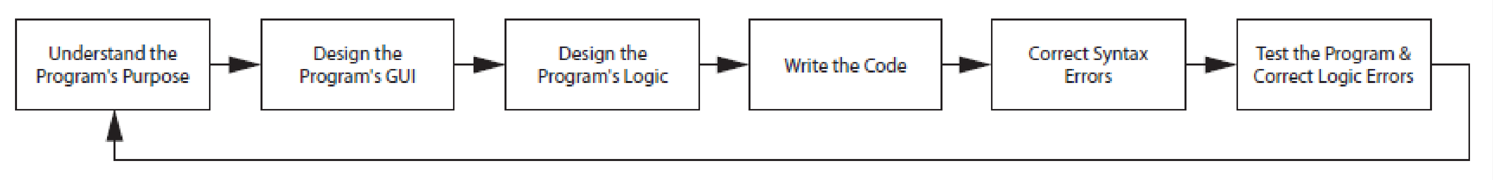
1.7 The Program Development Process



Understand the Program's Purpose

Design the Program's GUI

Design the Program's Logic

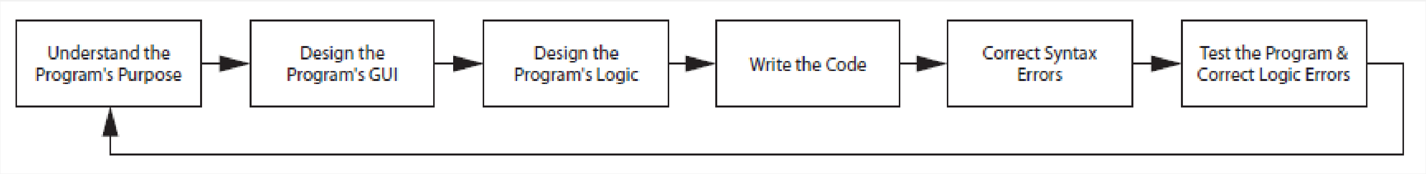
Write the Code

Correct Syntax Errors

Test the Program & Correct Logic Errors

AUBURN UNIVERSITY

1.7 The Program Development Process



Understand the Program's Purpose

Design the Program's GUI

Design the Program's Logic

Write the Code

Correct Syntax Errors

Test the Program & Correct Logic Errors

1. Understand the Program's Purpose

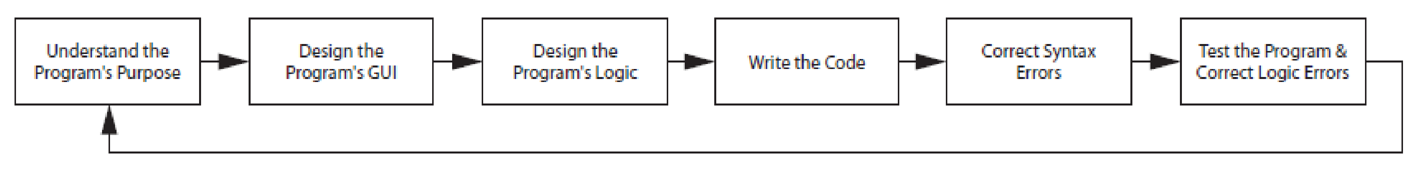
Most programs perform the following three-step process:

Step 1. Input is received.

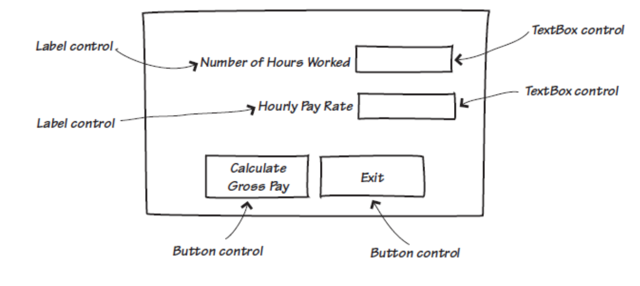
Step 2. Some process is performed on the input.

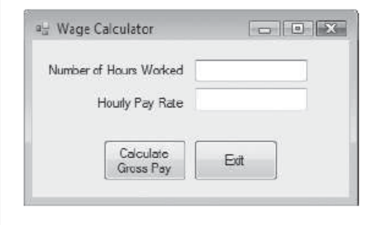
Step 3. Output is produced.

1.7 The Program Development Process

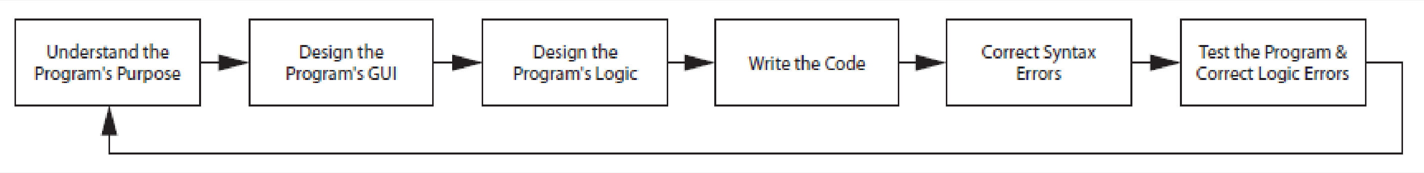

Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors

2. Design the Graphical User Interface (GUI)





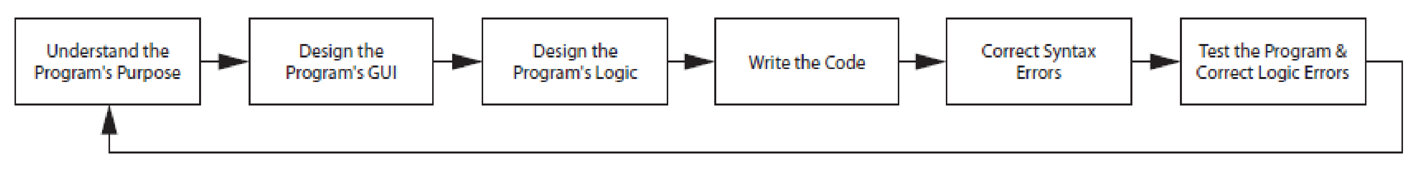
1.7 The Program Development Process


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors

3. Design the Program's Logic
Break down each task that the program must perform into a series of logical steps.

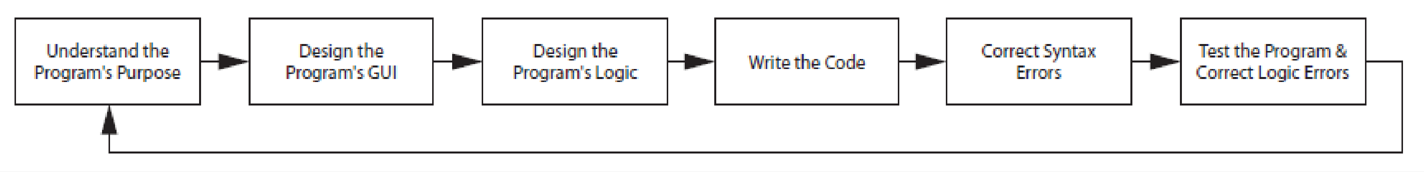
AUBURN UNIVERSITY

1.7 The Program Development Process


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors

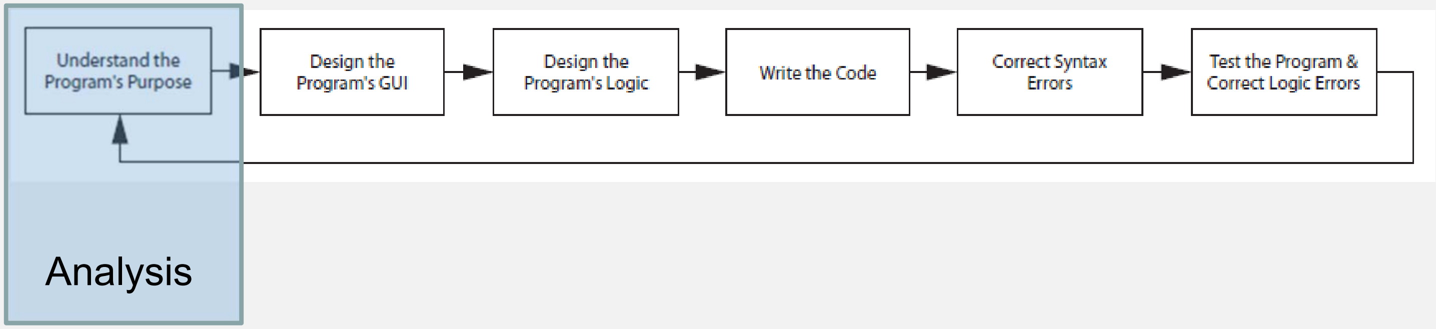
4. Write the Code
5. Correct Syntax Errors
6. Test the Program and Correct Logic Errors

1.7 The Program Development Process


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors

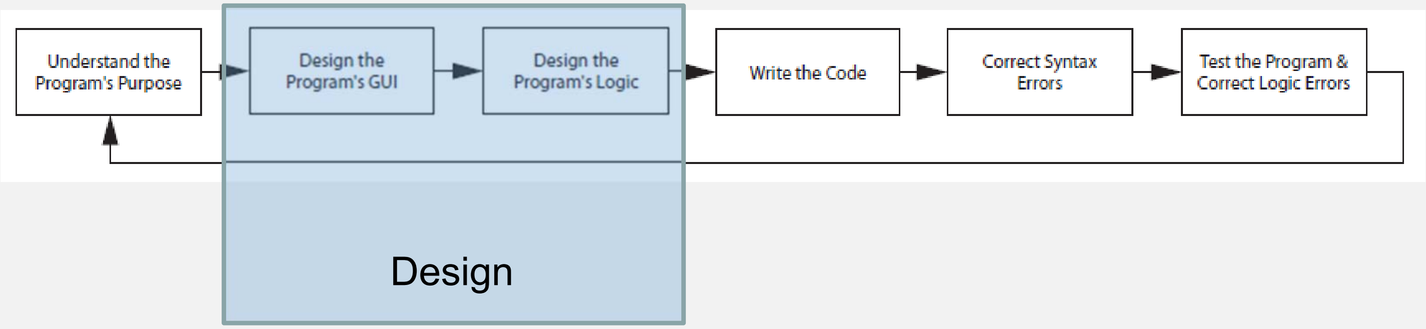
AUBURN UNIVERSITY

1.7 The Program Development Process


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors
Analysis

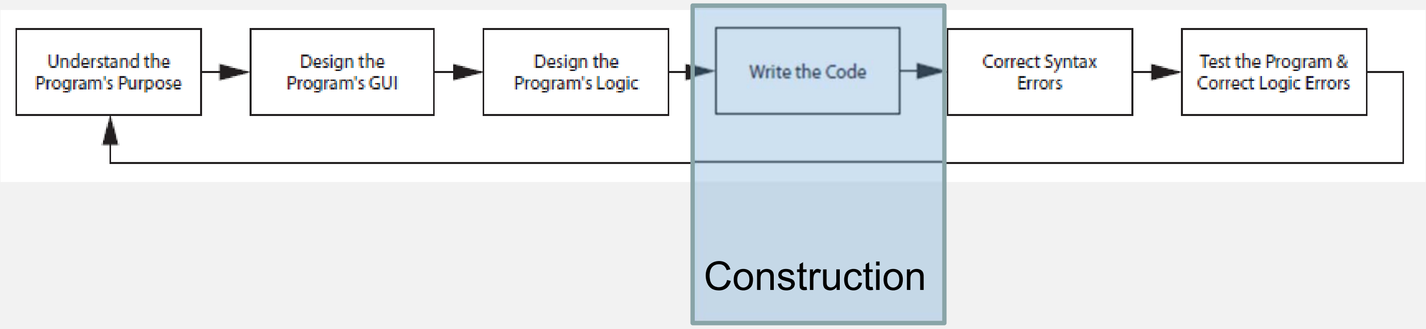
AUBURN UNIVERSITY

1.7 The Program Development Process


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors
Design

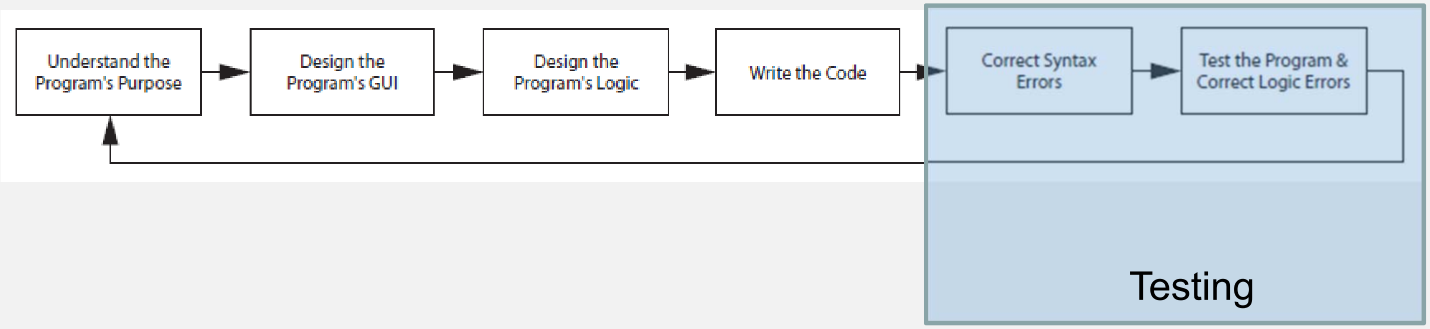
AUBURN UNIVERSITY

1.7 The Program Development Process


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors

Construction

1.7 The Program Development Process



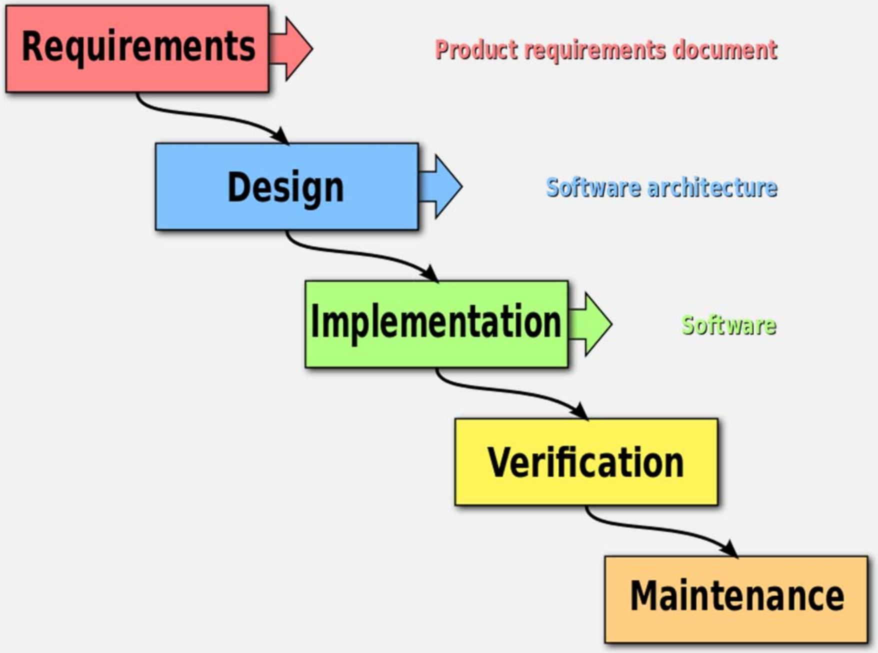
Understand the Program's Purpose → Design the Program's GUI → Design the Program's Logic → Write the Code → Correct Syntax Errors → Test the Program & Correct Logic Errors

Testing

Software
Process
Models

AUBURN
UNIVERSITY

Waterfall Model



Requirements

Product requirements document

Design

Software architecture

Implementation

Software

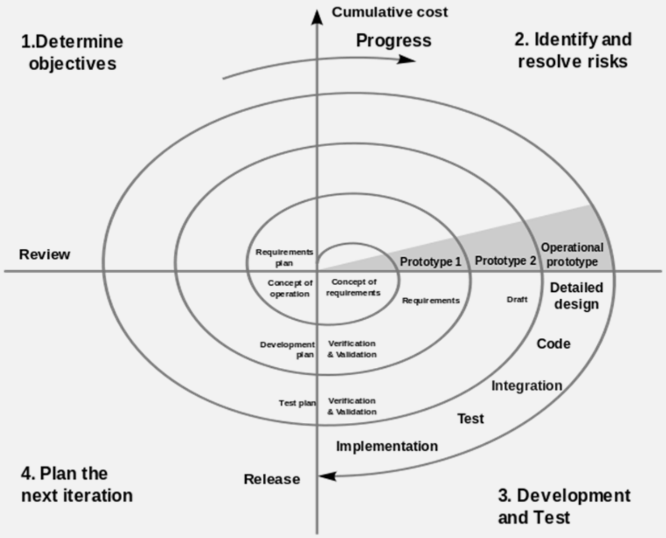
Verification

Maintenance

https://en.wikipedia.org/wiki/Software_development_process

Spiral Model (Boehm, 1988)

Software Process Models

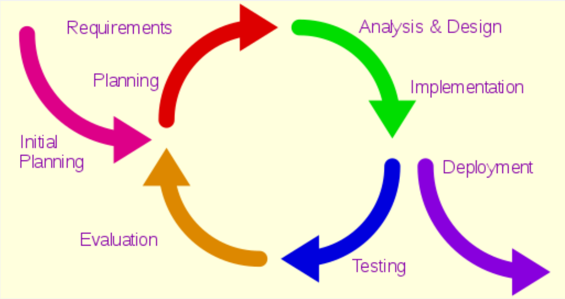


https://en.wikipedia.org/wiki/Software_development_process

Iterative and incremental development

Software Process Models

AUBURN UNIVERSITY

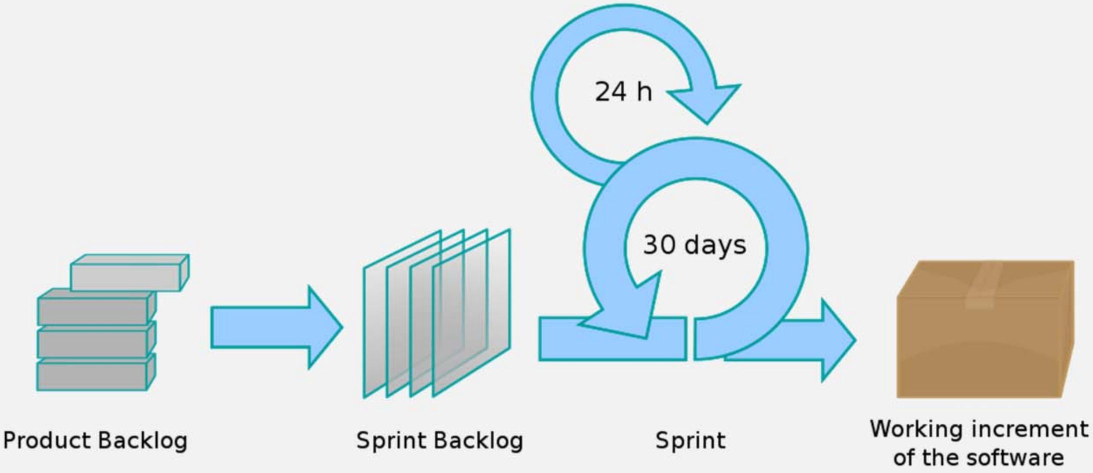


https://en.wikipedia.org/wiki/Iterative_and_incremental_development

Agile Software Development

Software Process Models

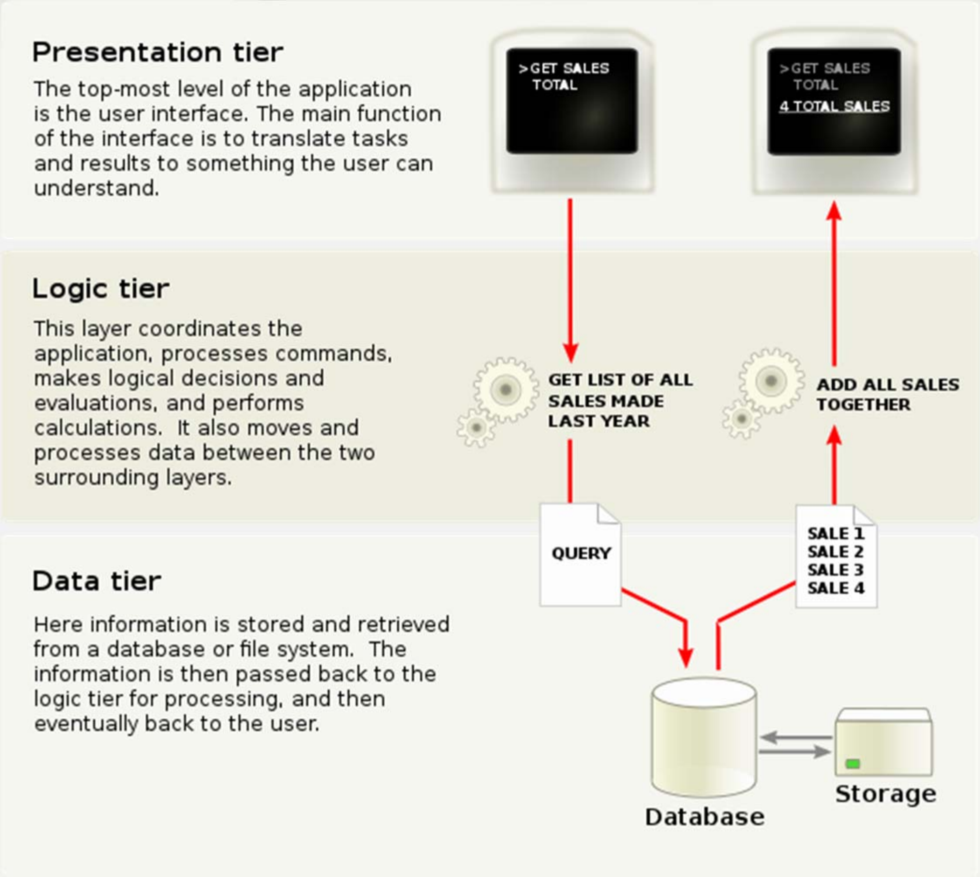
AUBURN UNIVERSITY



https://en.wikipedia.org/wiki/Scrum_(software_development)

https://en.wikipedia.org/wiki/Agile_software_development

3-Tier Architecture



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

https://en.wikipedia.org/wiki/Multitier_architecture#Three-tier_architecture

4.1 Decision Structures

- A **control structure** is a logical design that controls the order in which statements execute
- A **sequence structure** is a set of statements that execute in the order that they appear
- A **decision structure** execute statements only under certain circumstances
  - A specific action is performed only if a certain condition exists
  - Also known as a selection structure

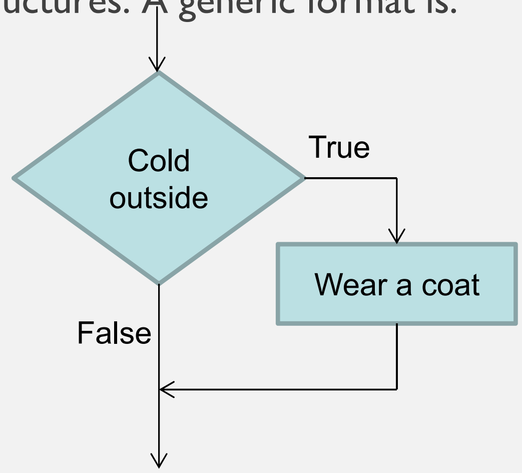
AUBURN UNIVERSITY

A Simple Decision Structure

AUBURN UNIVERSITY

- The flowchart is a single-alternative decision structure
- It provides only one alternative path of execution
- In C#, you can use the ***if*** statement to write such structures. A generic format is:

```
if (expression)
{
   Statements;
   Statements;
   etc.;
}
```



- The *expression* is a Boolean expression that can be evaluated as either true or false

Relational Operators

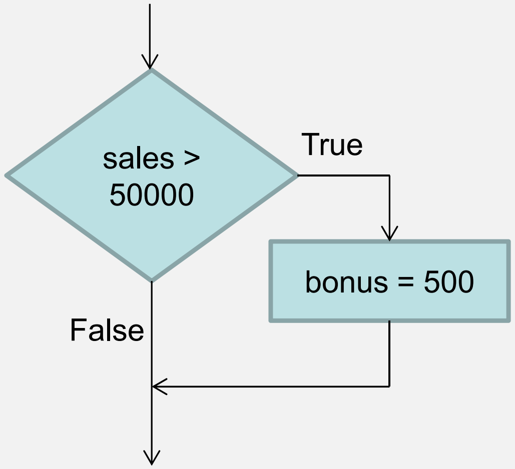
- A relational operator determines whether a specific relationship exists between two values

Operator	Meaning	Expression	Meaning
>	Greater than	x > y	Is x greater than y?
<	Less than	x < y	Is x less than y?
>=	Greater than or equal to	x >= y	Is x greater than or equal to y?
<=	Less than or equal to	x <= y	Is x less than or equal to you?
==	Equal to	x == y	Is x equal to y?
!=	Not equal to	x != y	Is x not equal to you?

AUBURN UNIVERSITY

if Statement with Boolean Expression

```
if (sales > 50000)
{
   bonus =  500;
}
```

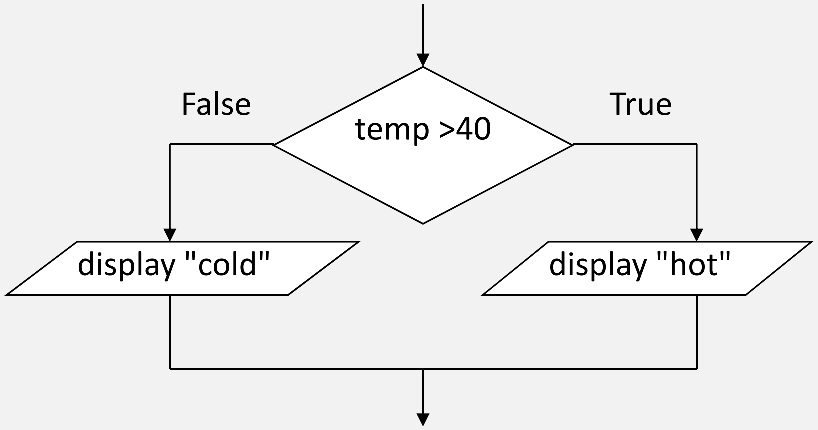


4.2 The *if-else* statement

- An ***if-else*** statement will execute one block of statement if its Boolean expression is true or another block if its Boolean expression is false
- It has two parts: an *if* clause and an *else* clause
- In C#, a generic format looks:

```
if (expression)
{
  statements;
}
else
{
  statements;
}
```

Example of *if-else* Statement



```
if (temp > 40)
{
    MessageBox.Show("hot");
}
else
{
    MessageBox.Show("cold");
}
```

AUBURN UNIVERSITY

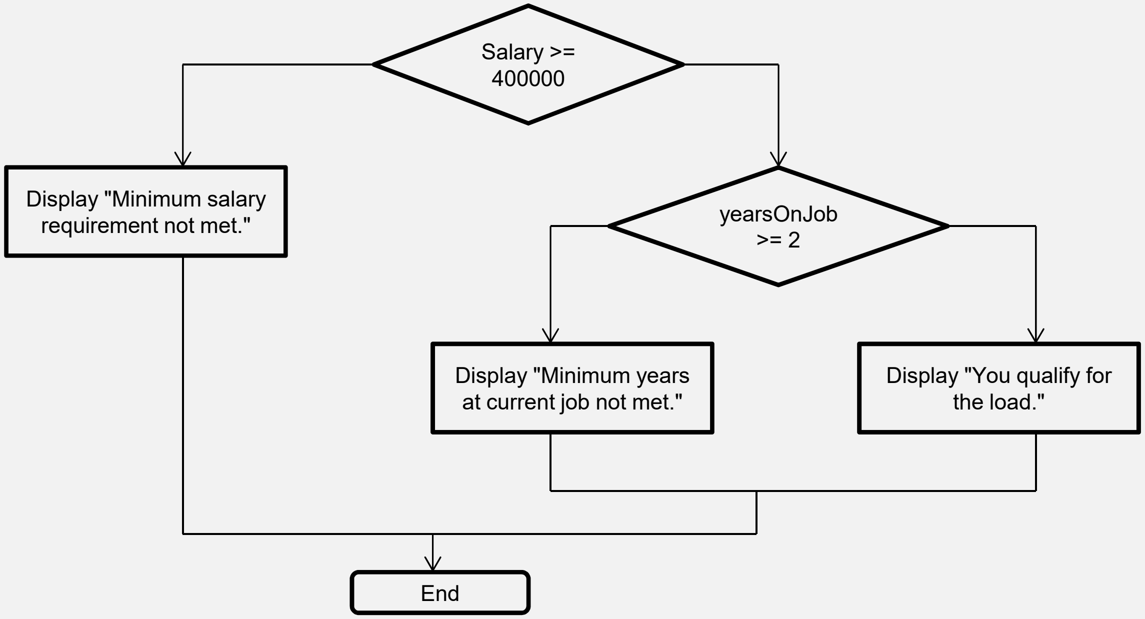
4.3 Nested Decision Structures

- You can create nested decision structures to test more than one condition.
- Nested means "one inside another"
- In C#, a generic format is:

```
if (expression)
{
  if (expression)
  {
    statements;
  }
  else
  {
    statements;
  }
}
else
{
  statements;
}
```

AUBURN
UNIVERSITY

A Sample Nested Decision Structure


Salary >= 400000
Display "Minimum salary requirement not met."
yearsOnJob >= 2
Display "Minimum years at current job not met."
Display "You qualify for the load."
End

Copyright © 2017 Pearson Education, Inc.

A Sample Nested Decision Structure

```
if (salary >= 40000)
{
  if (yearOnJob >= 2)
  {
    decisionLabel.Text = "You qualify for the load."
  }
  else
  {
    decisionLabel.Text = "Minimum years at current " + "job not met."
  }
}
else
{
  decisionLabel.Text = "Minimum salary requirement " + "not met."
}
```

The if-else-if Statement

- You can also create a decision structure that evaluates multiple conditions to make the final decision using the ***if-else-if*** statement
- In C#, the generic format is:

```
if (expression)
{
}
else if (expression)
{
}
else if (expression)
{
}
…
else
{
}
```

```
int grade =
double.Parse(textBox1.Text);
if (grade >=90)
{
   MessageBox.Show("A");
}
else if (grade >=80)
{
   MessageBox.Show("B");
}
else if (grade >=70)
{
   MessageBox.Show("C");
}
else if (grade >=60)
{
   MessageBox.Show("D");
}
else
{
   MessageBox.Show("F");
}
```

4.4 Logical Operators

- The logical **AND** operator (**&&**) and the logical **OR** operator (**||**) allow you to connect multiple Boolean expressions to create a compound expression
- The logical **NOT** operator (**!**) reverses the truth of a Boolean expression

Operator	Meaning	Description
&&	AND	Both subexpression must be true for the compound expression to be true
||	OR	One or both subexpression must be true for the compound expression to be true
!	NOT	It negates (reverses) the value to its opposite one.

Expression	Meaning
x >y && a < b	Is x greater than y AND is a less than b?
x == y || x == z	Is x equal to y OR is x equal to z?
! (x > y)	Is the expression x > y NOT true?

AUBURN

UNIVERSITY

Sample Decision Structures with Logical Operators

- The && operator

```
if (temperature < 20 && minutes > 12)
{
  MessageBox.Show("The temperature is in the danger zone.");
}
```

- The | | operator

```
if (temperature < 20 || temperature > 100)
{
  MessageBox.Show("The temperature is in the danger zone.");
}
```

- The ! Operator

```
if (!(temperature > 100))
{
  MessageBox.Show("The is below the maximum temperature.");
}
```

AUBURN UNIVERSITY

4.5 bool Variables and Flags

- You can store the values true or false in bool variables, which are commonly used as **flags**
- A flag is a variable that signals when some condition exists in the program
  - False – indicates the condition does not exist
  - True – indicates the condition exists

```
if (grandMaster)
{
  powerLevel += 500;
}
If (!grandMaster)
{
  powerLevel = 100;
}
```

AUBURN

UNIVERSITY

4.6 Comparing Strings

- You can use certain relational operators and methods to compare strings. For example,
  - The == operator can compare two strings

```
string name1 = "Mary";
string name2 = "Mark";
if (name1 == name2) { }
```

  - You can compare string variables with string literals, too

```
if (month ! = "October") { }
```

  - The **`String.Compare`** method can compare two strings

```
string name1 = "Mary";
string name2 = "Mark";
if (String.Compare(name1, name2) == 0) { }
```

4.7 Preventing Data Conversion Exception

- Exception should be prevented when possible
- The **`TryParse`** methods can prevent exceptions caused by users entering invalid data
  - `int.TryParse`
  - `doubel.TryParse`
  - `decimal.TryParse`
- The generic syntax is:

```
int.TryParse(string, out targetVariable)
```

- The **`out`** keyword is required; it specifies that the *`targetVariable`* is an output variable

Samples of TryParse Methods

```
// int.TryParse
int number;
if (int.TryParse(inputTextBox.Text, out number))
…
//double.TryParse
double number;
if (double.TryParse(inputTextBox.Text, out number))
…
//decimal.TryParse
decimal number;
if (decimal.TryParse(inputTextBox.Text, out number))
…
```

4.8 Input Validation

- **Input validation** is the process of inspecting data that has been entered into a program to make sure it is valid before it is used
- `TryParse` methods check if the user enters the data, but it does not check the integrity of the data. For example,
  - In a payroll program we might validate the number of hours worked.

```
if (hours > 0 && hours <= 168) { } else { }
```

  - In a program that gets test scores, we can limits its data to an integer range of 0 through 100.

```
if (testScore >= 0 && testScore <=100) { } else { }
```

4.10 The `switch` Statement

- The **`switch`** statement lets the value of a variable or an expression determine which path of execution the program will take
- It is a multiple-alternative decision structure
- It can be used as an alternative to an `if-else-if` statement that tests the same variable or expression for several different values

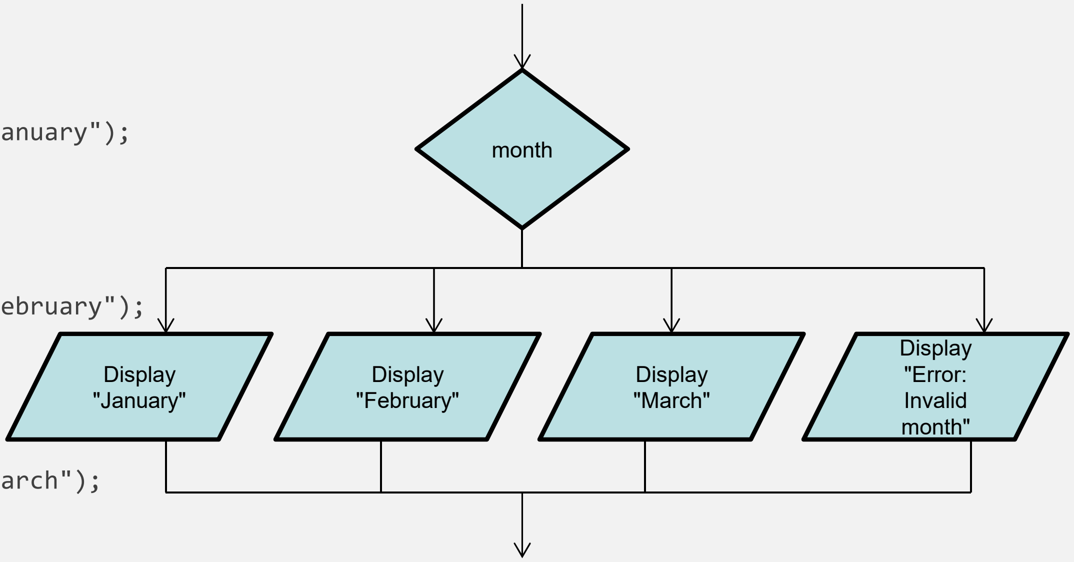
Generic Format of the `switch` Statement

```
swtich (testExpression)
{
  case value_1:
    statements;
    break;
 case value_2:
    statements;
    break;
…
 case value_n:
    statements;
    break;
 default:
    statements;
    break;
}
```

- The *`testExpression`* is a variable or an expression that given an integer, string, or bool value. Yet, it cannot be a floating-point or decimal value.
- Each `case` is an individual subsection containing one or more statements, followed by a `break` statement
- The `default` section is optional and is designed for a situation that the *`testExpression`* will not match with any of the `case`

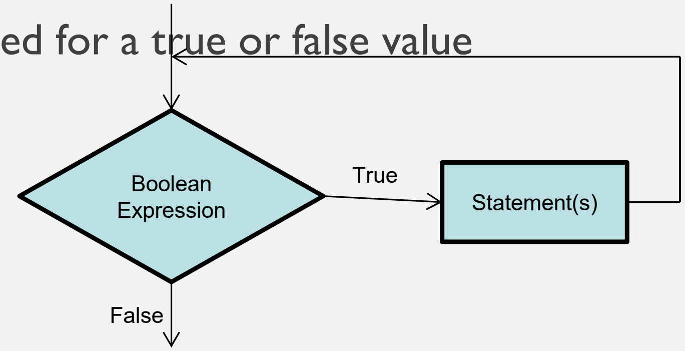
Sample `switch` Statement

```
switch (month)
{
  case 1:
           MessageBox.Show("January");
  break;
  case 2:
           MessageBox.Show("February");
  break;
  case 3:
           MessageBox.Show("March");
  break;
  default:
           MessageBox.Show("Error: Invalid month");
  break;
}
```



5.2 The `while` Loop

- The **`while`** loop causes a statement or set of statements to repeat as long as a Boolean expression is true
- The simple logic is: While a Boolean expression is true, do some task
- A `while` loop has two parts:
  - A Boolean expression that is tested for a true or false value
  - A statement or set of statements that is repeated a long as the Boolean expression is true



Structure of a while Loop

- In C#, the generic format of a while loop is:

```
while (BooleanExpression)
{
   Statements;
}
```

- The first line is called the **while clause**
- Statements inside the curly braces are the **body** of the loop
- When a while loop executes, the Boolean expression is tested. If true, the statements are executed
- Each time the loop executes its statement or statements, we say the loop is iterating, or performing an **iteration**

The while Loop is a Pretest Loop

- A while loop tests its condition before performing an iteration.
- It is necessary to declare a counter variable with initial value

```
int count = 1;
```

- So the while clause can test its Boolean expression

```
while (count < 5) { }]
```

- Inside the curly braces, there must exist a statement that defines increment (or decrement) of the counter

```
while (count < 5)
{
   …..
  counter = count + 1;
}
```

Sample Code

```
private void goButton_Click(object sender, EventArgs e)
{
  int count = 1;
  while (count <= 5)
  {
     MessageBox.Show("Hello!");
     count = count + 1;
  }
}
```

- The counter has an initial value of 1
- Each time the loop executes, 1 is added to counter
- The Boolean expression will test whether counter is less than or equal 5. So the loop will stop when counter equals 5.

Infinite Loops

- An **infinite loop** is a loop that will repeats until the program is interrupted
- There are few conditions that cause a while loop to be an infinite loop. A typical scenario is that the programmer forgets to write code that makes the test condition false
- In the following example, the counter is never increased. So, the Boolean expression is never false.

```
int count = 1;
while (count <= 5)
{
  MessageBox.Show("Hello");
}
```

5.3 The ++ and - - Operators

- To increment a variable means to increase its value, and to decrement a variable means to decrease its value
- C# provides the **++** and **- -** operator to increment and decrement variables
- Adding 1 to a variable can be written as:

```
count = count + 1;
```

or

```
count++;
```

or

```
count += 1;
```

- Subtracting 1 from a variable can be written as:

```
count = count – 1;
```

or

```
count --;
```

or

```
count -= 1;
```

Postfix Mode vs. Prefix Mode

- **Postfix mode** means to place the ++ and -- operators after their operands

```
count++;
```

- **Prefix mode** means to place the ++ and -- operators before their operands

```
--count;
```

AUBURN UNIVERSITY

5.4 The `for` Loop

- The **`for`** loop is specially designed for situations requiring a counter variable to control the number of times that a loop iterates
- You must specify three actions:
  - **Initialization**: a one-time expression that defines the initial value of the counter
  - **Test**: A Boolean expression to be tested. If true, the loop iterates.
  - **Update**: increase or decrease the value of the counter
- A generic form is:

```
for (initializationExpress; testExpression; updateExpression)
{ }
```

- The `for` loop is a pretest loop

Sample Code

```
int count;

for (count = 1; count <= 5; count++)

{

   MessageBox.Show("Hello");

}
```

- The initialization expression assign 1 to the `count` variable
- The expression `count <=5` is tested. If true, continue to display the message.
- The update expression add 1 to the `count` variable
- Start the loop over

```
// declare count variable in the
// initialization expression
for (int count = 1; count <= 5; count++)
{
   MessageBox.Show("Hello");
}
```

Other Forms of Update Expression

- In the update expression, the counter variable is typically incremented by 1. But, this is not a requirement.

```
// increment by 10
for (int count = 0; count <=100; count += 10)
{
   MessageBox.Show(count.ToString());
}
```

- You can decrement the counter variable to make it count backward

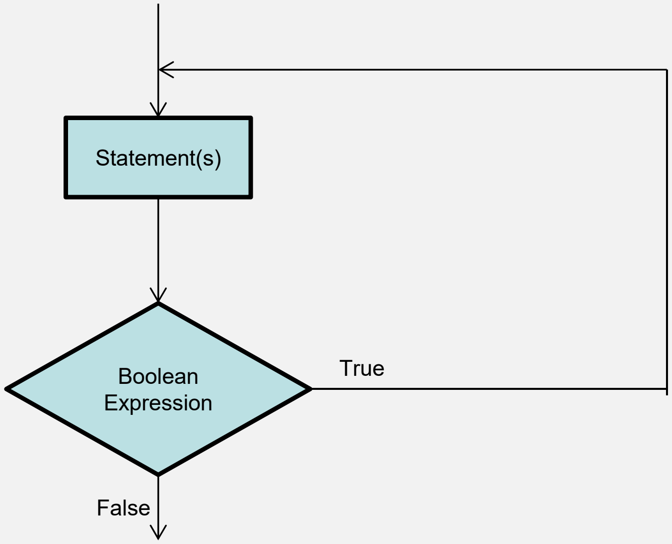
```
// counting backward
for (int count = 10; count >=0; count--)
{
   MessageBox.Show(count.ToString());
}
```

5.5 The do-while Loop

- The do-while loop is a posttest loop, which means it performs an iteration before testing its Boolean expression.
- In the flowchart, one or more statements are executed before a Boolean expression is

  tested
- A generic format is:

```
do
{
  statement(s);
} while (BooleanExpression);
```



Sample Code

- Will you see the message box?

```
int number = 1
do {
    MessageBox.Show(number.ToString());
} while (number < 0);
```

- Will you see the message box?

```
int number = 1
while (number < 0)
{
   MessageBox.Show(number.ToString());
}
```

6.1 Introduction to Methods

- **Methods** can be used to break a complex program into small, manageable pieces
  - This approach is known as **divide and conquer**
  - In general terms, breaking down a program to smaller units of code, such as methods, is known as **modularization**
- Two types of methods are:
  - A **void** method simply executes a group of statements and then terminates
  - A **value-returning** method returns a value to the statement that called it

Example

Using one long sequence of statement to perform a task

```
Namespace Example
{
 public partial class Form1 : Form
 {
  private void myButton_Click(object sender, EventArgs e)
  {
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    …
  }
 }
}
```

Example

Using methods to divide and conquer a problem

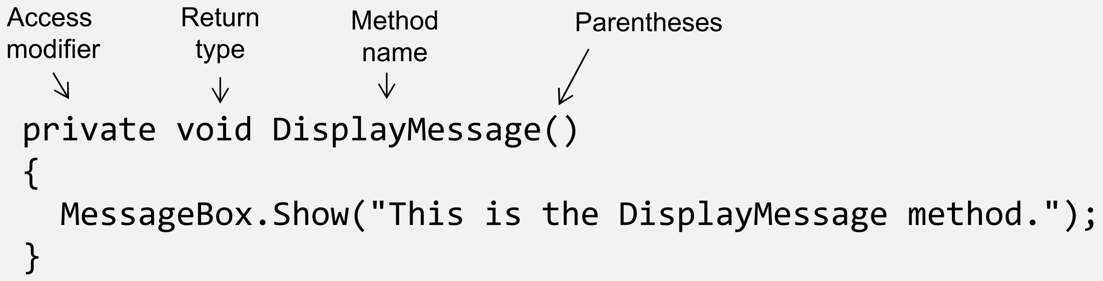
```
Namespace Example
{
  public partial class Form1 : Form
  {
     private void myButton_Click(object sender, EventArgs e)
     {
        Method2();
        Method3();
        …
     }

     private void Method2();
     {
        statements;
     }

     private void Method3();
     {
        statements;
     }
  }
}
```

6.2 void Methods

- A **void** method simply executes the statement it contains and then terminates. It does not return any value to the statement that called it
- To create a method you write its definitions
- A method definition has two parts:
  - **header**: the method header appears at the beginning of a method definition to indicate access mode, return type, and method name
  - **body**: the method body is a collection of statements that are performed when the method is executed

The Method Header

- The book separates a method header into four parts :
  - Access modifier: keywords that defines the access control
    - `private`: a private method can be called only by code inside the same class as the method
    - `public`: a public method can be called by code that is outside the class.
  - Return type: specifies whether or not a method returns a value
  - Method name: the identifier of the method; must be unique in a given program. This book uses Pascal case (aka camelCase)
  - Parentheses: A method's name is always followed by a pair of parentheses



Access modifier

Return type

Method name

Parentheses

```
private void DisplayMessage()
{
  MessageBox.Show("This is the DisplayMessage method.");
}
```

Declaring Method Inside a Class

- Methods usually belong to a class
- All Visual C# methods typically belong to applications' default `Form1` class
- In this book, methods are created inside the `Form1` class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Example
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // your method definition will appear here inside Form1 class
    }
}
```

Calling a Method

- A method executes when it is called
- Event handlers are called when specific events take place. Yet, methods are executed by **method call statements**.
- A method call statement is the name of the method followed by a pair of parentheses. For example:

```
private void goButton_Click(object sender, EventArgs e)
{
  MessageBox.Show("This is the goButton_Click method.");
  DisplayMessage();
}

private DisplayMessage()
{
  MessageBox.Show("This is the DisplayMessage method.");
}
```

Concept of Return Point

- When calling a method the system needs to know where the program should return after the method ends
- The system saves the memory address of the location called **return point** to which it should return
- The system jumps to the method and executes the statements in its body
- When the method ends, the system jumps back to the return point and resumes execution

AUBURN UNIVERSITY

Top-Down Design

- To modularize a program, programmers commonly use a technique known as **top-down design**
- It breaks down an algorithm to methods
- The process is performed in the following manner:
  - The overall task that the program is to perform is broken down into a series of subtasks
  - Each subtask is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified
  - Once all the subtasks have been identified, they are written in code

6.3 Passing Arguments to Methods

- An **argument** is any piece of data that is passed into a method when the method is called
  - In the following, the statement calls the `MessageBox.Show` method and passes the string "Hello" as an argument:

```
MessageBox.Show("Hello");
```

- A **parameter** is a variable that receives an argument that is passed into a method
  - In the following, *`value`* is an `int` parameter:

```
private void DisplayValue(int value)
{
    MessageBox.Show(value.ToString());
}
```

  - An example of a call to `DisplayValue` method with 5 as parameter is:

```
DisplayValue(5);
```

Contents of Variables as Arguments

- You can pass the contents of variables as arguments. For example,

```
int x = 5;
DisplayValue(x);
DisplayValue(x * 4);
```

```
private void DisplayValue(int value)
{
   MessageBox.Show(value.ToString());
}
```

- *value* is an int parameter in the DisplayValue method
- In this example, *x* is an int variable with the value 5. Its contents are passed as argument.
- The expression *x* * 4 also produces an int result, which can be passed as an argument
- Another example is:

```
DisplayValue(int.Parse("700"));
```

Argument and Parameter Data Type Compatibility

- An argument's data type must be assignment compatible with the receiving parameter's data type
- Basically,
  - You can pass only **string** arguments into **string** parameters
  - You can pass **int** arguments into **int** parameters, but you cannot pass **double** or **decimal** arguments into **int** parameters
  - You can pass either **double** or **int** arguments to **double** parameters, but you cannot pass **decimal** values to **double** parameters
  - You can pass either **decimal** or **int** arguments to **decimal** parameters, but you cannot pass **double** arguments into **decimal** parameters

Passing Multiple Arguments

- You can pass more than one argument to a method

```
private void showButton1_Click(object sender, EventArgs e)
{
  ShowMax(5, 10);
}

private void showButton2_Click(object sender, EventArgs e)
{
  int value1 = 2;
  int value2 = 3;
  ShowMax(value1, value2);
}

private void ShowMax(int num1, int num2) { }
```

Named Arguments

- C# allows you to specify which parameter an argument should be passed into. The syntax is:

```
parameterName : value
```

- An argument that is written using this syntax is known as a **named argument**

```
private void showButton_Click(object sender, EventArgs e)
{
  showName(lastName : "Smith", firstName : "Suzanne");
}
private void ShowName(string firstName, string lastName)
{
  MessageBox.Show(firstName + " " + lastNmae);
}
```

- Notice that you get the same result if the call statement is:

```
showName("Suzanne", "Smith");
```

Default Arguments

- C# allows you to provide a **default argument** for a method parameter

```
private void ShowTax(decimal price, decimal taxRate = 0.07m)
{
 decimal tax = price * taxRate;
}
```

- The value of taxRate is defaulted to 0.07m. You can simply call the method by passing only the price

```
showTax(100.0m);
```

- You can also override the default argument

```
showTax(100.0m, 0.08m);
```

AUBURN UNIVERSITY

6.4 Passing Arguments by Reference

- A **reference parameter** is a special type of parameter that does not receive a copy of the argument's value
- It becomes a **reference** to the argument that was passed into it
- When an argument is passed by reference to a method, the method can change the value of the argument in the calling part of the program
- In C#, you declare a reference parameter by writing the **`ref`** keyword before the parameter variable's data type

```
private void SetToZero(ref int number)
{
  number =0;
}
```

- To call a method that has a reference parameter, you also use the keyword **`ref`** before the argument

```
int myVar = 99;
SetToZero(ref myVar);
```

Using Output Parameters

- An **output parameter** works like a reference parameter with the following differences:
    - An argument does not have to be a value before it is passed into an output parameter
    - A method that has an output parameter must set the output parameter to some value before it finishes executing
- In C#, you declare an output parameter by writing the **out** keyword before the parameter variable's data type

```
private void SetToZero(out in number)
{
   number = 0;
}
```

- To call a method that has a output parameter, you also use the keyword **out** before the argument

```
int myVar;

SetToZero(out myVar);
```

6.5 Value-Returning Methods

- A **value-returning** method is a method that returns a value to the part of the program that called it
- A value-returning method is like a void method in the following ways:
  - It contains a group of statements that performs a specific task
  - When you want to execute the method, you call it
- The .NET Framework provide many value-returning methods, for example, the `int.Parse` method that accepts a string and returns an `int` value

argument

```
int number  = int.Parse("100");
```

Method call

Write Your Own Value-Returning Functions

AUBURN
UNIVERSITY

- In C# the generic format is:

```
AccessModifier DataType MethodName(ParameterList)
{
  statement(s);
  return expression;
}
```

- *AccessModifier*: `private` or `public`
- *DataType*: `int`, `double`, `decimal`, `string`, `bool`, etc.
- *MethodName*: the identifier of the method; must be unique in a program
- *ParameterList*: an optional list of parameter
- *Expression*: can be any value, variable, or expression that has a value

The `return` Statement

- There must be a **`return`** statement inside the method which is usually the last statement of the method. This return statement is used to return a value to the statement that called the method. For example:

```
private int sum(int num1, int num2)
{
   return num1 + num2;
}
```

- Notice that the returned value and the method's type must match

- In the above example, the method is an `int` method, so it can only return `int` value

Sample Code

```
// int type
private int Sum(int num1, int num2)
{
   return num1 + num2;
}
// double type
private double Sum(double num1, double num2)
{
   return num1 + num2;
}
// decimal type
private decimal Sum(decimal num1, decimal num2)
{
   return num1 + num2;
}
```

Returning Values to Variables

- A value-returning method returns a value with specific type. However, the method no longer keeps the value once it is returned.
- You can declare a variable to hold the returned value to use the value over and over again

```
int combinedAge = Sum (userAge, friendAge);


private int Sum(int num1, int num2)
{
  return num1 + num2;
}
```

- After execution, the value is kept in `combinedAge` variable

Boolean Methods

- A Boolean method returns either `true` or `false`. You can use a Boolean method to test a condition

```
private bool IsEven(int number)
{
   bool numberIsEven;
   if (number % 2 == 0)
   {
      numberIsEven = true;
   }
   else
   {
      numberIsEven = false;
   }
   return numberIsEven;
}
```

- With this code, an `int` value assigned to the `number` parameter will be evaluated by the *if* statement
- The `return` statement will return either `true` or `false`

Using the Modulus Operator in Boolean Expressions

- The book discusses the use of modulus operator to determine whether a whole number is odd or even

```
number % 2
```

- The modulus operator is a useful tool to write Boolean expression

•The expression `number % 2` has only two possible values: 0 or 1

```
if (number % 2 == 0)
{
   numberIsEven = true;
}
else
{
   numberIsEven = false;
}
```

```
switch (number % 2)
{
  case 0: numberIsEven = true;
          break;
  case 1: numberIsEven = false;
          break;
  // default is not needed in this case
}
```

Returning a String from a Method

- `string` is a primitive data type. A C# value-returning method can return a `string` to the statement that called it. For example,

```
private string FullName(string first, string middle, string last)
{
   return first + " " + middle + " " + last;
}
```

- A sample statement to call it is:

```
FullName("Lynn","Alisha","McCormick");
```

6.6 Debugging Methods

- The *Step Into* command allows you to single-step through a called method.
- Execute the *Step Into* command in any of the following ways:
  - Press the *F11* key
  - Select *Debug* from the menu bar, and then select *Step Into* from the *Debug* menu
  - Click the *Step Into* button on the *Debug* Toolbar, if the toolbar is visible
- Tutorial 6-6 demonstrates the *Step Into* command.

Debugging Methods

- The *Step Over* command allows you to call a method without single-stepping through its statements.
- Execute the *Step Over* command in any of the following ways:
  - Press the *F10* key
  - Select *Debug* from the menu bar, and then select *Step Over* from the *Debug* menu
  - Click the *Step Over* button on the *Debug* Toolbar, if the toolbar is visible
- Tutorial 6-7 demonstrates the *Step Over* command.

Debugging Methods

- When single-stepping through a method, the *Step Out* command causes the rest of the method's statements to execute without single-stepping.
- Execute the *Step Out* command in any of the following ways:
  - Press the *Shift+F11* keys
  - Select *Debug* from the menu bar, and then select *Step Out* from the *Debug* menu
  - Click the *Step Out* button ↑ n the *Debug* Toolbar, if the toolbar is visible
- Tutorial 6-8 demonstrates the *Step Out* command.

Debugging Methods

- Visual Studio can be configured in different ways.
  - Under some configurations, the *Step Into* command from the *Debug* menu might be activated by the F8 function key.
  - Under some configurations, the *Step Over* command may be activated by the Shift + F8 keys.
  - Under some configurations, the Step Out command might be activated by the Ctrl + Shift + F8 keys.
- To find out which keys are used, look carefully at these commands when you click on the *Debug* menu.

AUBURN UNIVERSITY

9.1 Introduction to Classes

- A **class** is the blueprint for an object.
  - It describes a particular type of object, yet it is not an object.
  - It specifies the fields and methods a particular type of object can have.
  - One or more objects can be created from the class.
  - Each object created from a class is called an **instance** of the class.

AUBURN UNIVERSITY

Creating a Class

- You can create a class by writing a **class declaration**. A generic form is:

```
class className
{
   Member declaration(s)…
}
```

- The **class header** is the first line. It starts with the keyword `class`, followed by the name of the class.
- **Member declarations** are statements that define the class's fields, properties, and/or methods.
- A class may contains a **constructor**, which is special method automatically executed when an object is created.

AUBURN
UNIVERSITY

Sample Code

```
class Coin
{
   private string sideUp; // field

   public Coin() // constructor
   {
      sideUp = "Heads";
   }

   public void Toss() // a void method
   {
      MessageBox.Show(sideUp);
   }

   public string GetSideUp() // a value-returning method
   {
       return sideUp;
   }
}
```

AUBURN
UNIVERSITY

Creating an Object

- Given a class named Coin, you can create a Coin object, use:

```
Coin myCoin = new Coin();
```

- where,

•myCoin is a variable that references an object of the Coin class;

•the **new** keyword creates an instance of the Coin class; and

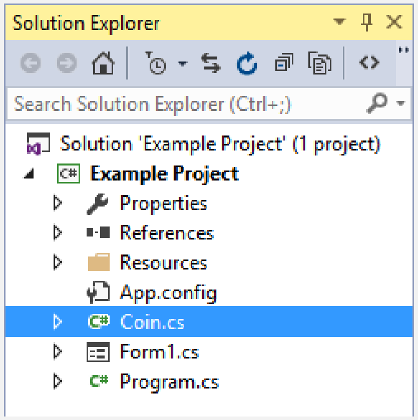
•the = operator assigns the reference that was returned from the new operator to the myCoin variable.

- Once a Coin object is created, you can access members of the class with it. E.g.

```
myCoin.Toss();
```

Where to Write Class Declarations

- In C# you have flexibility in choosing where to write class declarations. E.g.
- To create the Coin class, you can:
  - Save the class declaration is a separated .cs file; or
  - Add the Coin class next to the Form1 class inside the Form1.cs file.



```
Namespace Example
{
   public partial class Form1 : Form
   {
      public Form1()
      {
        InitializeComponent();
       }
       ….
   }
   class Coin
   {
      ….
   }
}
```

Passing an Object to a Method

- Objects of a class can be used as parameters of a method. E.g.

```
private void ShowCoinStatus(Coin coin)
{
   MessageBox.Show("Side is " + coin.GetSideUp());
}
```

- In this example, a method named `ShowCoinStatus` accepts a `Coin` object as an argument.
- To create a `Coin` object and pass it as an argument to the `ShowCoinStatus` method, use:

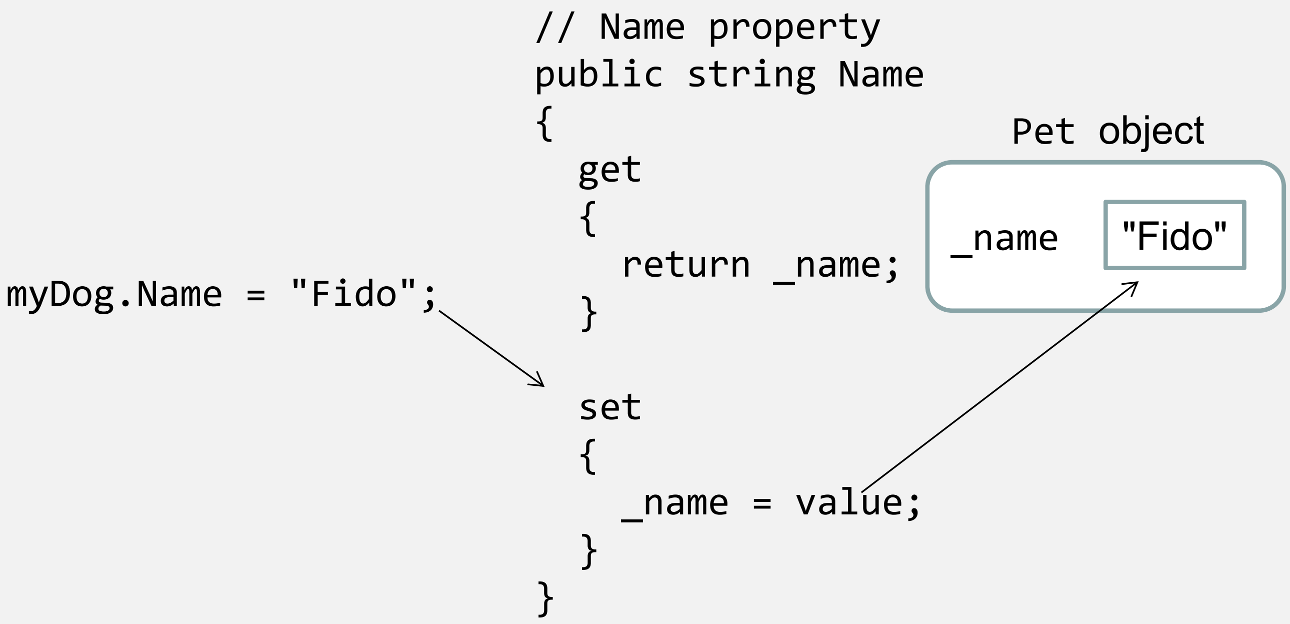
```
Coin myCoin = new Coin();
ShowCoinStatus(myCoin);
```

9.2 Properties

- A **property** is a class member that holds a piece of data about an object.
  - Properties can be implemented as special methods that set and get the value of corresponding fields.
  - In the code, there is a private field which is a known as the **backing field** and is used to hold any data assigned to the property.
  - The **value** parameter of set accessor is automatically created by the compiler.

```
class Pet
{
 private string _name; // backing field
 public Pet()
 {
  _name = "";
 }

 public string Name
 {
  get
  {
   return _name;
  }
  set
  {
   _name = value;
  }
 }
}
```

Setting the myDog object's Name Property to "Fido"



```
myDog.Name = "Fido";
```

```
// Name property
public string Name
{
  get
  {
    return _name;
  }

  set
  {
    _name = value;
  }
}
```

The Backing Field

- The **private backing field** is a variable that stores a value assigned to the property which the backing fields is associated with.
- It is declared to be private to protect it from accidental corruption.
- If a backing field is public, it can then be accessible directly by code outside the class without the need for accessors.

`get` vs `set` Methods

- The **`get`** method, if not empty, is a method that returns the property's value because it has a **`return`** statement.
  - It is executed whenever the property is read.
- The **`set`** method, if not empty, gets the value stored in the backing field and assigns the value to the property
  - It has an implicit parameter named **`value`**.
  - It is executed whenever a value is assigned to the property.

AUBURN
UNIVERSITY

Read-Only Properties

- A read-only property can be read, but it cannot be modified.
    - To set a read-only property, simply do no write a set method for the property. E.g.

```
// read and write
public double Diameter
{
   get { return _diameter; }
   set { _diameter = value; }
}
```

```
// read-only
public double Diameter
{
   get { return _diameter; }
}
```

AUBURN UNIVERSITY

Auto-Properties

- Sometimes a property simply gets and sets the value of a backing field, without performing any other operation.
- Auto-properties simplify the code for such a property. Example:

```
public string Name
{
    get;
    set;
}
```

is equivalent to

```
private string _name;

public string Name
{
    get
    {
        return _name;
    }

    set
    {
        _name = value;
    }
}
```

Auto-Properties

- With auto-properties, a hidden backing field is automatically created, as well as the code for the get and set methods.
- In fact, most programmers prefer to write an even shorter version of the property, like this:

```
public string Name { get;  set; }
```

- You can initialize an auto-property, like this:

```
public string Name { get;  set; } = "Fido";
```

Read-Only Auto-Properties

- If you leave out the set keyword in an auto-property, the property becomes read-only. Example:

```
class Pet
{
   // Name property
   public string Name { get; } = "Fido";
}
```

9.3 Parameterized Constructor & Overloading

- A constructor that accepts arguments is known as a **parameterized constructor**. E.g.

```
public BankAccount(decimal startingBalance) { }
```

- A class can have multiple versions of the same method known as **overloaded methods**.
- How does the compiler know which method to call?
  - Binding relies on the **signature** of a method which consists of the method's name, the data type, and argument kind of the method's parameter. E.g.

```
public BankAccount(decimal startingBalance) { }

public BankAccount(double startingBalance) { }
```

- The process of matching a method call with the correct method is known as **binding**.

Overloading Methods

- When a method is overloaded, it means that multiple methods in the same class have the same name but use different types of parameters.

```
public void Deposit(decimal amount) { }
public void Deposit(double amount) { } // overloaded
public void Deposit(int numbers) { } // overloaded
public void Deposit(string names) { } // overloaded
```

AUBURN UNIVERSITY

Overloading Constructors

- Constructors are a special type of methods. They can also be overloaded.

```
public BankAccount() { } // parameterless constructor
public BankAccount(decimal startingBalance) { } // overloaded
public BankAccount(double startingBalance) { } // overloaded
```

  - The parameterless constructor is the default constructor

- Compiler will find the matching constructors automatically. E.g.

```
BankAccount account = new BankAccount();
BankAccount account = new BankAccount(500m);
```

Lists of Class Type Objects

- You can create a List to hold a class object. E.g.

```
List<CellPhone> phoneList = new List<CellPhone>();
```

  - This statement creates a List object, referenced by the `phoneList` variable.

- Each object of the `CellPhone` class needs an instance of `CellPhone` class to hold data. E.g.

```
CellPhone myPhone = new CellPhone();
myPhone.Brand = "Acme Electronics";
myPhone.Model = "M1000";
myPhone.Price = 199;
```

- To add the `Cellphone` object to the List, use:

```
phoneList.Add(myPhone);
```

9.5 Finding the Classes & their Responsibilities in a Problem

AUBURN UNIVERSITY

- When developing an object-oriented program, you need to identify the classes that you will need to create.
- One simple and popular techniques involves the following steps:
  - Get a written description of the problem domain.
  - Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
  - Refine the list to include only the classes that are relevant to the problem.
- Once the classes have been identified, you need to identify each class's responsibilities. The responsibilities are:
  - The things that the class is responsible for knowing
  - The actions that the class is responsible for doing

Example

- In the textbook, there are three classes: `Customer`, `Car`, and `ServiceQuote`.
    - The `Customer` class has the following actions:
        - Create and initialize an object of the Customer class.
        - Get and set the customer's name.
        - Get and set the customer's address.
        - Get and set the customer's telephone number.
    - The `Car` class has the following actions:
        - Create and initialize an object of the Car class.
        - Get and set the car's make.
        - Get and set the car's model.
        - Get and set the car's year.
    - The `ServiceQuote` class has the following actions:
        - Create and initialize an object of the ServiceQuote class.
        - Get and set the estimated parts charges.
        - Get and set the estimated labor charges.
        - Get and set the sales tax rate.
        - Get the sales tax.
        - Get the total estimated charges.

AUBURN UNIVERSITY