C# - Ultimate Guide - Beginner to Advanced | Master class

Section 20 – Delegates & Events

Delegates

"Delegate type" is a "type" that represents methods that have specific parameters and return type.

The "delegate" (a.k.a. delegate object), is an object that stores reference (address) of a specific method of a specific class, with compatible parameters and return type, which is already defined in the delegate type.

1. Creating Delegate Type

public delegate ReturnType DelegateTypeName(param1, param2);

2. Creating Delegate Object

DelegateTypeName ReferenceVariable = new DelegateTypeName(MethodName);

3. Invoke Method using Delegate Object

ReferenceVariable.Invoke(arg1, arg2, ...);

Rules of Delegates

- You can invoke the methods using 'delegate objects' (or) 'delegates'.
- Delegates are used to pass methods as arguments to other methods.
- The method signature (parameters and return type) must match between the "method" and "delegate".
- Delegates can be used as "parameter type" or "return type" of a method.
- You can store references of non-static method or static method in the delegate object.
- The methods, which reference is stored in the "single-cast delegate object", can have return value.
- The methods, which reference is stored in the "multi-cast delegate object", can't have return value; in case, if they have return value, the return value of lastly-executed method only can be received; others will be ignored.
- All delegate types are derived from "System.Delegate" class.

Types of Delegates

Single-Cast Delegates

- Contains reference of only one method.
- When called, it directly invokes the referenced method.

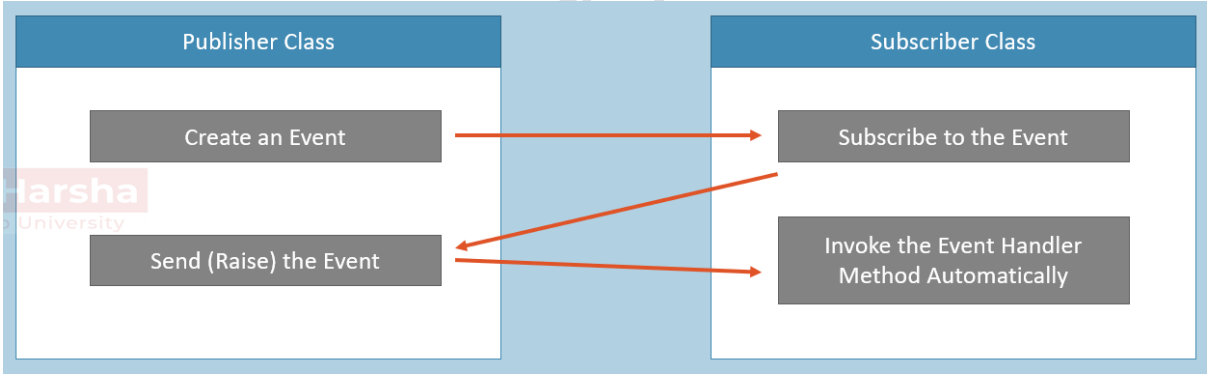
Multi-Cast Delegates

- Contains references of multiple methods.
- When called, it invokes all the referenced methods, one-by-one in a sequence.
- All methods' parameters and return type should be same.

Events

Event is a multi-cast delegate that stores one or more methods; and invoke them every time when the event is raised (called).

The event can be raised only in the same class, in which it is created.



- Publisher class is a class that sends (or raises) events (notifications), is called as "publisher class".
- Publisher class sends events; then Subscriber class receives events.
- Subscriber class is a class that receives (or subscribes or handles) events (notifications), is called as "subscriber class".

Events enable a class to send notifications to other classes, when something occurs.

Publisher class sends events; Subscriber class receives events.

Process of Events

1. The Publisher class creates an event.
2. The Subscriber class subscribes to the event; that means an "event handler" method is created in the subscriber class. The "event handler" method is nothing but, the method which is dedicated to be executed when the event is raised.
3. The publisher class can send (raise) events.
4. Every time, when the event is raised by the publisher, the corresponding "event handler" method executes automatically.

Creating Events

1. Create a Delegate

```
public delegate ReturnType DelegateTypeName(param1, param2, ...);
```

2. Create an Event in Publisher class

```
class Publisher
{
  private DelegateTypeName eventVariable;

  public event DelegateTypeName EventName
  {
    add
    {
      eventVariable += value;
    }

    remove
    {
      eventVariable -= value;
    }
}
```

3. Raise the event in Publishder class

```
if (EventName != null)
   EventName(arg1, arg2, ...);
```

4. Create Event Handler Method in Subscriber class

```
class Subscriber
{
  public ReturnType EventHandlerMethodName(param1, param2, ...)
  {
    Method body here
  }
}
```

5. Subscribe to the Event (Inside or outside the subscriber class)

```
EventName += EventHandlerMethodName;
```

Rules of Events

- The event should be created based on the delegate. That means, the event accepts the methods that are having specific parameters and return type, defined in the delegate.
- An event can have multiple subscribers.
- A subscriber can subscribe multiple events from multiple publishers.
- Events are basically signals to inform to other classes, that some important thing happened in the publisher class.
- Events are special kind of "multi-cast delegates", which can be raised only within the same class, in which they are created.
- Events can be static, virtual, sealed and abstract.
- Events will not be raised (throws exception), if there is no at least one subscriber.
- Events can be defined in interfaces.
- It's not a good idea to return value in events.

Auto-Implemented Events

"Auto-Implemented Events" provide a shortcut syntax to create events with less code.

In this case, you need not create "add" and "remove" accessors; the compiler does the same automatically.

Create an Auto-Implemented Event in Publisher Class

```
class Publisher
{
}
```

You also not required to create a private multi-cast delegate; the compiler does the same automatically.

Disadvantage / Limitation: We can't define custom logic for "add accessor" and "remove accessor".

Anonymous Methods

Anonymous methods are "name-less methods", that can be invoked by using the delegate variable or an event.

Subscribe to Event with Anonymous Method:

```
EventName += delegate(param1, param2, …)
{
  //method body here
}
```

Anonymous methods can be used anywhere within the method, to create methods instantly, without define a method at the class level.

Advantage: We need not create a "named method (normal method)" to quickly handle an event.

Rules:

- It can't be called without a delegate or event.
- It can't contain jump statements like goto, break, continue.
- It can access local variables and parameters of outer method.
- It can be passed as a parameter to any method; in this case, the delegate acts as data type for the anonymous method.
- It can't access ref or out parameter of an outer method.
- It is mainly used for event handlers.

Lambda Expressions

"Lambda Expressions" (a.k.a. Statement Lambda) are "name-less methods", that can be invoked by using the delegate variable or an event, much like anonymous methods.

Handle Event with Lambda Expressions:

```
EventName += (param1, param2, …) =>
{
  //method body here
}
```

Lambda Expressions can be used anywhere within the method, to create methods instantly, without define a method at the class level.

Advantage: It provides more easier and convenient syntax than "Anonymous methods".

=> operator is called as "goes to" or "goes into" operator.

Inline Lambda Expressions

"Inline Lambda Expressions" (a.k.a. Expression Lambda) are the lambda expressions, which performs a small calculation or condition check and returns a value.

Inline lambdas can receive one or more arguments and must return a value.

Advantage: It provides more easier and convenient syntax to create smaller methods that performs a single calculation or condition check.

Handle Event with Inline Lambda Expressions:

EventName += (param1, param2, …) => condition or calculation

Expression Bodied Members

"Expression Bodied Members" concept allows the developer to use "Inline Lambda Expressions" to create methods, property accessors, constructors, destructors, indexers in a class.

Method using Expression Bodied Members - without return value:

public ReturnType MethodName( ) => statement;

Method using Expression Bodied Methods - with return value

public ReturnType MethodName( ) => AnyValue;

Expression Bodied Members may have or parameters; may / may not have return value.

Expression Bodied Members can have only one statement.

Advantage: It provides more easier and convenient syntax to create smaller methods that performs a single calculation or condition check.

Expression Bodied Members - Usage

Constructor with Expression Bodied Members:

public ClassName(param1) => field = param1;

Property with Expression Bodied Members:

public type Property

{

```
  set => field = value;
  get => field;
}
```

Switch Expression

```
sourceVariable switch
{
  value1 => result1,
  value2 => result2,
  …
  _ => defaultResult
}
```

The variable used in switch expression is now coming before the switch keyword.

Colon (:) and case keyword are replaced with arrows (=>). Which makes the code more compact and readable.

The default case is now replaced with a discard(_).

And the body of the switch is expression, not a statement.

'Switch Expression' returns the result value based on the matching case.

It is only meant for getting a specific result value; doesn't let you to write multiple statements.

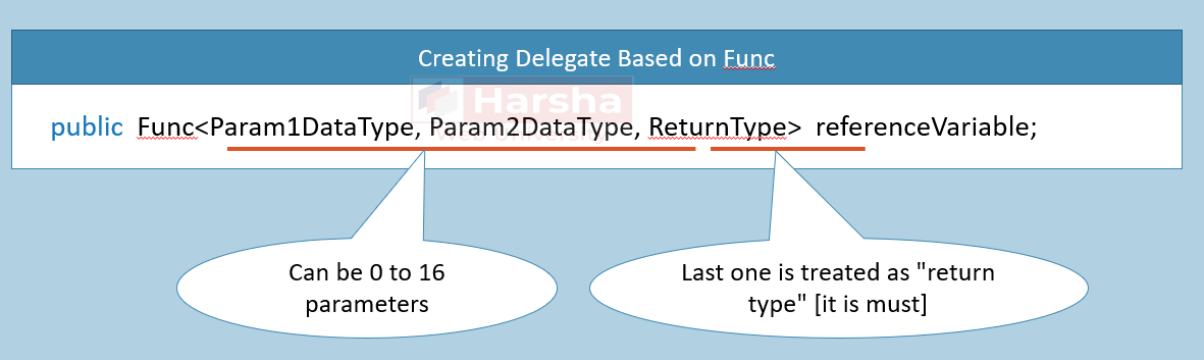
Func

"Func" is a pre-defined generic-delegate, which can be used to create events quickly.

Func supports parameters and return value also.

- Func must have 0 to 16 parameters.

- Func must have return value.



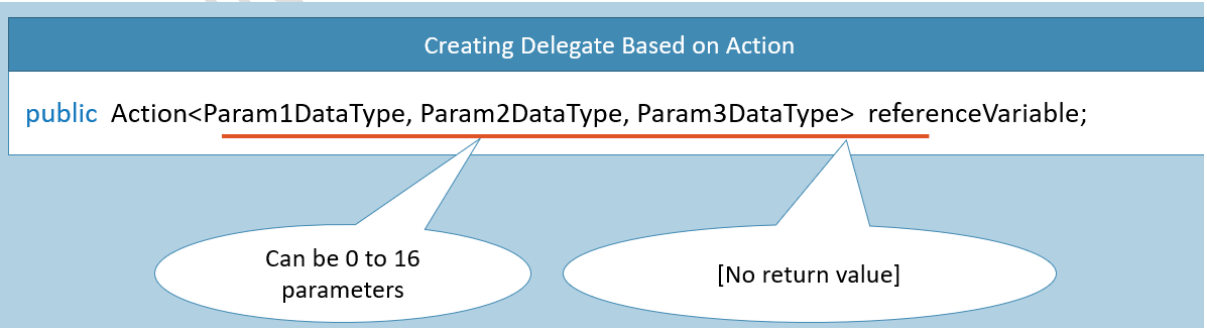
Action

"Action" is a pre-defined delegate, which can be used to create events quickly, similar to "Func".

The difference is:

1. Func must have return value; Action don't have return value.

2. Action must have 0 to 16 parameters.



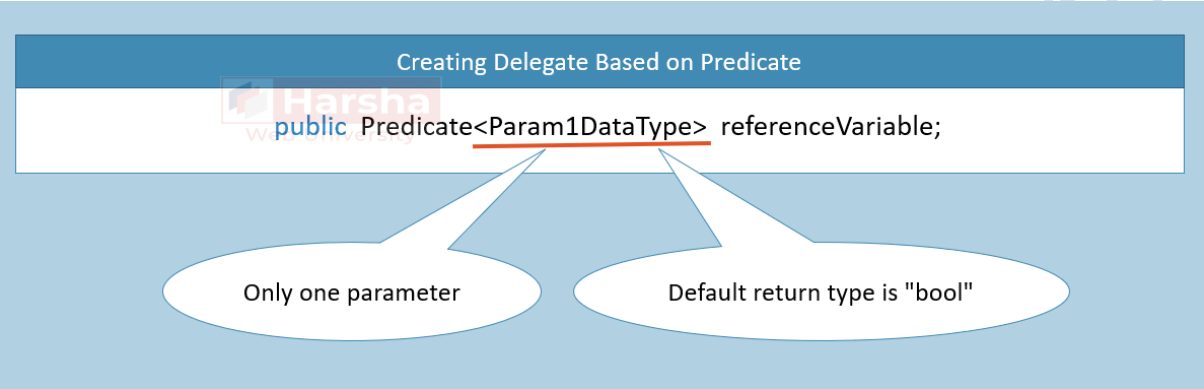
Predicate

"Predicate" is a pre-defined delegate, which can be used to create events quickly, similar to "Func".

The difference is:

1. Func must have return value of any type; Action don't have return value; Predicate must have return value of "bool" type.

2. Func can have 0 to 16 parameters of any type; Action can have 0 to 16 parameters of any type; Predicate must have only one parameter of any type.



EventHandler

'EventHandler' is a pre-defined delegate type, which has two parameters called "object sender" and "EventArgs e"; and no return.

object sender: Represents the source object, where the from where the event is originally raised.

EventArgs e: Represents additional parameters to pass to 'event handler method'. It is recommended to create a child class for 'EventArgs' class.

Creating Event Based on EventHandler:

public event EventHandler EventName;

Expression Trees

Expression Tree is a collection of delegates represented in tree-like structure.

Expression Tree only executes when we compile and execute it.

Expression Trees support all delegate types such as Func, Action, Predicate or custom delegate types.

1. Creating Expression Tree based on Func:

Expression< Func<type1, type2, …> > referenceVariable;

2. Compile and Execute Expression Tree:

Func<type1, type2, …> referenceVariable2 = referenceVariable.Compile( );

referenceVariable2.Invoke(arg1, arg2, …);