Kreiranje, inicijalizacija i uništavanje objekata

Programiranje korisničkih interfejsa

Bojan Furlan

Creating Objects

- **Step 1: Allocating memory**
  - Use **new** keyword to allocate memory from the heap
- **Step 2: Initializing the object by using a constructor**
  - Use the name of the class followed by parentheses

```
Date when = new Date( );
```

Using Initializer Lists

- **Overloaded constructors might contain duplicate code**
  - Refactor by making constructors call each other
  - Use the **this** keyword in an initializer list

```
class Date
{
    ...
    public Date( ) : this(1970, 1, 1) { }
    public Date(int year, int month, int day) { ... }
}
```

Declaring Readonly Variables and Constants

- Value of constant field is obtained at compile time

- Value of readonly field is obtained at run time

Keywords const & readonly

```
class MyClass {
    const int x = 5;
    readonly int y = 25;
    const int xx;
    readonly int yy; //ok
   readonly int zz = GetZ(); //ok
    public MyClass() {
       yy = 24;
    }
    public void MyMethod {
      yy = 10; x = 7;
    }
}
```

Objects and Memory

- Object Lifetime
- Objects and Scope
- Garbage Collection

Object Lifetime

- **Creating objects**
  - You allocate memory by using **new**
  - You initialize an object in that memory by using a constructor
- **Using objects**
  - You call methods
- **Destroying objects**
  - The object is converted back into memory
  - The memory is de-allocated

Objects and Scope

- **The lifetime of a local value is tied to the scope in which it is declared**
  - Short lifetime (typically)
  - Deterministic creation and destruction
- **The lifetime of a dynamic object is not tied to its scope**
  - A longer lifetime
  - A non-deterministic destruction

Garbage Collection

- **You cannot explicitly destroy objects**
  - C# does not have an opposite of **new** (such as **delete**)
  - This is because an explicit delete function is a prime source of errors in other languages
- **Garbage collection destroys objects for you**
  - It finds unreachable objects and destroys them for you
  - It finalizes them back to raw unused heap memory
  - It typically does this when memory becomes low

Resource Management

- Object Cleanup
- Writing Destructors
- Warnings About Destructor Timing
- IDisposable Interface and Dispose Method
- The using Statement in C#

Object Cleanup

- **The final actions of different objects will be different**
  - They cannot be determined by garbage collection.
  - Objects in .NET Framework have a **Finalize** method.
  - If present, garbage collection will call destructor before reclaiming the raw memory.
  - In C#, implement a destructor to write cleanup code. You cannot call or override **Object.Finalize**.

Writing Destructors

- **A destructor is the mechanism for cleanup**
  - It has its own syntax:
    - No access modifier
    - No return type, not even **void**
    - Same name as name of class with leading ~
    - No parameters

```
class SourceFile
{
    ~SourceFile( ) { ... }
}
```

Warnings About Destructor Timing

- **Destructors are guaranteed to be called**
  - Cannot rely on timing
- **Avoid destructors if possible**
  - Performance costs
  - Complexity
  - Delay of memory resource release
- **Use them for unmanaged non-memory resources**
  - E.g. DB connections, file lock, sessions etc.

IDisposable Interface and Dispose Method

- **To reclaim a resource:**
  - Inherit from **IDisposable** Interface and implement **Dispose** method that releases resources
  - Ensure that calling **Dispose** more than once is benign
  - Ensure that you do not try to use a reclaimed resource
  - Call **GC.SuppressFinalize** method
    - Requests that the system not call the finalizer for the specified object.

The using Statement in C#

- **Syntax**

```
using (Resource r1 = new Resource( ))
{
      r1.Method( );
}
```

- **Dispose is automatically called at the end of the using block**

Dispose design example

```
public class BaseResource: IDisposable
{
   // Pointer to an external resource
   private IntPtr handle;
   // Other resource this class uses
   private Component Components;
   // To track whether Dispose has been called
   private bool disposed = false;
    // Constructor for the BaseResource object
   public BaseResource( )
   {
      handle = // Insert code here to allocate on the
                              // unmanaged side
      Components = new Component (...);
   }
...
```

Dispose design example

```
// Implement IDisposable.
   // Do not make this method virtual.
   // A derived class should not be able to override
   // this method.
   public void Dispose( )
   {
      Dispose(true);
      // Take yourself off of the Finalization queue
      GC.SuppressFinalize(this);
   }
```

Dispose design example

```
protected virtual void Dispose(bool disposing)
{
   // Check to see if Dispose has already been
   // called
   if(!this.disposed)
   {
      // If this is a call to Dispose, dispose all
    // managed resources
      if(disposing)
      {
         Components.Dispose( );
      }
…
```

Dispose design example

```
        // Release unmanaged resources.
        // Note that this is not thread-safe.
        // Another thread could start disposing the
       // object after the managed resources are
        // disposed, but before the disposed flag is
        // set to true.
        this.disposed = true;
        Release(handle);
        handle = IntPtr.Zero;
    }
}
```

Dispose design example

```
// Use C# destructor syntax for finalization code.
// This destructor will run only if the Dispose
// method does not get called. It gives your base
// class the opportunity to finalize. Do not
// provide destructors in types derived from
// this class.
~BaseResource( )
{
   Dispose(false);
}

…
```

Dispose design example

```
// Design pattern for a derived class.
// Note that this derived class inherently implements
// the IDisposable interface because it is implemented
// in the base class.
public class MyResourceWrapper: BaseResource
{
   private bool disposed = false;

   public MyResourceWrapper( )
   {
      // Constructor for this object
   }
```

Dispose design example

```
protected override void Dispose(bool disposing)
{
   if(!this.disposed)
   {
      try
      {
         if(disposing)
         {
            // Release any managed resources here
         }
         // Release any unmanaged resources here
         this.disposed = true;
      }
   …
```

Dispose design example

```
…
finally
        {
          // Call Dispose on your base class
          base.Dispose(disposing);
        }
     }
  }
}

// This derived class does not have a Finalize method
// or a Dispose method with parameters because it
// inherits them from the base class
```