Swinburne University of Technology

COS20007 Object Oriented Programming

Key Object Oriented Concepts

PDF generated at 19:01 on Sunday 8th October, 2023

Produced by OOP Submission Formatter

Inheritance:
Inheritance is the mechanism that allows one class (child or derived class) to inherit the properties and behaviours of another class (parent or base class).
It promotes code reuse and the creation of hierarchical structure.

In this example, the **Dog** and **Cat** classes inherit from the **Animal** class, gaining access to its **Name** property and **Eat()** method.

```
class Animal
{
    public string Name { get; set; }

    public Animal(string name)
    {
        Name = name;
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }
}

class Dog : Animal
{
    public Dog(string name) : base(name) { }

    public void Bark()
    {
        Console.WriteLine($"{Name} says woof");
    }
}

class Cat : Animal
{
    public Cat(string name) : base(name) { }

    public void Meow()
    {
        Console.WriteLine($"{Name} says meow");
    }
}
```

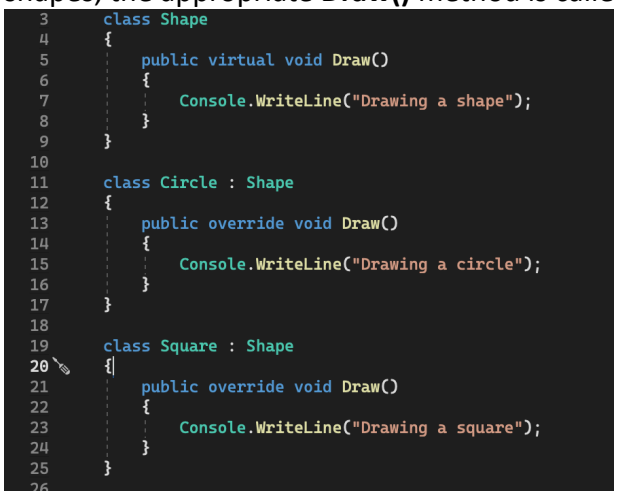
Encapsulation:
Encapsulation is the concept of bundling data, attributes and methods that operate into single unit (class).
It restricts direct access to the internal state of an object and promotes controlled access through methods.

In this example, the **BankAccount** class encapsulates the **accountNumber** and **balance** fields, allowing controlled access through methods like **Deposit()**, **Withdraw()**, and **GetBalance()**.

```
class BankAccount
{
    private string accountNumber;
    private double balance;

    public BankAccount(string accountNumber)
    {
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }

    public void Deposit(double amount)
    {
        if (amount > 0)
        {
            balance += amount;
        }
    }

    public void Withdraw(double amount)
    {
        if (amount > 0 && amount <= balance)
        {
            balance -= amount;
        }
    }

    public double GetBalance()
    {
        return balance;
    }
}
```

Polymorphism:
Polymorphism allows objects of different classes to be treated as objects of a common base class.
It allows for method overriding, where a subclass can override a specific implementation of a method defined in its base class, and dynamic method binding, which means that the appropriate method is determined at runtime based on the actual type of the object.

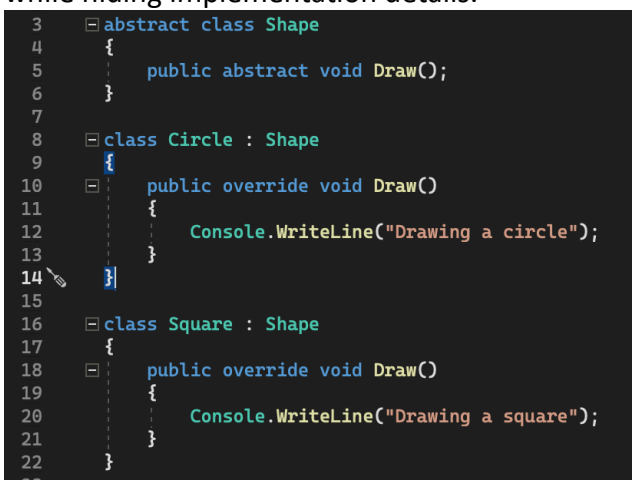
In this example, the **Shape**, **Circle**, and **Square** classes demonstrate polymorphism. The **Draw()** method is overridden in the subclasses, and when we iterate through an array of shapes, the appropriate **Draw()** method is called based on the actual object type.



```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

class Square : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a square");
    }
}
```

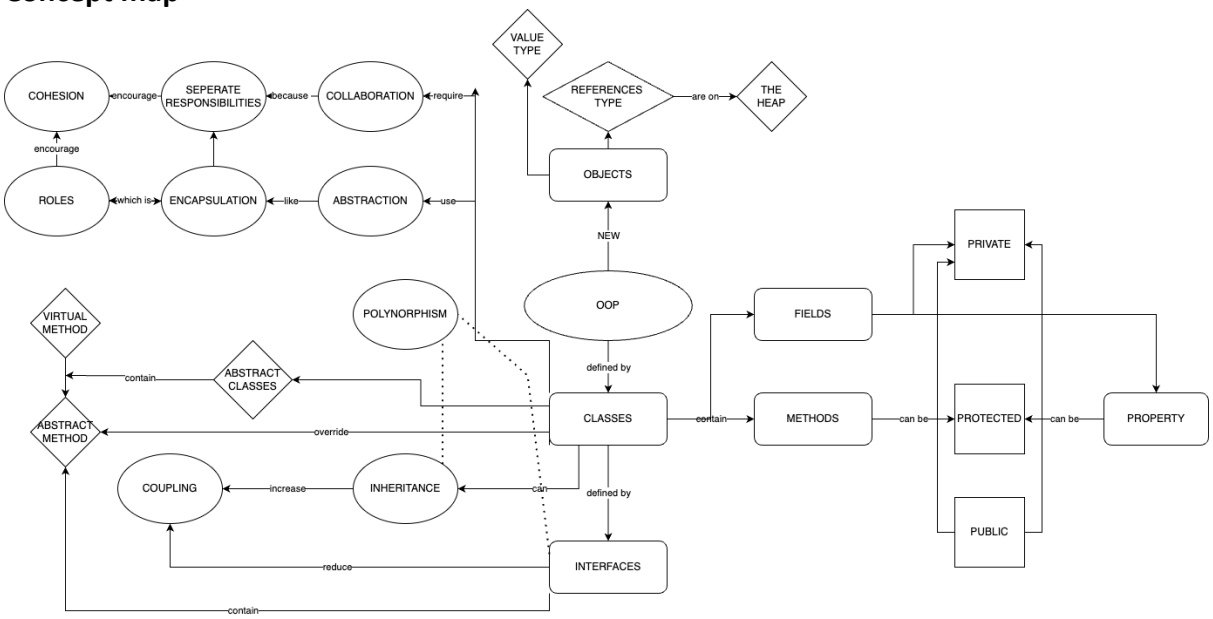
Abstraction:
Abstraction is the process of simplifying complex systems by breaking them down into more manageable and understandable parts.
It involves creating abstract classes or interfaces that define a set of methods or properties without specifying their implementation details. Derived classes then override concrete implementations while adhering to abstract class's contract, promoting modularity and flexibility in software design.

In this example, the **Shape** class is abstract, and it defines an abstract method **Draw()**. The **Circle** and **Square** classes inherit from **Shape** and provide concrete implementations of the **Draw()** method. Abstraction allows us to create a common interface for different shapes while hiding implementation details.



```
abstract class Shape
{
    public abstract void Draw();
}

class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

class Square : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a square");
    }
}
```

Concept map


VALUE TYPE
COHESION
encourage
SEPERATE RESPONSIBILITIES
because
COLLABORATION
require
REFERENCES TYPE
are on
THE HEAP
encourage
OBJECTS
ROLES
which is
ENCAPSULATION
like
ABSTRACTION
use
NEW
PRIVATE
VIRTUAL METHOD
POLYNORPHISM
OOP
FIELDS
contain
ABSTRACT CLASSES
defined by
ABSTRACT METHOD
override
CLASSES
contain
METHODS
can be
PROTECTED
can be
PROPERTY
COUPLING
increase
INHERITANCE
can
defined by
PUBLIC
reduce
INTERFACES
contain