Event driven architecture
Running programs can be "interrupted" by various "events": exceptions and events.

Exception and event handling changes “normal” execution flow.
Exceptions usually represent an error condition or unmet system request
Provide a mechanism for the application to recovery and continue.
Usage generates robust applications.

Events are usually generated by the user via the GUI.
There are internal events ( timers – animations).
Exceptions

Exception handling allows an application to respond to program states that are not attributable to errors in program construction or user input.

Exceptional situations "raise" , "fire", or throw exceptions.

Un-handled, caught, exceptions cause program termination.

When an exception is thrown the CLR looks for a handler (code block) in the current method. If not found the search for a handler continues by popping the call stack until one is found or the default CLR termination handler is invoked.

When an exception is caught and handled execution continues from the handler -- it does not return to the where the exception was thrown! (Not really an interrupt).
try-catch-finally

Thrown exceptions are typed and pass an exception Object to the catch block.

Exception classes can be derived from existing exceptions.
   Exception property Message can be used to set a display string

Code that could cause an exception to be thrown is placed inside a try block.

Code that handles the exception is placed inside a catch block.

There can be many catch blocks.

Catch blocks should be sequenced in the order of priority. The first one encountered will be used.

Optional code guaranteed to execute regardless of an exception being handled or not, can be placed after a catch block in a finally block.
public aMethod () {
    ...
    try {
        // exception possible code
    }
    catch (exception1 e1) {
        // code to handle exception1
    }
    catch (exception2 e2) {
        // code to handle exception2
    }
    ...
    catch (exceptionN eN) {
        // code to handle exceptionN
    }
    [ finally {
        // code to execute regardless of exception
    } ]
}
<< see example catchException.cs >>
event concept
An event is the encapsulation that "something has happened". Usually events are user responses to a GUI control (button, menu).

Events need event handlers to define behavior to do “On Event”.

Event Wiring Problem
The button class designer wants the button to have an action "be able to respond to" an event – they define the event (click), but can't define the future application's specific behavior.

The application designer wants to use a button event (click) and specify the method that defines the button's action when clicked.

Events are delegates that associate “event actions” with event handlers

```csharp
// GUI Library

// Button class
definition

Button aButton;

// Method that defines the action
void ButtonHandler()
{
    ...  // Application code
}
```
Delegates are type safe "function pointers"—
specify method to be invoked (called) at run time.

events are specialized (convience) delegates.

Delegates are ADTs (class types) – declared and instantiated.  
**can’t be derived.**

Delegates encapsulate a method, specific method signature and return type with a reference object.

```
public delegate type_d delegateClassName ([type_i arg_i]+);  
...  
type_d aMethodName ([type_i arg_i]+) {  
    ...  
    return type_d; }  
...  
public delegateClassName delegateName (aMethodName );  
...  
delegateName ([type_i arg_i]+); // invokes aMethodName();
```

Instantiated delegates are objects that can be passed to methods…
DelegateLiberty

ImageProcessor

Delegate DoEffect
Delegate[] arrayOfEffects
DoEffect *Effect

ImageProcessor()
    new arrayOfEffects[10]
    new DoEffect(method)
    ...

Test
    img = new Image
    ip = new ImageProcessor
    ip.AddToEffects(ip.*Effect)
    ...
    ip.ProcessImages()
delegate properties

Delegates have to be instantiated. The instantiation could be done “automatically” via the Class by defining the delegates as static.

To eliminate the instantiations of delegates that may not be used, the delegate instantiation can be replaced with properties (static or not). Now the delegate isn’t created unless it is requested (with a get).

```csharp
public static DelegateClassName delegateName =
    new delegateClassName(methodName);
```

is replaced with

```csharp
public static DelegateClassName DelegatePropertyName {
    get { return new DelegateName(methodName); } } }
```

<< see DelegateDemo.cs >>
registering methods

Delegates can be multicast – more than 1 method is called when delegate is activated.

Methods can be registerered with delegates when they are constructed and dynamically thereafter.

\[
\text{aDelegate} = \text{method1} + \text{method2};
\]

\[
\text{aDelegate} += \text{method3}; \quad // \text{method1, method2 and method3}
\]

Methods can be unregistered

\[
\text{aDelegate} -= \text{method2}; \quad // \text{method1 and method 3}
\]

Observer Pattern

The registering and unregistering of “handlers” with “event actions” is often referred to as publishing (event action) and subscribing (handlers) or the Observer Pattern. When the event observing object changes state its dependents (subscribers) are notified and updated automatically.
Events

Delegate problems: (solved with events)

Use of \(=\) instead of \(+=\) or \(-=\) can replace all registered methods.

Methods associated with delegates can be called directly – outside of the planned delegate usage.

Event keyword used to define “delegates” that can only be invoked by defining class and other classes can only subscribe to and unsubscribe to using \(+=\) and \(-=\) operators.

```csharp
public delegate typed delegateClassName ([type\(_i\) arg\(_i\)]+);
...
public event delegateClassName eventName;
...
typed aMethodName ([type\(_i\) arg\(_i\)]+) {
    ...
    return type\(_d\); }
...
eventName += new delegateClassName(aMethodName);
```

[see Liberty’s example 12-5]
In .NET Framework events are implemented as delegate properties of GUI controls.

The event handler methods associated with events by convention have the following form:

```csharp
access void eventHandlerName
    ( object sender, EventArgs eventArg) {
        ...
    }
```

The second argument can be any object derived from EventArgs.

Event handlers are "appended to" the delegate collection with:

```csharp
GuiControlObject.eventProperty +=
    new EventHandler(eventHandlerName);
```

For example

```csharp
cancelButton.Click += new EventHandler(doCancelEvent);
```
Frameworks
Software framework is a "template of a solution" to a category of problems.

The structure of the framework consists of a set of classes that cooperate closely to facilitate reusable solutions (or partial solutions).

  foundation methods are subclassed by rarely overridden
  specialization methods tailor the generic solution to a specific soln.

For example: GUIs, simulations, game engines …

Traditional library use

Framework library use

Application code 3rd party code