SIT771 – Lecture 2

Objects and classes

Further reading

- Paul Deitel and Harvey Deitel (2018). Visual C# how to Program (6th ed). Pearson. Ebook on Deakin Library – Chapter 4.

Outline

In this lecture...

- Objects
- Classes
- Abstraction (OOP Concept 1/4)
- Encapsulation (OOP Concept 2/4)
- Design principles
    - (Intra-class) Cohesion
    - (Inter-class) Coupling

OBJECTS

What are objects?

Real-world view

- Real-world problems contain a number of entities (similar or different) that interact with each other in specific ways.
- Objects provide a way to model real-world entities in your program.
  - Objects have a **state**: attributes and values
  - Objects have **behaviors**: methods
  - Object states and behaviors remain coupled
  - Objects interact with each other
  - Objects may contain other objects within them (more later!)

Robot
- name : string - model : string - age : int
+ GetName() : string + SetName(name : string) + GetTheJobDone() : boolean

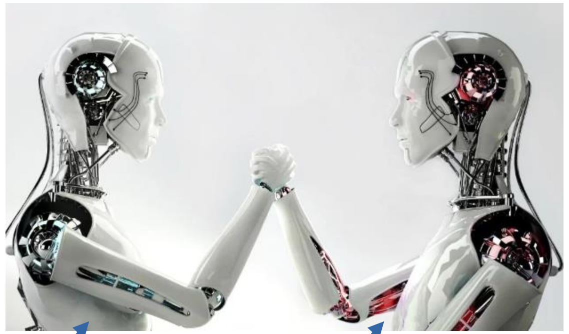


Image source: https://www.airedalesprings.co.uk/why-do-the-japanese-love-robots/

Gen2, modelY20, 4 Gen1, modelX21, 12

made up names/models not from the source

What are objects?

OOP view

- Objects…
  - **instantiate** entities from the problem domain
  - **encapsulate** states and behaviors
- Across some objects (e.g., two pandas, two robots)…
  - members (fields, properties, and methods) are the same
  - the behavior may slightly vary depending on the state, but is very similar
- A classification of objects can group them into separate **classes**

CLASSES

What are classes?

Object templates with states and behaviors

- Classes are created through…
  - finding commonalities across individual objects,
  - defining the state that the class instances (objects) **encapsulate**, and
  - defining the behaviors that the objects **exhibit**

- The class is then used to **instantiate objects** from it at run-time
- State information may be...
  - a single copy shared by all objects of a class [*what is this called in OOP?*]
  - created/assigned when the object is instantiated

Class: Planet

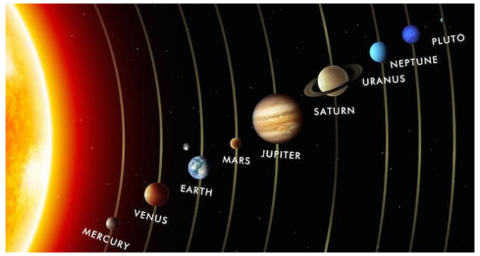


Image source: https://www.darksky.org/5-planets-align-in-celestial-treat/

Class: Robot

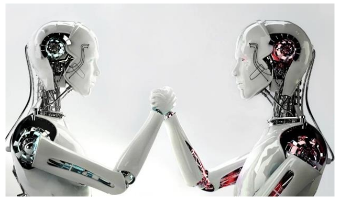


Image source: https://www.airedalesprings.co.uk/why-do-the-japanese-love-robots/

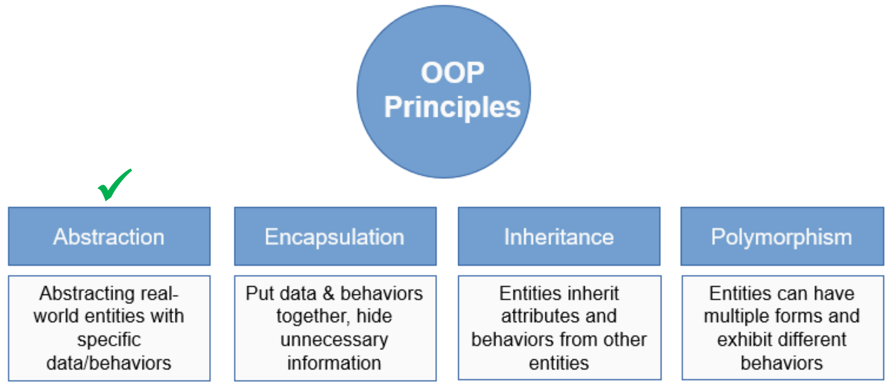
Image source: https://blogs.3ds.com/northamerica/future-robots-and-ensuring-human-safety/

State information in classes

Unique vs. shared

- Which of the following is the best option for the panda’s state information?
  - Name, species, type, and favorite food are shared.
  - Species and type are shared, each object separately records name and favorite food.
  - Each object separately records name, species, type, and favorite food.

OOP CONCEPT 1/4 … abstraction


OOP Principles
✓
Abstraction
Abstracting real-world entities with specific data/behaviors
Encapsulation
Put data & behaviors together, hide unnecessary information
Inheritance
Entities inherit attributes and behaviors from other entities
Polymorphism
Entities can have multiple forms and exhibit different behaviors

What is abstraction?

Representing/modelling complex real-world entities

- Abstraction in OOP serves two main purposes of programming:
  1. Creates a public-facing conceptual **representation** of real-world entities or objects
  2. Removes sharing of unnecessary **complexity** with other classes through...
     - a contextual, well-defined, and general definition of entities
     - coarse-grained public methods
     - public properties (or fields)

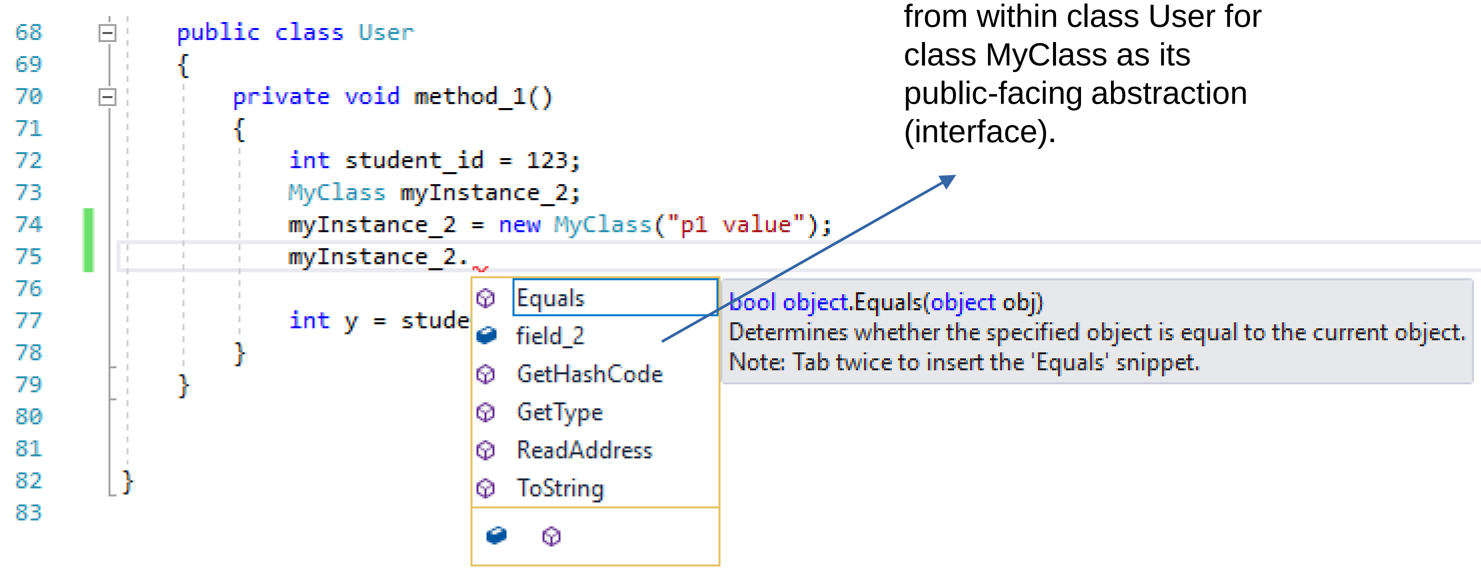
Remove complexity and create an abstract, brief view of a complex entity.

Source of images: Pinterest.com

What is abstraction?

Reduction into a simple model

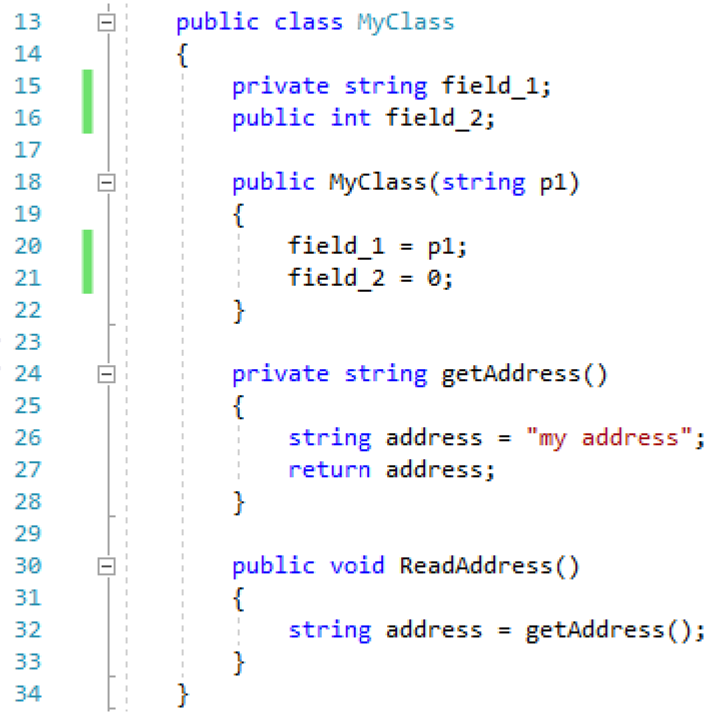
- Restricts what other classes need to know or can access from within the current class
- Note that abstraction differs from the following concepts in OOP:
  - abstraction is **<u>not</u>** the same as an abstract class (TBD)
  - abstraction is **<u>not</u>** the same as an abstract method (TBD)



```
public class User
{
    private void method_1()
    {
        int student_id = 123;
        MyClass myInstance_2;
        myInstance_2 = new MyClass("p1 value");
        myInstance_2.

        int y = stude
    }
}

}
```

```
bool object.Equals(object obj)
Determines whether the specified object is equal to the current object.
Note: Tab twice to insert the 'Equals' snippet.
```

This is all that can be seen from within class User for class MyClass as its public-facing abstraction (interface).



```
public class MyClass
{
    private string field_1;
    public int field_2;

    public MyClass(string p1)
    {
        field_1 = p1;
        field_2 = 0;
    }

    private string getAddress()
    {
        string address = "my address";
        return address;
    }

    public void ReadAddress()
    {
        string address = getAddress();
    }
}
```

Levels of abstraction

Creating different abstractions as needed

- The same real-world entity may have different abstractions or conceptualizations in your program based on the needs. This is part of a software designer's responsibilities to decide on.
- Abstraction, therefore, can have different levels or representations, e.g., ...
  - Classes Car, Mechanic, and Driver with two abstraction levels
    - Driver-level: does not need to know e.g., how brakes work
    - Mechanic-level: needs to know more, e.g., how the engine works

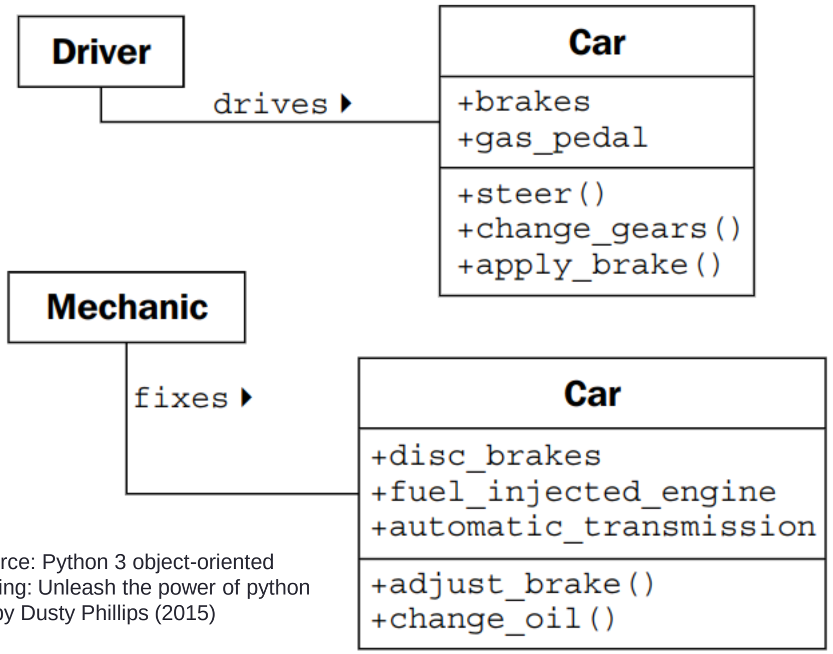


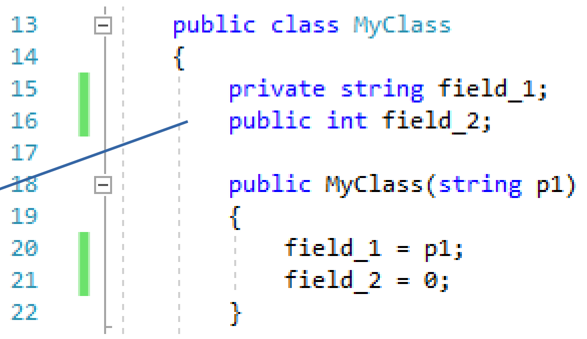
Image source: Python 3 object-oriented programming: Unleash the power of python 3 objects by Dusty Phillips (2015)

Abstraction implementation

Mechanism 1

- Access modifiers
    - **Public:** Accessible from outside the class. If a variable is public, you can set/read its value directly.
    - **Internal:** Accessible only within the same assembly (.dll or .exe)
    - **Private (by default for class members):** Not accessible (not even for reads) from outside of that class.
    - **Protected:** Accessible to methods of the class and its sub-classes (examined with **inheritance**).

```
[access_modifier] class class_name
{
        [access_modifier] class_member;
        ...
}
```



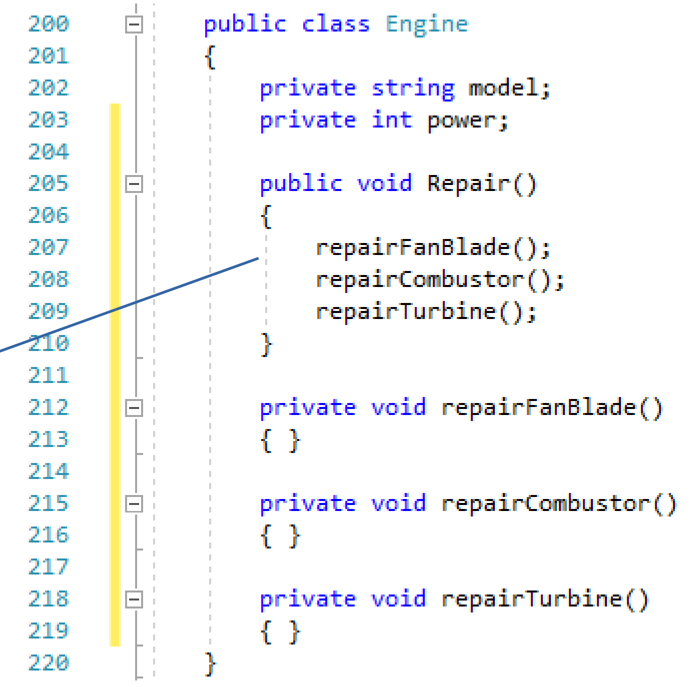
```
public class MyClass
{
    private string field_1;
    public int field_2;

    public MyClass(string p1)
    {
        field_1 = p1;
        field_2 = 0;
    }
```

Other classes will not see field_1.

Abstraction implementation

Mechanism 2

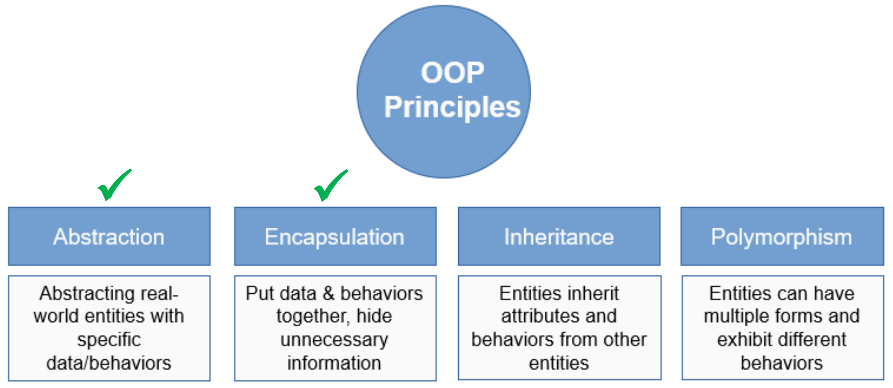
- Private detailed methods, e.g.,
  - Have some abstract and coarse-grained public-facing methods
  - Call some less abstract and more detailed private methods within the above methods



```
public class Engine
{
    private string model;
    private int power;

    public void Repair()
    {
        repairFanBlade();
        repairCombustor();
        repairTurbine();
    }

    private void repairFanBlade()
    { }

    private void repairCombustor()
    { }

    private void repairTurbine()
    { }
}
```

Repair is the public and more abstract functionality that relies on more detailed private methods inside the class.

OOP CONCEPT 2/4 … encapsulation


OOP Principles
✓
✓
Abstraction
Encapsulation
Inheritance
Polymorphism
Abstracting real-world entities with specific data/behaviors
Put data & behaviors together, hide unnecessary information
Entities inherit attributes and behaviors from other entities
Entities can have multiple forms and exhibit different behaviors

What is encapsulation?

Bundling states and behaviors

- Encapsulation is a powerful mechanism…
    - which means that an object is a **capsule** in which you find...
        - all data (fields) for the same object, and
        - all behaviors (methods) that operate on the data relevant to the same entity where the implementations remain hidden in the class
    - which ensures object's data are accessible to others through accessors/getters and mutators/setters
    - which is sometimes called **information hiding**, but it is broader than information hiding

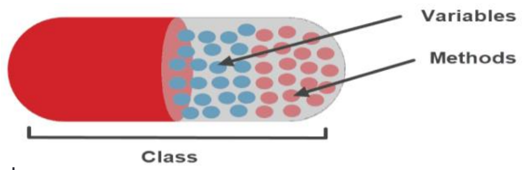
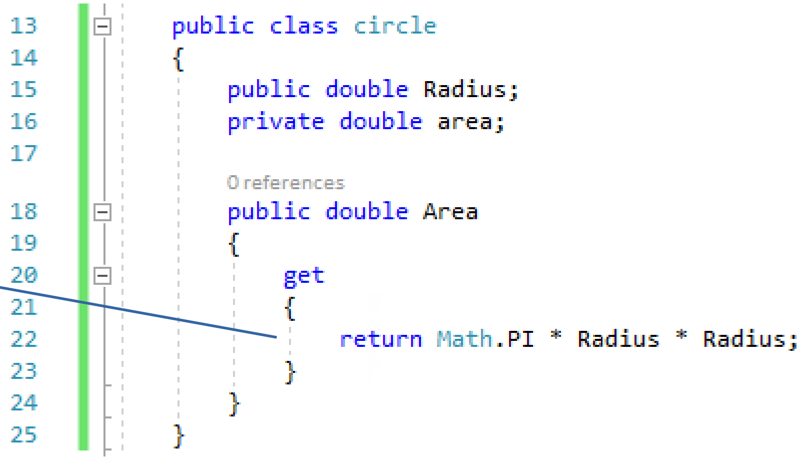


Image source: https://phoenixnap.com/kb/object-oriented-database

Barista does not need to know how the coffee maker makes the coffee. He just needs to know which buttons to push.

Image source: http://www.taliebrard.com/kids-magazine/article-illustrations/

Other classes will not know how the area of a circle object is calculated. They will not have access to "area" directly either but through the function that calculates Area.

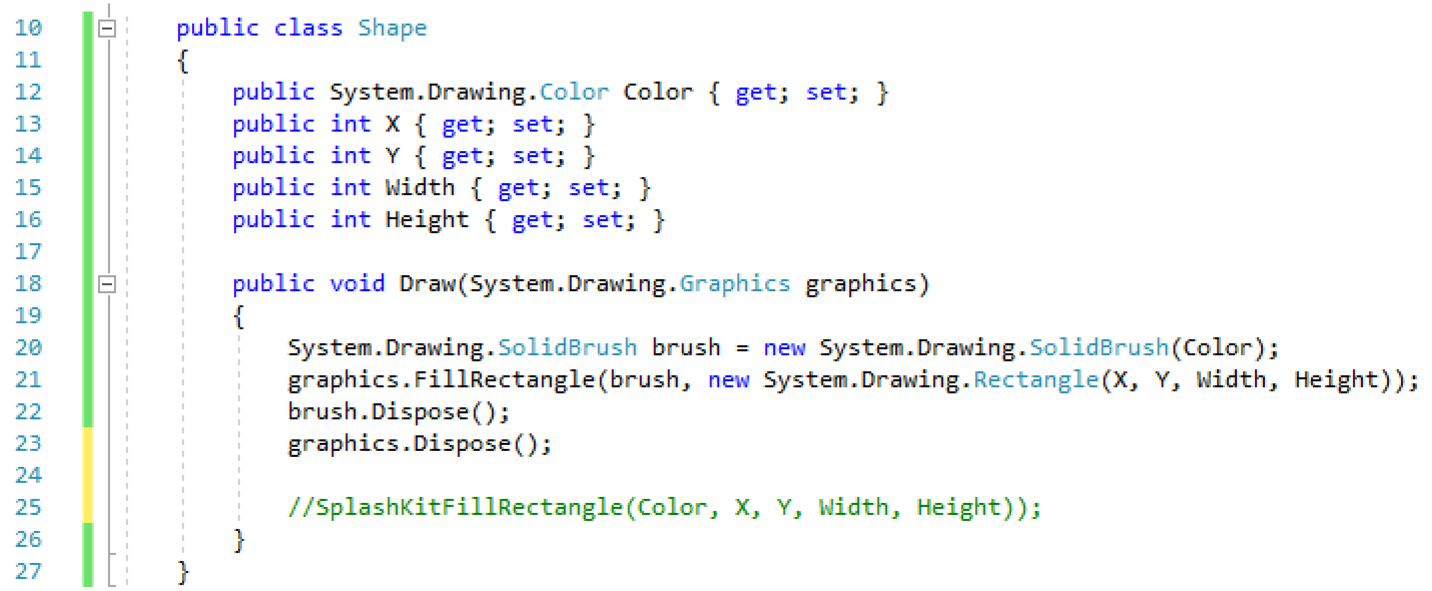


```
public class circle
{
    public double Radius;
    private double area;

    0 references
    public double Area
    {
        get
        {
            return Math.PI * Radius * Radius;
        }
    }
}
```

Encapsulation implementation

Mechanism 1

- Defining fields and methods
  - find and implement all necessary fields to represent an object’s state
  - find and implement all necessary methods that operate on the fields
  - bundle the above into what is called a class!

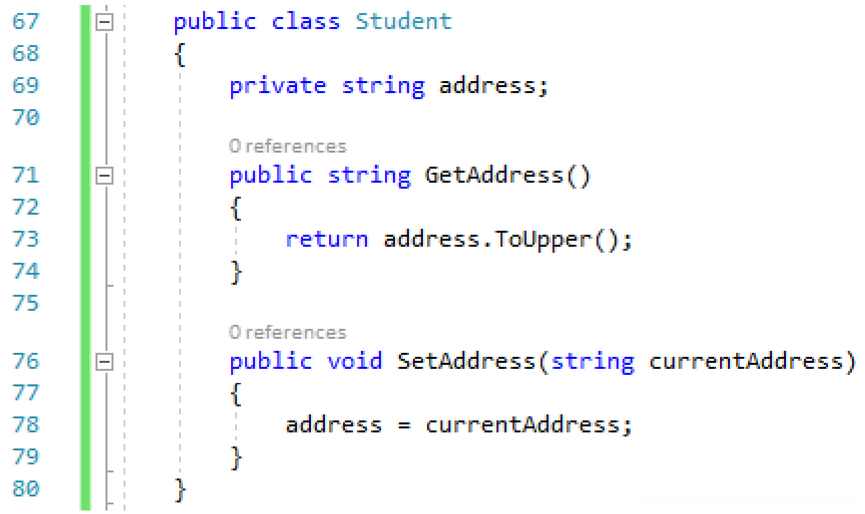


```
public class Shape
{
    public System.Drawing.Color Color { get; set; }
    public int X { get; set; }
    public int Y { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }

    public void Draw(System.Drawing.Graphics graphics)
    {
        System.Drawing.SolidBrush brush = new System.Drawing.SolidBrush(Color);
        graphics.FillRectangle(brush, new System.Drawing.Rectangle(X, Y, Width, Height));
        brush.Dispose();
        graphics.Dispose();

        //SplashKitFillRectangle(Color, X, Y, Width, Height));
    }
}
```

Encapsulation implementation

Mechanism 2

- Accessor and mutator methods
  - provide a public interface to private fields
  - define and implement data manipulation within the interface (logic hidden from outside)
  - Accesor
    - is usually prefixed with a **get**
    - is used for reading data
  - Mutator
    - is usually prefixed with a **set**
    - is used for storing or modifying data



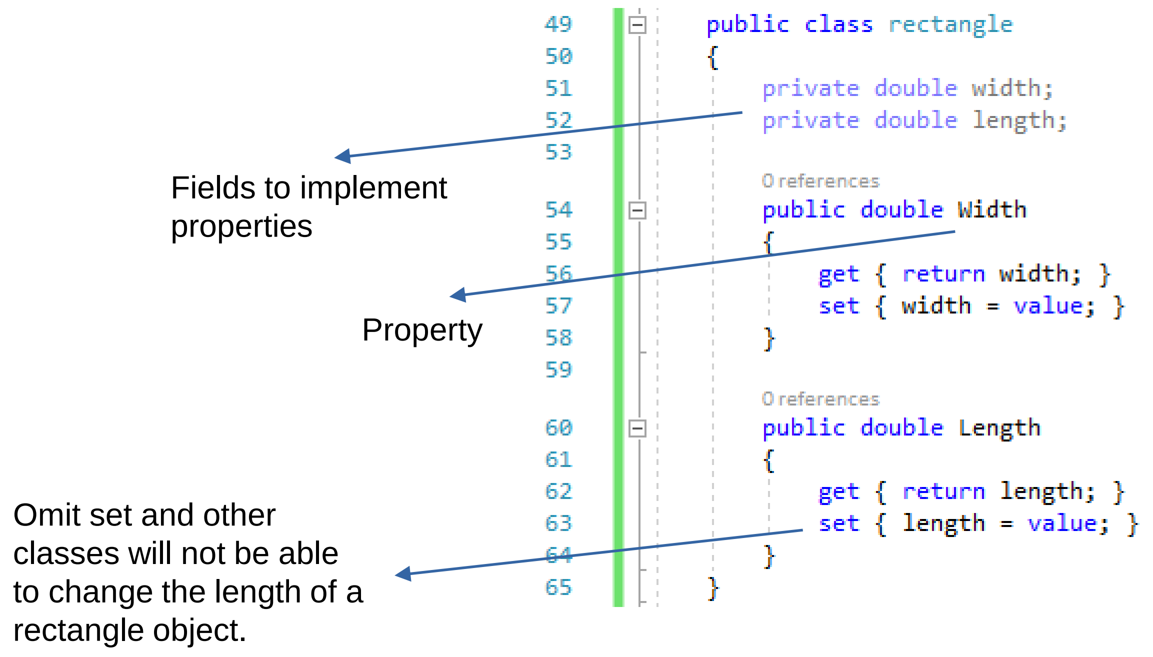
```
public class Student
{
    private string address;

    0 references
    public string GetAddress()
    {
        return address.ToUpper();
    }

    0 references
    public void SetAddress(string currentAddress)
    {
        address = currentAddress;
    }
}
```

Encapsulation implementation

Mechanism 3

- Properties
  - provide a more intuitive interface, e.g., sales.**count** = sales.**count** + 1;
  - instead of sales.**SetCount(**sales.**GetCount()** + 1**)**;

- Have optional **get** and **set** blocks
  - a read/write property defines both **get** and **set** blocks
  - a read-only property defines only the **get** block
  - a write-only property defines only the **set** block



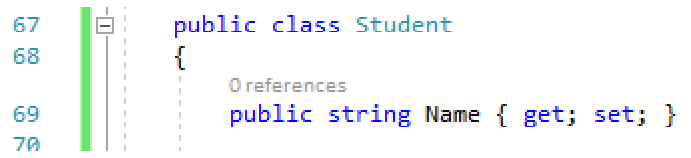
```
public class rectangle
{
    private double width;
    private double length;

    0 references
    public double Width
    {
        get { return width; }
        set { width = value; }
    }

    0 references
    public double Length
    {
        get { return length; }
        set { length = value; }
    }
}
```

Fields to implement properties
Property
Omit set and other classes will not be able to change the length of a rectangle object.

Encapsulation implementation

Mechanism 3 (cont.)

- Auto-properties
  - The C# programming language (since C# 3.0) supports **auto-implemented** properties. In this case, the compiler automatically creates a hidden variable (field) to store the data for the property.



```
public class Student
{
    0 references
    public string Name { get; set; }

```

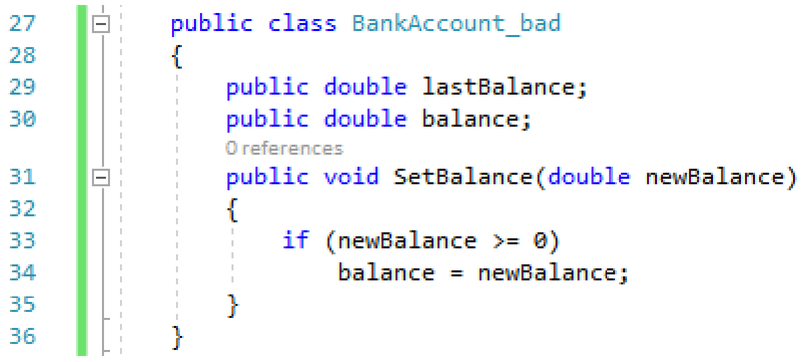
  - Advantages of auto-implemented properties
    - Read/write access levels can be set (contrary to public fields)
    - Flexibility is added if the logic is to change later
    - Data binding with UI fields will be possible (outside the scope of this subject)

OOP CONCEPT 1/4 … abstraction + OOP CONCEPT 2/4 … encapsulation

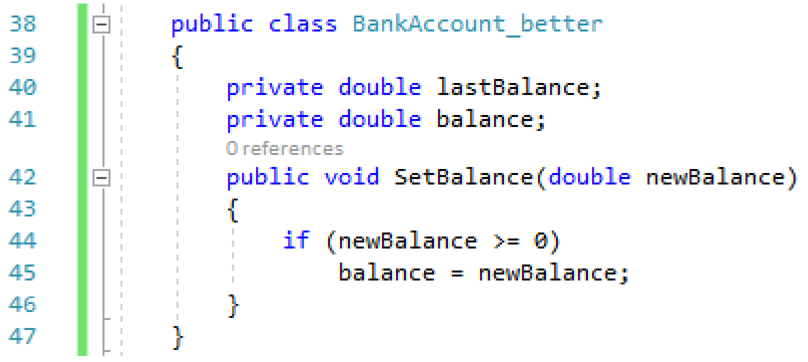
Encapsulation and abstraction together

Discussion

- Imagine we need a class called **BankAccount** that is used to keep and update the balance of an account…
  - How do we use **SetBalance**?
  - Is it good to share **lastBalance** with other classes?
  - How do we ensure that balance is always valid (in this case, not less than 0)?
  - How do we ensure that balance is not **accidentally** changed in other classes directly?
  - Why is the better implementation not the **best**?!!



```
public class BankAccount_bad
{
    public double lastBalance;
    public double balance;
    0 references
    public void SetBalance(double newBalance)
    {
        if (newBalance >= 0)
            balance = newBalance;
    }
}
```



```
public class BankAccount_better
{
    private double lastBalance;
    private double balance;
    0 references
    public void SetBalance(double newBalance)
    {
        if (newBalance >= 0)
            balance = newBalance;
    }
}
```

DESIGN PRINCIPLES

Cohesion

Single-purpose intra-class design

- Cohesion ensures that a class has a single purpose and is well-focused on that one single purpose.
- Highly cohesive classes…
    - have methods with much in common
    - are much easier to understand and maintain, with less frequent changes
    - are more (re-)usable in different contexts as they focus on a specific functionality

Image source: https://getsling.com/blog/group-cohesion-strategies/

Cohesion

Example

- We need to develop classes for an online library. The information about books and users are to be recorded in the library. Which scenario is preferred?
  - **Scenario 1-** To keep the information about books such as the title, author, isbn, and genre in one class, the information about users such as name, id, and borrowedBooks in a second class, and then use these pieces of information in a third class that keeps track of the books and users?
  - **Scenario 2-** To store the information of the books and the users such as the book title, book author, book genre, and borrower's id in one class and then have the second class to find books and borrowers?

Coupling

Separate abstractions

- Refers to the degree to which different classes know about each other.
    - **Loosely coupled:** Two classes interacting with each other through the interfaces
    - **Tightly coupled:** Two classes interacting with each other through non-interface attributes

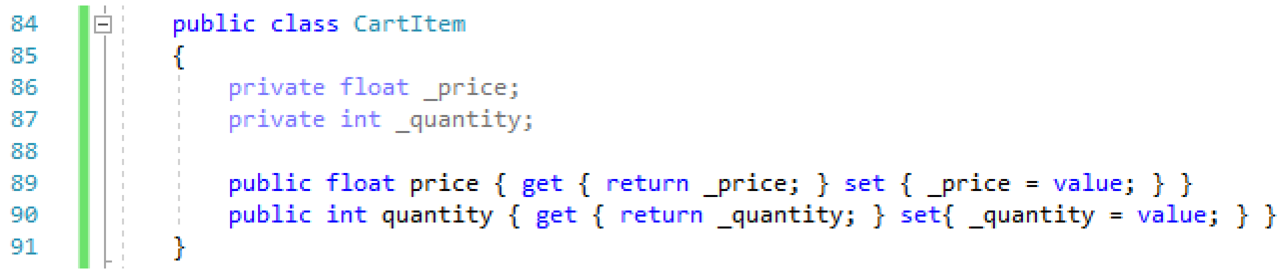
- The good design practice requires that…
    - classes be loosely coupled
    - the information and state of a class be not broken by the functions of another class
    - the implementation of the methods of one class be not dependent on those in another class
    - classes be replaced easily
    - buggy classes not drastically influence other classes (to crash)

Image source: https://www.reddit.com (Boeing's Starliner docks at International Space Station)

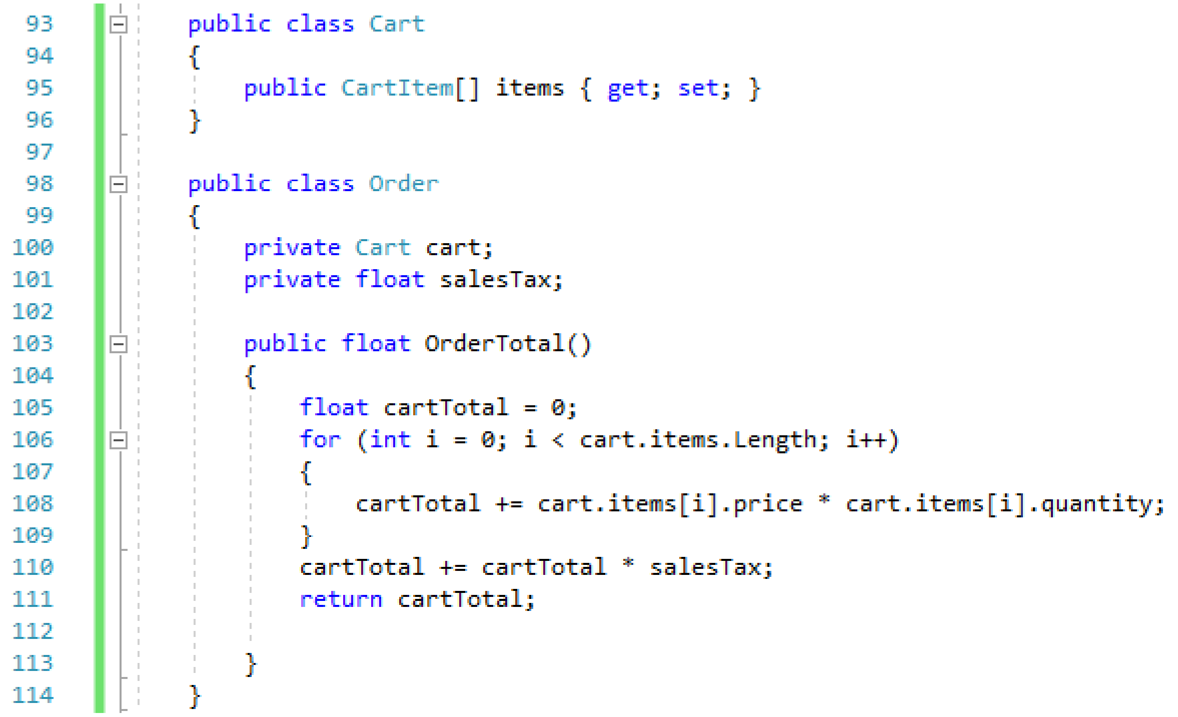
Coupling

Discussion

- Think about the example given…
  - What do you think is not right in this code? Are the classes loosely coupled?
  - Would it be easy to introduce fixed discounts on carts? How would that impact the implementation?



```
public class CartItem
{
    private float _price;
    private int _quantity;

    public float price { get { return _price; } set { _price = value; } }
    public int quantity { get { return _quantity; } set{ _quantity = value; } }
}
```



```
public class Cart
{
    public CartItem[] items { get; set; }
}

public class Order
{
    private Cart cart;
    private float salesTax;

    public float OrderTotal()
    {
        float cartTotal = 0;
        for (int i = 0; i < cart.items.Length; i++)
        {
            cartTotal += cart.items[i].price * cart.items[i].quantity;
        }
        cartTotal += cartTotal * salesTax;
        return cartTotal;

    }
}
```

A+E vs. C+C

Abstraction + encapsulation (A+E)

- Focus on the design of one class
- Ensure accuracy and safety of data of a class and its objects

Cohesion + coupling (C+C)

- Focus on the design of multiple classes in relation to each other
- Ensure classes implement only the necessary functionality
- In a good design:
    - classes are **highly** cohesive
    - classes are **loosely** coupled

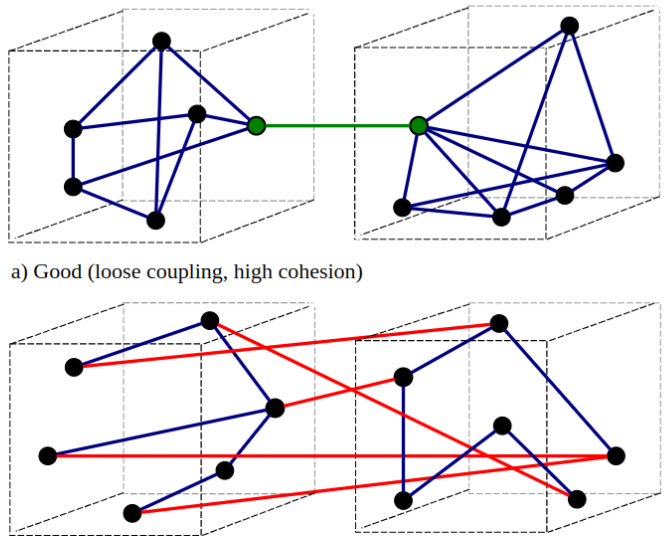

a) Good (loose coupling, high cohesion)
b) Bad (high coupling, low cohesion)

Image source:
https://upload.wikimedia.org/wikipedia/commons/0/09/CouplingVsCohesion.svg

Epilogue

PROGRAMMING IS NOT ABOUT WHAT YOU KNOW, IT IS ABOUT WHAT YOU CAN FIGURE OUT...

CHRIS PINE