Properties and Basic Generics

January 30, 2008

Recall last lecture:

```
// Create a form (i.e. a window)
Form theForm = new Form();

// Set the title
theForm.Text = "My Window";
```

Recall last lecture:

```
// Create a form (i.e. a window)
Form theForm = new Form();

// Set the title
theForm.Text = "My Window";
```

Q. Is theForm.Text really a member?

Recall last lecture:

```
// Create a form (i.e. a window)
Form theForm = new Form();

// Set the title
theForm.Text = "My Window";
```

Q. Is theForm.Text really a member?
A. No. theForm.Text is a *property*.

Properties provide special syntax for methods.

- A property consists of two methods: get and set.
- Clients call set with assignment notation
  e.g. theForm.Text = "My␣Window";
- Clients call get with member read notation
  e.g. WriteLine(theForm.Text)
- Each property access runs a method.

Property Example (I/III)

```
public class Temperature{
  private double myKelvin;

  public double Kelvin{
    get{
      //Think  "public double get()"
      return myKelvin;
    }
    set{
      //Think  "public void set(double value)"
      myKelvin = value;
    }
  }
  ...
```

Property Example (II/III)

```
    ...
    public double Fahrenheit{
      get{
          return myKelvin*(9.0/5.0) - 459.67;
      }
      set{
         myKelvin = (5.0/9.0)*(value + 459.67);
      }
    }
}
```

Property Example (III/III)

```
public class Runner{
  public static void Main(string[] args)
  {
    Temperature Temp = new Temperature();
    Temp.Fahrenheit = 32.0;
    Console.Out.WriteLine(Temp.Kelvin);
  }
}
```

Output: 273.15 (that's the right answer)

C# 3.0 has special syntax for declaring simple properties.

```
public class Temperature{

  // Compiler automatically generates private
  // member, getter, and setter
  public double Kelvin { get; set; }

  public double Fahrenheit{
    get{
      return Kelvin*(9.0/5.0) - 459.67;
    }
    set{
     Kelvin = (5.0/9.0)*(value + 459.67);
    }
  }
}
```

When should I use a . . .

- . . . public member?
- . . . property?
- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?

- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?
  - The getter has no (observable) side effects.
  - The getter does not throw exceptions.
  - Both get and set return almost immediately (no long computations or database queries)
- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?
  - The getter has no (observable) side effects.
  - The getter does not throw exceptions.
  - Both get and set return almost immediately (no long computations or database queries)
- . . . method? Any other time.

Access limited properties.

```
public class Misc {
  int myNumber;

  // A property with a private getter.  Only
  // members of Misc can read .DropBox
  public int DropBox {
    set{
      myNumber = value;
    }
    private get{
      return myNumber;
    }
  }

  public int PrivateSet { get; private set; }
}
```

Read-only and write-only properties

```
public class GetSetOnly{
  private int myX, myY;

  // A read-only property: a common pattern
  public int X { get { return myX; } }

  // Write only patterns are considered bad style
  public int Y { set { myY = value; } }
}
```

Technical notes about properties

- Properties compile to method calls, not member access
- So properties can't implement members in interfaces
- Properties are optimized to be roughly as fast as member access

1 Properties

2 Basic Generics

Generics allow types (e.g. classes and delegates) to be parameterized by types.

- Provide extra compile-time type information
- Provide opportunities for compiler optimizations.
- Allow the compiler to catch bugs that would otherwise happen at runtime.
- Enhance code readability.
- Reduce need for downcasts (which are expensive and can throw exceptions).

Example: a specialized “option” class.

```
public class IntOption{
  private bool isFull; private int contents;

  public bool Empty { get {return !isFull;} }

  public int GetValue() {
    if (isFull) return contents;

    throw new Exception("GetValue of Empty");
  }

  public IntOption() { isFull = false; }

  public IntOption(int x){
      isFull = true; contents = x; }
}
```

Example: Using the specialized option class.

```
public class Runner{
  public static IntOption div(int x, int y){
    if (y==0)
        return new IntOption();
    else
        return new IntOption(x / y);
  }

  public static void Main(string[] args)
  {
    Console.Out.WriteLine(div(3,4).Empty); //false
    Console.Out.WriteLine(div(3,0).Empty); //true
  }
}
```

Example: A generic option class.

```
public class GenOption<T>{
  private bool isFull; private T contents;

  public bool Empty { get {return !isFull;} }

  public T GetValue() {
    if (isFull) return contents;

    throw new Exception("GetValue␣of␣Empty");
  }

  public GenOption() { isFull = false; }

  public GenOption(T x){
      isFull = true; contents = x; }
}
```

Example: Using the generic option class.

```
public class Runner{
  public static GenOption<int> div(int x, int y){
    if (y==0)
        return new GenOption<int>();
    else
        return new GenOption<int>(x / y);
  }

  public static void Main(string[] args)
  {
    Console.Out.WriteLine(div(3,4).Empty); //false
    Console.Out.WriteLine(div(3,0).Empty); //true
  }
}
```

Generics vs. Generics vs. Templates

- C++ Templates
  - Template expansion is static: each template instantiation creates a new compile-time class.
  - Templates can't live in compiled libraries—only headers.
  - Templates expansion = Turing complete programming language(!)
- Java Generics
  - Similar semantics to C#
  - Implemented by type erasure; no runtime support in JVM
  - Poor support for reflection
  - Legacy code can break apprent type guarantees for generic objects.
- C#Generics
  - CLR (.Net virtual machine) has support for generics
  - Generics can be specialized to used native types at runtime
  - Type parameters preserved at runtime, and can be queried by reflection.