C#

UNIT 2 : Object Oriented Programming in C#

SOFTWARE DEVELOPMENT TOOLS(C# and ASP.NET)

II MSc CS

V.B.Buvaneswari

CONTENTS OF THIS UNIT

Object Oriented Programming in C#:

- Methods
- Classes and objects
- Access specifier
- Inheritance
- Abstract class
- Sealed classes
- Interfaces
- Delegates
- Namespaces
- Exceptions

METHODS

METHOD DECLARATION

The general form of a method declaration is

```
modifiers type methodname (formal-parameter-list) {
        method _ body
}
```

METHOD DECLARATION

Part 01

Name of the Method

Part 02

Type of the value the method returns

Part 03

List of Parameters

Part 04

Body of the Method

Part 05

Method Modifiers

METHOD DECLARATION(cont.)

- The method name is a valid C# identifier.
- The type specifies the type of value the method will return.
- This can be a simple data type such as int as well as any class type. If the method does not return anything, a return type of void is specified.
- Note that we cannot omit the return type altogether.

METHOD DECLARATION(cont.)

- The formal-parameter-list is always enclosed in parentheses.
- This list contains variable names and types of all the values we want to give to the method as input.
- The parameters are separated by commas.
- In the case where no input data are required, the declaration must still include an empty set of parantheses() after the method name.

METHOD DECLARATION(cont.)

- int Fun1 (int m, float x, float y) / /three parameters
- void Display ( ) //no parameters
- The body enclosed in curly braces actually describes the operations to be performed on the data.
- For example, the code segment below computes the product of two integer values and returns the result.

```
int Product ( int x, int y)
{ int m = x * y; //operation, m is a local variable
return(m); // returns the result (int type)
}
```

METHOD DECLARATION(cont.)

- The return statement in the body may be omitted if the method does not return any value.
- Example:

```
void Display ( int x )
{
    Console.Writeline(x);
}
```

- The formal parameters should be declared for their types individually.
- For example, the method header int Product (int m, float x, y) is invalid.
- The modifiers specify keywords that decide the nature of accessibility and the mode of application of the method.

THE MAIN METHOD

- C# programs start execution at a method named Main().
- This method must be the static method of a class and must have either int or void as return type
- public static int Main( ) Or
- public static void Main( )
- The modifier public is used as this method must be called from outside the program.
- The Main can also have parameters which may receive values from the command line at the time of execution.
- public static int Main (string [ ] args)
- public static void Main (string [ ] args)

INVOKING METHODS

- The actual-parameter-list is a comma separated list of 'actual values' (or expressions) that must match in type, order and number with the formal parameter list of the method name declared in the class.
- The invoking statement is an executable one and therefore must end with a semicolon.
- The values of the actual parameters are assigned to the formal parameters at the time of invocation.

INVOKING METHODS

- Once methods have been defined, they must be activated for operations.
- The process of activating a method is known as invoking or calling.
- The invoking is done using the dot operator

  objectname.methodname(actual-parameter-list);
- Here, objectname is the name of the object on which the method methodname is called.

DEFINING AND INVOKING A METHOD

```
// Client class to invoke the cube method
class MethodTest
{
      public static void Main( )
     {
           //Create object for invoking cube
          Method M = new Method ( );
          //Invoke the cube method
          int y= M.Cube (5);
          //Method call
          // Write the result
          Console.Writeline( y)
     }
}
```

C#

C#

C#

C#

DEFINING AND INVOKING A METHOD

```
using System;
class Method // class containing the method
 {
    //Define the Cube method
     int Cube ( int x)
    {
        return ( x * x * x );
    }
}
```

C#

C#

C#

NESTING OF METHODS

```
class NestTesting
{
    public static void Main( )
    {
        Nesting next = new Nesting ( );
        next.Largest ( 100, 200) ; //Method
    }
}
```

C#

C#

NESTING OF METHODS

#

C#

C#

```
using System;
class Nesting

{


      void Largest ( int m, int n )
      {
            int large= Max ( m , n ); //Nesting Console.Writeline( large);
       }
       int Max (int a, int b)
      {
            int x = ( a > b ) ? a : b ;
            return ( x );
      }
```

C#

METHOD PARAMETERS

PARAMETERS

Manages the process of passing values and getting back the results

Value Parameters

Pass parameters into methods by value

Reference Parameters

Pass parameters into methods by reference

Output Parameters

Used to pass results back from a method

Parameter Arrays

Used in a method definition to enable it to receive variable number of arguments when called

METHODS OVERLOADING

- Method overloading is used when methods are required to perform similar tasks but using different input parameters.
- Overloaded methods must differ in number and/or type of parameters they take.
- This enables the compiler to decide which one of the definitions to execute depending on the type and number of arguments in the method call.
- The method's return type does not play any role in the overload resolution.
- Using the concept of method overloading, a family of methods can be designed with one name but different argument lists.

METHODS OVERLOADING

An an overloaded add() method handles different types of data.

```
//Method definitions
int add ( int a, int b ) { ... }          //Method1
int add ( int a, int b, int c ) { ... }   //Method2
double add ( float x, float y ) { ... }   //Method3
double add ( int p, float q ) { ... }     //Method4
double add (float p, int q ) { ... }      //Method5
```

METHODS OVERLOADING

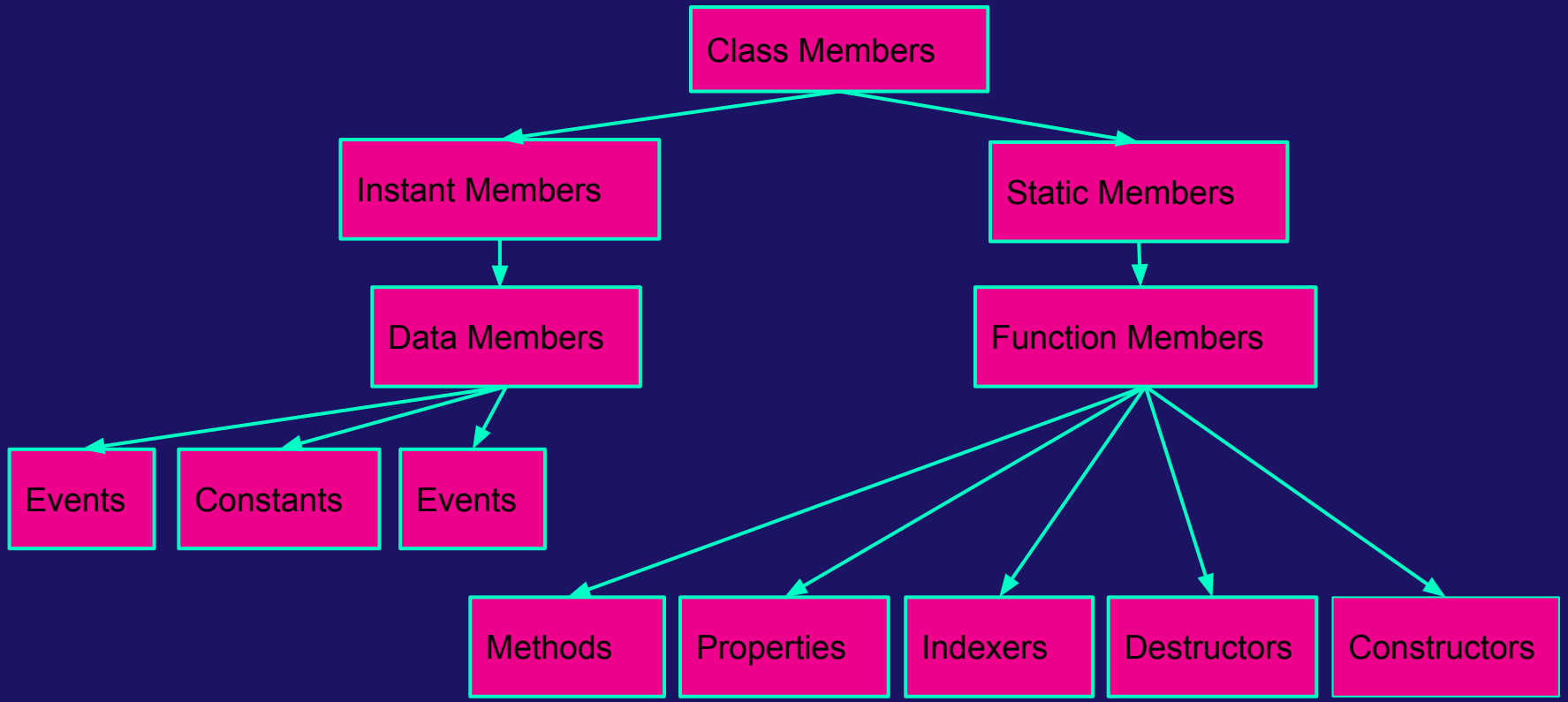
```
//Method calls
int m = add ( 5, 10) ;              //calls method1
double x = add ( 15, 5.0F );        //calls method4
double x = add ( 1.0F, 2.0 F );     //calls method3
int m = add ( 5, 10., 15);          //calls method2
double x = add ( 2.0F, 10 );        //calls method5
```

The method selection involves the following steps:

- The compiler tries to find an exact match in which the types of actual parameters arc the same and uses that method.
- If the exact match is not found, then the compiler tries to use the implicit conversions to the actual arguments and then uses the method whose match is unique. If the conversion creates multiple matches, then the compiler will generate an error message.

CLASSES AND OBJECTS

CATEGORIES OF CLASS MEMBERS



DEFINING A CLASS

```
Class classname
{
    [ variables declaration;]
    [ methods declaration;]
}
```

ADDING VARIABLES

- Data is encapsulated in a class by placing data fields inside the body of the class definition and these variables are called instance variables because they are created whenever an object of the class is instantiated.
- The instance variables are declared exactly the same way as local variables are declared.
- Example

```
Class Rectangle
{
int length , width; // instance variables or member variables
}
```

ADDING METHODS

```
type methodname (parameter-list)
{
method-body;
}
```

CREATING OBJECTS

Action	Statement	Result
Instantiate	Rectangle rect1;	null rect1
Declare	rect1 = new Rectangle;	rect1 → Rectangle object

- rect1 is a reference to Rectangular object

ACCESSING CLASS MEMBERS

```
Objectname.variable name
Objectname.methodname
(parameter-list);
```

CONSTRUCTORS

- Constructors have the same name as the class itself.
- They do not specify a return type, not even void because they do not return any value.

TYPES OF CONSTRUCTORS

CONSTRUCTORS

Enables an object to initialize itself when created

Overloaded Constructors

More than one method having same name but different parameter lists and definitions.

Static Constructors

Assign initial values to static data members.

Private Constructors

Preventing objects to be created for classes containing only static members

Copy Constructors

Creates an object by copying variables from another object

OVERLOADED CONSTRUCTORS

- Used when objects are required to perform similar tasks but using different input parameters.
- When a method in an object is called, C# matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute and this process is known as polymorphism.
- The concept of method overloading can be extended to provide more than one constructor to a class.

OVERLOADED CONSTRUCTORS

```
class Room
{
        public double length;
        public double breadth;
        public Room(double x, double y) //constructor 1
        {
                length=x;
                breadth=y;
        }
        public Room(double x) // constructor 2
        {
                length = breadth=x;
        }
        public int Area()
        {
                return(length * breadth);
        }
}
```

OVERLOADED CONSTRUCTOR

```
Room room1 = new Room(25.0, 15.0); //
                                using constructor 1
Room room2 = new Room(20.0); // using
                                constructor 2
```

STATIC CONSTRUCTORS

```
class Abc
{
     static Abc()  // No parameters
     {
          .....      // set values for static
                        members here
     }
     .....
}
```

COPY CONSTRUCTORS

```
public Item(Item item)
{
    code=item.code;
    price=item.price;
}
```

Eg: Item item2= new item (item1);
Item2 is a copy of item1

DESTRUCTORS

- A destructor is opposite to a constructor, It is a method called when an object is no more required.
- The name of the destructor is the same as the class name and is preceded by a tilde (~).
- A destructor has no return type.
- C# manages the memory dynamically and uses a garbage collector running on a separate thread, to execute all destructors on exit.
- The process of calling a destructor when an object is reclaimed by the garbage collector is called finalization.

DESTRUCTORS

```
Class Fun
{
      ....
      ....
      ~Fun() // No arguments
      {
      ...
      }
}
```

PROPERTIES

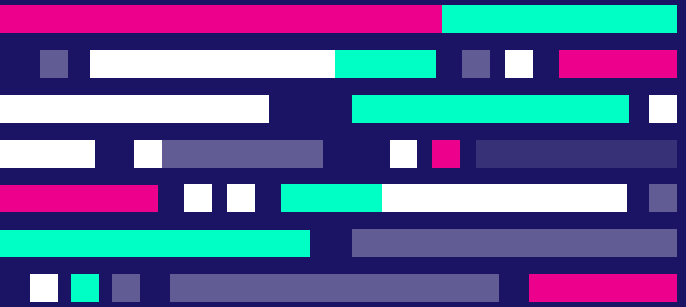
- C# provides a mechanism known as properties that has the same capabilities as accessor methods, but is much more elegant and simple to use.
- Using a property, a programmer can get access to data members as though they are public fields.
- Properties are referred to as 'smart fields'

INDEXERS

- Indexers are location indicators and arc used to access class objects,just like accessing elements in an array.
- They are useful in cases where a class is a container for other objects.
- An indexer looks like a property and is written the same way a property is written, but with two differences:
  - The indexer takes an index argument and looks like an array.
  - The indexer is declared using the name this.
- The indexer is implemented through get and set accessors for the [ ] operator.

INDEXER

```
public double this [int  idx]
{
      get
      {
            //Return desired data
      }
      set
      {
            //Set desired data
      }
}
```



PROGRAM LINKS

Click Execute to run programs

1. Classes and Objects
   http://tpcg.io/vi8OkT9d
2. Properties
   http://tpcg.io/V9wXs9zr
3. Indexer
   http://tpcg.io/a7vJ0oa4

ACCESS SPECIFIERS

C# ACCESS SPECIFIERS

MODIFIER	ACCESSIBILITY CONTROL
private	Member is accessible only within the class containing the member
public	Member is accessible from anywhere outside the class as well. It is also accessible io derived classes
protected	Member is visible only to its ownclassand its derived classes
internal	Member is available within the assembly or component that is being created but not to the clients of that component
protected internal	Available in the containing program or assembly and in the derived classes.

INHERITANCE

CLASSICAL INHERITANCE IMPLEMENTATION

CLASSICAL INHERITANCE

Represents a kind of relationship between two classes

Single Inheritance

Only one base class

Multiple Inheritance

Several base classes

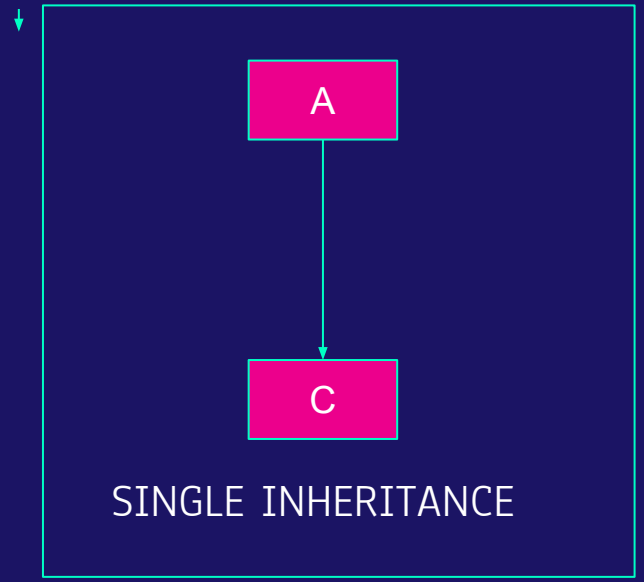
Hierarchical Inheritance

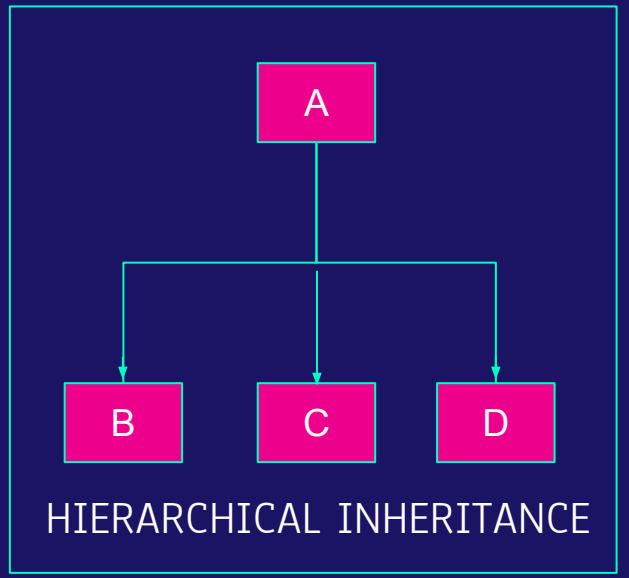
One base class, many sub classes

Multilevel Inheritance

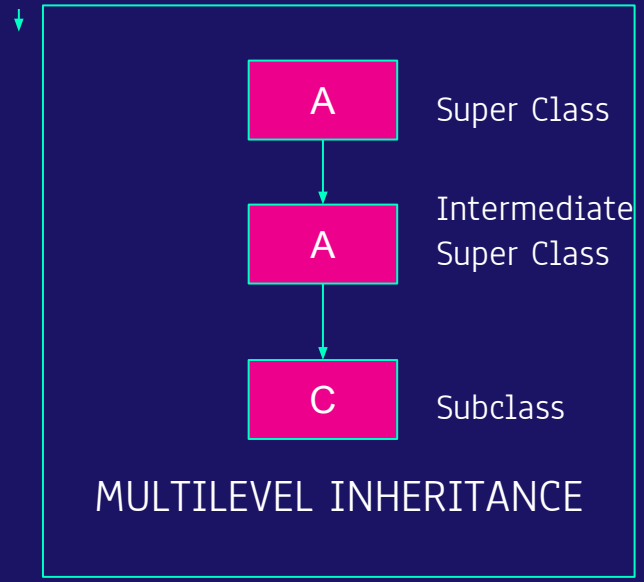
Derived from a derived class

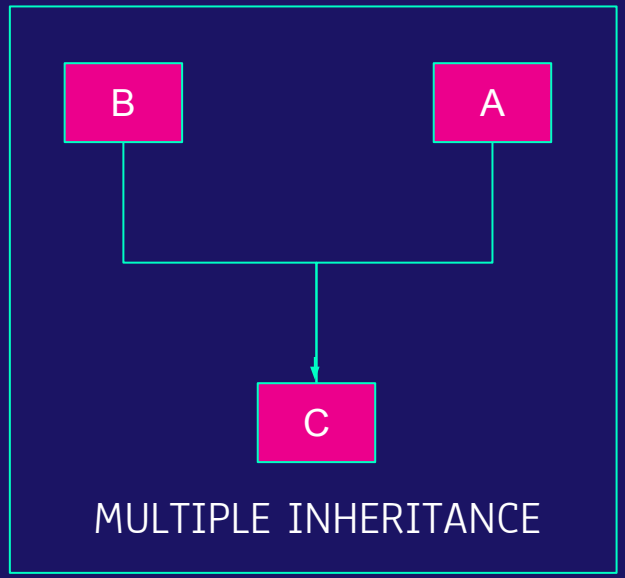
IMPLEMENTATION OF INHERITANCE





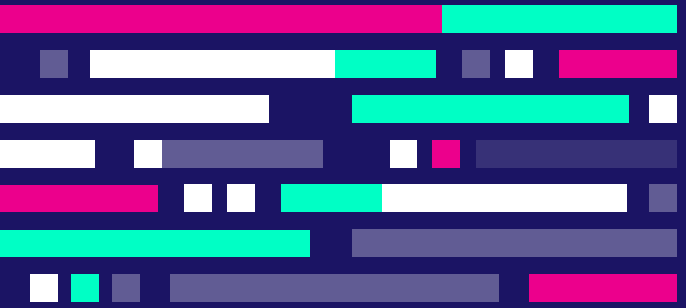
IMPLEMENTATION OF INHERITANCE





CONTAINMENT INHERITANCE

```
Class A
{
        …
}
Class B
{
        …
        A a; // a is contained in b
}
B b;
….
```



PROGRAM LINKS

Click Execute to run programs

1. Simple Inheritance
   http://tpcg.io/rfhNWQfF
2. Multilevel Inheritance
   http://tpcg.io/gpfNLIQu
3. Multiple Inheritance
   http://tpcg.io/2m6UjDz4

ABSTRACT CLASS

ABSTRACT CLASS

- In a number of hierarchical applications, there would be one base class and a number of different derived classes.
- The top-most base class simply acts as a base for others and is not useful on its own.
- ln such situations, creation of objects can be avoided by making base class abstract.
- The abstract is a modifier and when used to declare a class indicates that the class cannot be instantiated.
- Only its derived classes (that are not marked abstract) can be instantiated.

ABSTRACT CLASS

Example:

```
abstract class Base
 {
        ….
}
 class Derived : Base
{
        ….
}
….
Base b1;       //Error
Derived d1; //OK
```

ABSTRACT CLASS

- Objects of Base type cannot be created but subclasses can be derived which can be instantiated.
- Some characteristics of an abstract class are:
  - It cannot be instantiated directly
  - lt can have abstract members
  - A sealed modifier cannot be applied to it

SEALED CLASSES

SEALED CLASSES

- A class may be prevented being further subclassed for security reasons.
- A class that cannot be subclassed is called a sealed class.
- Declaring a class sealed prevents any unwanted extensions to the class.
- It also allows the compiler to perform sorne optimizations when a method of a sealed class is invoked.
- Usually standalone utility classes are created as sealed classes.
- A sealed class cannot also be an abstract class

SEALED CLASSES

```
Sealed class Aclass
 {
   ...
 }
 Sealed class Bclass : Someclass
 {
   ...
 }
```

SEALED METHODS

```
class A
{
        public virtual void Fun()
        {
        ….
        }
}
class B : A
{
        public sealed override void Fun()
        {
        ….
        }
}
```

INTERFACES

INTERFACE

- An interface supports the concept of multiple inheritance
- An interface in C# is a reference type.
- It is basically a kind of class with some differences that include :
  - All the members of an interface are implicitly public and abstract.
  - An interface cannot contain constant fields, constructors and destructors.
  - Its members cannot be declared static.
  - Since the methods in an interface are abstract, they do not include implementation code.
  - An interface can inherit multiple interfaces.

DEFINING AN INTERFACE

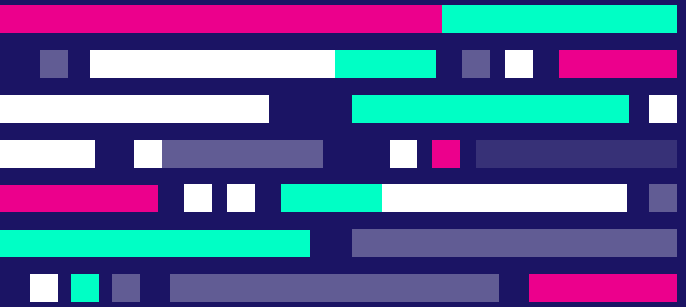
```
interface InterfaceName
{
    Member of declarations;
}
```

EXTENDING AN INTERFACE

```
Interface name2 : name1
{
    Members of name2;
}
```

IMPLEMENTING INTERFACES

```
Class classname : interfacename
{
    Body of classname
}
```



PROGRAM LINKS

Click Execute to run programs

1. Multiple Inheritance
   http://tpcg.io/2m6UjDz4

DELEGATES

DELEGATE

DELEGATES

A method acting for another method

Delegate Declaration

A type declaration

Delegate Methods Definition

References are encapsulated into a delegate instance

Delegate Instantiation

Create new instance of a delegate

Delegate Invocation

Invokes the method whose reference has been encapsulated into the delegate

DELEGATE DECLARATION

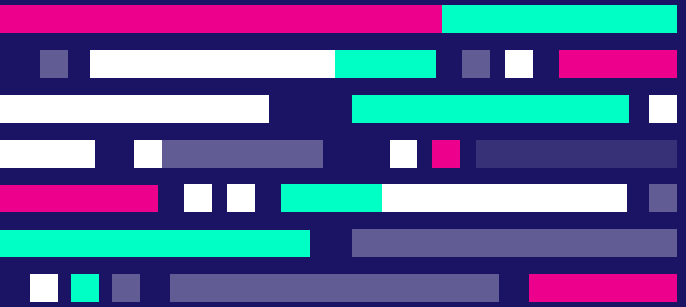
Modifier delegate return-type delegate name (parameters);

DELEGATE INSTANTIATION

new delegate-type (expression)

DELEGATE INVOCATION

```
delegate_object (parameters list)
```



PROGRAM LINKS

Click Execute to run programs

1 Creating and Implementing Delegate
http://tpcg.io/xoinHJsQ

NAMESPACES

NAMESPACES

- Namespaces are the way C# segregates the .NET library classes into reasonable groupings.
- C# supports feature known as using directive that can be used to import the namespace **System** into the program.
- Once the namespaces is imported, the elements of the namespaces can be used without using the namespace as prefix
- Example
  - System.Console.WriteLine();
  - System is the namespace (scope) in which the Console class is located.

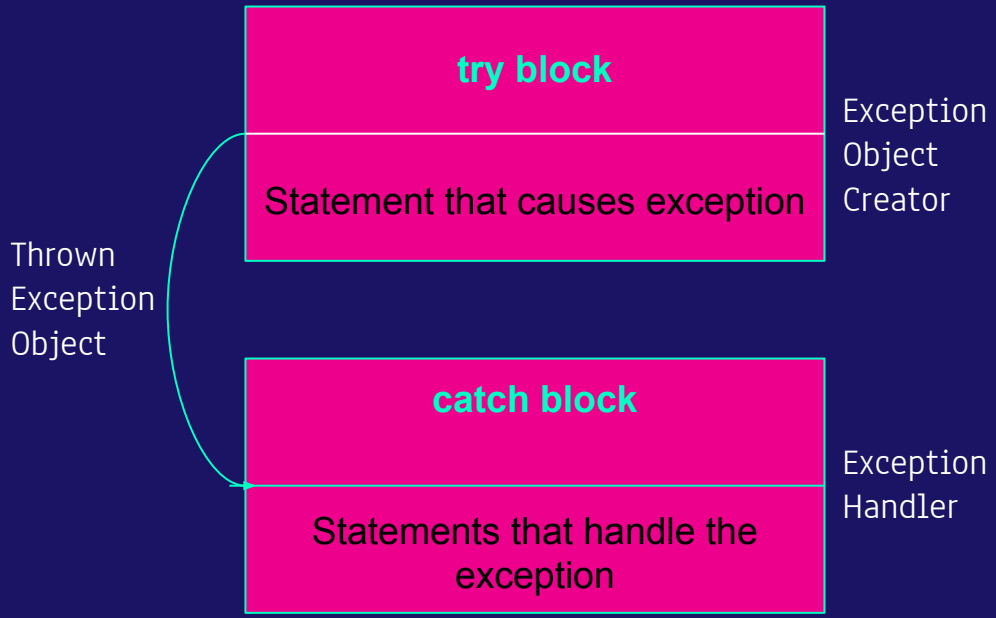
EXCEPTIONS

EXCEPTIONS

- An exception is a condition that is caused by a run-time error in the program.
- The purpose of the exception handling mechanism is to provide a means to detect and report an exceptional circumstance so that appropriate action can be taken.
- The error handling code performs the following tasks.
  - Find the problem (Hit the exception)
  - Inform that an error has occurred (Throw the exception)
  - Receive the error information (Catch the exception)
  - Take corrective actions. (Handle the exception)

SYNTAX OF EXCEPTION HANDLING CODE

Basic concepts of exception handling are throwing an exception and catching it.

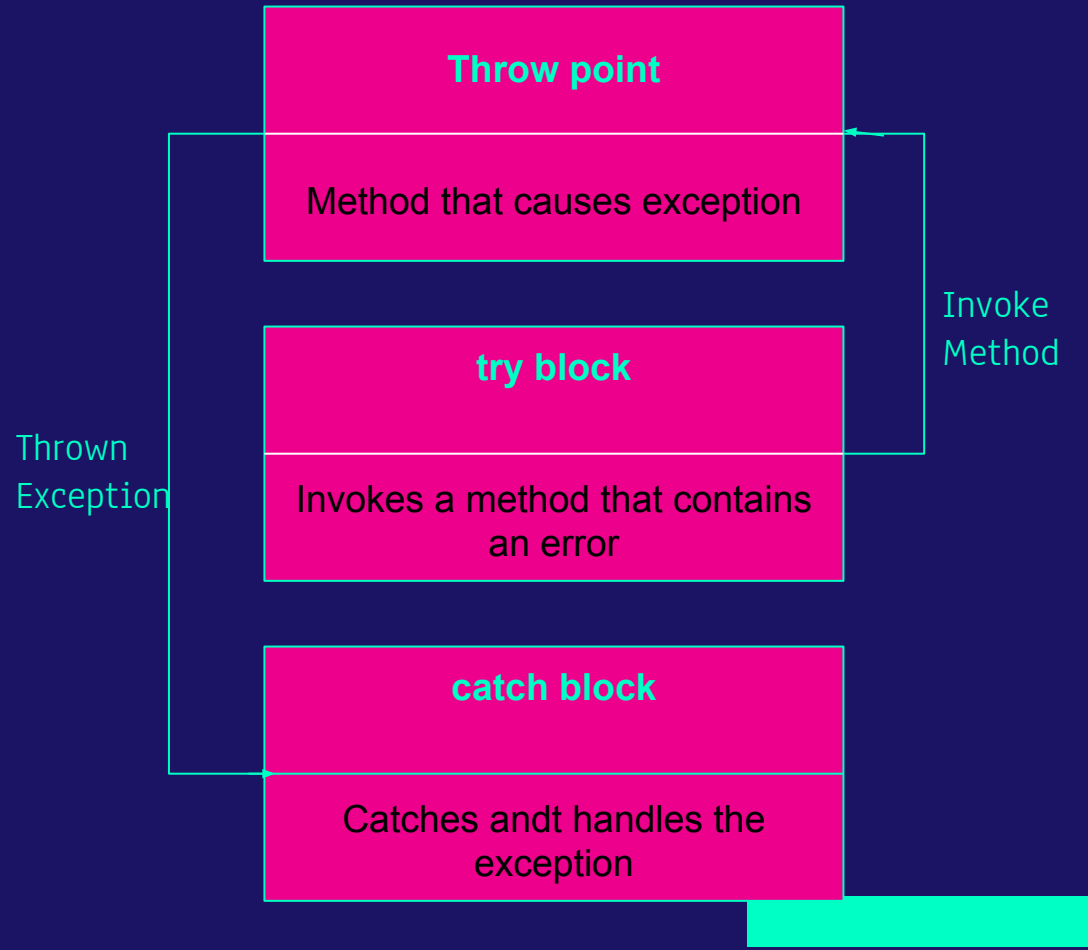


SIMPLE TRY AND CATCH STATEMENTS

```
try
{
        Statement;      // generates an exception
}
catch (Exception e)
{
        Statement;      // processes the exception
}
```

INVOKING METHOD THAT CONTAINS EXCEPTION

Exceptions are thrown by methods that are invoked within the try blocks. The point at which the exception is thrown is called the throw point. Once an exception is thrown at the catch block, control cannot return to the throw point.



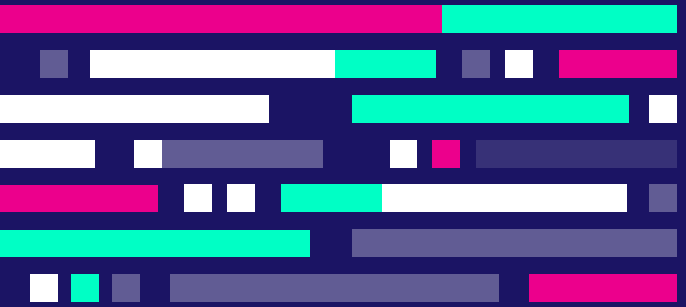
FINALLY STATEMENT

```
try
{
        ...
}
finally
{
        ...
}
```

```
        try
        {
                ...
        }
        catch (...)
        {
                ...
        }
        catch (...)
        {
                ...
        }
        ..
        finally
        {
                ..
        }
```

THROWING OUR OWN EXCEPTIONS

```
Throw new Throwable_subclass;
```



PROGRAM LINKS

Click Execute to run programs

1. Using try and catch for Exception
   http://tpcg.io/mq9vpaSt
2. Throwing our own Exception
   http://tpcg.io/w81wvlEq

THANKS !

Do you have any questions?
buvann@gmail.com

Credits

- E.Balagurusamy , "Programming in C#" , Third Edition, Tata McGraw Hill Education Private Limited, New Delhi.
- www.slidesgo.com