

Graphics

In the examples shown in class, there were two ways to get the form repainted:

- Write a handler for the Paint event and register for the event
- Override the OnPaint method that is inherited from the class Control

The second approach is the suggested one. It avoids creating a delegate and adding it to the handler. But your code should do this:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    ...
}
```

This guarantees that any registered delegates receive the event.

Graphics

Question: what do you do when you need to redraw the screen?

Answer: force a Paint event.

```
Invalidate();
Update();
```

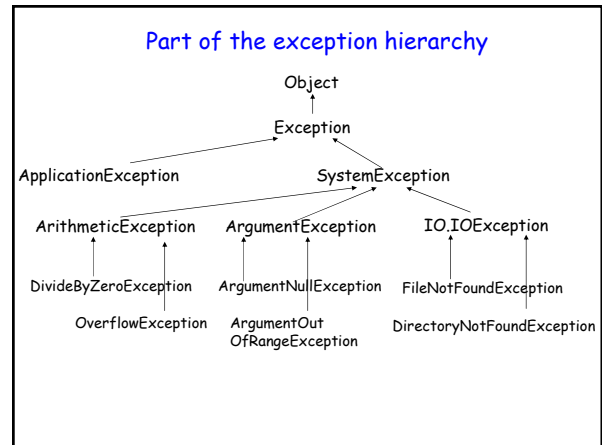
Causes an immediate call to OnPaint.

Exceptions

```
public void SomeMethod(...)
{
    File file = new File("Readme.txt");
    .
    .
    .
    file.Close();
}
```

What happens if the file doesn't exist?

What happens if an error occurs in here?



Exceptions: the general idea

```
try
{
    some code that might throw an exception
    more code
}
catch (most specific exception)
{
    handle the exception
}
catch (less specific exception)
{
    handle the exception
}
catch (any exception)
{
    handle the exception
}
finally
{
    do this no matter what
}
still more code
```

```
try
{
    some code that might throw an exception
    more code
}
catch (most specific exception)
{
    handle the exception
}
catch (less specific exception)
{
    handle the exception
}
catch (any exception)
{
    handle the exception
}
finally
{
    do this no matter what
}
still more code
```

No exception thrown

```
try
{
    some code that might throw an exception
    more code
}
catch (most specific exception)
{
    handle the exception
}
catch (less specific exception)
{
    handle the exception
}
catch (any exception)
{
    handle the exception
}
finally
{
    do this no matter what
}
still more code
```

This exception thrown

```
try
{
    some code that might throw an exception
    more code
}
catch (most specific exception)
{
    handle the exception
}
catch (less specific exception)
{
    handle the exception
    return;
}
catch (any exception)
{
    handle the exception
}
finally
{
    do this no matter what
}
still more code
```

This exception thrown

```
try
{
    some code that might throw an exception
    more code
}
catch (most specific exception)
{
    handle the exception
}
catch (less specific exception)
{
    handle the exception
    throw;
}
catch (any exception)
{
    handle the exception
}
finally
{
    do this no matter what
}
still more code
```

This exception thrown

```
using System.IO;
public void SomeMethod(...)
{
    File file = null;
    try
    {
        file = new File("Readme.txt");
        more code
    }
    catch (FileNotFoundException e)
    {
        Console.WriteLine("File " + e.FileName + " not found");
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
    }
    finally
    {
        if (file != null)
            file.Close();
    }
}
```

A property of System.IO.FileNotFoundException.

Exceptions

In a catch block, you can:

- Rethrow the same exception, notifying code higher in the call stack
- Throw a different exception, giving additional information to code higher in the call stack
- Handle the exception and fall out the bottom of the catch block

Throwing Exceptions

```
public void MyMethod(...)
{
    if (you can't do what is requested)
    {
        throw new InvalidOperationException("bad operation request");
    }
    ...
}
```

We instantiate the appropriate exception

Be sure to check the constructor details. All exceptions have a Message property that lets you attach useful information for debugging (this should not be shown to the end user of the application).

Exceptions

Remember that:

- Exceptions are not always "errors".
- Exceptions are not always infrequent.
- Exceptions indicate that a method was unable to complete it's task.
- Sometimes it's best not to catch an exception where it occurs. That may be the place where you are least likely to know what to do.
- There is a performance hit for exceptions.

Exceptions

```
public void SomeMethod(...)  
{  
    File file = null;  
    try  
    {  
        file = new File("Readme.txt");  
        more code  
    }  
    finally  
    {  
        if (file != null)  
            file.Close();  
    }  
}
```

Since there is no catch block, the exception is propagated to the caller

Exceptions: Similarities to Java

- **try** block can have one or more **catch** blocks
- **finally** block executes regardless of whether an exception occurs
- With multiple **catch** blocks, you must catch the most specific exception first
- A **try** block must have at least one **catch** block or a **finally** block
- All exception classes are subclasses of a main exception class: **System.Exception**

Exceptions: Differences from Java

- **C#** does not differentiate between checked and unchecked exceptions
- Java insists that checked exceptions be handled or thrown. In **C#**, no exception handling is necessary
- **C#** has no **throws** keyword. All exceptions are "unchecked" and are propagated if not handled locally
- When an exception is caught, it is not necessary to assign the exception object to a variable name, and the **throw** keyword by itself rethrows the same exception

```
catch (Exception)  
{  
    . . .  
    throw;  
}
```

Strings

What you already know:

String constants:

```
"hi there"  
"hi there\r\n"
```

Concatenation:

```
"hi" + " " + "there"
```

If one operand of + is a string, the other is converted:

```
"5" + 2    is    "52"  
5.0 + "5"  is    "55"
```

Strings

What you may not know:

String constants:

```
"C:\\Program Files\\BJ\\Form1.cs"  
@"C:\Program Files\BJ\Form1.cs"
```

Strings have indexers:

```
string s = "Hi";  
char ch = s[1];
```

Strings are immutable:

```
string s = "Hi";  
s[0] = 'F'; //won't compile
```

Strings

What you may not know:

String constants: Use \" to get \" in one of these

```
"C:\\Program Files\\BJ\\Form1.cs"  
@"C:\\Program Files\\BJ\\Form1.cs"
```

Use: Use "" to get "" in one of these

```
textBox1.Text = "Hi" + Environment.NewLine + "there";
```

rather than:

```
textBox1.Text = "Hi\\r\\nthere";
```

Strings

What you may not know:

Strings are immutable:

```
string s = "Hi";  
s += " there"; //orphans "Hi", g.c. reclaims  
Console.WriteLine(s + "!"); //allocates new string
```

Strings are reference types, but ==, !=, Equals work on string value

```
string s1 = "Hi";  
s1 += " there";  
string s2 = "Hi there";  
bool b1 = s1 == s2; //true  
bool b2 = s1.Equals(s2); //true  
bool b3 = Object.Equals(s1, s2); //true  
bool b4 = Object.ReferenceEquals(s1, s2); //false  
bool b5 = (object) s1 == s2; //false
```

Strings

- string is an alias for System.String
- < and > don't work for strings
- Some constructors:

```
public string(char[] value);  
public string(char[] value,  
              int startIndex, int length);  
public string(char c, int count);
```
- Or just

```
string str = "hello";
```

Strings

- Public field:

```
public static readonly string Empty
```
- Properties and indexers

```
Length
```

 is a public property returning the number of chars

```
[ ]
```

 is defined as a read-only indexer for strings

Strings

- Public methods (many overloads exist):

Compare	Remove
Concat	Replace
Copy	Substring
CopyTo	ToLower
Equals	ToUpper
EndsWith	Trim
StartsWith	PadLeft
IndexOf	PadRight
LastIndexOf	
Insert	

The Best Way to Compare Strings

```
public bool Equals (string value,  
                  StringComparison comparisonType  
                  )  
  
public static bool Equals (string a,  
                           string b,  
                           StringComparison comparisonType  
                           )  
  
public static int Compare (string strA,  
                           string strB,  
                           StringComparison comparisonType  
                           )
```

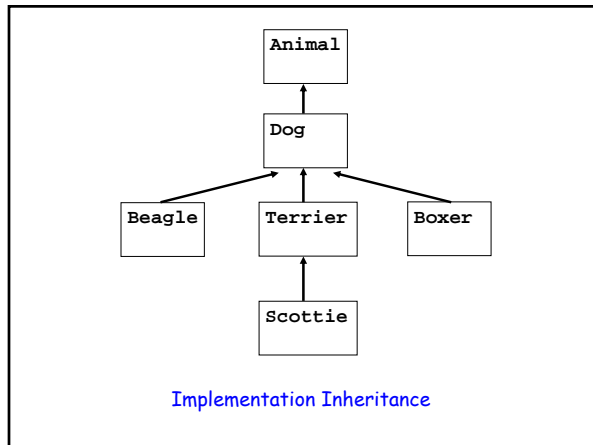
The Best Way to Compare Strings

```
public enum StringComparison {  
    CurrentCulture,  
    CurrentCultureIgnoreCase,  
    InvariantCulture,  
    InvariantCultureIgnoreCase,  
    Ordinal,  
    OrdinalIgnoreCase  
}
```

See:
New Recommendations for Using Strings in Microsoft .NET 2.0
<http://msdn2.microsoft.com/en-us/library/ms973919.aspx>

Inheritance

```
abstract class Dog  
{  
    public string name;  
    public abstract void Bark();  
    public virtual void RollOver()  
    {  
        . . .  
    }  
}  
  
public class Beagle : Dog  
{  
    public string color;  
  
    public override void Bark()  
    {  
        . . .  
    }  
}
```



Implementation Inheritance

- Code reuse
- Code maintenance
- Polymorphism

Implementation Inheritance

- Code reuse
- Code maintenance
- Polymorphism

```
ArrayList dogs = new ArrayList();  
. . .  
dogs.Add(myBeagle);  
dogs.Add(myScottie);  
dogs.Add(myBoxer);  
. . .  
foreach (Dog dog in dogs)  
    dog.Bark();
```

Polymorphism lets us use a superclass reference to invoke methods on subclass instances.

Problems with Implementation Inheritance

- Implementation inheritance is actually a form of *white-box reuse*
- The subclass may know some details of the superclass
- In particular, when you use the names of **protected** methods or properties in subclasses, you have created a layer of inflexible dependency: you cannot change the signature or remove them in the superclass without breaking the subclasses.
- This is an example of the *fragile superclass* scenario

Interface-based programming

- An interface is a contract—a promise to implement
- Interfaces may contain:
 - Methods
 - Properties
 - Indexers
 - Events
- A good reference:

[http://msdn2.microsoft.com/en-us/library/aa260635\(VS.60\).aspx](http://msdn2.microsoft.com/en-us/library/aa260635(VS.60).aspx)

Understanding Interfaced-based Programming, by Ted Pattison

Interface Inheritance

- Separation of interface and implementation
- *Black-box* reuse
- To do this, we need to make the interface a type of its own
- Since it contains no implementation, an interface does not create the tight coupling that we saw with implementation inheritance
- More than one class can implement an interface, creating the opportunity for polymorphism
- A class can implement several interfaces

```
public interface IDog
{
    string Name
    {
        get;
        set;
    }
    void Bark();
    void RollOver();
}

class Beagle : IDog
{
    private string name;
    public string Name
    {
        get {return name;}
        set {name = value;}
    }
    public void Bark()
    { . . . }
    public void RollOver()
    { . . . }
}

class Boxer : IDog
{
    private int name;
    public string Name
    {
        get {return name.ToString();}
        set { . . . }
    }
    public void Bark()
    { . . . }
    public void RollOver()
    { . . . }
}
```

```
static void Main()
{
    Boxer bowser = new Boxer(...);
    Beagle ralphie = new Beagle(...);
    List<IDog> dogs = new List<IDog>();

    dogs.Add(bowser);
    dogs.Add(ralphie);
    foreach (IDog dog in dogs)
        dog.Bark();
}
```

Tradeoffs and Comparisons

- Interfaces offer flexibility
- Interfaces reduce tight coupling
- Implementation inheritance provides an IS-A relationship (single inheritance)
- Interfaces provide a CAN-DO relationship
- All members of an interface must be implemented by each class that uses it
- Adding a member to an interface requires adding it to each class that implements the interface

Garbage Collection

The basic idea: memory in the managed heap that is no longer needed is automatically reclaimed. The programmer is not responsible for freeing it.

Object life cycle:

- Allocate memory
- Initialize the object to a useful state
- Use the object
- Tear the object down
- Free memory

Garbage Collection

The basic idea: memory in the managed heap that is no longer needed is automatically reclaimed. The programmer is not responsible for freeing it.

Object life cycle:

- Allocate memory
- Initialize the object to a useful state
- Use the object
- Tear the object down ← The garbage collector can't do this
- Free memory ← The garbage collector is solely responsible for this

C# "Destructors" : the Finalize method

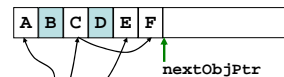
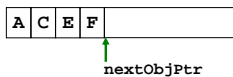
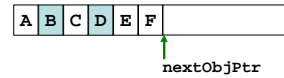
- `System.Object` has a protected virtual method `Finalize`
- Called when the garbage collector determines that the object is garbage but before the memory is reclaimed
- Types that require cleanup should override this method (e.g., to close files or network connections)
- In C#, the syntax is like the destructor syntax of C++:

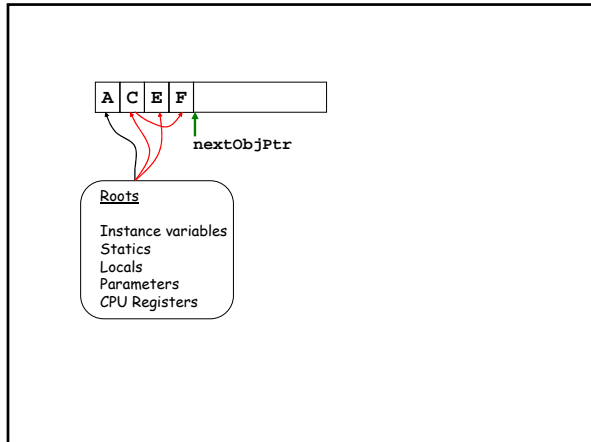
```
~MyClass()
{ //cleanup here }
```

- The finalization function is never explicitly called, and destruction of an object is non-deterministic

Use Finalization only when necessary

- Finalizable objects take longer to allocate
- Forcing the garbage collector to call a `Finalize` method can hurt performance. If there is an array of 10,000 objects, the finalization method gets called 10,000 times
- You have no control over when finalization takes place
- The CLR doesn't guarantee the order in which finalization methods are called





What if WE want to initiate cleanup?

```
class MyClass : IDisposable
{
    ~MyClass()
    {
        Dispose( );
    }

    protected void Dispose( )
    {
        ← Cleanup code goes here
    }
}
```

```
class MyClass : IDisposable
{
    ~MyClass()
    {
        Dispose( ); ← Call from finalizer
    }

    protected void Dispose( )
    {
        ← Cleanup code goes here
    }
}
```

We would also like clients to initiate cleanup, and we would like to know which calls are which.

```
class MyClass : IDisposable
{
    private bool disposed = false;
    ~MyClass()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
    }

    protected void Dispose(bool clientCall)
    {
        if (!disposed)
        {
            if(clientCall)
            {
                {
                }
                ...
            }
            disposed = true;
        }
    }
}
```

```
class MyClass : IDisposable
{
    private bool disposed = false;
    ~MyClass()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Dispose(true);
    }

    protected void Dispose(bool clientCall)
    {
        if (!disposed)
        {
            if(clientCall)
            {
                {
                }
                ...
            }
            disposed = true;
        }
    }
}
```

```

class MyClass : IDisposable
{
    private bool disposed = false;
    ~MyClass()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Dispose(true);
    }

    protected void Dispose(bool clientCall)
    {
        if (!disposed)
        {
            if(clientCall)
            { ← Here, the object is not garbage
            {
                ...
            }
            disposed = true;
        }
    }
}
    
```

```

class MyClass : IDisposable
{
    private bool disposed = false;
    ~MyClass()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Dispose(true);
    }

    protected void Dispose(bool clientCall)
    {
        if (!disposed)
        {
            if(clientCall)
            { ← Here, the object is not garbage
            {
                ... ← Here, it might be
            }
            disposed = true;
        }
    }
}
    
```

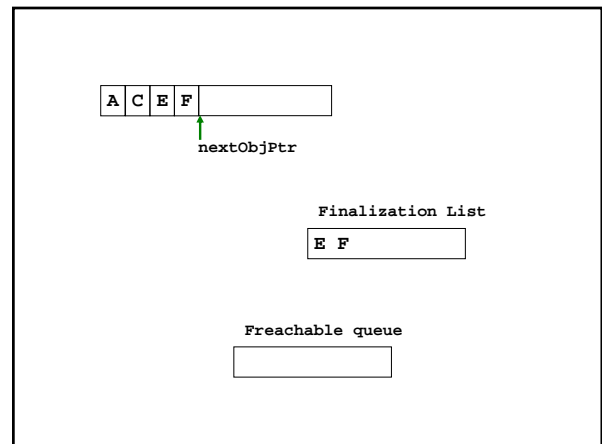
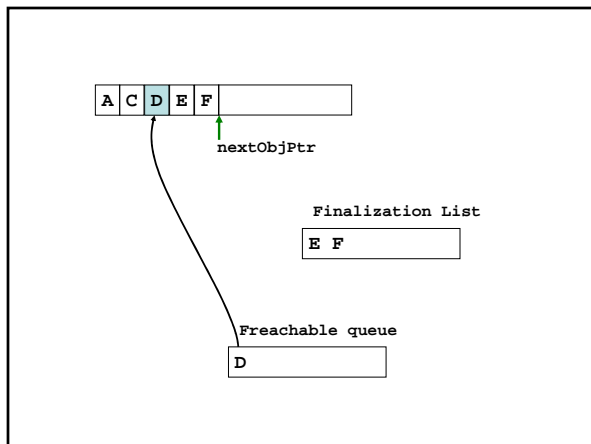
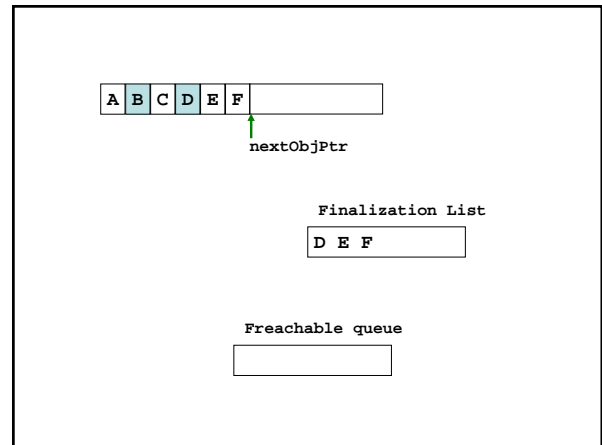
```

class MyClass : IDisposable
{
    private bool disposed = false;
    ~MyClass()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Dispose(true);
    }

    protected void Dispose(bool clientCall)
    {
        if (!disposed)
        {
            if(clientCall)
            {
                ...
            }
            disposed = true;
        }
    }
}
    
```

Dispose only deals with resources wrapped by the object.
 The garbage collector is still responsible for freeing the object's memory.



Parameter Passing in C#

Output parameters

```

void Foo(out int x, out int y)
{
    x = 2;
    y = 3;
}
    
```

Work like reference parameters except:

- Need not be assigned a value in the caller before the call
- Considered unassigned in the called method, and must be assigned before being used in an expression.
- Must be assigned before the called method returns.

```

{
    int a, b;
    Foo(out a, out b);
}
    
```

Parameter Passing in C#

Output parameters

```

static void GetRandomNumbers(int n, out int[] nums)
{
    Random r = new Random();
    nums = new int[n];
    for (int i = 0; i < n; i++)
        nums[i] = r.Next(1, 1000);
}

{
    int [] randNums;
    GetRandomNumbers(5, out randNums);
}
    
```

Parameter Passing in C#

Parameter arrays

```

public static void SetSum(out int sum, params int[] values)
{
    sum = 0;
    foreach (int n in values)
        sum += n;
}

{
    int [] randNums;
    int sum;
    GetRandomNumbers(5, out randNums);
    SetSum(out sum, randNums);
}
    
```

Parameter Passing in C#

Parameter arrays

```

public static void SetSum(out int sum, params int[] values)
{
    sum = 0;
    foreach (int n in values)
        sum += n;
}

{
    int [] randNums;
    int sum;
    GetRandomNumbers(5, out randNums);
    SetSum(out sum, randNums);
}
    
```

- Indicate with the keyword `params`
- Must be the last parameter
- Must be a single-dimensional array type

If the argument is an array, it acts just like a value parameter.

Parameter Passing in C#

Parameter arrays

```

public static void SetSum(out int sum, params int[] values)
{
    sum = 0;
    foreach (int n in values)
        sum += n;
}

{
    int [] randNums;
    int sum;
    GetRandomNumbers(5, out randNums);
    SetSum(out sum, randNums);
    SetSum(out sum, 1, 2, 3, 4);
}
    
```

- Indicate with the keyword `params`
- Must be the last parameter
- Must be a single-dimensional array type

If the argument is a list of values, an array of the appropriate length is created and passed as a value parameter.

An array of objects as 'params'

```

ShowTypes(1, 2.0, "Hi there", sum, randNums, new Foo());
    
```

```

public static void ShowTypes(params object[] obj)
{
    foreach (object o in obj)
    {
        Console.WriteLine(o.GetType());
        if (o is Foo)
            ((Foo) o).Hi();
    }
}
    
```

```

System.Int32
System.Double
System.String
System.Int32
System.Int32[]
Params.Foo
Hi from Foo
    
```