Properties

(modified)

developmentor

Public data

- **Public data can be both good and bad**
  - good because it allows clear and simple access syntax
  - bad because it exposes implementation details of class

public field →

```
class Person
{
  public string name;
  ...
}
```

```
Person p = new Person();

p.name = "Bob";

string n = p.name;
```

write →

read →

Private data

- **Class typically has private fields and public access methods**
  - clients use methods to access fields

```
class Person
{
  private string name;

  public string GetName()
  {
    return name;
  }

  public void SetName(string newName)
  {
    name = newName;
  }
}
```

private field →

public read method →

public write method →

Advantage: access control

- **Private data lets class offer only desired access methods**
  - read-only
  - write-only
  - read and write

```
class Person
{
  private string name;

  public string GetName()
  {
    return name;
  }
}
```

supply only read method for read-only access →

Advantage: data validation

- **Private data allows class to perform data validation**
  - methods can do error checking
  - clients can not avoid validation code by accessing data directly

```
class Person
{
  public void SetName(string newName)
  {
    if (newName == "")
      // error ...

    name = newName;
  }
  ...
}
```

error checking →

Disadvantage: access method names

- **Typical to prefix access method names with `Get` and `Set`**
  - follows common wisdom to make method names verb phrases
  - but conflicts with concept of field as representing data property

```
Person p = new Person();

p.SetName("Bob");

string n = p.GetName();
```

write →

read →

Disadvantage: access method usage

- **Syntax of access methods can be undesirable**
  - set does not use = so is not obviously a write operation
  - get typically has awkward looking empty parentheses

no assignment operator →
empty parentheses →

```
Person p = new Person();
p.SetName("Bob");
string n = p.GetName();
```


Properties

- **Properties combine the advantages of public and private data**
  - have clean access syntax like public fields
  - provide error checking opportunity like private fields

`Person` has `Name` property →
write →
read →

```
Person p = new Person();
p.Name = "Ann";
string n = p.Name;
```


Structure of property definition

- **Property definition has**
  - declaration of access level, type, and name
  - code block with implementation

declaration of `Name` property →
implementation →

```
class Person
{
  public string Name
  {
    ...
  }
  ...
}
```


Properties storage

- **Property does not provide any needed data storage**
  - typically define field to store data

field to store data →

```
class Person
{
  private string name;

  public string Name
  {
    ...
  }
  ...
}
```


Property accessors

- **Property can define `set` and `get` accessors**
  - definitions go inside implementation code block
  - no explicit arguments or return types

write method →
read method →

```
class Person
{
  public string Name
  {
    set
    {
      ...
    }

    get
    {
      ...
    }
  }
  ...
}
```


Set implementation

- **Set typically records new data for property**
  - new data passed in hidden argument called `value`

use `value` argument →

```
class Person
{
  private string name;

  public string Name
  {
    set
    {
      name = value;
    }
    ...
  }
  ...
}
```

calls set →

```
Person p = new Person();
p.Name = "Ann";
```

Get implementation

- **Get typically returns current value of property**
  - uses **`return`** to return value

return value of property →

```
class Person
{
  private string name;

  public string Name
  {
    get
    {
      return name;
    }
    ...
  }
  ...
}
```

calls get →

```
Person p = new Person();
...
string n = p.Name;
```


Get implementation

- **Shortcut – if no special get, set logic needed.**

```
class Person
{

  public string Name
  {
    get; set;
  }
  ...
}
```


Validation

- **Property accessors can contain validation code**

validation →

```
class Person
{
  private string name;

  public string Name
  {
    set
    {
      if (value == "")
        // error ...

      name = value;
    }
    ...
  }
  ...
}
```


Access control

- **Property may provide any access desired**
  - read-only
  - write-only
  - read and write

`Age` property →

supply **`get`** and omit **`set`** for read-only →

```
class Person
{
  public int Age
  {
    get
    {
      ...
    }
  }
  ...
}
```


Data storage

- **Property does not need to be backed by matching field**
  - value may be produced in any way that is appropriate

store date of birth →

calculate age →

birthday occurred yet this year? →

```
class Person
{
  private DateTime dob;

  public int Age
  {
    get
    {
      int age = DateTime.Today.Year - dob.Year;

      DateTime birthday = dob.AddYears(age);

      if (birthday > DateTime.Today)
        age--;

      return age;
    }
  }
  ...
}
```


Static property

- **Property can be static**
  - must be accessed through type name

define static property →

```
class Item
{
  private static int revenue;

  public static int Revenue
  {
    get { return revenue;  }
    set { revenue = value; }
  }
  ...
}
```

access using type name →

```
Item.Revenue = 5;
```

Access level

- **Property accessors have same access level**
  - both **`public`**, both **`private`**, etc.