

Operators, Delegates and Events

• 10.04.2014

Agenda

- Introduction to Operators
- Operator Overloading
- Creating and Using Delegates
- Defining and Using Events

Operators, Delegates and Events

Agenda

- Introduction to Operators
- Operator Overloading
- Creating and Using Delegates
- Defining and Using Events

Introduction to Operators

- Operators and Methods
- Predefined C# Operators

Operators are different from methods. They have special requirements that enable them to function as expected. C# has a number of predefined operators that you can use to **manipulate** the types and classes supplied with the Microsoft® .NET Framework.

Operators and Methods

- Using methods
 - Reduces clarity
 - Increases risk of errors, both syntactic and semantic

```
myIntVar1 = Int.Add(myIntVar2,  
    Int.Add(Int.Add(myIntVar3,  
        myIntVar4), 33));
```

- Using operators
 - Makes expressions clear

```
myIntVar1 = myIntVar2 + myIntVar3 + myIntVar4 + 33;
```

Operators and Methods

- The purpose of operators is to make expressions clear and easy to understand.
- We can use Method for adding two numbers:

```
myIntVar1 = Int.Add(myIntVar2, myIntVar3);
```

```
myIntVar2 = Int.Add(myIntVar2, 1);
```

- We can use Operator+:

```
myIntVar1 = myIntVar2 + myIntVar3;
```

```
myIntVar2 = myIntVar2 + 1;
```

Predefined C# Operators

Operator Categories	
Arithmetic	Member access
Logical (Boolean and bitwise)	Indexing
String concatenation	Cast
Increment and decrement	Conditional
Shift	Delegate concatenation and removal
Relational	Object creation
Assignment	Type information
Overflow exception control	Indirection and address

Predefined C# Operators

The C# language provides a large set of predefined operators. Following is the complete list.

Operator category	Operators
Arithmetic	+, -, *, /, %
Logical (Boolean and bitwise) &, , ^, !, ~, &&, , true, false	
String concatenation	+
Increment and decrement	++, --
Shift	<<, >>
Relational	==, !=, <, >, <=, >=
Assignment	=, +=, -=, *=, /=, %=, &=, =, <<=, >>=
Member access	.
Indexing	[]

Predefined C# Operators

The C# language provides a large set of predefined operators. Following is the complete list.

Operator category	Operators
▪ Cast	()
▪ Conditional	?:
▪ Delegate concatenation and removal	+, -
▪ Object creation	new
▪ Type information	is, sizeof, typeof
▪ Overflow exception control	checked, unchecked
▪ Indirection and address	*, ->, [], &

◆ Operator Overloading

- Introduction to Operator Overloading
- Overloading Relational Operators
- Overloading Logical Operators
- Overloading Conversion Operators
- Overloading Operators Multiple Times
- Quiz: Spot the Bugs

Operator Overloading

- We should only define operators when it makes **sense** to do so. Operators should **only** be overloaded when the class or struct is a piece of data (like a number), and **will be used** in that way.
- An operator should always be unambiguous in **usage**; there should be only one **possible** interpretation of what it means.
- For example, you should not define an increment operator (++) on an **Employee class** (emp1++;) because the semantics of such an operation on an Employee e are not **clear**.

Syntax for Overloading Operators

- All operators must be **public static methods** and their names follow a particular pattern:

`operator@`

`@` - specifies exactly which operator is being overloaded.

- For example, the method for overloading the addition operator is **`operator+`**.

Operator Overloading. Example

- All **arithmetic** operators return an **instance** of the class and manipulate objects of the class.

```
public static Time operator+(Time t1, Time t2)  
{  
    int newHours = t1.hours + t2.hours;  
    int newMinutes = t1.minutes + t2.minutes;  
    return new Time(newHours, newMinutes);  
}
```

Overloading Relational Operators

- Relational operators **must be paired**

< and >

<= and >=

== and !=

For consistency, create a **Compare** method first and define all the relational operators by using Compare.

- Override the **Equals** method if overloading == and !=
- Override the **GetHashCode** method if overriding Equals method

Overloading Relational Operators

- The following code shows how to implement the relational operators, the Equals method, and the GetHashCode method for the Time struct:

```
public struct Time
{ // Equality
    public static bool operator==(Time lhs, Time rhs)
        { return lhs.Compare(rhs) == 0;}
    public static bool operator!=(Time lhs, Time rhs)
        { return lhs.Compare(rhs) != 0;}
    // Relational
    public static bool operator<(Time lhs, Time rhs)
        { return lhs.Compare(rhs) < 0;}
    public static bool operator>(Time lhs, Time rhs)
        { return lhs.Compare(rhs) > 0;}
    public static bool operator<=(Time lhs, Time rhs)
        { return lhs.Compare(rhs) <= 0;}
    public static bool operator>=(Time lhs, Time rhs)
        { return lhs.Compare(rhs) >= 0;}
```


Overloading Relational Operators

- The following code shows how to implement the relational operators, the Equals method, and the GetHashCode method for the Time struct:

```
// Inherited virtual methods (from Object)
public override bool Equals(object obj)
{
    return (obj is Time) && Compare((Time)obj) == 0;
}
public override int GetHashCode( )
{
    return TotalMinutes( ) ;
}
private int Compare(Time other)
{
    int lhs = TotalMinutes( );
    int rhs = other.TotalMinutes( );
    int result;
    if (lhs < rhs)
        result = -1;
    else if (lhs > rhs)
        result = +1;
    else
        result = 0;
    return result;
} . . .
}
```

Overloading Logical Operators

- Operators `&&` and `||` cannot be overloaded directly
 - They are evaluated in terms of `&`, `|`, `true`, and `false`, which can be overloaded
 - `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`
 - `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`

Overloading Conversion Operators

- Overloaded conversion operators

```
public static explicit operator Time (float hours)
{ ... }
public static explicit operator float (Time t1)
{ ... }
public static implicit operator string (Time t1)
{ ... }
```

- You can define **implicit** and **explicit** conversion operators for your own classes and create programmer-defined cast operators that can be used to convert data from one type to another.

Overloading Conversion Operators

`explicit operator Time (int minutes)`

- It is explicit operator because **not all int** can be converted; a negative argument results in an exception being thrown.

`explicit operator Time (float minutes)`

- It is explicit operator because a **negative parameter** causes an exception to be thrown.

`implicit operator int (Time t1)`

- It is implicit operator because **all Time** values can **safely be converted** to int.

Overloading Conversion Operators

```
implicit operator string  
(Time t1)
```

- This operator converts a Time into a string. This is also implicit because there is **no danger of losing** any information in the conversion.
- If a class defines a string conversion operator - the class should `override ToString`

Overloading Conversion Operators

The following code shows how to implement the conversion operators and ToString method.

```
public struct Time { ...
public static explicit operator Time (int minutes) //
    Conversion operators
    { return new Time(0, minutes); }
public static explicit operator Time (float minutes)
    { return new Time(0, (int)minutes); }
public static implicit operator int (Time t1)
    { return t1.TotalMinutes( ); }
public static explicit operator float (Time t1)
    { return t1.TotalMinutes( ); }
public static implicit operator string (Time t1)
    { return t1.ToString( ); }
public override string ToString( ) // Inherited virtual
    methods (from Object)
    { return String.Format("{0}:{1:00}", hours, minutes); }
... }
```

Overloading Operators Multiple Times

- The same operator can be overloaded multiple times to provide **alternative implementations** that take different types as parameters. At compile time, the system establishes the method to be called depending upon the types of the parameters being used to **invoke the operator**.

```
public static Time operator+(Time t1, int hours)
{...}
```

```
public static Time operator+(Time t1, float hours)
{...}
```

```
public static Time operator-(Time t1, int hours)
{...}
```

```
public static Time operator-(Time t1, float hours)
{...}
```

Quiz: Spot the Bugs

```
public bool operator != (Time t1, Time t2)
{ ... }
```

1

```
public static operator float(Time t1) { ... }
```

2

```
public static Time operator += (Time t1, Time t2)
{ ... }
```

3

```
public static bool Equals(Object obj) { ... }
```

4

```
public static int operator implicit(Time t1)
{ ... }
```

5

Quiz: Spot the Bugs. Answers

1 Operators must be static. The definition for the != operator should be:
`public static bool operator != (Time t1, Time t2) { ... }`

2 The “type” is missing. Conversion operators must be implicit or explicit.
`public static implicit operator float (Time t1) { ... }`

3 You **cannot** overload the += operator. However, += is evaluated by using the + operator, which you can overload.

4 The Equals method should be an instance method rather than a class method. However, if you remove the static keyword, this method will hide the virtual method inherited from Object and not be invoked as expected, so the code should use override instead, as follows:

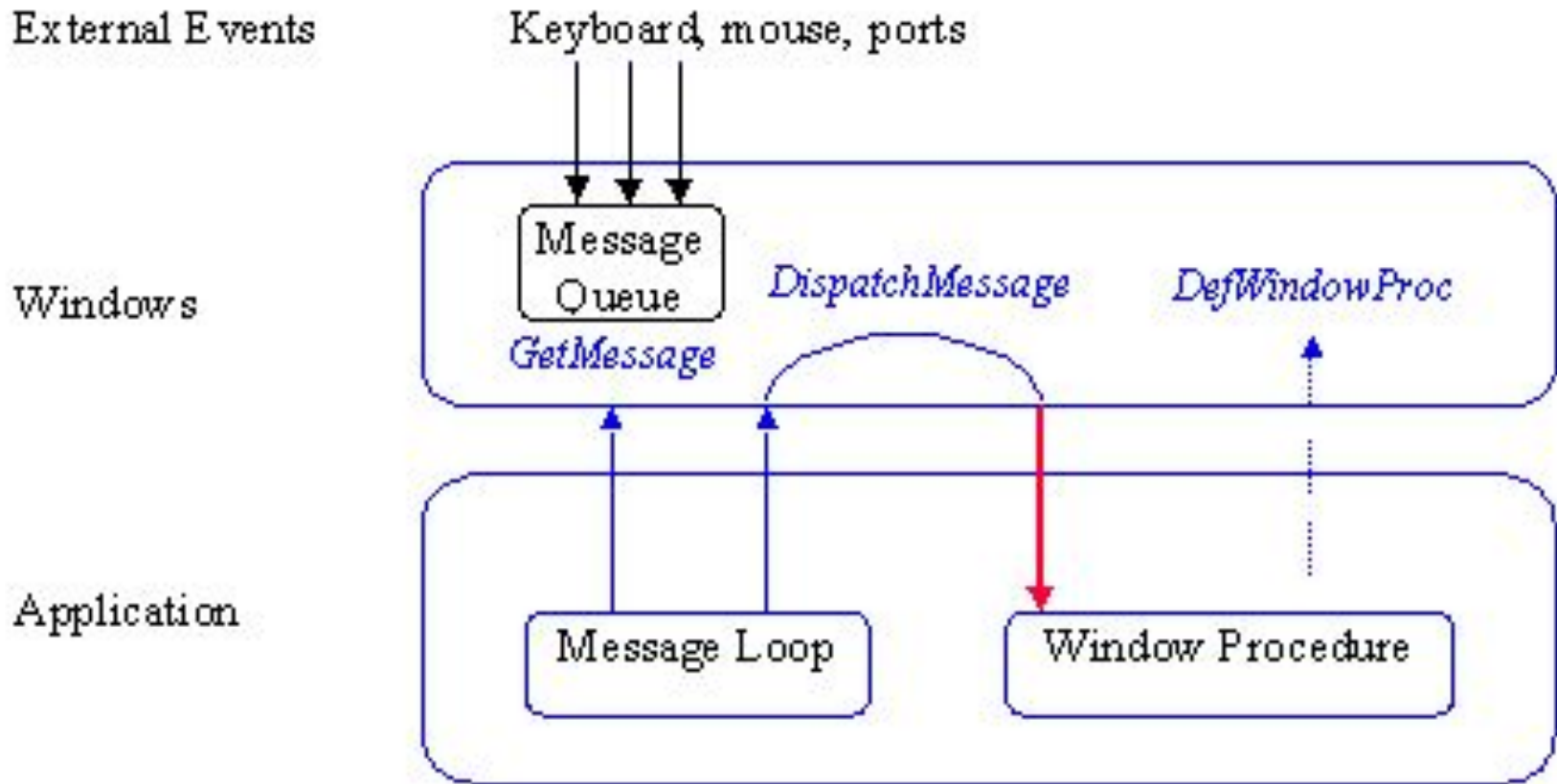
```
public override bool Equals(Object obj) { ... }
```

5 The int and implicit keywords have been transposed. The name of the operator should be int, and its type should be implicit, as follows:
`public static implicit operator int(Time t1) { ... }`

2. Windowing system

- Modern graphical environments use **event model** for communicating between interactive objects and the input/output system. The event model was developed to support direct manipulation interfaces.
- In a windowing system a user interface of an application is built of
 - top level windows, and
 - controls (ui components, child windows, widgets, ...).
- User actions with the input devices are translated into software *events* (messages) and distributed to the appropriate window. Events (or messages) are identified by an *event type*.

Process of WA execution



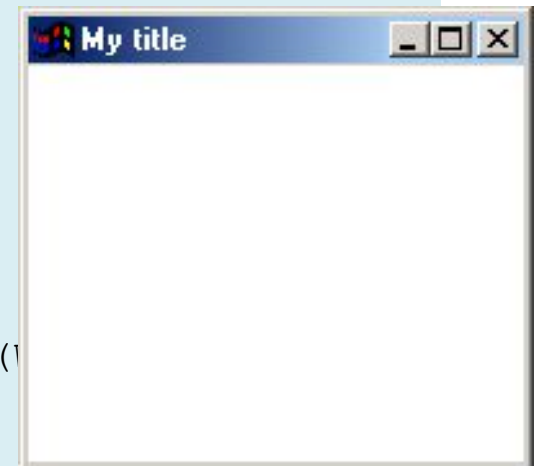
Analyzing the Problem. WinAPI

How create a **simple WIN32 window**

```
#include <windows.h>

LONG WINAPI WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR      lpCmdLine,
                  int        nCmdShow)
{
    HWND hMainWnd, hWndButton;
    MSG msg;
    WNDCLASS w;
    memset(&w, 0, sizeof(WNDCLASS));
    w.style = CS_HREDRAW | CS_VREDRAW;
    w.lpfnWndProc = WndProc;
    w.hInstance = hInstance;
    w.hbrBackground = (HBRUSH)GetStockObject(
    w.lpszClassName = "My Class";
    RegisterClass(&w);
```



Analyzing the Problem. WinAPI

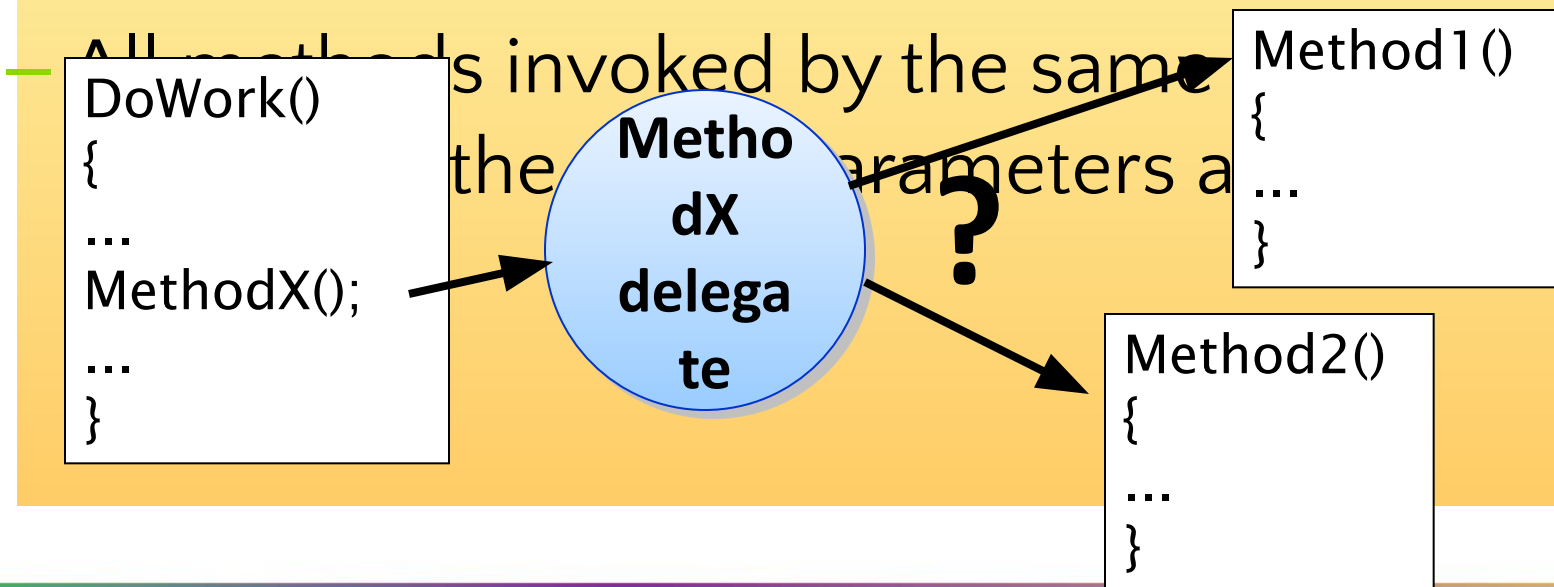
```
hMainWnd = CreateWindow("My Class", "My title",
    WS_OVERLAPPEDWINDOW,
    300, 200, 200, 180, NULL, NULL, hInstance, NULL);
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
while(GetMessage(&msg, NULL, 0,
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    return msg.wParam;
}
LONG WINAPI WndProc(HWND hwnd, UINT Message,
    WPARAM wParam,
    LPARAM lparam)
{
    switch (Message)
    {
        case WM_DESTROY: PostQuitMessage(0);
            return 0;
        default: return DefWindowProc(hwnd, Message, wParam, lparam);
    }
}
return 0;
```

```
hWndButton =
CreateWindow("BUTTON", "Copy",
    BS_PUSHBUTTON | WS_CHILD |
    WS_VISIBLE, 10, 10, 90, 20, hMainWnd,
    (HMENU)ID_BUTTON,
    (HINSTANCE)hInstance, NULL);
```

```
case WM_PAINT: ...
case WM_CLOSE: ...
case WM_DESTROY: ...
case WM_COMMAND: // Command from Child
    windows
    switch(wParam) // The ID is
    wParam
    { case ID_BUTTON: ... }
```

Delegates and event handlers in .NET

- A delegate allows a method to be called indirectly
 - A delegate is a special kind of class that holds a reference to a method with a pre-defined signature.



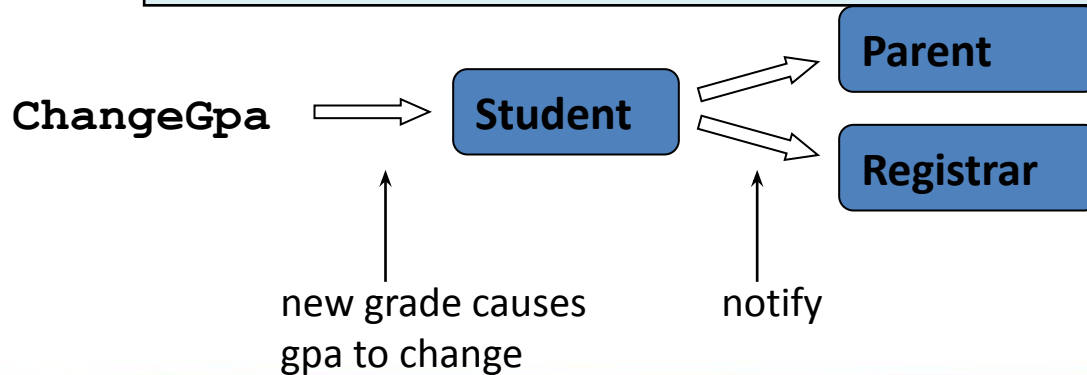
Using Delegates. Example.

```
class Student
{
    string name;
    double gpa;
    int    units;

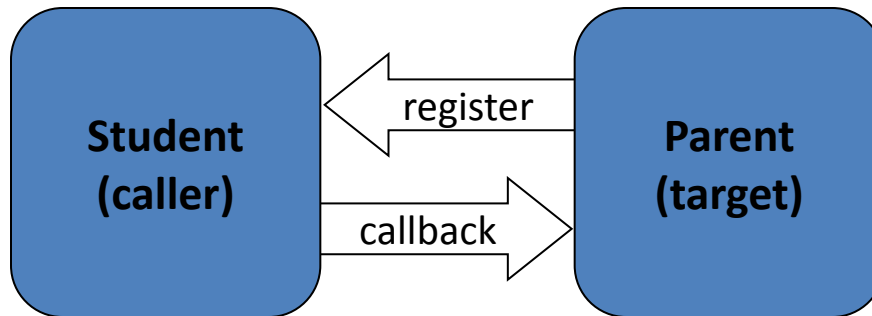
    public void ChangeGpa(int grade)
    {
        gpa = (gpa * units + grade) / (units + 1);
        units++;
    }
    ...
}
```

store state →

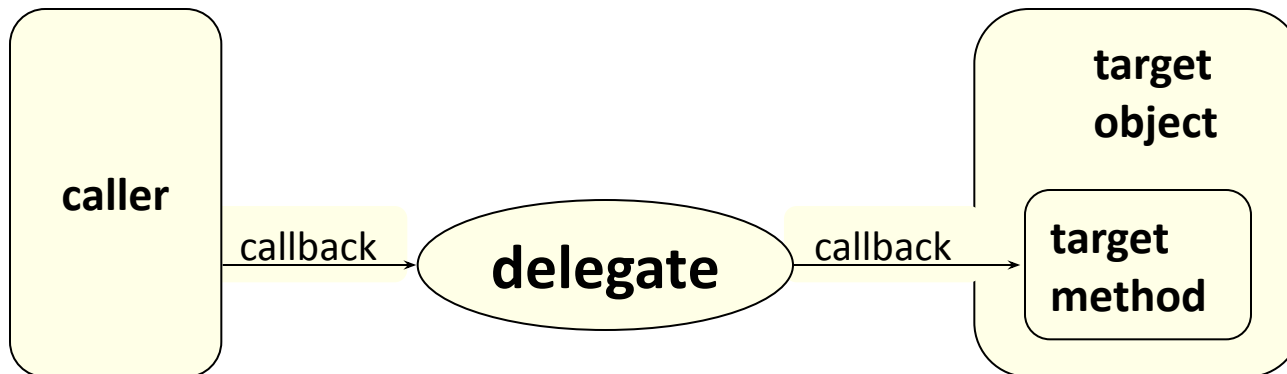
change state →



Using Delegates. Example.



.NET Framework uses delegates for callback supporting :



Using Delegates. Example.

```
class Parent
{
    public void Report(double gpa)
    { ...
    }
```

```
class Registrar
{
    public static void Log(double gpa)
    { ...
    }
```

target methods



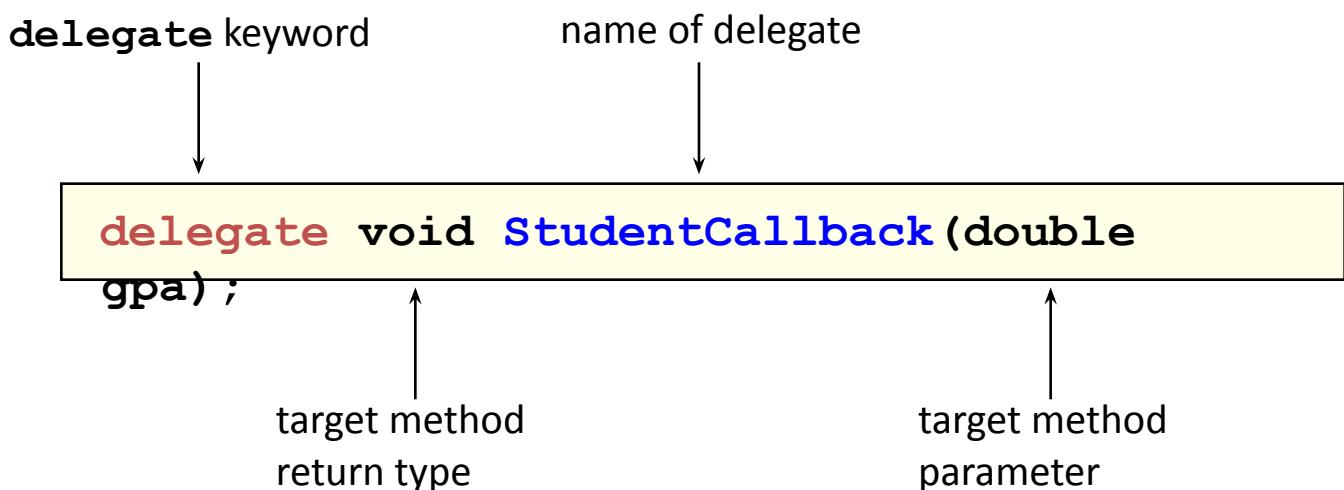
delegate keyword

name of delegate

```
delegate void StudentCallback(double  
gpa);
```

target method
return type

target method
parameter



Using Delegates. Example.

define delegate



```
delegate void StudentCallback(double gpa);
```

target method



```
class Parent  
{  
    public void Report(double gpa) { ... }  
}
```

caller stores delegate



```
class Student  
{  
    public StudentCallback GpaChanged;
```

caller invokes delegate



```
    public void ChangeGpa(int grade)  
    {  
        // update gpa  
        ...  
        GpaChanged(gpa);  
    }  
}
```

create and install delegate



```
Student ann = new Student("Ann");  
Parent mom = new Parent();  
  
ann.GpaChanged = new StudentCallback(mom.Report);  
ann.ChangeGpa(4);
```

Using Delegates. Example.

- Null reference

```
class Student
{
    public StudentCallback
    GpaChanged;

    public void ChangeGpa(int grade)
    {
        // update gpa
        ...
        if (GpaChanged != null)
            GpaChanged(gpa);
    }
}
```

test before call



Using Delegates. Example.

Static methods

static method →

```
class Registrar
{
    public static void Log(double gpa)
    {
        ...
    }
}
```

register →

```
void Run()
{
    Student ann = new Student("Ann");

    ann.GpaChanged = new StudentCallback(Registrar.Log);
    ...
}
```

Using Delegates. Example.

- Multiple delegates
- Overloading operator+= and operator+

```
targets → Parent mom = new Parent();  
          Parent dad = new Parent();  
  
          Student ann = new Student("Ann");  
  
first →  ann.GpaChanged += new  
second → StudentCallback(mom.Report);  
          ann.GpaChanged += new  
          StudentCallback(dad.Report);  
          ...
```

Using Delegates. Example.

- Removing delegate
- Overloading operator+= and operator-

```
Parent mom = new Parent();  
Parent dad = new Parent();  
  
Student ann = new Student("Ann");  
  
ann.GpaChanged += new  
StudentCallback(mom.Report);  
ann.GpaChanged += new  
StudentCallback(dad.Report);  
...  
ann.GpaChanged -= new  
StudentCallback(dad.Report);  
...
```

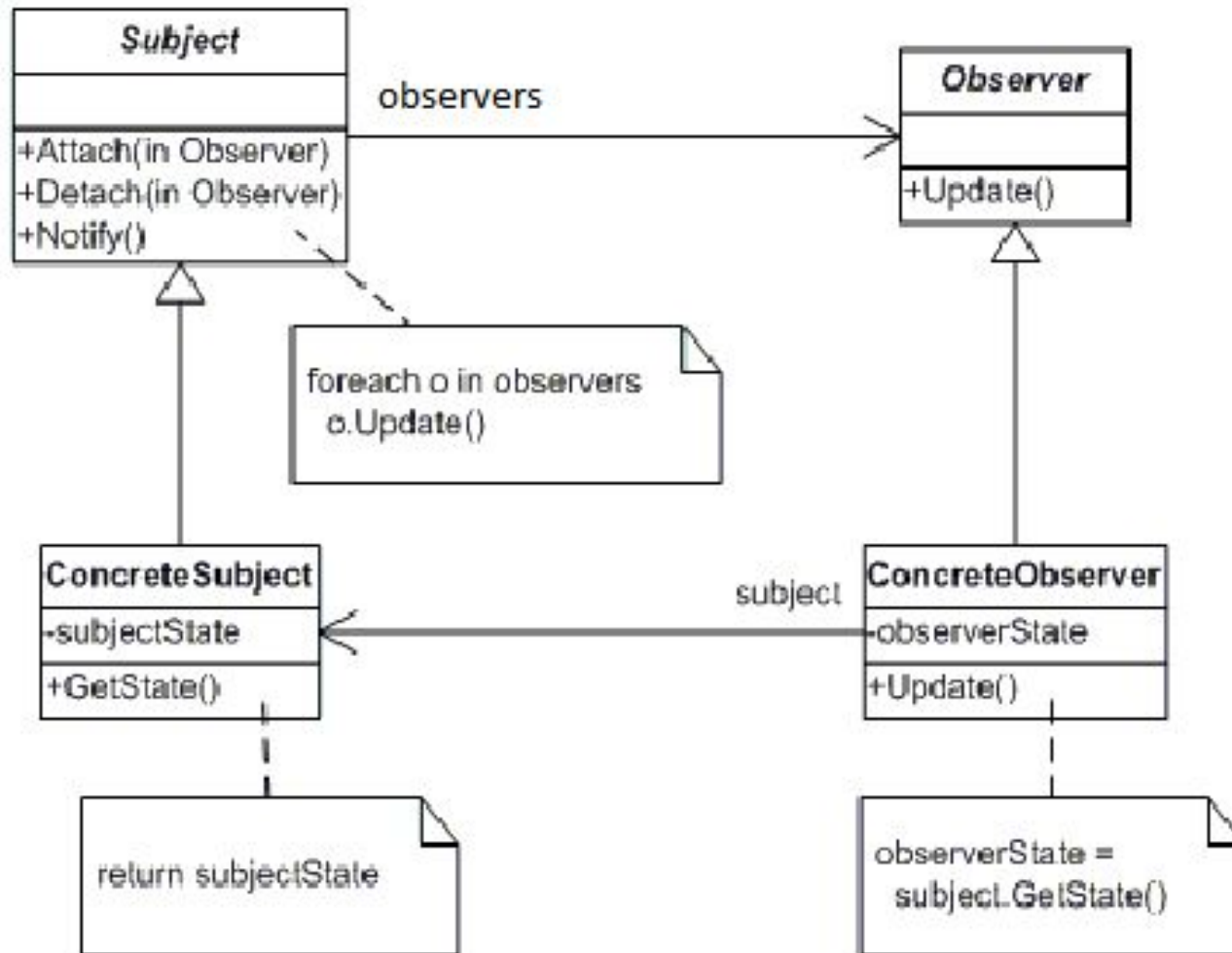
add →

remove →

Defining and Using Events

- How Events Work
- Defining Events
- Passing Event Parameters
- Demonstration: Handling Events

Pattern Observer



How Events Work

- Publisher (Student)
 - Raises an event to alert all interested objects (subscribers)
- Subscriber (Parents, Registrar)
 - Provides a method to be called when the event is raised

Defining Events

- Defining an event

```
public delegate void ChangedEventHandler (object sender, EventArgs e );  
private event ChangedEventHandler Changed;
```

- Subscribing to an event

```
List.Changed += new ChangedEventHandler(ListChanged);
```

```
protected virtual void OnChanged(EventArgs e)  
    {  
        if (Changed != null)  
            Changed(this, e);  
    }
```

Passing Event Parameters

- Parameters for events should be passed as EventArgs
 - Define a class descended from EventArgs to act as a container for event parameters
- The same subscribing method may be called by several events
 - Always pass the event publisher (sender) as the first parameter to the method

.NET Delegates

```
[SerializableAttribute]  
[ComVisibleAttribute(true)]  
public delegate void EventHandler ( Object sender,  
                                   EventArgs e )
```

```
[SerializableAttribute]  
public delegate  
void EventHandler<TEventArgs> ( Object sender,  
                                TEventArgs e )  
    where TEventArgs : EventArgs
```

```
[Serializable]  
public class EventArgs {  
    public static readonly EventArgs Empty = new EventArgs();  
    public EventArgs() { }  
}
```

- I am pretty sure you all must have seen these delegates when writing code. IntelliSense shows methods that accept Actions, Func<TResult> and some accept Predicate<T>. So what are these? Let's find out.

Let's go by a simple example. I have following "Employee" class and it has a helper method which will return me a list of Employees.

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }
    public int Age { get; set; }

    public static List<Employee> GetEmployees()
    {
        return new List<Employee>()
        {
            new Employee()
            {
                FirstName = "Jaliya",
                LastName = "Udagedara",
                Birthday = Convert.ToDateTime("1986-09-11")
            },
            new Employee()
            {
                FirstName = "Gary",
                LastName = "Smith",
                Birthday = Convert.ToDateTime("1988-03-20")
            }
        };
    }
}
```

In my Main method I am getting the list of type employees into a variable

```
List<Employee> employees = Employee.GetEmployees();
```

- Action series of delegates are pointers to methods which take zero, one or more input parameters, and do not return anything.
- Let's consider [List<T>.ForEach](#) method, which accepts a Action of type T. For my list of type Employee, it accepts an Action of type Employee.

```
List<Employee> employees = Employee.GetEmployees();  
employees.ForEach()
```

```
void List<Employee>.ForEach(Action<Employee> action)
```

Performs the specified action on each element of the System.Collections.Generic.List<T>.

action: The System.Action<T> delegate to perform on each element of the System.Collections.Generic.List<T>.

- So let's create an Action now. I have the following method which will calculate the age of the employee when the employee is passed in.
- `static void CalculateAge(Employee emp) { emp.Age = DateTime.Now.Year - emp.Birthday.Year; }`
- So I can create an Action, pointing to above method.
- `Action<Employee> empAction = new Action<Employee>(CalculateAge); employees.ForEach(empAction);`
- `foreach (Employee e in employees) { Console.WriteLine(e.Age); }`
- This will print me the calculated age for each employee. With the use of Lambda Expressions, I can eliminate writing a separate method for calculating the age and put it straight this way.
- `employees.ForEach(e => e.Age = DateTime.Now.Year - e.Birthday.Year);`

Func<TResult>

- Func<TResult> series of delegates are pointers to methods which take zero, one or more input parameters, and return a value of the type specified by the TResult parameter.
- For this, let's consider [Enumerable.First<TSource>](#) method, which has an overloading method which accepts a Func.

```
employees.First()
```

```
▲ 2 of 2 ▼ (extension) Employee IEnumerable<Employee>.First(Func<Employee,bool> predicate)  
Returns the first element in a sequence that satisfies a specified condition.  
predicate: A function to test each element for a condition.
```

- In my scenario, this particular method accepts Func which accepts an Employee and returns a bool value. For this, let's create a method which I am going to point my Func to. Following method accepts an employee and checks whether his/her FirstName is equal to "Jaliya" and returns true or false.
- `static bool` NamesEqual(Employee emp)
- `{ return emp.FirstName == "Jaliya"; }`
- Now I can create `aFunc<Employee, bool> myFunc = new Func<Employee, bool>(NamesEqual);`
`Console.WriteLine(employees.First(myFunc).FirstName);`
- Again with the use of Lambda Expressions, I can make my code simple.
- `Console.WriteLine(employees.First(e => e.FirstName == "Jaliya").FirstName);`
- Func and get the first employee **which satisfies the condition on Func.**

Predicate<T>

Predicate<T> represents a method that defines a set of criteria and determines whether the specified object meets those criteria.

For this, let's consider [List<T>.Find](#) Method which accepts a Predicate.

```
employees.Find()  
Employee List<Employee>.Find(Predicate<Employee> match)  
Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire System.Collections.Generic.List<T>. match: The System.Predicate<T> delegate that defines the conditions of the element to search for.
```

- In here it's a Predicate of type Employee. So let's create a method which accepts a Employee and check whether he/she is born in "1986". If yes, it will return true or else false.
- `static bool BornInNineteenEightySix(Employee emp)`
- `{ return emp.Birthday.Year == 1986; }`
- Now I am creating a Predicate pointing to above method.
- `Predicate<Employee> predicate = new Predicate<Employee>(BornInNineteenEightySix);`
- `Console.WriteLine(employees.Find(predicate).FirstName);`
- Again with the use of Lambda Expressions, I can simplify the code.
`Console.WriteLine(employees.Find(e => e.Birthday.Year == 1986).FirstName);`

Func Vs. Predicate<T>

- Now you must be wondering what is the difference between Func and Predicate. Basically those are the same, but there is a one significant difference.
-
- Predicate can only be used point to methods which will return bool. If the pointing method returning something other than a bool value, you can't use predict. For that, you can use Func. Let's take a look at following method.
- `static string MyMethod(int i)`
- `{ return "You entered: " + i; }`
- The method accepts a integer value and returns a string. I can create the following Func and use it to call the above method.
- `Func<int, string> myFunc = new Func<int, string>(MyMethod);`
`Console.WriteLine(myFunc(3));`
- This will compile and print the desired output. But if you try to create a Predicate for this, you can't.

Questions?

