Thread-safe Events in C#

Mark Pelf
11 Mar 2022 MIT

Discussion on proper method to check for null-value and raise Event in C#

In this article, we discuss three most common ways to check for null-value and raise Event in C#. Thread safety is analyzed. Then, in a small demo program, by creating thread race situation, we attack each solution and demo its thread-safety.

Download source code - 81.5 KB

Three Most Common Ways to Check for null-value and Raise an Event

In articles on the internet, you will find a lot of discussions on what is the best and thread-safe way to check for `null`-value and raise `Event` in C#. Usually, there are three methods mentioned and discussed:

C#

```
public static event EventHandler<EventArgs> MyEvent;

Object obj1 = new Object();
EventArgs args1 = new EventArgs();

//Method A
if (MyEvent != null)            //(A1)
{
    MyEvent(obj1, args1);       //(A2)
}

//Method B
var TmpEvent = MyEvent;         //(B1)
if (TmpEvent != null)           //(B2)
{
    TmpEvent(obj1, args1);      //(B3)
}

//Method C
MyEvent?.Invoke(obj1, args1);   //(C1)
```

Let us immediately give an answer: method A is not thread-safe, while methods B and C are thread-safe ways to check for `null`-value and raise an `Event`. Let us provide an analysis of each of them.

Analyzing Method A

In order to avoid `NullReferenceException`, in (A1) we check for `null`, then in (A2) we raise the `Event`. The problem is that in the time between (A1) and (A2), some other thread can access `Event MyEvent` and change its status. So, this approach is not thread safe. We demo that in our code (below) where we successfully launch race-thread attack on this approach.

Analyzing Method B

Key to understanding this approach is to really understand well what is happening in (B1). There, we have objects and assignment between them.

At first, one might think, we have two C# object references and assignment between them, So, they should be pointing to the same C# object. That is not the case here, since then there would be no point of that assignment. Events are C# objects (you can assign `Object obj=MyEvent`, and that is legal), but that assignment in (`B1`) is different there.

The real type of `TmpEvent` generated by compiler is `EventHandler<EventArgs>`. So, we basically have assignment of an `Event` to a delegate. If we assume that Events and Delegates are different types (see text below), conceptually compiler is doing implicit cast, that is the same as if we wrote:

C#

```
//not needed, just a concept of what compiler it is implicitly doing
EventHandler<EventArgs> TmpEvent = EventA as EventHandler<EventArgs>;  //(**)
```

As explained in [1], Delegates are immutable reference types. This implies that the reference assignment operation for such types creates a copy of an instance unlike the assignment of regular reference types which just copies the values of references. The key thing here is what really happens with `InvocationList` (that is of type `Delegate[]`) which contains list of all added delegates. What it seems is that list is Cloned in that assignment. That is the key reason why method B will work, because nobody else has access to newly created variable `TmpEvent` and its inner `InvocationList` of type `Delegate[]`.

We demo that this approach is thread safe in our code (below) where we launch race-thread attack on this approach.

Analyzing Method C

This method is based on `null`-conditional operator that is available from C#6. For thread safety, we need to trust Microsoft and its documentation. In [2], they say:

> *"The '?.' operator evaluates its left-hand operand no more than once, guaranteeing that it cannot be changed to* `null` *after being verified as non-null.... Use the* `?.` *operator to check if a delegate is non-null and invoke it in a thread-safe way (for example, when you raise an event)."*

We demo that this approach is thread safe in our code (below) where we launch race-thread attack on this approach.

Are Events same as Delegates?

In the above text at (**), we were arguing that in (B1), we have implicit cast from `Event` to a `Delegate`. But, are `Event`s and `Delegate`s the same or different type in C#?

If you look at [3], you will find the author Jon Skeet strongly arguing that `Event`s and `Delegate`s are not the same. To quote:

> *"Events aren't delegate instances. It's unfortunate in some ways that C# lets you use them in the same way in certain situations, but it's very important that you understand the difference. I find the easiest way to understand events is to think of them a bit like properties. While properties look like they're fields, they're definitely not ..... Events are pairs of methods, appropriately decorated in IL to tie them together ......"*

So, based on the text above by Jon Skeet and comments on this article below by Paulo Zemek, we can accept the interpretation that “events are like special kind of properties”. Following on that analogy, we can in our demo program below replace:

C#

```
public static event EventHandler<EventArgs> EventA;
public static event EventHandler<EventArgs> EventB;
public static event EventHandler<EventArgs> EventC;
```

with:

C#

```
public static EventHandler<EventArgs> EventA { get; set; } = null;
public static EventHandler<EventArgs> EventB { get; set; } = null;
public static EventHandler<EventArgs> EventC { get; set; } = null;
```

and everything will still work. Also, it is interesting to try this code:

C#

```
public static event EventHandler<EventArgs> EventD1;
public static EventHandler<EventArgs> EventD2 { get; set; } = null;
public static EventHandler<EventArgs> EventD3;

EventD1 = EventD2 = EventD3 = delegate { };
Console.WriteLine("Type of EventD1: {0}", EventD1.GetType().Name);
Console.WriteLine("Type of EventD2: {0}", EventD2.GetType().Name);
Console.WriteLine("Type of EventD3: {0}", EventD3.GetType().Name);
```

You will get a response:

```
Type of EventD1: EventHandler`1
Type of EventD2: EventHandler`1
Type of EventD3: EventHandler`1
```

But, going back to reality, events are created by “`event`“ keyword and therefore they are separate construct in C# language, then properties or delegates. We can “interpret” them that they are “alike” properties or delegates, but they are not the same. Truth is Events are whatever compiler is doing with that keyword “`event`”, and it seems that it makes them look like C# Delegates.

I am inclined to think like this: Events and Delegates are strictly speaking not the same, but in C# language, it seems that they are treated interchangeably in a very similar manner, so it has become accustomed in the industry to talk about them as they are the same, interchangeably. Even in Microsoft documentation [2], the author is interchangeably using terms Event and Delegate when discussing null-conditional operator “?.”. In one moment, the author talks about “..raise an event”, then the next sentence says “…delegate instances are immutable…” etc.

Race-Thread Attack on Three Proposed Approaches

In order to verify thread safety of the three proposed approaches, we created a small demo program. This program is not a definite answer for all cases and cannot be considered as a “proof”, but still can show/demo some interesting points. In order to setup race situations, we slow down threads with some `Thread.Sleep()` calls.

Here is the demo code:

C#

```
internal class Client
{
    public static event EventHandler<EventArgs> EventA;
    public static event EventHandler<EventArgs> EventB;
    public static event EventHandler<EventArgs> EventC;
    public static void HandlerA1(object obj, EventArgs args1)
    {
        Console.WriteLine("ThreadId:{0}, HandlerA1 invoked",
            Thread.CurrentThread.ManagedThreadId);
    }
    public static void HandlerB1(object obj, EventArgs args1)
    {
        Console.WriteLine("ThreadId:{0}, HandlerB1 invoked",
            Thread.CurrentThread.ManagedThreadId);
    }

    public static void HandlerC1(object obj, EventArgs args1)
    {
        Console.WriteLine("ThreadId:{0}, HandlerC1 - Start",
            Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(3000);
        Console.WriteLine("ThreadId:{0}, HandlerC1 - End",
            Thread.CurrentThread.ManagedThreadId);
    }
    public static void HandlerC2(object obj, EventArgs args1)
    {
        Console.WriteLine("ThreadId:{0}, HandlerC2 invoked",
            Thread.CurrentThread.ManagedThreadId);
    }

    static void Main(string[] args)
    {
        // Demo Method A for firing of Event-------------------------------
        Console.WriteLine("Demo A ========================");

        EventA += HandlerA1;

        Task.Factory.StartNew(() =>  //(A11)
        {
            Thread.Sleep(1000);
            Console.WriteLine("ThreadId:{0}, About to remove handler HandlerA1",
                Thread.CurrentThread.ManagedThreadId);
            EventA -= HandlerA1;
            Console.WriteLine("ThreadId:{0}, Removed handler HandlerA1",
                Thread.CurrentThread.ManagedThreadId);
        });

        if (EventA != null)
        {
            Console.WriteLine("ThreadId:{0}, EventA is null:{1}",
                Thread.CurrentThread.ManagedThreadId, EventA == null);
            Thread.Sleep(2000);
            Console.WriteLine("ThreadId:{0}, EventA is null:{1}",
                Thread.CurrentThread.ManagedThreadId, EventA == null);

            Object obj1 = new Object();
            EventArgs args1 = new EventArgs();

            try
            {
                EventA(obj1, args1);  //(A12)
            }
            catch (Exception ex)
            {
```

```
            Console.WriteLine("ThreadId:{0}, Exception:{1}",
                Thread.CurrentThread.ManagedThreadId, ex.Message);
        }
    }

    // Demo Method B for firing of Event-------------------------------
    Console.WriteLine("Demo B ========================");

    EventB += HandlerB1;

    Task.Factory.StartNew(() =>  //(B11)
    {
        Thread.Sleep(1000);
        Console.WriteLine("ThreadId:{0}, About to remove handler HandlerB1",
            Thread.CurrentThread.ManagedThreadId);
        EventB -= HandlerB1;
        Console.WriteLine("ThreadId:{0}, Removed handler HandlerB1",
            Thread.CurrentThread.ManagedThreadId);
    });

    var TmpEvent = EventB;
    if (TmpEvent != null)
    {
        Console.WriteLine("ThreadId:{0}, EventB is null:{1}",
            Thread.CurrentThread.ManagedThreadId, EventB == null);
        Console.WriteLine("ThreadId:{0}, TmpEvent is null:{1}",
            Thread.CurrentThread.ManagedThreadId, TmpEvent == null);
        Thread.Sleep(2000);
        Console.WriteLine("ThreadId:{0}, EventB is null:{1}",   //(B13)
            Thread.CurrentThread.ManagedThreadId, EventB == null);
        Console.WriteLine("ThreadId:{0}, TmpEvent is null:{1}",   //(B14)
            Thread.CurrentThread.ManagedThreadId, TmpEvent == null);

        Object obj1 = new Object();
        EventArgs args1 = new EventArgs();

        try
        {
            TmpEvent(obj1, args1);  //(B12)
        }
        catch (Exception ex)
        {
            Console.WriteLine("ThreadId:{0}, Exception:{1}",
                Thread.CurrentThread.ManagedThreadId, ex.Message);
        }
    }

    // Demo Method C for firing of Event-------------------------------
    Console.WriteLine("Demo C ========================");

    EventC += HandlerC1;
    EventC += HandlerC2;  //(C11)

    Task.Factory.StartNew(() =>   //(C12)
    {
        Thread.Sleep(1000);
        Console.WriteLine("ThreadId:{0}, About to remove handler HandlerC2",
            Thread.CurrentThread.ManagedThreadId);
        EventC -= HandlerC2;
        Console.WriteLine("ThreadId:{0}, Removed handler HandlerC2",
            Thread.CurrentThread.ManagedThreadId);
    });

    Console.WriteLine("ThreadId:{0}, EventC has EventHandlers:{1}",
        Thread.CurrentThread.ManagedThreadId, EventC?.GetInvocationList().Length);
        try
        {
            Object obj1 = new Object();
            EventArgs args1 = new EventArgs();

            EventC?.Invoke(obj1, args1);

            Console.WriteLine("ThreadId:{0}, EventC has EventHandlers:{1}",
            Thread.CurrentThread.ManagedThreadId, EventC?.GetInvocationList().Length);  //(C13)
        }
        catch (Exception ex)
        {
            Console.WriteLine("ThreadId:{0}, Exception:{1}",
                Thread.CurrentThread.ManagedThreadId, ex.Message);
        }

        Console.WriteLine("End =========================");
        Console.ReadLine();
    }
}
```

And here is the execution result:

A) In order to attack Method A, we at (A11) launch new racing thread that is going to do some damage. We will see that it succeeds to create `NullReferenceException` at (A12)

B) In order to attack Method B, we at (B11) launch new racing thread that is going to do some damage. We will see that at (B12) nothing eventful will happen and this approach will survive this attack. Key thing is printout at (B13) and (B14) that will show that `TmpEvent` is not affected by changes to `EventB`.

C) We will attack method C in a different way. We know that `EventHandler`s are invoked synchronously. We will create 2 `EventHandler`s (C11) and will during execution of the first one, attack with racing thread (C12) and try to remove the second handler. We will from printouts see that attack has failed and both `EventHandler`s were executed. It is interesting to look at output at (C13) that shows that AFTER `EventC`, reports decreased number of `EventHandler`s.

Conclusion

The best solution is to avoid thread-racing situations, and to access Events from a single thread. But, if you need, Method C based on `null`-conditional operator is the preferred way to check for `null`-value and raise an `Event`.

References

- [1] https://stackoverflow.com/questions/55322255/what-if-i-will-copy-a-reference-to-an-event-object-to-another-object-and-will-ch
- [2] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#null-conditional-operators--and-
- [3] https://jonskeet.uk/csharp/events.html

History

- 10th March, 2022: Initial version

License

About the Author

Mark Pelf
Serbia

Mark Pelf is pseudonym of just another programmer from Belgrade, Serbia.

Comments and Discussions

10 messages have been posted for this article Visit **https://www.codeproject.com/Articles/5327025/Thread-safe-Events-in-Csharp** to post and view comments on this article, or click **here** to get a print view with messages.