in uppercase.	`label1.Text =sValue.ToUpper( );`
		`//Displays`
		`C# PROGRAMMING`

`Trim( )`
`TrimEnd( )`
`TrimStart( )`
	Overloaded. Removes all occurrences of a set of specified characters from the beginning and end.	`s = sValue.Trim('g','i','n','m','C' );`
		`label1.Text = s;`
		`//Displays`
		`# Progra`

- **Table :** Members of the string class

String Class

- Class methods: such as Compare, Concat, and Copy, prefix the name of the method in the call with the string data type (e.g. s = string.Copy(sValue);).
- Instance methods: Svalue.ToUpper()
- Most string member arguments that take a string object accept a string literal
  - @-quoted string literals, start with the @ symbol

```
label1.Text =(@"hello \t world");

//Displays hello \t world
```

String Interpolation

- New from C# 6.0
- Process of evaluating a string literal containing one or more placeholders and replacing with values
  - string.Format("{0} {1}\nAmt: {2 :F2}", first, last, amt);
- Placing $ before the string literal to indicate string interpolation should occur
  - Place identifier insider curly braces

    return $"{identifier};

String Interpolation Example

```
string ReturnInterperpolatedString( )
{
 string first = "Joann";
 string first = "Smith";
 double amt = 23.45675;
 return $"{first} {last}\nAmt: {amt : F2}";
}
```

Previously you would have written,

return string.Format("{0} {1}\nAmt: {2 :F2}", first, last, amt);

- Also previously needed to invoke the string.Format( ) method to get **amt** formatted

Printing Windows Form:

COLLECTIONS

Collection are classes that enable you to store and retrieve groups of objects. Collections are an essential tool for managing multiple items. They are also central to developing graphical applications. Controls such as drop-down list boxes and menus are typically data-bound to collections. To instantiate objects of these collections, you need to include a using System.Collections and System.Collections.Generic; to the list of namespaces.

TYPES OF COLLECTIONS

UNTYPED COLLECTIONS (.NET 1X)

The collection classes in the System.Collections namespace allow you to create untyped collections. With an untyped collection, you can store any type of object in the collection.

Example : –**`ArrayList numbers = new ArrayList();`**

Class	Description
ArrayList	Represents an array of objects whose size is dynamically increased as req
Hashtable	Represents a collection of key/value pairs that are organized based on the
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

TYPED COLLECTION (.NET 2.0 – CURRENT .NET VERSION)

The collection classes in the System.Collections.Generic namespace uses a feature known as generics to allow you to create typed collections that can only store the specified data type. With a typed collection, you specify the data type in angle brackets (<>) after the name of the collection class.

Example : –**`List<int> numbers = new List<int>();`**

Class	Description
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized bas
List<T>	Represents a list of objects that can be accessed by index. Provi
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

TYPE VS UNTYPED COLLECTIONS

Typed collections have two advantages over untyped collections. First, they check the type of each element at compile-time and prevent runtime errors from occurring. Second, they reduce the amount of casting that's needed when retrieving objects. If you want to use untyped collections, you can edit the using directive at the beginning of the form class so it refers to the System.Collections namespace instead of the System.Collections.Generic namespace that's often included by default in new applications. Then, you need to make sure to cast the elements to the appropriate type when you retrieve them from the collection.

INF1003F- COMMERCIAL PROGRAMING

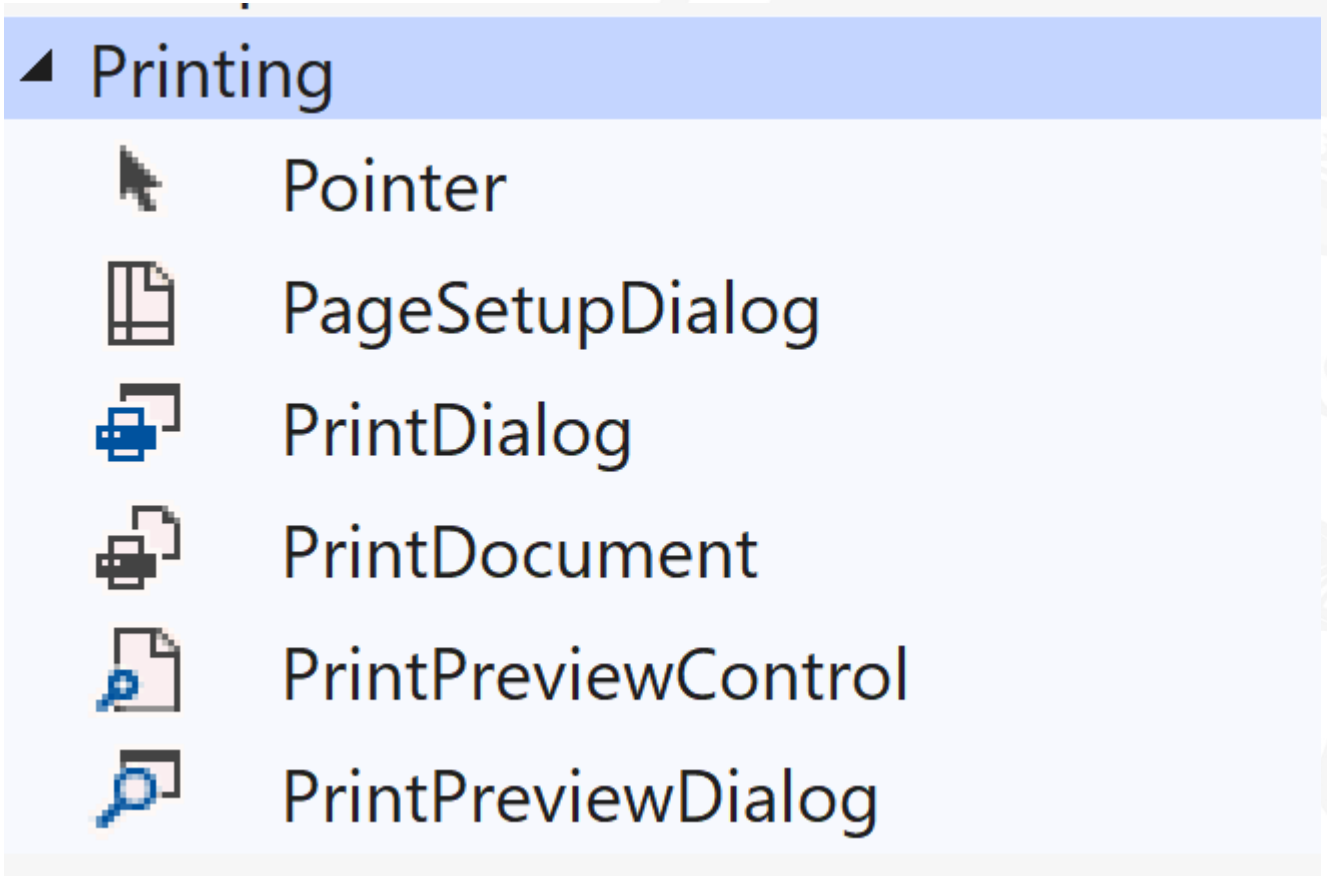
INF 1003F: COMMERCIAL PROGRAMMING
Zainab Ruhwanya
Department of Information Systems

Printing in Windows Forms

INF 1003F: COMMERCIAL PROGRAMMING

INFORMATION SYSTEMS DEPARTMENT
University of Cape Town

Printing in a Window Form



Recap Using Dialog Boxes

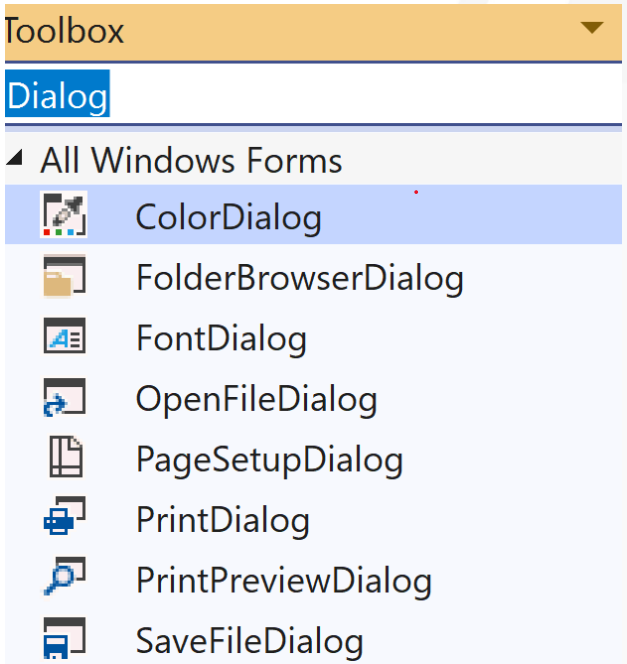
Using Dialog Boxes

You use dialog boxes to display information and prompt for input from the user.

Recap Dialog Boxes

- A dialog box is a temporary window an application creates to retrieve user input.
- An application typically uses dialog boxes to prompt the user for additional information for menu items.
- A dialog box usually contains one or more controls (child windows) with which the user enters text, chooses options, or directs the action.
- https://docs.microsoft.com/en-us/windows/win32/dlgbox/dialog-boxes

Dialog Boxes Control



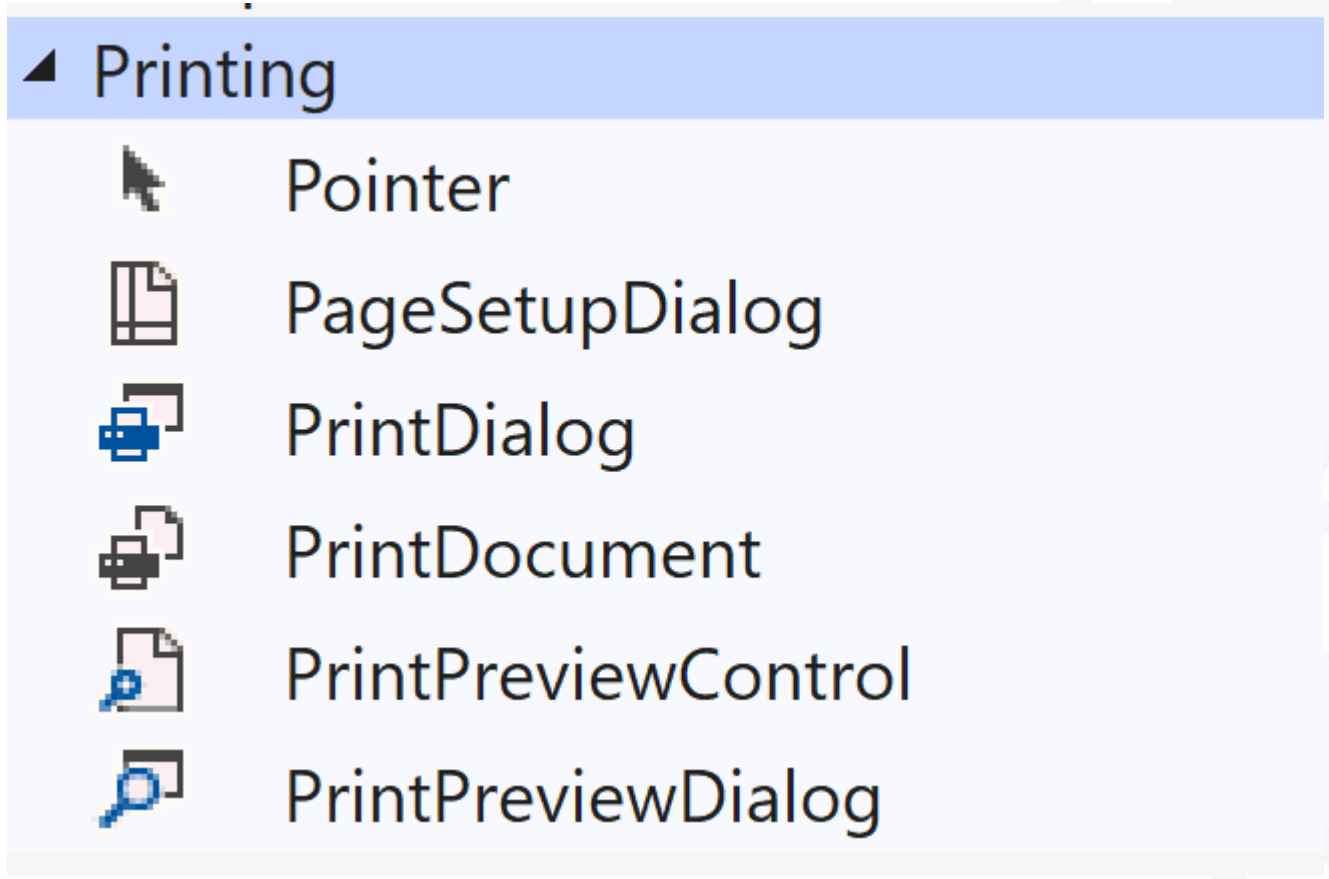
Recap MessageBox

MessageBox

Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

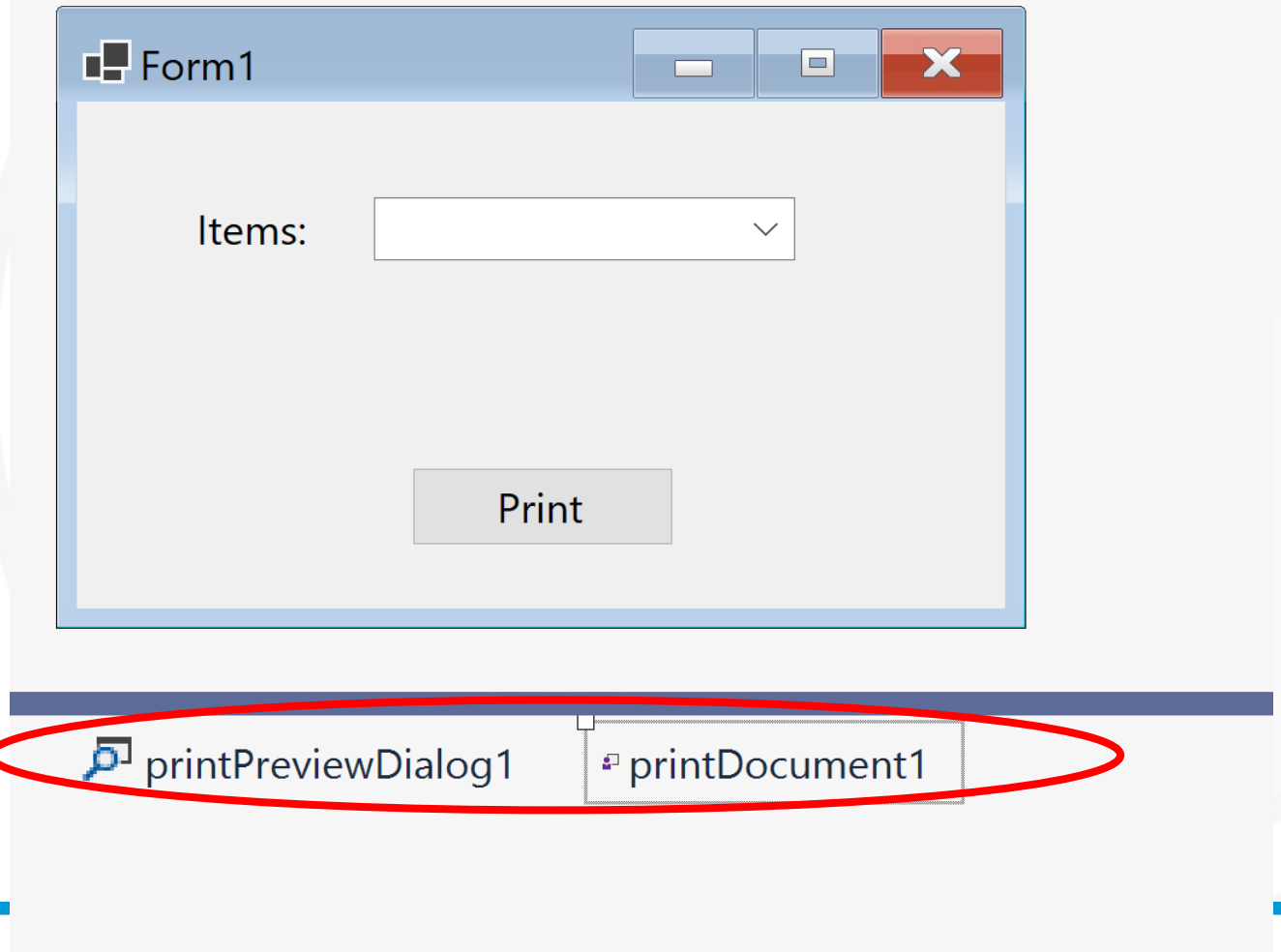
How to: Print in Windows Forms Using Print Preview

Printing


Printing
Pointer
PageSetupDialog
PrintDialog
PrintDocument
PrintPreviewControl
PrintPreviewDialog

Add Printing

- Add printPreviewDialog and PrintDocument to your form



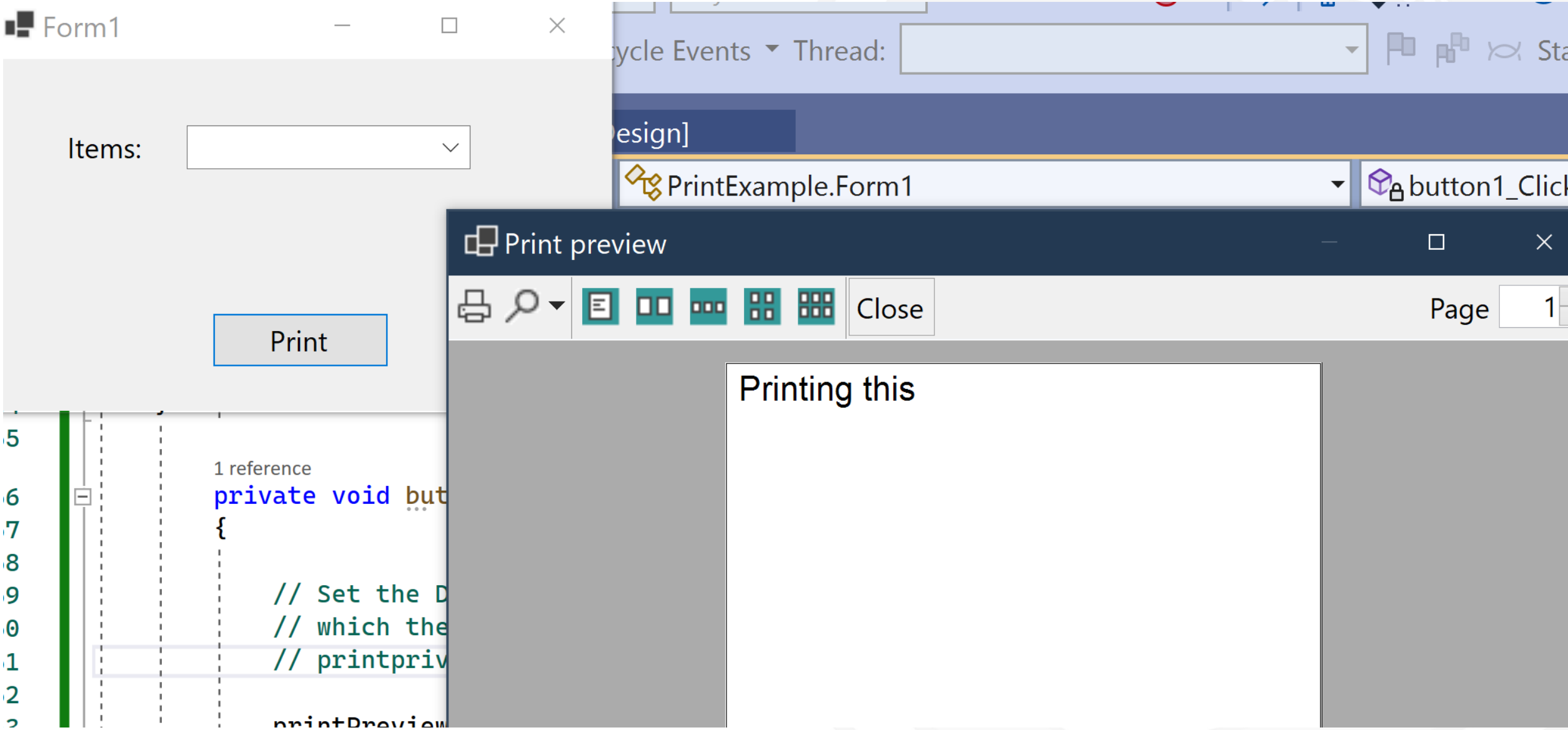
Create a click event from a PrintDocument1 object

```
using System.Drawing.Printing;

1 reference
private void printDocument1_PrintPage(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
        // Insert code to render the page here.
                    // This code will be called when the control is drawn.

                    // The following code will render a simple
                    // message on the printed/previewed document.
                    string text = "Printing this";
                        Font printFont = new System.Drawing.Font
                         ("Arial", 35, System.Drawing.FontStyle.Regular);

                    // Draw the string .
                    e.Graphics.DrawString(text, printFont,
                                                Brushes.Black, 10, 10);
}
```

Calling ShowDialog method on the PrintPreviewDialog control.

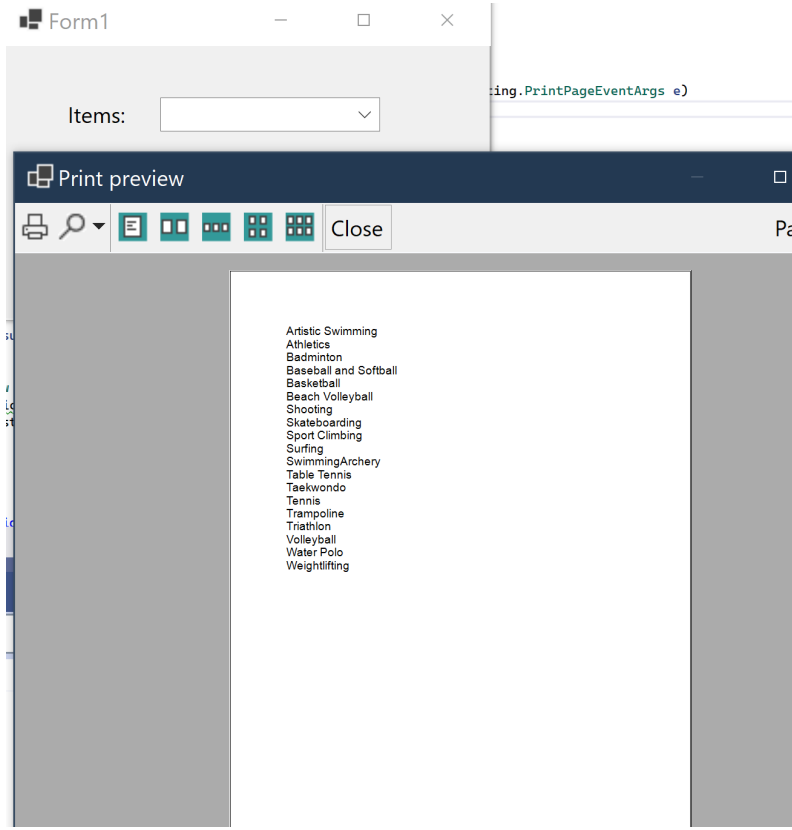
- The following code example shows the Click event-handling method for a button on the form. This event-handling method calls the methods to read the document and show the print preview dialog.

```
1 reference
private void button1_Click(object sender, EventArgs e)
{

    // Set the Document property to the PrintDocument for
    // which the PrintPage Event has been handled. To display the
    // printprivew either this property must be set

    printPreviewDialog1.Document = printDocument1;
    printPreviewDialog1.ShowDialog();



}
```

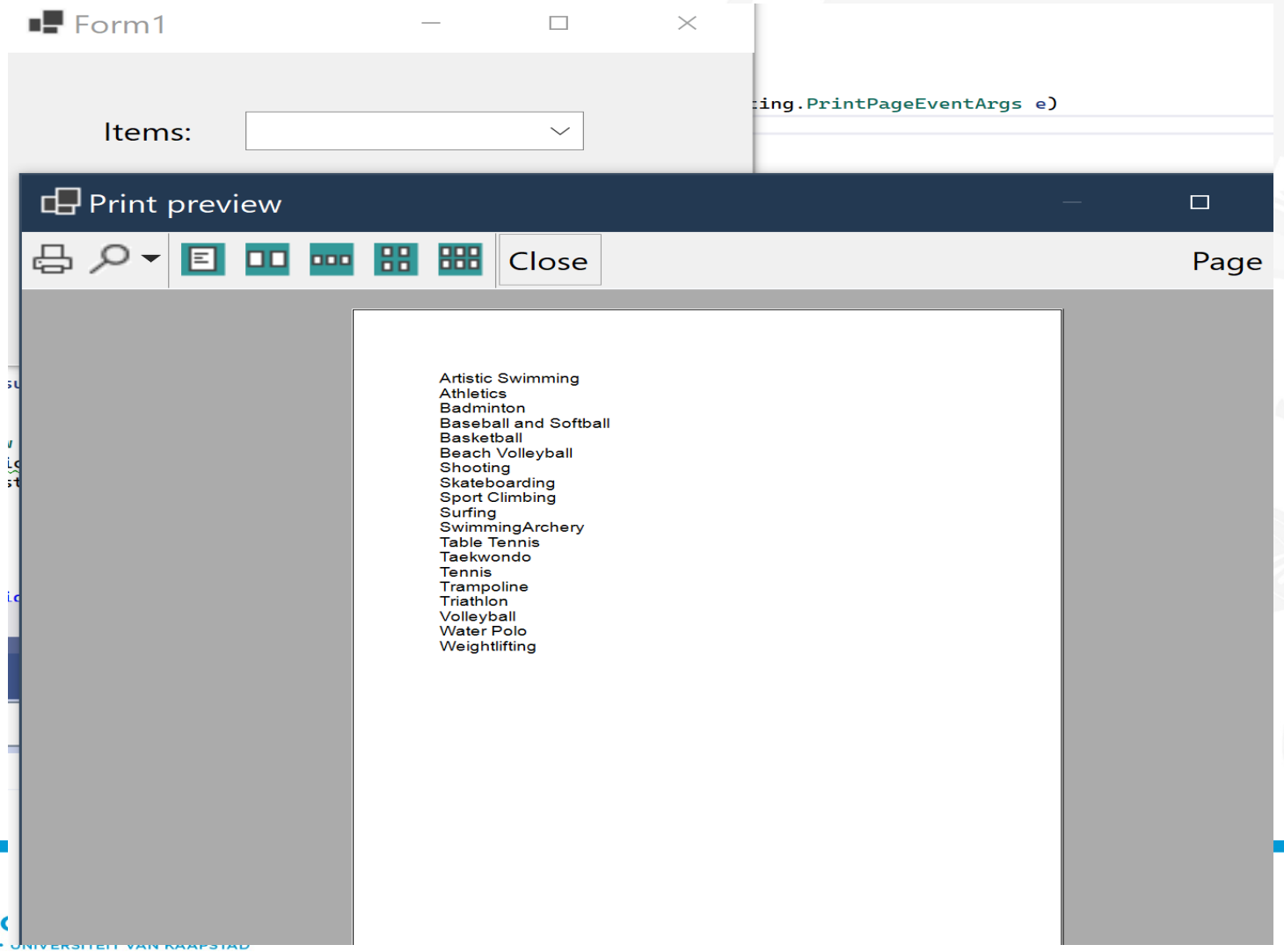

Form1
Items:
Print
ycle Events
Thread:
esign]
PrintExample.Form1
button1_Click
Print preview
Close
Page
1
Printing this
1 reference
private void but
{
// Set the D
// which the
// printpriv

Example using Collections of items

```
1 reference
private void printDocument1_PrintPage(object sender, System.Drawing.Printing.PrintPageEventArgs e)
{
    // Insert code to render the page here.
    // This code will be called when the control is drawn.

    // The following code will render a simple
    // message on the printed document.
   // string text = "Printing this";
    string result="";
    System.Drawing.Font printFont = new System.Drawing.Font
        ("Arial", 15, System.Drawing.FontStyle.Regular);

    foreach (string sports in comboBox1.Items)
    {
        result += sports + " \n";

    }
    // Draw the content.
   e.Graphics.DrawString(result, printFont,
        System.Drawing.Brushes.Black, 100, 100);


}
```



Output

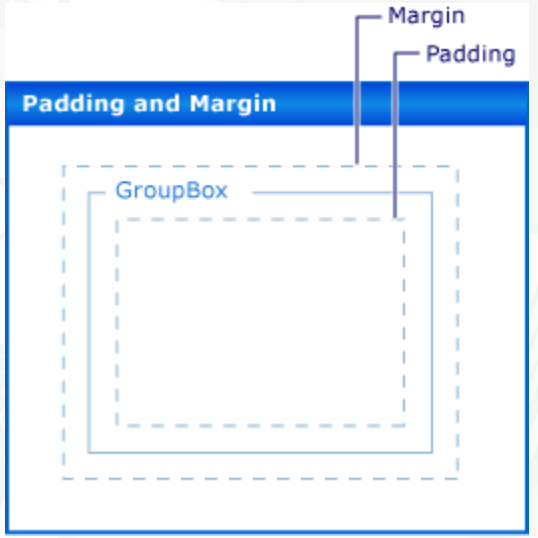

Form1
Items:
ing.PrintPageEventArgs e)
Print preview
Close
Page
Artistic Swimming
Athletics
Badminton
Baseball and Softball
Basketball
Beach Volleyball
Shooting
Skateboarding
Sport Climbing
Surfing
SwimmingArchery
Table Tennis
Taekwondo
Tennis
Trampoline
Triathlon
Volleyball
Water Polo
Weightlifting

Important Properties

- Font - Gets or sets the font used for the control.
- Example
-
```
Font printFont = new Font
                ("Arial", 15, FontStyle.Regular);
 Font BodyFont = new Font("Arial", 12);
```

Margins

- Margins
- A Margins that represents the margins, in hundredths of an inch, for the page. The default is 1-inch margins on all sides
- Left, Right, Top, and Bottom are properties that define the margins



MarginBounds

- Property of PrintPageEventArgs Class
  - Provides data for the PrintPage event.
- MarginBounds
  - Gets the rectangular area that represents the portion of the page inside the margins.

Graphics.DrawString Method (Overloads – 5+)

- Draws the specified text string at the specified location with the specified Brush and Font objects.
- **DrawString(String, Font, Brush, Single, Single)**

String
String to draw.
Font
Font that defines the text format of the string.
brush
Brush
Brush that determines the color and texture of the drawn text.
x
Single
The x-coordinate of the upper-left corner of the drawn text.
y
Single
The y-coordinate of the upper-left corner of the drawn text.

Drawing Content on the screen

```
// Draws the content on the screen.
e.Graphics.DrawString(text, printFont,
   Brushes.Black, 10, 10);
```

- VS – DEMO.

Classes

To simplify development and maintenance, many applications use a three-layered architecture to separate the application's user interface, business rules, and database processing.

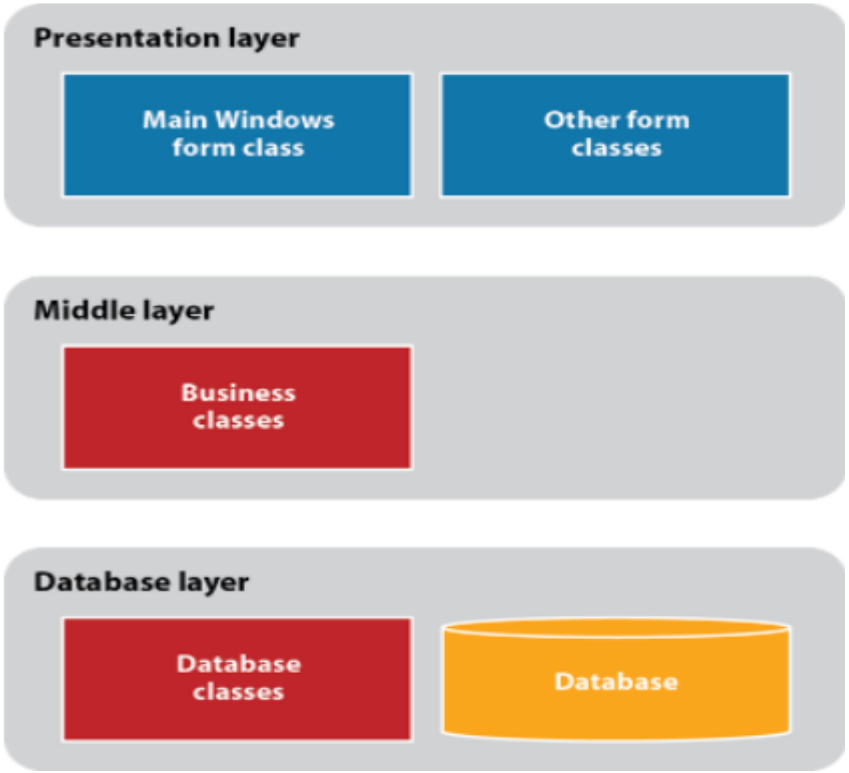
Classes are used to implement the functions performed at each layer of the architecture.

• The classes in the **presentation laye**r control the application's user interface. For a Windows Forms application, the user interface consists of the various forms that make up the application.
• The classes in the **database layer handle** all of the application's data processing.

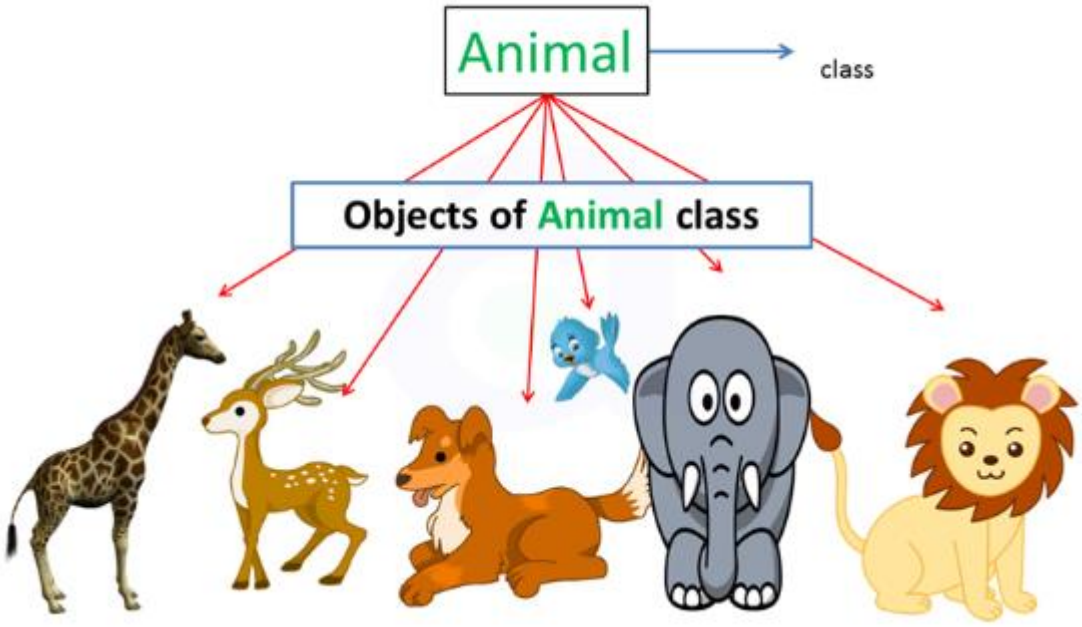
• The classes in the **middle layer**, sometimes called the business layer or domain layer, act as an interface between the classes in the presentation and database layers.

These classes can represent business entities, such as customers or products, or they can implement business rules, such as discount or credit policies.

• When the classes in the middle layer represent business entities, the classes can be referred to as business classes, and the objects that are created from these classes can be referred to as business objects.
• Often, the classes that make up the database layer and the middle layer are implemented in class libraries that can be shared among applications.



CLASSES AND OBJECTS



Definitions:

Classes: Classes are the main focus in Object oriented programming (OOP).

- A class is like a blueprint of a specific ***object.***
- A class is a collection of ***objects*** of similar type.
- A class is a model for creating ***objects***.

Objects: an *instance* of a class, and the process of creating an object is called *instantiation*.

- What that means is that ⇒ an *Object is an actual physical existence of a Class.*

  It is a real entity whose structure, identity and behavio r are decided by a Class.

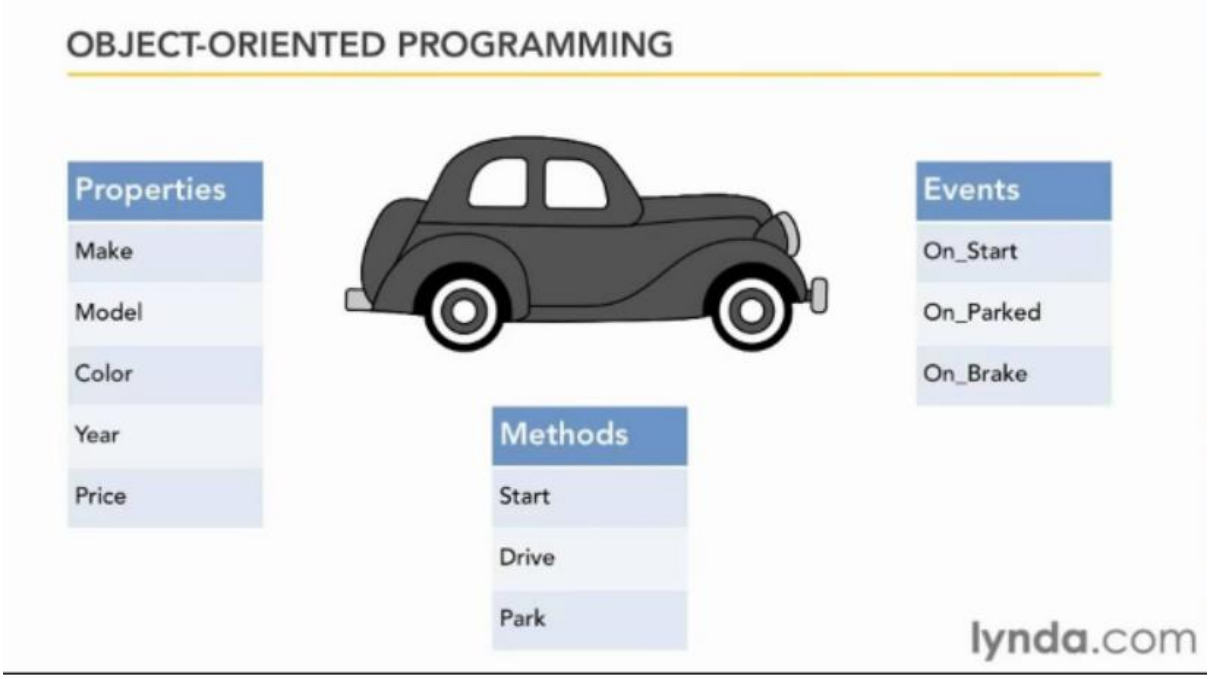
- An object is a block of memory that has been allocated and configured according to the ***blueprint***.
- Blueprint: An object is a combination of ***data/fields and methods***.
  - Fields are actual variables in your object that stores a particular piece of information.

    - The data and the methods are called *members* of an object.
- These objects ***communicate together through methods*** **-** actions they perform.
- Each object can ***receive messages, send messages and process data.***
- A program may ***create many objects of the same class.***
- Objects are also called ***instances,*** and they can be stored in either a named variable or in an array or collection
- Creating an object means ***allocating memory to store the data of variables temporarily***. i.e. we create an object of class to store data temporarily.

Examples

If a class can be defined as ***a template/blueprint that describes the behavior/state that an object of its type support*** - what does this mean?

- If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc.
- All these objects have a ***state*** and a ***behavior***.
- If we consider a dog, then its ***state or characteristics/attributes*** include - name, breed, color.
- If we consider a dog, then its ***behavior*** is - barking, wagging the tail, running
- If we consider an Employee:
  the ***state/fields/ characteristics/attributes*** include Name, Id, Salary etc respectively.
- If we consider a Car as a class then examples of objects of this car include Toyota, Nissan, Audi, Golf etc
- If we consider a Car as a class
  then: ***state/fields/ characteristics/attributes*** include wheels, doors, seating capacity.
- If we consider a Car as a class then its ***behavior*** is: accelerate, stop, display how much fuel is left etc.



A class and its *Properties*

- `private` variables can only be accessed within the same class (an outside class has no access to it). *However, sometimes we need to access them - and it can be done with* ***properties*****.**
- A property is a public class member
- It allows controlled access to the internal state of an object through fields and methods within that object.
- Properties have ***get*** and ***set*** procedures, which provide more control on how values are set or returned.
- We use a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures

EXAMPLE

```
class Product
{
  private string code; // field

  public string code   // property
  {
    get { return code; }   // get method
```

```
    set { code = value; }  // set method
  }
}
```

- The `code` property is associated with the `code` field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.
- The `get` method returns the value of the variable `code`.
  - The body of the `get` accessor resembles that of a method.
  - It must return a value of the property type.
  - The execution of the `get` accessor is equivalent to reading the value of the field.

- The `set` method assigns a `value` to the `code` variable. The `value` keyword represents the value we assign to the property.
  - The `set` accessor resembles a method whose return type is void.
  - It uses an implicit parameter called `value`, whose type is the type of the property. In the example, a `set` accessor is added to the code property:

Instantiating a class

- A class can be used to create many objects.
- Objects created at runtime from a class are called *instances* of that particular class.

Code that creates these two object instances

```
Product product1, product2;

product1 = new Product("CS15", "Murach's C# 2015", 56.50m);

product2 = new Product("VB15", "Murach's Visual Basic 2015",56.50m);
```

Code that creates the object instances with object initializers

```
product1 = new Product { Code = "CS15",Description = "Murach's C# 2015",
Price = 56.50m };

product2 = new Product { Code = "VB15",Description = "Murach's Visual Basic
2015", Price = 56.50m };
```

Statements that call these constructors

```
Product product1 = new Product();

Product product2 = new Product("CS15", "Murach's C# 2015", 56.50m);

Product product3 = new Product(txtCode.Text);
```

Summary on classes

- Software objects have a ***state*** and a ***behavior***.
- A software object's **state is stored in fields** and ***behavior is shown via methods***.
- Methods operate on the internal state of an object and the *object-to-object communication is done via methods*.
- A good rule of thumb to identify classes in Object-Oriented programming is that:
    - Classes are the *Nouns* in your analysis of the problem
    - Methods in a class correspond to the *Verbs* that the noun does
    - Properties are the *Adjectives* that describe the noun
- Each class has Fields and properties which represent information that an object contains.
- Fields are like variables because they can be read or set directly, subject to applicable access modifiers.
- Properties have `get` and `set` accessors, which provide ***more control*** on how values are set or returned.
- Benefit: The fields of a class can be made read-only or write-only.
- Benefit: A class can have total control over what is stored in its fields.

- In short, a class has:
    - Accessibility level: Public, private, etc
    - Class name
    - Fields: actual variables in your object that stores a particular piece of information,
    - Properties and
    - Methods/ functions

Encapsulation

- is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- is a *mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit*.
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Because of this, it is also known as **data hiding**
- It lets you control the data and operations within a class that are exposed to other classes.

To achieve encapsulation

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

⇒⇒Fields are like variables because they can be read or set directly, subject to applicable access modifiers.

⇒⇒ Properties have get and set accessors, which provide ***more control*** on how values are set or returned.

A class and its *Properties*

- private variables can only be accessed within the same class (an outside class has no access to it). *However, sometimes we need to access them - and it can be done with* ***properties.***
- A property is a public class member
- It allows controlled access to the internal state of an object through fields and methods within that object.
- Properties have ***get*** and ***set*** procedures, which provide more control on how values are set or returned.
- We use a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures

EXAMPLE

```
class Product
{
  private string code; // field

  public string code   // property
```

```
 {
   get { return code; }   // get method
   set { code = value; }  // set method
 }
}
```

- The code property is associated with the code field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.
- The get method returns the value of the variable code.
    - The body of the get accessor resembles that of a method.
    - It must return a value of the property type.
    - The execution of the get accessor is equivalent to reading the value of the field.
- The set method assigns a value to the code variable. The value keyword represents the value we assign to the property.
    - The set accessor resembles a method whose return type is void.
    - It uses an implicit parameter called value, whose type is the type of the property. In the example, a set accessor is added to the code property:

Constructors

- A constructor is a method whose name is the same as the name of its type - A special type of method that's executed when an object is instantiated.
- Its method signature includes only the method name and its parameter list; it does not include a return type.
- Whenever a class is created, its constructor is called.
- A class may have multiple constructors that take different arguments.
- *Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read*.
- For more on constructors, see the summary here.

Example

A constructor with one parameter

```
public Product(string code)
{
```

```
    Product p = ProductDB.GetProduct(code);
    this.Code = p.Code;
    this.Description = p.Description;
    this.Price = p.Price;
}
```

A constructor with three parameters

```
public Product(string code, string description, decimal price)
{
    this.Code = code;
    this.Description = description;
    this.Price = price;
}
```

Instantiating a class

- A class can be used to create many objects.
- Objects created at runtime from a class are called *instances* of that particular class.

Code that creates these two object instances

```
Product product1, product2;
product1 = new Product("CS15", "Murach's C# 2015", 56.50m);
product2 = new Product("VB15", "Murach's Visual Basic 2015",56.50m);
```

Code that creates the object instances with object initializers

```
product1 = new Product { Code = "CS15",Description = "Murach's C# 2015", Price = 56.50m };
product2 = new Product { Code = "VB15",Description = "Murach's Visual Basic 2015", Price =
56.50m };
```

Statements that call these constructors

```
Product product1 = new Product();
Product product2 = new Product("CS15", "Murach's C# 2015", 56.50m);
Product product3 = new Product(txtCode.Text);
```

Summary

Each class should be able to control its data - *what does this mean?*

- The data of a class should be encapsulated within a class using *data hiding*.
- This means that a group of related properties, methods, and other members are treated as a single unit or object.
- Each class has Fields and properties which represent information that an object contains.

- Fields are like variables because they can be read or set directly, subject to applicable access modifiers.
- Properties have get and set accessors, which provide ***more control*** on how values are set or returned.
- Benefit: The fields of a class can be made read-only or write-only.
- Benefit: A class can have total control over what is stored in its fields.

The Product class

```
namespace ProductMaintenance
{
    public class Product
    {
        private string code;
        private string description;
        private decimal price;

        public Product(){}

        public Product(string code, string description,
                       decimal price)
        {
            this.Code = code;
            this.Description = description;
            this.Price = price;
        }

        public string Code
        {
            get
            {
                return code;
            }
            set
            {
                code = value;
            }
        }

        public string Description
        {
            get
            {
                return description;
            }
            set
            {
                description = value;
            }
        }
```

Fields

An empty constructor

A custom constructor

The Code property

The Description property

```
public decimal Price
{
    get
    {
        return price;
    }
    set
    {
        price = value;
    }
}

public string GetDisplayText(string sep)
{
    return code + sep + price.ToString("c") + sep +
           description;
}
```

The Price property

The GetDisplayText method

Revisit Week 11 for Examples