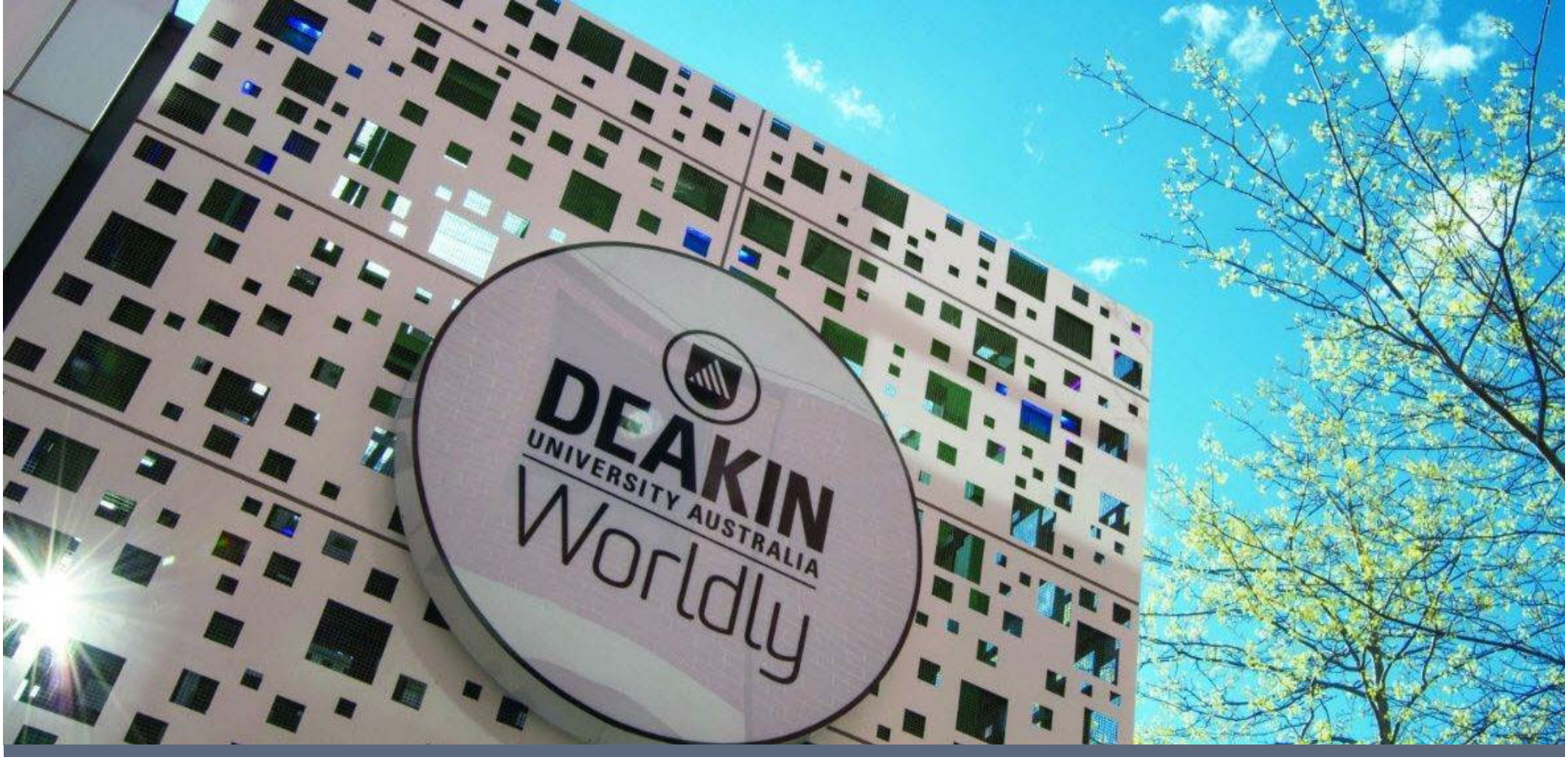


Lecture 08. Delegates and Callback Functions

SIT232 Object-Oriented Development

Delegates

- **Delegates** are similar to object references, except they reference a method instead.
- Using a delegate allows the programmer to encapsulate a reference to a method inside a **delegate object**.
- A delegate in C# is a concept very similar to a function pointer in C or C++.
- Most commonly used for event handlers for Windows GUI components.

Declaring a Delegate Data Type

Defining a delegate means telling the compiler what kind of method a delegate of that type will represent.

[access_modifier] **delegate** return_type delegate_name ([parameter[, ...]]);

- Must match the methods to be referenced
- Generally, one delegate type per method signature and return type

Example:

```
public delegate int PerformCalculation (int x, int y);
```

Create and Initialize a Delegate Variable

After a delegate type has been declared, a **delegate object** must be created and associated with a particular method.

[access_modifier] delegate_name variable_name;

You may then assign the method reference to the delegate variable

variable_name = method_name;

Example:

```
public PerformCalculation calculator = ProductValue;
```

- ProductValue must match the delegate's signature, i.e. be of the form int ProductValue (int x, int y);

Calling a Delegate

- After a delegate object is created, the delegate object is typically passed to other code that will call the delegate.
- A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate.

```
variable_name (int x, int y);
```

Example:

```
calculator(5,7);
```

- No different to a method call, just use the delegate variable name followed by parameters to invoke the related ProductValue(5,7) method.

Callback Functions: The Idea

Objects typically maintain state, which **changes** over time

store state →

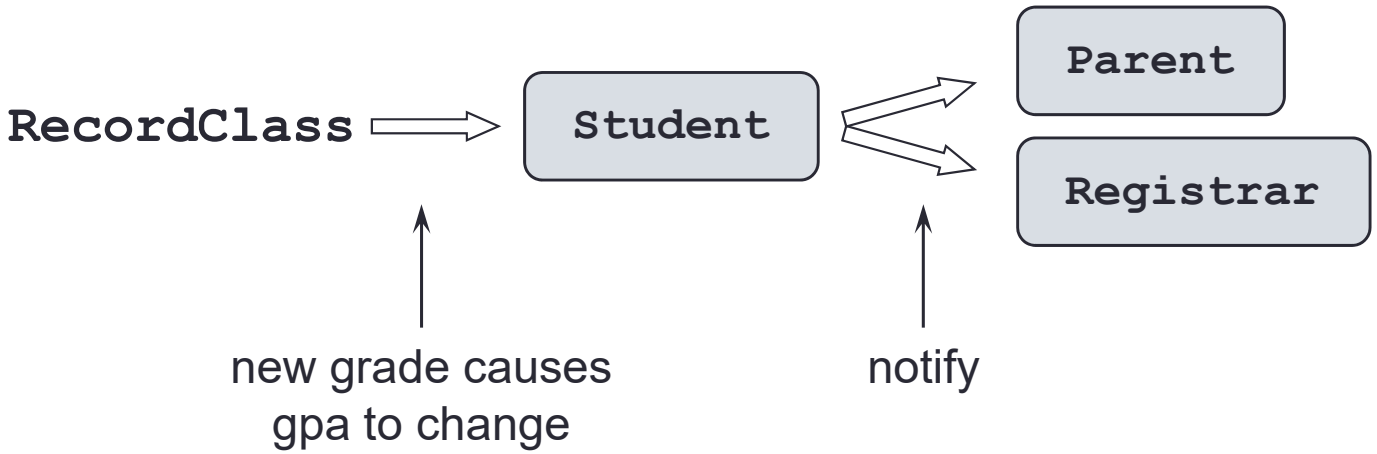
change state →

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

  public void RecordClass(int grade)
  {
    gpa = (gpa * units + grade) / (units + 1);

    units++;
  }
  ...
}
```

Callback Functions: Notification

Objects may want to notify interested parties of state change

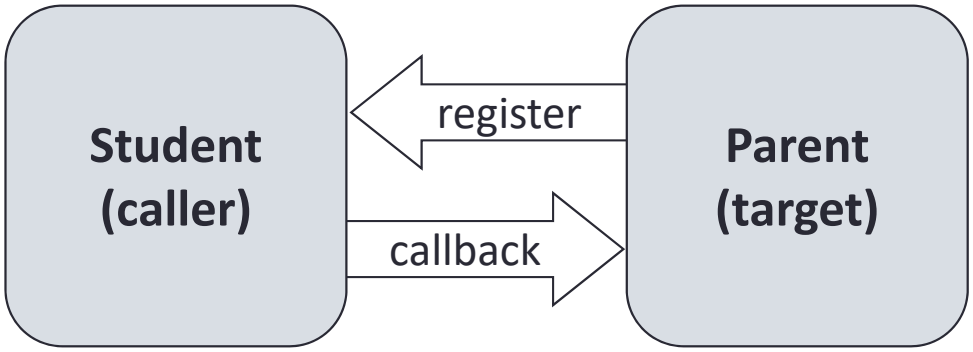
- notification widely used throughout .NET framework
- GUI event handling is the most common example



Callback Functions: The Design Pattern

Notification typically involves *registration* and *callback*

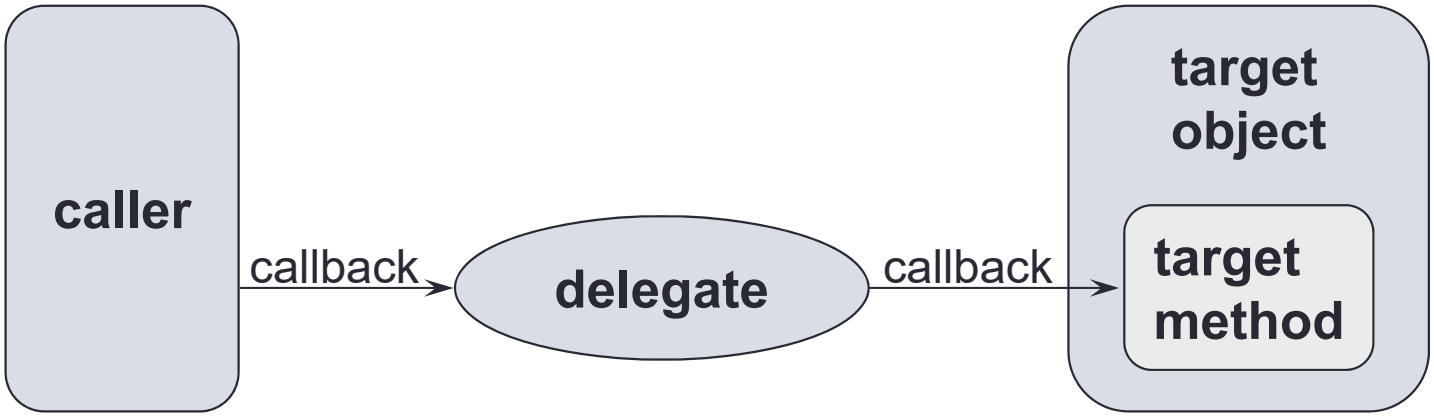
- target registers with caller
- caller calls back target when state changes
- this design pattern is also called as **publish/subscribe**



Callback Functions: Use of Delegates

.NET Framework uses delegates to implement callbacks

- intermediary between caller and target
- declaration defines callback method signature
- instance stores object reference and method token



Callback Functions: Setting Delegate

Delegate name is a type name

- can declare references
- can create objects

define delegate →

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

```
StudentCallback a = new StudentCallback(...);
```

reference object

Callback Functions: Preparing 'Target'

Target defines method with signature specified by delegate

– parameters and return type must match

delegate defines required signature →

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

target →

method signature matches delegate →

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

Callback Functions: Preparing 'Caller'

Caller typically defines delegate reference (a.k.a. delegate object)

caller →

delegate reference →

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

Callback Functions: Binding 'Target' and 'Caller'

Create delegate object and store in caller to register

- pass target object and method to delegate constructor

caller →
target →
create →
store →

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

  Parent  mom = new Parent();

  StudentCallback a = new StudentCallback(mom.Report);

  ann.GpaChanged = a;
  ...
}
```

target object
target method

Callback Functions: Invocation

Caller invokes callback indirectly through delegate

- uses method call syntax on delegate
- delegate calls target method on target object

```
class Student
{
  public StudentCallback GpaChanged;

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

invoke callback through delegate

callback takes **double** argument

Callback Functions: Summary

define delegate →

target method →

caller stores delegate →

caller invokes delegate →

```
delegate void StudentCallback(double gpa);

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

class Student
{
  public StudentCallback GpaChanged;

  public void RecordClass(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.RecordClass(4);
```

Callback Functions: Multiple targets

We can combine delegates using operator += or operator +

- creates **invocation list of delegates**
- all targets called when delegate invoked
- targets called in order added
- use of += ok even when left-hand-side is null

targets →

first →

second →

```
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);
...
```

Callback Functions: Multiple targets

We can remove delegate from invocation list

- use operator -= or operator -
- identity of target object/method determines which is removed

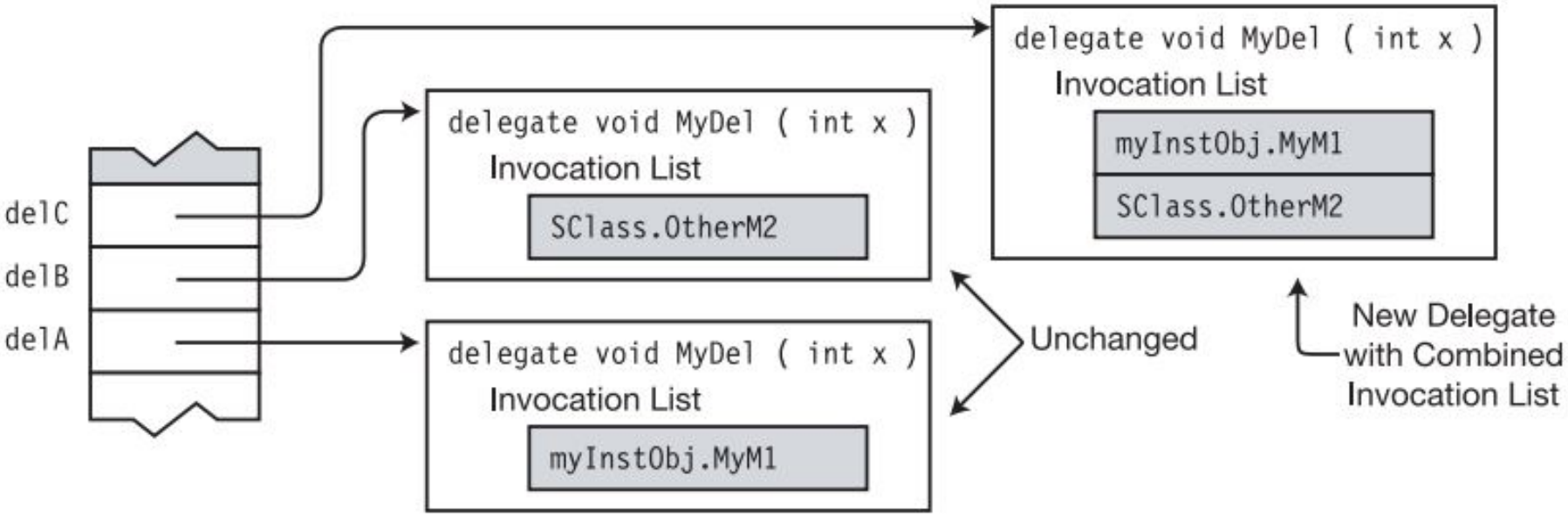
```
Parent mom = new Parent();
Parent dad = new Parent();

Student ann = new Student("Ann");

ann.GpaChanged += new StudentCallback(mom.Report);
ann.GpaChanged += new StudentCallback(dad.Report);
...
ann.GpaChanged -= new StudentCallback(dad.Report);
...
```

add → ann.GpaChanged += new StudentCallback(dad.Report);

remove → ann.GpaChanged -= new StudentCallback(dad.Report);

Callback Functions: Multiple targets

Illustration of the invocation list



Example

```
MyDel delA = myInstObj.MyM1;
MyDel delB = SClass.OtherM2;
MyDel delC = delA + delB;                    // Has combined invocation list
```