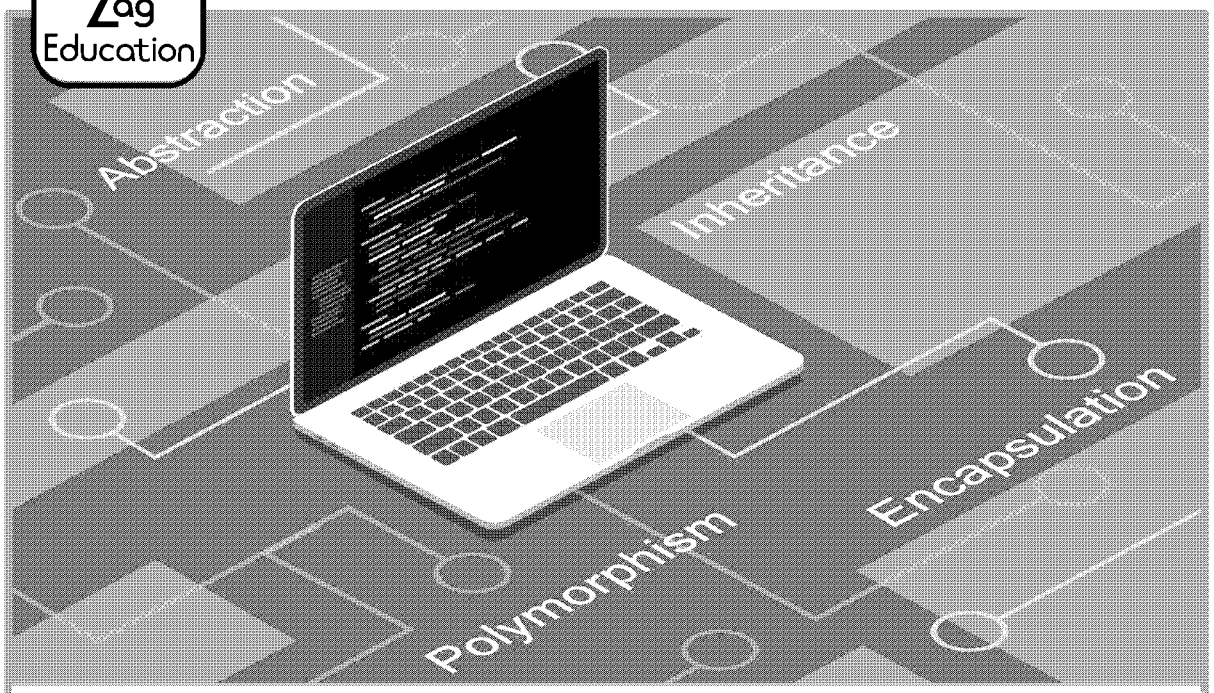
Zig Zag Education

Computer Science



Introduction to Object-Oriented Programming (C# Edition)

Cameron Standring

DA11/
9754

POD
9754

Contents

TEACHER'S INTRODUCTION

This resource is designed as an introduction to object-oriented programming (OOP), with the aim of taking students with some experience of procedural programming, through to having the required OOP knowledge and skills required for a KS5 course in Computer Science. It is best used by reading and working through the **five topics** in order, as later topics build on the knowledge and skills that students learn in earlier topics.

There are examples of the content and **programming tasks** throughout each topic which provide stretch and challenge for all students through repetition of the topic skills. Review questions have been provided at the end of each topic to test students understanding and application of the theory covered. The answers to these written questions are included in the answer section towards the back of this resource. 'C# Notes' are also included throughout to highlight how C# specifically deals with object-oriented concepts.

Each topic contains pseudocode examples written in these boxes

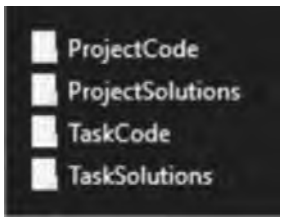
```
These black boxes show where C# code is being used instead of pseudocode
```

There are three **programming projects** for students to test their object-oriented programming skills. These programming projects are independent from each other, can be done in any order (although they are given in order of increasing complexity). For each project, there are two versions of the skeleton code:

1. The *Extended* versions provide only the main method of the program as a starting point
2. The *Basic* versions, in addition to the main method, also include the program's classes and select methods

The basic task provides a version of each project that is less complex; ideal for using when time is limited, or for use with weaker students. Each project also comes with a model solution (with marking guidance).

C# code files are provided electronically on the accompanying CD.



TaskCode contains the skeleton code for the four C# tasks in topics 1-4.
TaskSolutions contains the answer files for each task.

ProjectCode contains the skeleton code for the three C# programming projects.
ProjectSolutions contains the answer files (with marking information written as comments) for each project.

The answer files for both the C# tasks and programming projects provide working programs that contain comments to show where marks should be awarded. In the case of the programming projects, any marks followed by 'ETO' (Extended Task Only) should be awarded only to students attempting the extended version of the project.

In addition to the code files, a **HTML version of the student resources** is also provided. It is recommended that you copy the *IntroToOOP* folder onto your school's secure network, and provide a shortcut to the index.html inside it.

C Standring, July 2019

1. FUNDAMENTALS OF OBJECT-PROGRAMMING

In this chapter you will learn:

- ☑ What object-oriented programming means
- ☑ The differences between procedural programming and object-oriented programming
- ☑ Why the object-oriented programming paradigm is used
- ☑ What classes, objects, attributes and methods (including static attributes and methods)
- ☑ What constructors are and how they are used to instantiate a class
- ☑ How to create basic object-oriented programs

Introduction to object-oriented programming

There are many different programming styles that can be used to create computer programs (known as **programming paradigms**) that you are likely to be familiar with is **procedural**. In procedural programming, every variable, constant and subroutine is defined separately and relationships between each other.

Another commonly used programming paradigm is **object-oriented programming**. This allows you to define separate **objects** that have their own associated values and subroutines. This means that subroutines can be easily grouped together in a logical way.

Consider this interactive map application:

In an object-oriented program, each location pin would be defined as a different object that would have its own associated values, such as its name, its location and what it is marking (e.g. a restaurant). Each pin object would also have some associated subroutines, such as displaying detailed information if the pin is clicked, or allowing for the pin's information to be edited.

Object-oriented programming is primarily used because of the advantages of three concepts explained further later chapters): **encapsulation**, which allows different parts of the program to be made easier to understand and work on; **inheritance**, which allows different objects to still share the same core code; and **polymorphism**, which allows subroutines to know which object is using the subroutine and what data is passed to it.

Objects and classes

As object-oriented programs can have many different objects, many of which sha
not need to be written to define the properties of each individual object. Instead,
(known as a **class**) is created.

For example, the `Pin` class may look like this:

```
class Pin
    private name //Data belonging to [illegible] in [illegible]ject
    private location
    private markerTyp[illegible]

    public[illegible]du[illegible] new(pinName, pinLocation, pinMarker)
        th[illegible]e = pinName //Method to create a new Pin object
        this.location = pinLocation
        this.markerType = pinMarker
    endprocedure
    ...
endclass
```

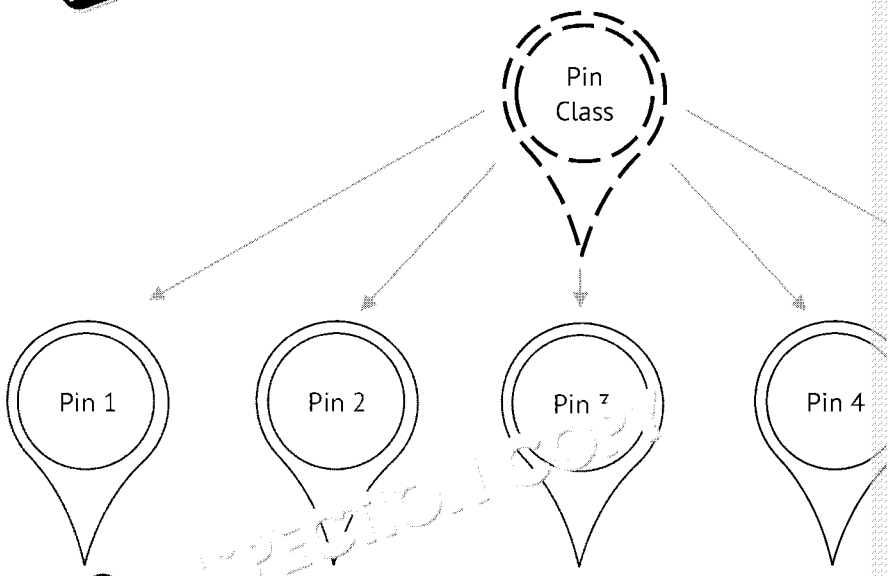
The keyword 'this' is used by an object to refer to itself, so when an object runs the 'new' subroutine, 'th
to the value of 'pinName'

A `Pin` object could then be created using the `Pin` class:

```
templeMeads = new Pin("Bristol Temple Meads", [51.449760, -2.581080], "T
```

The `Pin` class defines the values associated with a `Pin` object (in this case, `name`
`markerType`) and the subroutines that a `Pin` object ca[illegible] form. The values as
known as **attributes**, and the subroutines associa[illegible] an object are known as

The `new` method is a **constructor** [illegible] that creates an object of a particula
values. The process of c[illegible]ject from a class is known as **instantiation**, a
instance of [illegible]. [illegible] think of classes as blueprints, and objects as the indi
blueprints.



A class provides a template from which many objects can be crea

Most attributes and methods are only relevant to a particular object. However, so
attribute or method that is relevant to the class as a whole. These are known as s
methods.

For example, the `Pin` class could include a static attribute to count the number o

```
class Pin
    private name
    private location
    private markerType
    public static noOfPins = 0 //Static attribute that belongs to the class

    public procedure new(pinName, pinLocation, pinMarker)
        this.name = pinName
        this.location = pinLocation
        this.markerType = pinMarker
        Pin.count = count + 1
    endprocedure
    ...
endclass
```

Notice that the static attribute is set using 'Pin.count' and not 'this.count' because the attribute belongs t the class. Similarly, static methods are called by 'ClassName.method()' whereas non-static methods are

If there is an attribute or method in a class that you may want to use, even if ther it should be static.

C# Note

The following pseudocode program:

```
class Pin
    private name
    private location
    private markerType
    public static noOfPins = 0

    public procedure new(pinName, pinLocation, pinMarker)
        this.name = pinName
        this.location = pinLocation
        this.markerType = pinMarker
        Pin.noOfPins = Pin.noOfPins + 1
    endprocedure
    ...
endclass
...
pinObject = new Pin(name, location, marker)
```

... would be written in C# as:

```
public class Pin {
   private string name;
   private double[] location;
   private string markerType;
   public static int noOfPins = 0;

   public Pin(string pinName, double[] pinLocation, string
       this.name = pinName;
       this.location = pinLocation;
       this.markerType = pinMarker;
       Pin.noOfPins = Pin.noOfPins + 1;
   }
   ...
   Pin pinObject = new Pin(name, location, marker);
}
```

Note that, in C#, you must declare the types of all attributes and return types of a the exception of the constructor method, which is declared using the class name

Questions (Fundamentals of Object-Oriented Programming)

1. Define the term *programming paradigm*.

.....

.....

2. Explain the difference between *object-oriented programming* and *procedural p*

.....

.....

.....

.....

3. Explain the difference between a *class* and an *object*.

.....

.....

.....

.....

4. Identify a situation where a *static method* may be

.....

.....

5. Use pseudocode to write a *class* with relevant *attributes* and *methods* to repre
should represent the time as a 24-hour clock, and include methods to create
manually, display the time, and update the time at the end of each minute.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

C# Task 1

The *Task 1* skeleton code (`Skeleton`) is part of a program that allows a user to c
their account, check their balance and deposit or withdraw money from their acc
class, that stores information about the individual bank accounts, and a `Bank` cla
accounts and performs operations on individual accounts when asked to by the u

Add the missing attributes and method logic to the *T* [illegible] skeleton code to comp

No changes should be made to ma[illegible] methods should be defined, and th
methods use should not b[illegible] deleted or added to.

Account

```
public class Account {
    private int
    private string
    private double

    /*A new bank account should be defined with a given accou
    balance*/
    public Account (int number, string password, double balan
        this.accountNumber = number;
        this.
        this.
    }

    public int getNumber() {
    //This method should return the account number of this a
    }

    public bool checkPassword( [illegible] password) {
    /*This method should [illegible] a given password is equal t
    account*/
    }

    publi[illegible]ble getBalance() {
    //This method should return the balance of this account
    }

    public void setBalance(double newBalance) {
    //This method should change the balance of this account t
    }
}
```

Bank.cs

```
using System.Collections.Generic;

public class Bank {
    private List<Account> accounts;
    private static int latestAccount;

    Bank() { /*A new bank is defined wit[illegible]st of bank acco
              keeps track of the a[illegible]umber of the most
        this.accounts = new [illegible]ount>();
        Bank.latestAccou[illegible]
    }

    publ[illegible]gin() {
    /*Thi[illegible]hod should ask the user to give their account
    returning the account number if they match, or returning
    }

    public void deposit(int number) {
    /*This method should ask the user how much money they wa
    account, and correctly update the of their account*/
    }

    public void withdraw(int number) {
    /*This method should ask the user how much money they wa
    account, and correctly update the balance of their accou
    }

    public void checkBalance(int number) {
    /*This method should display a message telling the user
    their account*/
    }

    public void addAccou[illegible]
    /*This method sh[illegible]e a new account with an accoun
    the accou[illegible] the last account created, a passwo
    a bal[illegible]. The account should be added to the bank's
    }
}
```

Program.cs

```
using System;

public class Program {

    public static void Main() {
        Bank bank = new Bank();
        bool loggedIn = false;
        bool quitting = false;
        int accountNo = -1;

        while (!logged[illegible] && !quitting) {
            Cons[illegible]teLine("Do you have an account? (y/n/qu
            string response = Console.ReadLine();
            Console.WriteLine();
            if (string.Equals(response, "y")) {
                accountNo = bank.login();
                if (accountNo != -1)
                    loggedIn = true;
            }
            else if (string.Equals(response, "n"))
                bank.addAccount();
            else if (string.Equals(response, "quit"))
                quitting = true;
        }

        while (!quitting) {
            Console.WriteLine("Press 1 to check your balance\n
                                money\nPress 3 to withdraw mone
            string option = Console.ReadLine();
            Console.WriteLine();
            if (string.Equals(option, "1"))
                bank.checkBalan[illegible]cNo);
            else if (strin[illegible](option, "2")) {
                bank.[illegible]+accountNo);
                b[illegible]eckBalance(accountNo);
            }
            else if (string.Equals(option, "3")) {
                bank.withdraw(accountNo);
                bank.checkBalance(accountNo);
            }
            else if (string.Equals(option, "4"))
                quitting = true;
            else
                Console.WriteLine("Invalid option selected");
                Console.WriteLine();
        }
    }
}
```

2. ENCAPSULATION

In this chapter you will learn:

- ☑ What encapsulation is
- ☑ Why encapsulation is used
- ☑ How to use private and protected attributes and methods
- ☑ How to properly encapsulate a program

Encapsulation in object-oriented programming

As mentioned in chapter 1, **encapsulation** is the idea of grouping data and subroutines together to work on and understand. In object-oriented programming, encapsulation is achieved by ensuring that a class should only contain the attributes and methods that it needs, and none of the logic should depend on the internal processing in another class.

Imagine a company has a system that stores various information about different employees. If we don't use encapsulation, any data can be used or altered in any part of the program. This means:

1. If any errors occur it will be much harder to identify the source of the error, because it could be anywhere. In a properly encapsulated program, any errors will originate either from a class that isn't working correctly, or from an error in how different parts of the program interact.
2. It means that some parts of the system will have access to attributes and methods they shouldn't have access to. In the example of a company's employee information system, it might be fine for an employee to be able to update some of their own records (such as their name), but they shouldn't have access to change other information (such as their salary) or information about other employees (such as their addresses).

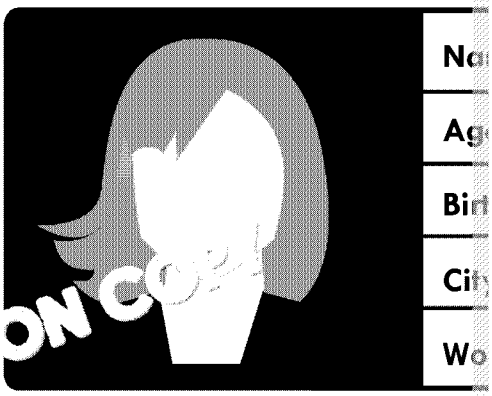
Private attributes and methods

```
class Account
    private accountPassword //This attribute is private
    ...
    public function checkPassword(password) //This method is public
    ...
endclass

class Bank
    private accounts //This attribute is private
    ...
    public procedure withdraw(number) //This method is public
    ...
endclass
```

Take the above program from *Task 1*. Notice how **private** and **public** keywords are used. If a method is private, it can only be accessed within the class. If an attribute or method is public, it can be accessed by other classes. In *Task 1*, the attribute `accountPassword` in the `Account` class and `Bank` class cannot access passwords which it shouldn't have access to. Instead, the `Account` class has a method `checkPassword` to check whether a user has entered their password correctly.

This is useful for security purposes (in a real-world system, the more a password is stored, the more vulnerable it is to being stolen) and for encapsulating the program (the `Bank` class doesn't need to know what the password is, it only needs to be able to check that a password is correct, without having access to that data).

```
class Account
    public profileImage
    public name
    private age
    public birthday
    public city
    private work
    ...
```



Private attribu[illegible]e h[illegible]en from other parts of the system. There are many situations where you [illegible] stored about a [illegible]r object.

When an attribute from another class is needed, instead of making that attribute [illegible] method that returns the value of the attribute. Similarly, to change the value of a [illegible] public method to change its value rather than directly altering it. Methods that r[illegible] attribute are known as **accessors** (or 'getters'), and methods that alter the value o[illegible] as **mutators** (or 'setters'). These may at first appear unnecessary, but can be usefu[illegible] functionality of the class.

For example, imagine you have the following code:

```
class Clock
    public currentTime //This attribute is public
    ...
endclass

class Display()
    clock = new Clock()
    ...
     public procedure [illegible]
         pri[illegible]ck [illegible]me)//currentTime is public so can be called in Display
    endpr[illegible]e
endclass
```

The `Display` class directly accesses the clock's `currentTime` attribute to displ[illegible] if you wanted to make a change to how the clock's time is displayed (e.g. by maki[illegible] 24-hour clock, or changing whether seconds or milliseconds are displayed) and th[illegible] `currentTime` in multiple places, then formatting or other checks would need to [illegible] throughout the `Display` class, which could mean changing a lot of code.

The program could be instead be written as:

```
class Clock
    private currentTime //currentTime is made private
    ...
     public function getTime() //This public metho[illegible]s [illegible]ccess to currentTi[illegible]
         return this.currentTime
    endfunction
endclass

class Disp[illegible]
    clock = [illegible]lock()
    ...
     public procedure showTime()
         print(clock.getTime()) //The public method is called in Display
    endprocedure
endclass
```

With this version of the program, the change could be made to the `getTime` acc
does not need to be updated. Accessors and mutators should not just be used to
but to hide information from other classes or limit the ability of other classes to

C# Note

In C#, methods and attributes are made public by default

```
public class Class {
   int publicAttribute;

   void setAttribute(int value) {
      this.publicAttribute = value;
   }
}
```

... is equivalent to:

```
public class Class {
   private int publicAttribute;

   private void setAttribute(int value) {
      this.publicAttribute = value;
   }
}
```

While it is therefore not necessary to declare an attribute or method as public, it
make it clearer how you intend that attribute or method to be used.

Questions (Encapsulation)

1. Define the term *encapsulation*.

.....

.....

2. Explain the difference between a *private* attribute or method and a *public* attribute.

.....

.....

.....

.....

3. Explain one reason why an attribute may be made *private*.

.....

.....

4. Define the terms *accessor* and *mutator*.

.....

.....

.....

.....

5. Identify when *accessors* and *mutators* should be used.

.....

.....

.....

.....

6. Explain why you might make an attribute public instead of using *accessors* and *mutators*.

.....

.....

.....

.....

C# Task 2

The *Task 2* skeleton code is a system that manages a hotel and its staff. Custome their rooms, and leave feedback depending on how their stay was (if they are suc room is clean they become happier with their stay, and if their room is overbooke happy with their stay).

Recreate the *Task 2* non-encapsulated code (`Non-Encapsulated`) so that it kee properly encapsulated. There should be classes for `Hotel`, `Room`, `Customer`, Ma `Cleaner`. The manager should be responsible for processing feedback; the clean cleaning rooms; the receptionist should be responsible for checking customers in attributes should be private (although you may add any methods that you th

You may use provided *Task 2* encapsulated skeleton code (`Encapsulated S` converted main method and constructors for each class that do not need to be alte

---

Non-Encapsulated

`Cleaner.cs`

```
public class Cleaner {
    public string name;

    public Cleaner(string name) {
        this.name = name;
    }
}
```

`Customer.cs`

```
public class Customer {
    public int roomBooking;
    public string name;
    public int feedback;

    public Customer(int roomBooking, string name) {
        this.roomBooking = roomBooking;
        this.name = name;
        this.feedback = 0;
    }
}
```

`Hotel.cs`

```
using System.Collections.Generic;

public class Hotel {
    public List<Room> rooms;

    public Hotel(List<Room> rooms) {
        this.rooms = rooms;
    }
}
```

`Manager.cs`

```
public class Manager {
    public string name;

    public Manager(string name) {
        this.name = name;
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;

public class Program {

    static void addOccupant(Room room, Custom[illegible] occupantIn) {
        if (room.occupants.Count < room.[illegible]
            room.occupants.Add(occup[illegible]an In[illegible]
             occupantIn.feedbac[illegible]
        }
        else {
            [illegible]cu[illegible] feedback--;
            [illegible]tu[illegible]n;
        }
        if (room.clean == true)
            occupantIn.feedback++;
        else
            occupantIn.feedback--;
        room.clean = false;
    }

    static void removeOccupant(Room room, Customer occupantOu
        int index = -1;
        for (int i = 0; i < room.occupants.Count; i++) {
            if (string.Equals(room.occupants[i], occupantOut)
                index = i;
        }
        if (index != -1)
            room.occupants.RemoveAt(index);
    }

    static void takeFeedback(M[illegible]nager, Customer custo
        if (customer.feedb[illegible]
            Console.W[illegible]e(manager.name + " says: " + cust
                                with their stay!");
        e[illegible]f (customer.feedback < 0)
            [illegible]sole.WriteLine(manager.name + " says: " + custo
                                with their stay!");
        else
            Console.WriteLine(manager.name + " says: " + custo
                                stay ok.");
    }

    static void cleanRooms(Cleaner cleaner, List<Room> hotel)
        for (int i = 0; i < hotel.Count; i++) {
            if (hotel[i].occupants.Count == 0) {
                hotel[i].clean = true;
                 Console.WriteLine(cleaner.name + " cleared Ro
            }
        }
    }

    static void checkIn(Reception[illegible]t [illegible]ionist, List<Room>
                        cust[illegible]er
        addOccupant(hot[illegible] [illegible]u[illegible]er.roomBooking - 1], customer
        Console.Wri[illegible]eceptionist.name + " checked in "
    }

    stati[illegible]d checkOut(Receptionist receptionist, List<Roo
                        customer, Manager manager) {
        removeOccupant(hotel[customer.roomBooking - 1], custo
        Console.WriteLine(receptionist.name + " checked out "
        takeFeedback(manager, customer);
    }
```

```
    public static void Main() {
        Room room1 = new Room(1, 1, false);
        Room room2 = new Room(2, 2, false);
        Room room3 = new Room(3, 1, false);

        List<Room> hotel = new List<Room>();
        hotel.Add(room1);
        hotel.Add(room2);
        hotel.Add(room3);

        Customer customer1 = new Customer(1, "Mrs. White");
        Customer customer2 = new Customer(2, "Mr. Green");
        Customer customer3 = new Customer(2, "Miss. Scarlett");
        Customer customer4 = new Customer(3, "Mrs. Peacock");
        Customer customer5 = new Customer(2, "Prof. Plum");
        Customer customer6 = new Customer(3, "Col. Mustard");

        Receptionist receptionist = new Receptionist("Jane");
        Cleaner cleaner = new Cleaner("Michael");
        Manager manager = new Manager("Janhavi");

        checkIn(receptionist, hotel, customer1);
        checkIn(receptionist, hotel, customer2);
        checkIn(receptionist, hotel, customer3);
        checkOut(receptionist, hotel, customer1, manager);

        cleanRooms(cleaner, hotel);

        checkIn(receptionist, hotel, customer4);
        checkOut(receptionist, hotel, customer4, manager);
        checkIn(receptionist, hotel, customer5);
        checkOut(receptionist, hotel, customer5, manager);
        checkOut(receptionist, hotel, customer2, manager);
        checkOut(receptionist, hotel, customer3, manager);

        cleanRooms(cleaner, hotel);

        checkIn(receptionist, hotel, customer6);
        checkOut(receptionist, hotel, customer6, manager);
        Console.ReadLine();
    }
}
```

Receptionist.cs

```
public class Receptionist {
    public string name;

    public Receptionist(string name) {
        this.name = name;
    }
}
```

Room.cs

```
using System.Collections.Generic;

public class Room {
    public int number;
    public int size;
    public List<Customer> occupants;
    public bool clean;

    public Room(int number, int size, bool clean) {
        this.number = number;
        this.size = size;
        this.occupants = new List<Customer>();
        this.clean = clean;
    }
}
```

Encapsulated Skeleton

Cleaner.cs

```
public class Cleaner {
    private string name;

    public Cleaner(string name) {
        this.name = name;
    }
}
```

Customer.cs

```
public class Customer {
    private int roomBooking;
    private string name;
    private int feedback;

    public Customer(int roomBooking, string name) {
        this.roomBooking = roomBooking;
        this.name = name;
        this.feedback = 0;
    }
}
```

Hotel.cs

```
using System.Collections.Generic;

public class Hotel {
    private List<Room> rooms;

    public Hotel(List<Room> rooms) {
        this.rooms = rooms;
    }

    public List<Room> checkRooms() {
        return this.rooms;
    }
}
```

Manager.cs

```
public class Manager {
    private string name;

    public Manager(string name) {
        this.name = name;
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;

public class Main {

    public static void Main() {
        List<Room> rooms = new List<Room>();
        rooms.Add(new Room(1, 1, false));
        rooms.Add(new Room(2, 2, true));
        rooms.Add(new Room(3, 1, false));
        Hotel hotel = new Hotel(rooms);
        Customer customer1 = new Customer(1, "Mrs. White");
        Customer customer2 = new Customer(2, "Mr. Green");
```

```
        Customer customer3 = new Customer(2, "Miss. Scarlett");
        Customer customer4 = new Customer(3, "Mrs. Peacock");
        Customer customer5 = new Customer(2, "Prof. Plum");
        Customer customer6 = new Customer(3, "Col. Mustard");
        Receptionist receptionist = new Receptionist("Jane");
        Cleaner cleaner = new Cleaner("Michael");
        Manager manager = new Manager("Janhavi");

        receptionist.checkIn(hotel, customer1);
        receptionist.checkIn(hotel, customer2);
        receptionist.checkIn(hotel, customer3);
        receptionist.checkOut(hotel, customer1, manager);

        cleaner.cleanRooms(hotel);

        receptionist.checkIn(hotel, customer4);
        receptionist.checkOut(hotel, customer4, manager);
        receptionist.checkIn(hotel, customer5);
        receptionist.checkOut(hotel, customer5, manager);
        receptionist.checkOut(hotel, customer2, manager);
        receptionist.checkOut(hotel, customer3, manager);

        cleaner.cleanRooms(hotel);

        receptionist.checkIn(hotel, customer6);
        receptionist.checkOut(hotel, customer6, manager);
        Console.ReadLine();
    }

    public static void main(string[] args) {
        new Main();
    }
}
```

Receptionist.cs

```
public class Receptionist {
    private string name;

    public Receptionist(string name) {
        this.name = name;
    }
}
```

Room.cs

```
using System.Collections.Generic;

public class Room {
    private int number;
    private int size;
    private List<Customer> occupants;
    private bool clean;

    public Room(int number, int size, bool clean) {
        this.number = number;
        this.size = size;
        this.occupants = new List<Customer>();
        this.clean = clean;
    }
}
```

3. INHERITANCE AND ABSTRACT

In this chapter you will learn:

- ☑ What inheritance is and what parent and child classes are
- ☑ What super methods are
- ☑ What interfaces and abstract methods are and how they are used
- ☑ How to create object-oriented programs with inheritance

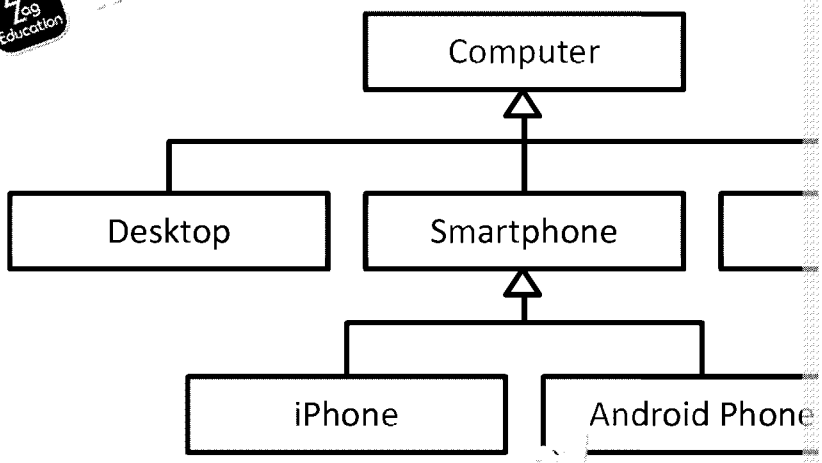
Inheritance

When creating classes, you may begin to realise that some classes have certain similarities, with code being duplicated across classes.

For example, in *Task 2*, the `Manager`, `Receptionist`, `Cleaner` and `Customer` classes share a `name` attribute:

```
class Customer
    private roomBooking
    private name //Customer has a 'name' attribute
    ...
endclass

class Receptionist
    private name //Receptionist has a 'name' attribute
    ...
endclass
```

This is not too much of an issue when there are very few similarities, but as soon as there are multiple identical attributes or even identical methods, a lot of time can be spent copying from one class to another. Object-oriented programming solves this dilemma through inheritance.



You can find examples of inheritance everywhere. The above image shows the inheritance between different types of computer. Desktops, smartphones and laptops are all specific types of computer, and Android phones are also specific types of smartphone. Inheritance only works when we come up with a definition for a smartphone, then that definition will also work for an iPhone, and work for every computer.

Simply put, inheritance refers to when one class copies the characteristics of another class and/or alter that class's methods or attributes. This means that a generic class can be created with basic characteristics of a class, as shown on the next page.

```
class Bird
    protected featherColour

    public procedure new(colour)
        featherColour = colour
    endprocedure

    public procedure fly()
        print("This bird is flying")
    endprocedure
endclass
```

Note the use of the protected keyword on the `featherColour` attribute. If an a as protected, it can only be accessed from within that class or from a class that in `Bird` class defines the attributes and abilities that every bird has. However, we m specific class that builds on this one:

```
class SwimmingBird inherits Bird //Inherits the attributes and methods of Bird
    public procedure swim()
        print("This bird is swimming")
    endprocedure
endclass
```

Note that this class has no defined constructor. It uses the same constructor as its parent class and so

A class that inherits from another class is known as a **child** class, and the class th its **parent** class. In this case, the class `SwimmingBird` is a child of the parent cla `SwimmingBird` class keeps all of the characteristics defined in the `Bird` class,

In the following code:

```
public procedure main()
    bird1 = new Bird("Brown")
    bird2 = new SwimmingBird("Brown")

    bird1.fly()
    bird2.fly() //SwimmingBirds can use the 'fly' method defined in Bird
    bird2.swim()
    bird1.swim() //Birds cannot use the 'swim' method defined in Swimming
endprocedure
```

`bird1.fly()` and `bird2.fly()` will both print 'This bird is flying' because alth does not define the `fly` method, it can still use it because it is defined in its pare

`bird2.swim()` will print 'This bird is swimming', `bird1.swim()` will cause an does not have access to the methods defined in the child class `SwimmingBird`.

Inheritance is not just used to add to a parent class, but can also be used to chan parent class act in the child class:

```
class Flamingo inherits SwimmingBird
    public procedure new()
        super.new("Pink") //Calls the 'new' method from the SwimmingBird
        print("New flamingo created")
    endprocedure
endclass
```

A class can inherit from a class that inherits from yet another class

The `Flamingo` class inherits from the `SwimmingBird` class, but defines a new c
a `Flamingo` object is created, the inherited `Bird` constructor will be replaced by

The `Flamingo` constructor uses what is known as a **super method**. Super metho
class's parent class. So, in the previous case, `super.new()` calls the `new` metho
(which in this case is just the `new` method from the `Bird` class).

A parent class can have multiple child classes that in[illegible], along with the a

```
class FlightlessBird inherits Bi[illegible]ingBird and FlightlessBird both inh
    public procedure fl[illegible]
        pri[illegible]Thi[illegible]not fly")
    endpr[illegible]e
endclass
```

Any number of child classes can expand on the same parent class in different wa

Abstract classes

While a class can have multiple child classes without causing any problems, if a c
than one class (known as **multiple inheritance**) there can be issues regarding wha
should inherit. For example, if a class tried to inherit from multiple classes as foll

```
class A
    public procedure method()
        print("Do this")
    endprocedure
endclass

class B inherits A //Inherits 'met[illegible] fro[illegible] A
endclass

class C in[illegible]A, [illegible]erits 'method' from A
    public [illegible]dure method() //Changes 'method'
        pri[illegible] Do that")
    endprocedure
endclass

class D inherits B, C //Class D inherits 'method' from B and C
endclass
```

It is not clear if the result of `method()` in class `D` should be 'Do this' or 'Do that'.
won't allow a class to inherit multiple classes, while others try to solve this issue
parent class depending on how the classes are arranged (but this can be confusin

One way to get around the issues with multiple inheritance is to use **abstract clas**
that declares methods without specifying how they w[illegible]e methods are kno
example, the following class would be an abstra[illegible]

```
class AbstractClass
    public procedur[illegible]e[illegible]method() //This method is not abstract
        p[illegible]es [illegible]
    endpr[illegible]e

public abstract procedure abstractMethod(number) //This method is abstra
    endprocedure
endclass
```

An abstract method only defines the method's name and parameters (and the dat parameters in some programming languages) of the method. Any class that conta abstract class, but an abstract class can contain non-abstract methods.

You cannot create an object from an abstract class, and any class that inherits fro all of its parent class' abstract methods using the specified parameters for each m any of the abstract methods undefined, then it will be an abstract class itself.

```
class ConcreteClass inherits AbstractClass
    public procedure abstractMethod(number) //This method is no longer ab
        print(number)
    endprocedure
endclass
```

`ConcreteClass` defines the abstract method in `AbstractClass`, and as it ha methods remaining, `ConcreteClass` objects can be created.

Abstract methods are useful because they tell the programmer what functionality without defining a generic method in the parent class that may not be useful. For class `Dog`, with child classes for different breeds of dog, there are some functions to implement, although they may all implement it differently.

```
class Dog
    public abstract procedure whatBreed()
    endprocedure
endclass

class Labrador inherits Dog
    public procedure whatBreed()
        print("This dog is a Labrador")
    endprocedure
endclass
```

A generic version of the `whatBreed` method could be defined in the `Dog` class, but `Dog` would need to replace the method anyway, so declaring it as an abstract method Abstract methods should be used when all child classes require a certain method implementation.

Some languages refer to classes that only contain abstract methods as interfaces, multiple interfaces but only one actual class. This avoids the issues with multiple have implementations that can cause conflicts with each other.

C# Note

C# does not allow for multiple inheritance, so the following would not be a valid

```
public class First : Second, Third {
...
```

In C#, abstract methods cannot be given any functionality and must be declared with the keyword:

```
public class Abstract {
   ...
   public abstract void method();
}
```

`method` has been declared as an abstract method with return type void and no parameters. Child classes will need to implement `method` in order to be instantiated.

To implement an abstract method, the 'override' keyword must be used as follows:

```
public class Concrete : Abstract {
   ...
   override public void method() {
      ...
   }
}
```

Note that the non-abstract method must have the same return type and take the same parameters as the method that it is implementing.

If parent and child classes are declared as follows:

```
public class Parent {
   private int a;
   protected int b;

   public Parent() {
      this.a = 1;
      this.b = 2;
   }
}

public class Child : Parent {

   public Child() : base(){
      this.b++;
   }
}
```

the Child constructor will use : base() to run the constructor method of its parent and increase the value of b by 1. When a Child object is created it will not have an attribute a, as it is not inherited, and it will have an attribute b with a value of 3.

Questions (Inheritance and Abstract Methods)

1. Define the term *inheritance*.

   .....

   .....

2. Draw a diagram to show inheritance relationships between at least three things.

3. Use the pseudocode below to answer the questions that follow:

```
class Guitar
    private noOfStrings = 6

    public procedure holdFret()
    endprocedure

    public procedure strum()
    ...
    endprocedure
endclass

class ElectricGuitar inherits Guitar
    public procedure adjustVolume()
        ...
    endprocedure
endclass
```

   a) Identify the parent class and the child class.

   .....

   b) Identify the attributes and methods that ElectricGuitar inherits from Guitar.

   .....

   .....

   .....

4. Describe what happens when a class calls a *super* method.

.....

.....

5. Explain the issue caused by allowing *multiple inheritance*

.....

.....

.....

.....

6. Define the term *abstract method*, and explain when you might use an *abstract*

.....

.....

.....

.....

C# Task 3

The *Task 3* skeleton code (`Skeleton`) contains classes for various animals, desc
and what actions they can do.

Recreate the *Task 3* skeleton code so that it keeps the same functionality but add
`Animal`, `Reptile` and `Mammal`. The `Animal` class should include abstract meth

Classes should inherit from other classes as appropriate and as much functionali
moved to the three new classes. The main should not be altered.

Bat.cs

```
using System;

public class Bat {
    private bool coldBlooded;
    private string skinType;
    private bool tail;
    private int legs;
    private int arms;
    private int wings;

    public Bat() {
        this.coldBlooded = false;
        this.skinType = "fur";
        this.tail = true;
        this.legs = 2;
        this.arms = 0;
        this.wings = 2;
    }

    private void fly() {
        Console.WriteLine("This animal flies");
    }

    private void eat() {
        Console.WriteLine("This animal is an omnivore");
    }

    private void birth() {
        Console.WriteLine("This animal gives birth to live y
    }

    private void hibernate() {
        Console.WriteLine("This animal hibernates");
    }

    public void getInfo() {
        Console.WriteLine("Bat:");
        if (this.coldBlooded)
            Console.WriteLine("This animal is cold-blooded");
        else
            Console.WriteLine("This animal is warm-blooded");
        if (this.skinType != null)
            Console.WriteLine("This animal is covered in " +
        if (this.tail)
            Console.WriteLine("This animal has a tail");
        if (this.legs > 0)
            Console.WriteLine("This animal has " + this.legs
```

```
        if (this.arms > 0)
            Console.WriteLine("This animal has " + this.arms
        if (this.wings > 0)
            Console.WriteLine("This animal has " + this.wings
        this.move();
        this.eat();
        this.birth();
        this.hibernate();
        Console.WriteLine();
    }
}
```

Gorilla.cs

```
public class Gorilla {
    private boolean coldBlooded;
    private String skinType;
    private boolean tail;
    private int legs;
    private int arms;
    private int wings;

    Gorilla() {
        this.coldBlooded = false;
        this.skinType = "fur";
        this.tail = false;
        this.legs = 2;
        this.arms = 2;
        this.wings = 0;
    }

    private void move() {
        System.out.println("This animal walks and climbs");
    }

    private void eat() {
        System.out.println("This animal is a herbivore");
    }

    private void birth() {
        System.out.println("This animal gives birth to live
    }

    public void getInfo() {
        System.out.println("Gorilla:");
        if (this.coldBlooded)
            Console.WriteLine("This animal is cold-blooded");
        else
            Console.WriteLine("This animal is warm-blooded");
        if (this.skinType != null)
            Console.WriteLine("This animal is covered in " +
        if (this.tail)
            Console.WriteLine("This animal has a tail");
        if (this.legs > 0)
            Console.WriteLine("This animal has " + this.legs
        if (this.arms > 0)
            Console.WriteLine("This animal has " + this.arms
        if (this.wings > 0)
            Console.WriteLine("This animal has " + this.wings
        this.move();
        this.eat();
        this.birth();
        System.out.println();
    }
}
```

Otter.cs

```
using System;

public class Otter {
    private bool coldBlooded;
    private string skinType;
    private bool tail;
    private int legs;
    private int arms;
    private int wings;

    public Otter() {
        this.coldBlooded = false;
        this.skinType = "fur";
        this.tail = true;
        this.legs = 4;
        this.arms = 0;
        this.wings = 0;
    }

    private void move() {
        Console.WriteLine("This animal walks and swims");
    }

    private void eat() {
        Console.WriteLine("This animal is an omnivore");
    }

    private void birth() {
        Console.WriteLine("This animal gives birth to live y
    }

    public void getInfo() {
        Console.WriteLine("Otter");
        if (this.coldBlooded)
            Console.WriteLine("This animal is cold-blooded");
        else
            Console.WriteLine("This animal is warm-blooded");
        if (this.skinType != null)
            Console.WriteLine("This animal is covered in " +
        if (this.tail)
            Console.WriteLine("This animal has a tail");
        if (this.legs > 0)
            Console.WriteLine("This animal has " + this.legs
        if (this.arms > 0)
            Console.WriteLine("This animal has " + this.arms
        if (this.wings > 0)
            Console.WriteLine("This animal has " + this.wings
        this.move();
        this.eat();
        this.birth();
        Console.WriteLine();
    }
}
```

Program.cs

```
using System;

public class Program {
    public static void Main() {
        Tortoise tortoise = new Tortoise();
        Turtle turtle = new Turtle();
        Snake snake = new Snake();
        Otter otter = new Otter();
        Gorilla gorilla = new Gorilla();
```

```
        Bat bat = new Bat();

        tortoise.getInfo();
        turtle.getInfo();
        snake.getInfo();
        otter.getInfo();
        gorilla.getInfo();
        bat.getInfo();
        Console.ReadLine();
    }
}
```

Snake.cs

```
using Sy

public class Snake {
    private bool coldBlooded;
    private string skinType;
    private bool tail;
    private int legs;
    private int arms;
    private int wings;

    public Snake() {
        this.coldBlooded = true;
        this.skinType = "scales";
        this.tail = true;
        this.legs = 0;
        this.arms = 0;
        this.wings = 0;
    }

    private void move() {
        Console.WriteLine("This animal slithers");
    }

    priv  eat() {
        le.WriteLine("This animal is a carnivore");
    }

    private void birth() {
        Console.WriteLine("This animal lays eggs");
    }

    private void hibernate() {
        Console.WriteLine("This animal hibernates");
    }

    public void getInfo() {
        Console.WriteLine("Snake:");
        if (this.coldBlooded)
            Console.WriteLine("This animal is cold-blooded");
        else
            Console.WriteLine("This animal is warm-blooded");
        if (this.skinType != n
            Console.Write  "This animal is covered in " +
        if (this.ta
            iceLine("This animal has a tail");
        i. legs > 0)
            nsole.WriteLine("This animal has " + this.legs
        if (this.arms > 0)
            Console.WriteLine("This animal has " + this.arms
        if (this.wings > 0)
            Console.WriteLine("This animal has " + this.wi
        this.move();
```

```
        this.eat();
        this.birth();
        this.hibernate();
        Console.WriteLine();
    }
}
```

Tortoise.cs

```
using System;

public class Tortoise {
    private bool coldBlooded;
    private string skinType;
    private bool tail;
    private int legs;
    private int arms;
    private int wings;

    public Tortoise() {
        this.coldBlooded = true;
        this.skinType = "scales";
        this.tail = true;
        this.legs = 4;
        this.arms = 0;
        this.wings = 0;
    }

    private void move() {
        Console.WriteLine("This animal walks");
    }

    private void eat() {
        Console.WriteLine("This animal is herbivore");
    }

    private void birth() {
        Console.WriteLine("This animal lays eggs");
    }

    private void hibernate() {
        Console.WriteLine("This animal hibernates");
    }

    public void getInfo() {
        Console.WriteLine("Tortoise:");
        if (this.coldBlooded)
            Console.WriteLine("This animal is cold-blooded");
        else
            Console.WriteLine("This animal is warm-blooded");
        if (this.skinType != null)
            Console.WriteLine("This animal is covered in " +
        if (this.tail)
            Console.WriteLine("This animal has a tail");
        if (this.legs > 0)
            Console.WriteLine("This animal has " + this.legs
        if (this.arms > 0)
            Console.WriteLine("This animal has " + this.arms
        if (this.wings > 0)
            Console.WriteLine("This animal has " + this.wings
        this.move();
        this.eat();
        this.birth();
        this.hibernate();
        Console.WriteLine();
    }
}
```

Turtle.cs

```
using System;

public class Turtle {
    private bool coldBlooded;
    private string skinType;
    private bool tail;
    private int legs;
    private int arms;
    private int wings;

    public Turtle() {
        this.coldBlooded = true;
        this.skinType = "scales";
        this.tail = true;
        this.legs = 4;
        this.arms = 0;
        this.wings = 0;
    }

    private void move() {
        Console.WriteLine("This animal crawls and swims");
    }

    private void eat() {
        Console.WriteLine("This animal is an omnivore");
    }

    private void birth() {
        Console.WriteLine("This animal lays eggs");
    }

    private void hibernate() {
        Console.WriteLine("This animal hibernates");
    }

    public void getInfo() {
        Console.WriteLine("Turtle:");
        if (this.coldBlooded)
            Console.WriteLine("This animal is cold-blooded");
        else
            Console.WriteLine("This animal is warm-blooded");
        if (this.skinType != null)
            Console.WriteLine("This animal is covered in " +
        if (this.tail)
            Console.WriteLine("This animal has a tail");
        if (this.legs > 0)
            Console.WriteLine("This animal has " + this.legs
        if (this.arms > 0)
            Console.WriteLine("This animal has " + this.arms
        if (this.wings > 0)
            Console.WriteLine("This animal has " + this.wings
        this.move();
        this.eat();
        this.birth();
        this.hibernate();
        Console.WriteLine();
    }
}
```

4. POLYMORPHISM

In this chapter you will learn:

- ☑ What polymorphism is
- ☑ What the different types of polymorphism are and how they are used
- ☑ What virtual methods are and how they are used
- ☑ How to create object-oriented programs with polymorphism

Polymorphism

Object-oriented languages allow for multiple methods with different implementations but the same name. This is known as **polymorphism** and can come in one of two forms:

1. **Overriding** (which replaces one method with a new method of the same name)
2. **Overloading** (which allows multiple methods with the same name to exist)

Polymorphism is important for object-oriented programming because it means that there can be multiple implementations depending on how it is being used, instead of declaring a new method for each different implementation. This is useful because it allows different parts of a program without needing to know which specific implementation is required, allowing for a more flexible program.

Overriding

One of the most common uses for polymorphism is to allow a child class to 'override' a method in its parent class. This is an example of overriding.

Overriding simply replaces one implementation of a method with another. For example, consider a parent and child classes:

```
class Lizard
    private legs
...
    public procedure new()
        this.legs = 4
        ...
    endclass

class SlowWorm inherits Lizard
...
    public procedure new() //This replaces the 'new' method from Lizard
        this.legs = 0
        ...
endclass
```

The `SlowWorm` class inherits every method and attribute in the `Lizard` class by default, including the constructor method `new`.

Because a `SlowWorm` does not have any legs, it cannot use the same `new` method as `Lizard`; however, most object-oriented programming languages require constructors to have the same name (in this case 'new'), and so the only way a constructor can be defined for the `SlowWorm` class is to override the `new` method of its parent class, `Lizard`.

The constructor is not the only method that can be overridden. Any method that c class is known as a **virtual method**. Virtual methods are declared differently depe language; for example, methods in C# are virtual by default and are made non-vi keyword (which prevents child classes from overriding it), whereas in C# the 'virtu the method to be overridden.

```
class Lizard
...
    public procedure move()
        print("The lizard was ")
    endprocedure
endclass

class SlowWorm inherits Lizard
...
    final public procedure move() //This method cannot be overridden
        print("The lizard slithers")
    endprocedure
endclass
```

This program overrides the virtual method `move` in the `Lizard` class. The inclus definition for the `move` method in the `SlowWorm` class means that if another cla `SlowWorm` it would not be able to override `move` because it is defined as a non-

C# Note

There is no 'final' keyword in C#, instead methods are final by default, so the foll

```
using System;

public class Parent {
    void method() {
        Console.WriteLine("Do something");
    }
}
```

For a method to be overridden, it must be declared virtual, and the 'override' key class when it overrides the method (in the same way as implementing an abstrac

```
using System;

public class Parent {
    virtual void method() {
        Console.WriteLine("Do something");
    }
}

public class Child : Parent {
    override void method() {
        Console.WriteLine("Do something else");
    }
}
```

Note that an error will occur if you attempt to use the override keyword to overri

Overloading

The other type of polymorphism is overloading. Overloading allows multiple met
parameters to use the same name. For example, you may want to make some valu
an object:

```
class Cat
    private age
    private legs

    public procedure new() //This 'new' method takes 1 argument
        this.age = 1
        this.legs = 4
    endprocedure

    public procedure new(age, legs) //A second 'new' method takes 2 argur
        this.age = age
        this.legs = legs
    endprocedure
endclass
...
    mia = new Cat(6)
    percy = new Cat(12, 3) //Both 'new' methods can be called
```

The `Cat` class is defined with two different constructors. The constructor to be us
depending on the arguments that are passed to it; so, when `mia` is instantiated, t
(because one argument is given), but when `percy` is instantiated the second con
arguments are given).

Unlike overriding, which allows one methc ac place of another, overloadir
methods which simply share a na , overloading is not considered to be 'true'

Questions (Polymorphism)

1. Define the term *polymorphism*, and explain why polymorphism is useful.

   .....

   .....

   .....

2. Use the pseudocode below to answer the questions that follow:

```
class Object
    public procedure type()
        print("Type: Object")
    endprocedure

    final public procedure display()
        print(this.value)
    endprocedure
endclass

class Number inherits Object
    ...
    public procedure type()
        print("Type: Number")
    endprocedure

    public function add(num1, num2)
        return(num1 + num2)
    endfunction

    public function add(num1, num2, num3)
        return(num1 + num2 + num3)
    endfunction
endclass
```

   a) Statement: `Object` cannot override any of the methods in `Number`.

   .....

   .....

   b) Identify the name of an overridden method, and explain why it is an over

   .....

   .....

   c) Identify the name of an overloaded method, and explain why it is an over

   .....

   .....

   d) Identify the name of a virtual method, and explain why it is a virtual meth

   .....

   .....

3. Explain when you would choose to make a method virtual.

   .....

   .....

C# Task 4

The *Task 4* skeleton code (`Skeleton`) provides a series of calls to various metho
and provides expected results for each method call.

Use polymorphism (method overriding/overloading) to implement the methods c
skeleton code so that the expected results are produced. No changes should be m

---

Program.cs

```
using System;

public c[illegible]Pr[illegible] {
    publi[illegible]tic void Main() {

        //Circles have one value: radius
        Shape circle1 = new Shape(2);
        Shape circle2 = new Shape("three");

        //Rectangles have two values: width and height
        Shape rectangle1 = new Shape(5, 3);
        Shape rectangle2 = new Shape("seven", "two");

        //Triangles have three values: the lengths of each si
        Shape triangle1 = new Shape("four", "six", "nine");
        Shape triangle2 = new Shape(3, 6, 5);

        /*You can assume that shapes are either given only in
        strings with one of the following values:*/
        //"one", "two", "three", "four", "five", "six", "seve
        //The perimeter of a circle is: 2 x [illegible] x radius
        //The area of a circle is: pi [illegible]s^2
        //You can use 'Math.PI' [illegible] value of pi
        //You can use 'Mat[illegible]ue,2)' to square a value

        circle1.pe[illegible]; //Should print "This circle has a
        ci[illegible]1.[illegible]; //Should print "This circle has an ar

        ci[illegible].perimeter(); //Should print "This circle has a
        circle2.area(); //Should print "This circle has an ar

        //The perimeter of a rectangle is: 2 x (width + heigh
        //The area of a rectangle is: width x height

        rectangle1.perimeter(); //Should print "This rectangl
        rectangle1.area(); //Should print "This rectangle has

        rectangle2.perimeter(); //Should print "This rectangl
        rectangle2.area(); //Should print "This rectangle has

        //The perimeter of a triangle with sides of length a,
        /*The area of a triangle with sides of length a, b an
        the square root of:*/
        //p/2 x (p/2-a) x (p/2-b) x (p/2-[illegible]
        //You can use 'Math.sq[illegible]e,' to get the square ro

        triangle1.pe[illegible]; //Should print "This triangle
        tri[illegible]); //Should print "This triangle has a

        tr[illegible]e2.perimeter(); //Should print "This triangle
        triangle2.area(); //Should print "This triangle has a

        Console.ReadLine();
    }
}
```

Shape.cs

```
public class Shape {
    private StringToNumber strToNum = new StringToNumber();

}
```

StringToNumber.cs

```
class StringToNumber {
    public int convert(string numString) {
        if (string.Equals(numString, "one"))
            return 1;
        else if (string.Equals(numString, "two"))
            return 2;
        else if (string.Equals(numString, "three"))
            return 3;
        else if (string.Equals(numString, "four"))
            return 4;
        else if (string.Equals(numString, "five"))
            return 5;
        else if (string.Equals(numString, "six"))
            return 6;
        else if (string.Equals(numString, "seven"))
            return 7;
        else if (string.Equals(numString, "eight"))
            return 8;
        else if (string.Equals(numString, "nine"))
            return 9;
        else
            return -1;
    }

    public int convert(int number) {
        if (number >= 1 && number <= 9)
            return 9;
        else
            return -1;
    }
}
```

5. CLASS RELATIONSHIPS

In this chapter you will learn:

☑ What class diagrams are, why they are used, and how to create and understa
☑ What composition and aggregation are
☑ When composition should be used over inheritan

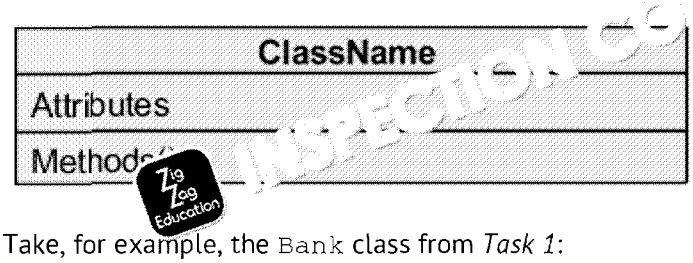
Class diagrams

A very impor as, object-oriented programs is the relationships betwee There are m fferent types of relationship in object-oriented programs, and a scope, it can difficult to understand the relationships between different classe by using Unified Modelling Language (UML) class diagrams – visualisations that methods and relationships that form systems.

In a UML class diagram, the following symbols represent the following visibility a attribute or a method may have:

- Public (+)
- Private (-)
- Protected (#)
- Static (underlined)
- *Abstract* (italics)

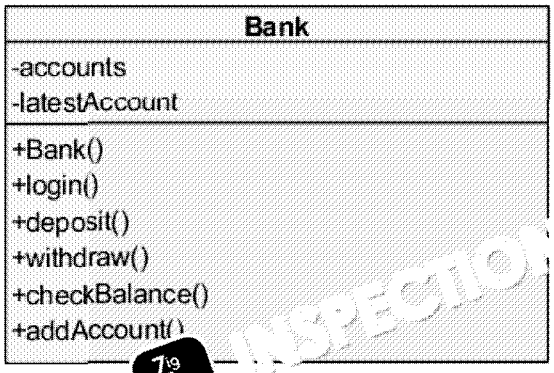
Classes are defined in UML diagrams as follows:



Take, for example, the `Bank` class from *Task 1*:

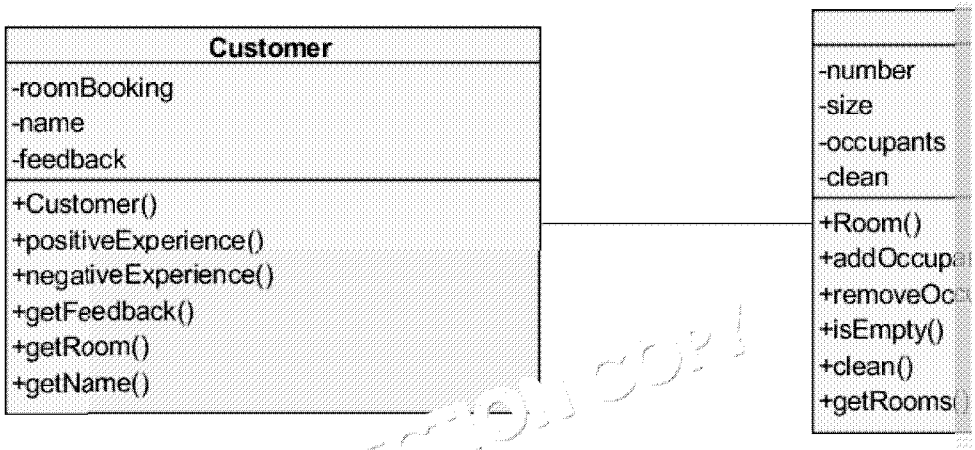
```
class Bank
    private accounts
    private latestAccount

    public procedure new()
        ...
     public function login()
        ...
    public procedure deposit(number)
        ...
    public procedure withdraw(number)
        ...
     public function checkBalance(nu
        ...
     public procedure
        ...
endclass
```

A UML class diagram would represent the `Bank` class as:

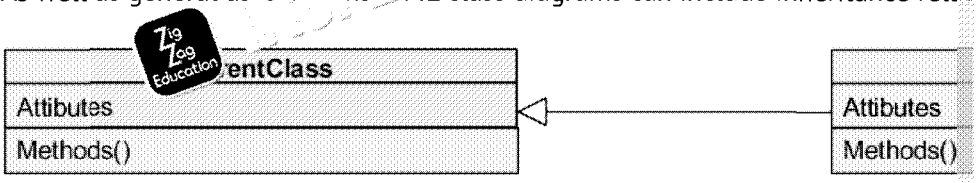


UML class diagrams can sometimes inc
(e.g. '+ +Draw(number)'), but this is not s
mod ether or not to include the par
or understanding the system.

UML class diagrams can also show association between classes. For example, in t
`customer` class is associated with the `room` class because there is a relationship
(i.e. customers have bookings for certain rooms, and each room can contain differ
his is demonstrated by a line connecting these classes in the diagram as follows:



As well as general asso UML class diagrams can include inheritance relat

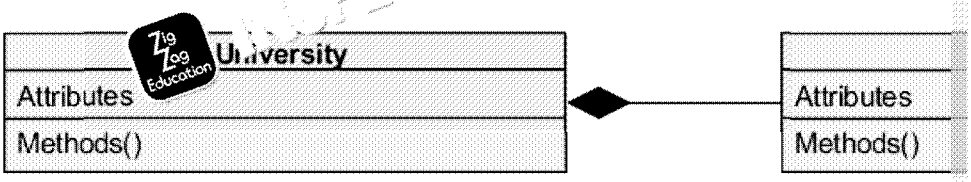


In a UML class diagram, a child class does not need to display the methods and a
parent class, although it can be helpful to display any inherited methods that hav

Composition and aggregation

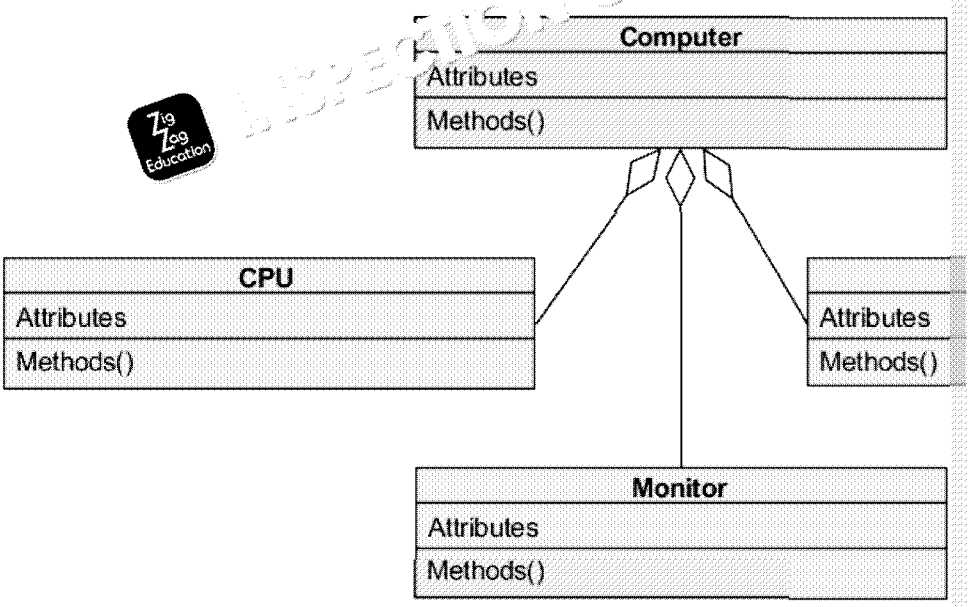
Another type of association between classes that is often seen is called **compositi**
composite object is formed from a collection of different component objects, whe
objects can only exist as part of a composite object.

For example, a university is formed by a collection of different departments. Each de
a department will only exist for as long as the university does. If the university close
This relationship would be shown as follows:



The end of the line with the filled-in diamond is the composite class, and the oth
component class. In this case, `University` is a composite class and `Departme`

There is another type of association whereby several component objects combine
as **aggregation**. Aggregation is very similar to composition, but the component ob
the aggregated object. For example, a computer is made up from a number of sep
component exists as an object in its own right; therefore if you disassemble the c
continue to exist even though the computer no longer exists. This relationship wo



The end of each line with the hollow diamond is the aggregation class, and the o
component classes. In this case, `Computer` is an aggregated class, while `CPU`, `Mo`
component classes.

There is not always a distinction between composition and aggregation. For
between lecturers and universities a composition, as a lecturer is no longer a lect
or is it an aggregation, because a lecturer still exists as a person without a univer
whatever model is most useful for the system that you are designing.

When designing an object-oriented program, there are some basic principles that

- **Encapsulate what varies** – if the implementation of a particular aspect of a pro
being developed, then it should be encapsulated from the rest of the program
- **Favour composition over inheritance** – inheritance relationships can get comp
created, especially if using multiple inheritance. Instead, it is often better to u
can use parts of many different components while avoiding this complexity.
- **Program to interfaces, not implementation** – use abstract methods wherever p
implementation from a parent class so that you don't have to change child cla
the parent class changes.

Questions (Class Relationships)

1. Explain what *UML class diagrams* are and why they are used.

   .....

   .....

   .....

2. State the similarity between *composition* and *aggregation*.

   .....

   .....

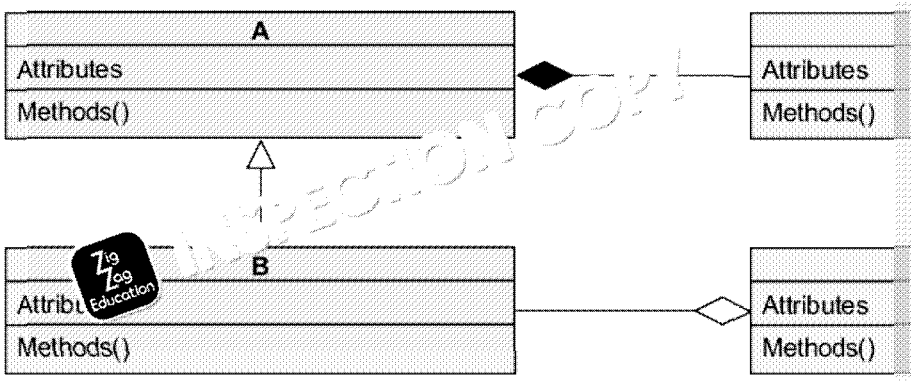
3. Explain the difference between *composition* and *aggregation*.

   .....

   .....

   .....

4. Use the *UML class diagram* below to answer the questions that follow:



   a) Describe the relationship between class A and class B.

   .....

   .....

   b) Describe the relationship between class A and class C.

   .....

   .....

   c) Describe the relationship between class B and class D.

   .....

   .....

UML Class Diagram Tasks

1. Draw a *UML class diagram* for the system created by the '*Task 1 (Answers)*' co
2. Draw a *UML class diagram* for the system created by the '*Task 2 (Answers)*' co
3. Draw a *UML class diagram* for the system created by the '*Task 3 (Answers)*' co
4. Draw a *UML class diagram* for the system created by the '*Task 4 (Answers)*' co

PROJECT 1: FOUR IN A ROW

Introduction

Four in a Row is a game in which players take turns adding tokens to one of the columns on the game board.

Tokens fall to the lowest position in the ch[illegible] umn that does not already have a token in it. Once on [illegible] players has placed four of their tokens in a straight l[illegible] (e[illegible] vertically, horizontally or diagonally), they [illegible] game.

If the boa[illegible]ll and no player has won, then the game ends in a draw.

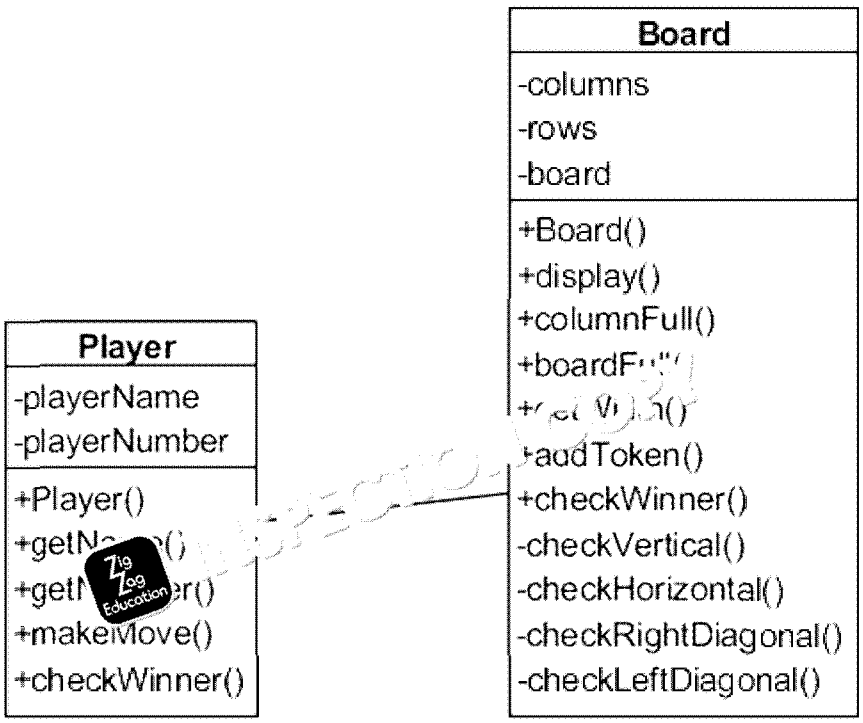
Task

Using the following UML class diagram and class descriptions to help [illegible] in a Row.

- The game must allow for a minimum of two and a maximum o[illegible]
- The game must allow each player to enter their name (duplicat[illegible] accepted)
- The game should give the players the ability to choose how ma[illegible] 10), and how many columns (between fo[illegible] 10) the game bo[illegible]

You may use the Four in a Row [illegible] ode to help you.

UML class [illegible]n[illegible]



Player

Attribute/Method	Description
`playerName`	Specifies this player's name.
`playerNumber`	Specifies the number of this player's token.
`Player()`	Creates a new `Player` object.
`getName()`	Accessor for `playerNam`
`getNumber()`	Accessor for `r` `ber`.
`makeMove()`	As `e` `o` pick a column to place their token in `w`, and adds their token to the given column.
`checkWi` `()`	Returns the player's name if they have won, or "Nobo

Board

Attribute/Method	Description
`columns`	Specifies the number of columns on the game b
`rows`	Specifies the number of rows on the game boar
`board`	Keeps track of which player's token (if any) is st game board.
`Board()`	Creates a new `Board` object.
`display()`	Displays the current state of the board.
`columnFull()`	Checks whether a given column is full.
`boardFull()`	Checks whether the entire board is full.
`getWidth()`	Accessor for `colum`
`addToken()`	Adds a gi `t` `k` a given column.
`checkWinner()`	board for a winner, returning the wi `ere` is no winner.
`checkVe` `al`	Checks for vertical lines of four matching tokens `playerNumber` of the player who made the l lines of four.
`checkHorizontal()`	Checks for horizontal lines of four matching tok `playerNumber` of the player who made the l horizontal lines of four.
`checkRightDiagonal()`	Checks for left-to-right diagonal lines of four m `playerNumber` of the player who made the l to low-right diagonal lines of four.
`checkLeftDiagonal()`	Checks for right-to-left diagonal lines of four m `playerNumber` of the player who made the l right to low-left lines of four.

PROJECT 2: SINKING SHIPS

Introduction

Sinking Ships is a game in which two players place a number of ships of various length on their own board, which is hidden from the other player.

Players then take turns calling out one of the tiles on their opponent's board. Their opponent then tells them whether the shot hit or missed any of their ships.

Once one player has hit every tile that contains a ship on their opponent's board.

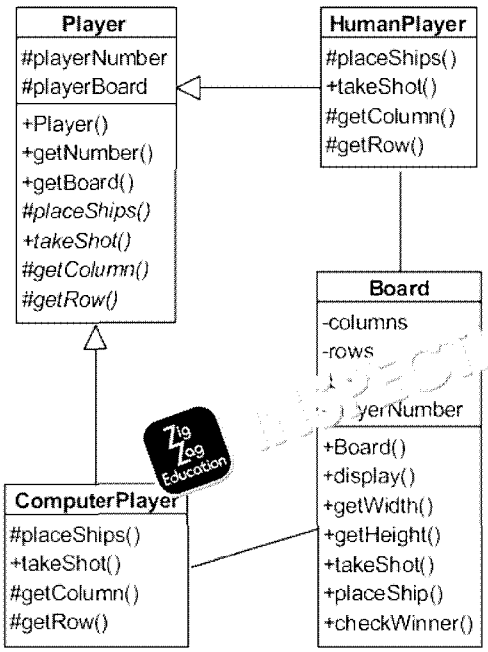
Task

Using the following UML class diagram and class descriptions to help you create *Sinking Ships*.

- The game must have one human player, and one player controlled by the computer (the computer player can be made to target random tiles).
- The player should be able to choose how many rows (between 10 and 26) and columns (between 10 and 26) the game boards should have.
- Players cannot shoot tiles that they have already shot, and before they take their shot, they should have the option to look at their own board (showing their ships and the tiles that the computer player has taken shots at, and whether those shots hit or missed) or the opponent's board (showing the tiles that they have taken shots at, and whether those shots hit or missed, but not showing the opponent's ships).
- Player input should be given as a number, to indicate the column, and a letter, to indicate the row (first row is A, second row is B, etc.).

You may use the Sinking Ships skeleton code to help you.

UML Class Diagram



Class Descriptions

Board

Attribute/Method	Description
`columns`	Specifies the number of columns on the game board.
`rows`	Specifies the number of rows on the game board.
`board`	Keeps track of the ship locations and shot locations on the
`playerNumber`	Specifies the number of the player that the board belongs
`Board()`	Creates a new `Board` object.
`display()`	Displays the current state of the board, only showing ship at their own board.
`getWidth()`	Accessor for `columns`.
`getHeight()`	Accessor for `rows`.
`takeShot()`	Takes a shot at the given location on the board.
`placeShip()`	Asks the player to pick a location on the board and an orie for a ship of a given length until a valid location and orier ship on the board, and then adds the ship to the board as
`checkWinner()`	Checks whether all of the ships on the board have been su

Player

Attribute/Method	Description
`playerNumber`	Specifies this player's number.
`playerBoard`	Specifies this player's board.
`Player()`	Creates a new `Player` object.
`getNumber()`	Accessor for `playerNumber`.
`getBoard()`	Accessor for `playerBoard`.

HumanPlayer

Attribute/Method	Description
`placeShips()`	Places all of this player's ships onto their board, displayin which ship they are placing and to confirm that they have
`takeShot()`	Gets a valid location on a board and takes a shot at it, dis the player another chance to take a shot if they select an
`getColumn()`	Gets a valid column on a board from player input, displayi player another chance to select a column if they select an
`getRow()`	Gets a valid row on a board from player input, displaying another chance to select a row if they select an invalid ro

ComputerPlayer

Attribute/Method	Description
`placeShips()`	Places all of this player's ships onto their board, displayin computer is placing its ships and another message to say have been placed.
`takeShot()`	Gets a valid location on a board and takes a shot at it.
`getColumn()`	Gets a valid column on a board.
`getRow()`	Gets a valid row on a board.

PROJECT 3: CHESS

Introduction

Chess is a game played on an 8 × 8 game board whereby two players h
including one King. Players take turns to move one of their pieces.

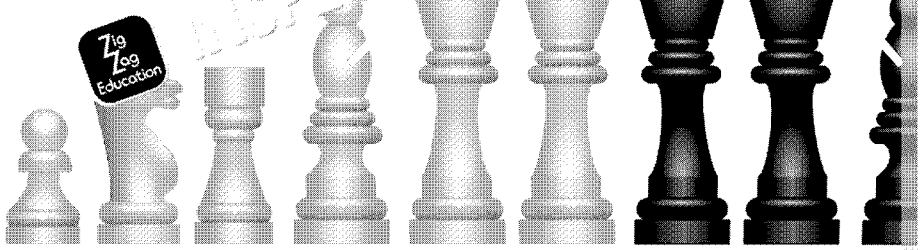
Each piece has different rules for how it ca[illegible] around the board, a
onto a tile containing one of the op[illegible]'s pieces, the opponent's pie

The winner is the pla[illegible] manages to take their opponent's King.

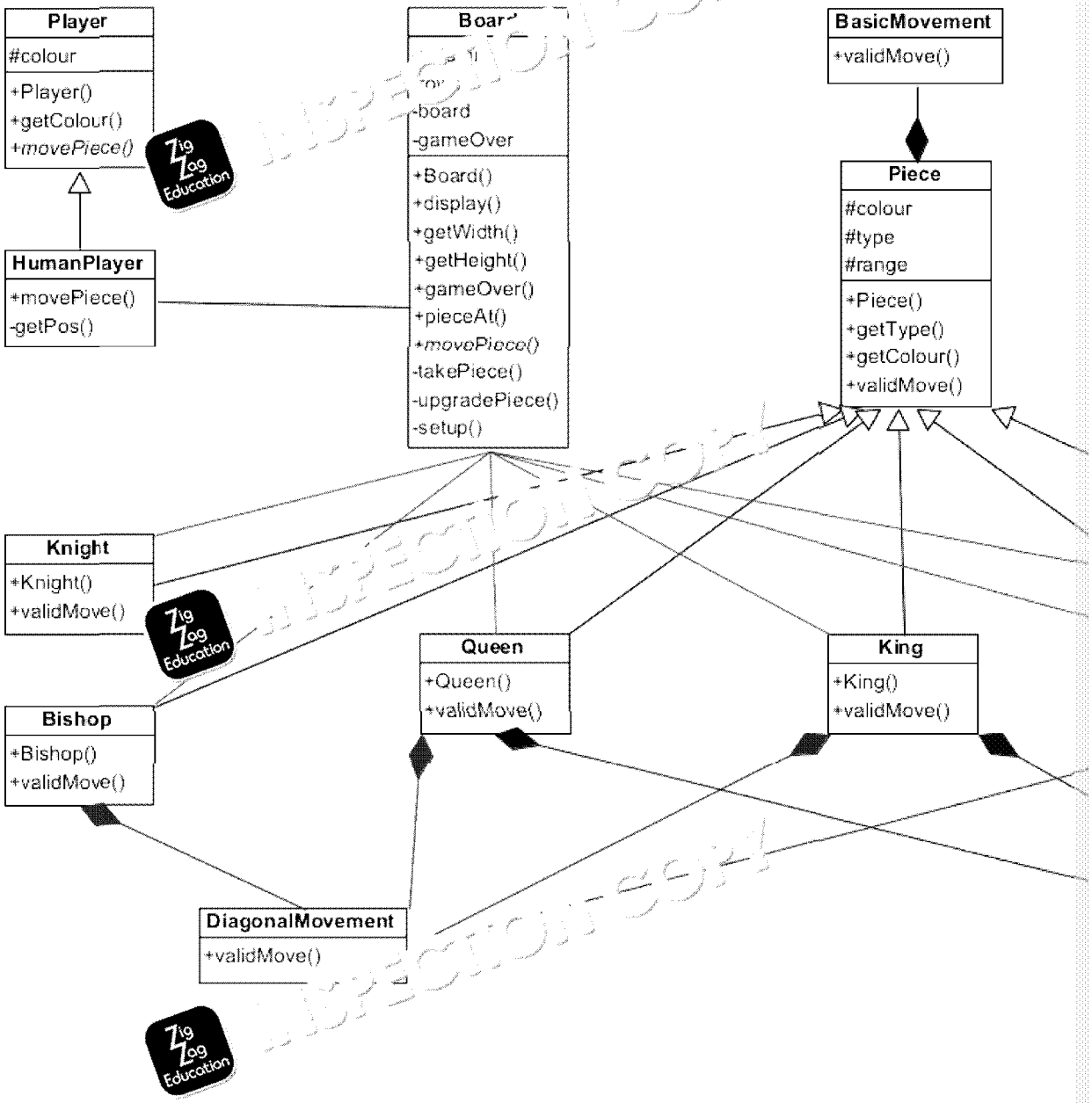
Task

Using the following UML class diagram and class descriptions to help
Chess.

- When it is a player's turn, they must select the tile with the piec
move (if this tile doesn't contain one of their pieces, a message
the player that they don't have a piece on that tile and they sho
selection again) and then select the tile that they would like to
not a valid move, a message should be displayed to tell the play
and they should be asked for their move again).
- Player input should be given as a number, to indicate the colum
and a letter, to indicate the row (first row is A, second row is B,
- You may use the Chess skeleton code t[illegible]u.



UML Class Diagram



Class Descriptions

Board

Attribute/Method	Description
`columns`	Specifies the number of columns on the game board.
`rows`	Specifies the number of rows on the game board.
`board`	Keeps track of the piece locations on the game board.
`gameOver`	Specifies whether or not the game has been won.
`Board()`	Creates a new `Board` object.
`display()`	Displays the current state of the board.
`getWidth()`	Accessor for `columns`.
`getHeight()`	Accessor for `rows`.
`gameOver()`	Accessor for `gameOver`.
`pieceAt()`	Returns the piece at a given location, or none if that locati
`movePiece()`	Tries to make a given move, displaying a message saying v message saying that the move is invalid.
`takePiece()`	Displays a message to say which piece has been taken by `gameOver = True` if the piece which has been taken
`upgradePiece()`	Replaces a pawn with a queen if it reaches the end of the
`setUp()`	Sets up the board with all pieces in their starting positions

Player

Attribute/Method	Description
`colour`	Specifies this player's colour.
`Player()`	Creates a new `Player` object.
`getColour()`	Accessor for `colour`.

HumanPlayer

Attribute/Method	Description
`movePiece()`	Asks a player for the start and end locations of their move locations, then makes the move and returns whether or not
`getPos()`	Checks that the player has given a valid location, returning numbers if one location has been passed to `getPos` or ma two locations have been passed.

Piece

Attribute/Method	Description
`colour`	Specifies the colour of this piece.
`type`	Specifies the name of this type of piece.
`range`	Specifies the number of tiles this piece can move in a turn.
`Piece()`	Creates a new `Piece` object.
`getType()`	Accessor for `type`.
`getColour()`	Accessor for `colour`.
`validMove()`	When moving, a piece must end on a tile on the board, can started on, and cannot land on a tile containing a friendly p

Pawn

Attribute/Method	Description
`Pawn()`	Creates a new `Pawn` object, with `type = "pawn"`.
`validMove()`	A pawn can move one tile straight forward if there is no pie diagonally forward if there is an enemy piece in that locati straight forward if it is in its starting position and there are tiles in front of it.

Knight

Attribute/Method	Description
`Knight()`	Creates a new `Knight` object, with `type = "knight"`
`validMove()`	A knight can move two tiles vertically and one tile horizont one tile vertically. A knight can jump over pieces.

Rook

Attribute/Method	Description
`Rook()`	Creates a new `Rook` object, with `type = "rook"`.
`validMove()`	A rook can move any number of tiles in a straight line as lo between it and the end tile.

Bishop

Attribute/Method	Description
`Bishop()`	Creates a new `Bishop` object, with `type = "bishop"`
`validMove()`	A bishop can move any number of tiles in a diagonal line as between it and the end tile.

Queen

Attribute/Method	Description
`Queen()`	Creates a new `Queen` object, with `type = "Queen"`.
`validMove()`	A Queen can make any move that a rook or bishop can mak

King

Attribute/Method	Description
`King()`	Creates a new `King` object, with `type = "King"`.
`validMove()`	A King can move one tile in any direction.

BasicMovement

Attribute/Method	Description
`validMove()`	Makes sure that the given move meets the basic criteria fo

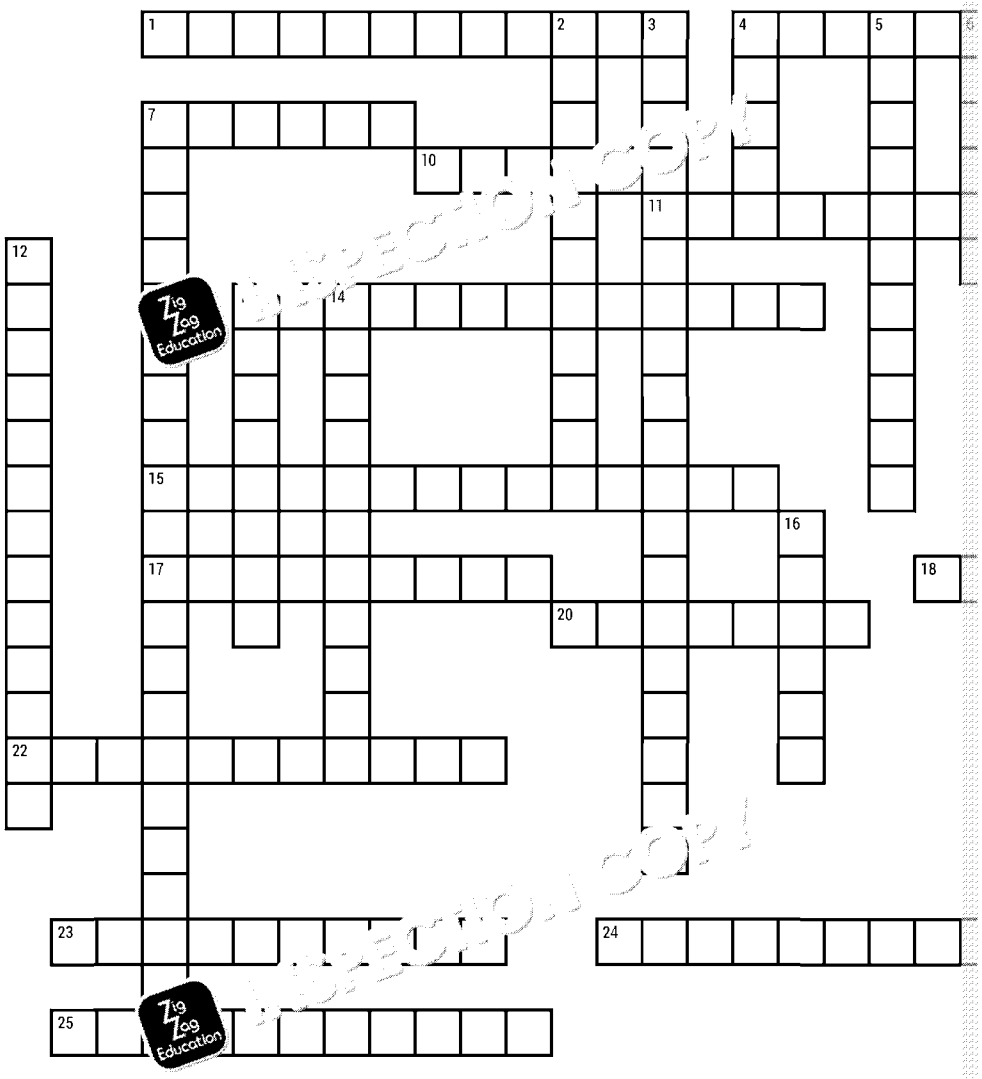
StraightMovement

Attribute/Method	Description
`validMove()`	Makes sure that the given move meets the criteria for a vali straight line.

DiagonalMovement

Attribute/Method	Description
`validMove()`	Makes sure that the given move meets the criteria for a vali diagonal line.

CROSSWORD (OOP CONCEPTS)



Across	Down
1 Allowing different implementations of a method to use the same method name (12)	**2** When one type of o a different type of ob
4 Forming one larger object from multiple smaller objects, where the smaller objects cannot exist separately to the larger object (11)	**3** When a child class (8,11)
7 A method or attribute of a particular class that can be called by any other class (6)	**4** A template defining object from which o
10 An instance of a data structure that has its own attribute values and associated methods (6)	**5** A class that is inher
11 The process of creating an object from a particular class (13)	**6** Class method that c class have been ins
13 A class that contains abstract methods and cannot be instantiated (8,5)	**7** A programming par series of steps that
15 A method whose name and parameters are defined but that d have any implementation (8,6)	**8** A variable or consta object (9)
17 A method or attribute of a particular class tha be called within that class or its child classes	**9** A method that creat
18 A public method that chang or a private attribute (7)	**12** Any method that ca
20 A method or attribu cular class that can only be called within th (7)	**13** A public method tha attribute (8)
22 Creating implementations of the same method that take different a ent types (11)	**14** When a child class a method (5,6)
23 A class that inherits from another class (5,5)	**16** A subroutine belon
24 Grouping together related data and subroutines into classes, and providing controlled access to that class's private attributes (13)	**19** Superseding the im in its child class (10
25 Forming one larger object from multiple smaller objects, where the smaller objects can exist separately to the larger object (11)	**21** A particular approa (8)

ANSWERS

Questions (Chapters 1 – 5)

1 – Fundamentals of Object-Oriented Programming

1. A programming paradigm is a particular style of programming. **(1)**
2. Object-oriented programming groups to gether values and subroutines as objects; programming runs through a series of subroutines in sequence. **(1)**
3. A class is a template that defines what attributes and methods an object should of a class.
4. A static method may be used to perform an operation that corresponds to the class particular object of that class, **(1)** or when the method may be used even if the c **mark**
5. **1 mark** for suitable attributes; **1 mark** for including a constructor method; **1 mark** the time; **1 mark** for including a method to display the time; **1 mark** for displayin **1 mark** for including a method that updates the display time each minute.
   Accept any sensible approach that meets the requirements of the question. For e

```
class DigitalClock
    private hours
    private minutes

    public procedure new(currentHour, currentMinute)
        this.hours = currentHour
        this.minutes = currentMinute
    endprocedure

    public procedure setHour(currentHour)
        this.hours = currentHour
    endprocedure

    public procedure setMinute(currentMinute)
        this.minutes = currentMinute
    endprocedure

    public procedure displayTime(currentMinute)
        print(this.hours + ":" + this.minutes)
    endprocedure

    public procedure newMinute()
        if this.minutes < 60 then
            this.minutes = this.minutes + 1
        else
            this.minutes = 0
            this.hours = this.hours + 1
    endprocedure

    public procedure newHour()
        if this.hours < 24 then
            this.hours = this.hours + 1
        else
            this.hours = 0
    endprocedure
endclass
```

2 – Encapsulation

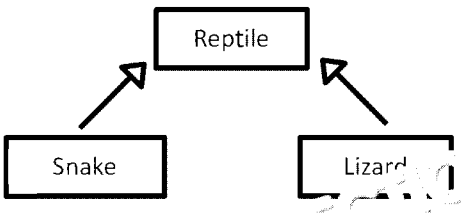
1. Encapsulation is the grouping of data and subroutines that relate to each other.
2. If an attribute or method is public, it will be available to any part of the program private, it can only be accessed from within the class in which it is defined. **(1)**
3. An attribute may be made private to prevent it from bein incorrectly altered el
4. An accessor is a method that returns the value of a private attribute. **(1)** A mutat value of a private attribute. **(1)**
5. Accessors are used to value of a private attribute outside of its class. (value of a private attribute outside of its class. **(1)**
6. If *full* a the attribute is needed outside of the class, the attribute should accessors and mutators, **(1)** because accessors and mutators should be used to attributes. **(1)**

3 – Inheritance and Abstract Methods

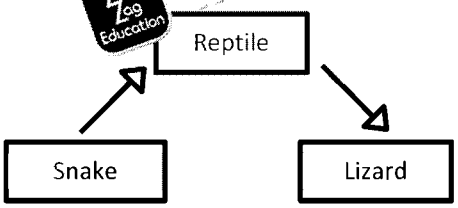
1. Inheritance is when one class uses another class as a base which can then be e
2. 1 mark for showing a correct inheritance relationship; 1 mark if all inheritance r

   Accept any diagram that shows a clear inheritance structure between at least th

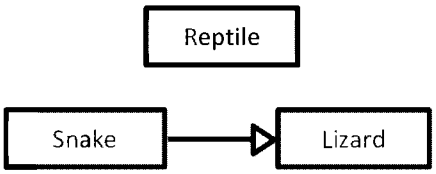
   2 marks: all inheritance relationships are correct between at least three things, e



   1 mark: tance relationship shown, e.g.:



   0 marks: no correct inheritance relationships shown, e.g.:



3.
   a) The parent class is `Guit` hild class is `ElectricGuitar`. **(1 m the parent class d class)**
   b) `El` inherits the attribute `noOfStrings` **(1)** and the meth fr tar. **(Max. 2 marks if `adjustVolume` is given as an inherited m**
4. When a super method is called, the version of the method in the current class's
5. Multiple inheritance can cause conflicts if a particular method has one impleme different implementation in another parent class, **(1)** as the program may be un the method should be inherited. **(1)**

6. An abstract method is a method that has been left undefined in a particular clas any child class that inherits it. **(1)** You might use an abstract method to highligh implemented in child classes but that will have a different implementation in ea

4 – Polymorphism

1. Polymorphism is a way of allowing a particular method to have multiple differe Polymorphism is useful for allowing methods to be called on different data type the specific implementation of the method is **(1**

2. a) `Object` cannot override any of the methods in `Number` because `Object`
   b) `type` is an overloaded method **(1)** because the implementation in the chil in implementation in the parent class `Object`. **(1)**
   c) `type/add` is an overloaded method **(1)** because it has multiple impleme arguments. **(1)**
   d) `type/add` is a virtual method **(1)** because it can be overridden in a child c

3. A method would be made final to prevent child classes that inherit the method f

5 – Class Relationships

1. UML class diagrams are visual representations of object-oriented systems **(1)** th and understanding object-oriented systems. **(1)**

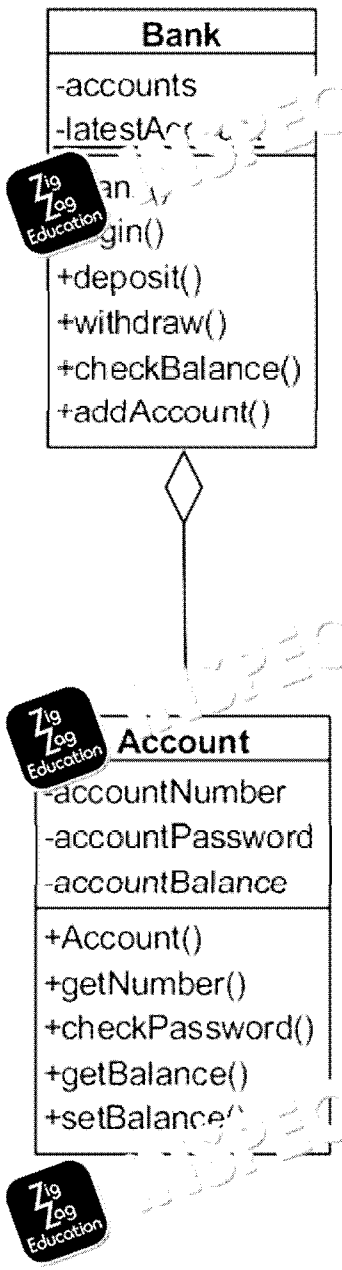
2. Composition and aggregation both form an object from multiple smaller objects

3. The component objects that form a composite object only exist for as long as th whereas the component objects that form an aggregated object can exist as sep aggregated object is destroyed. **(1)**

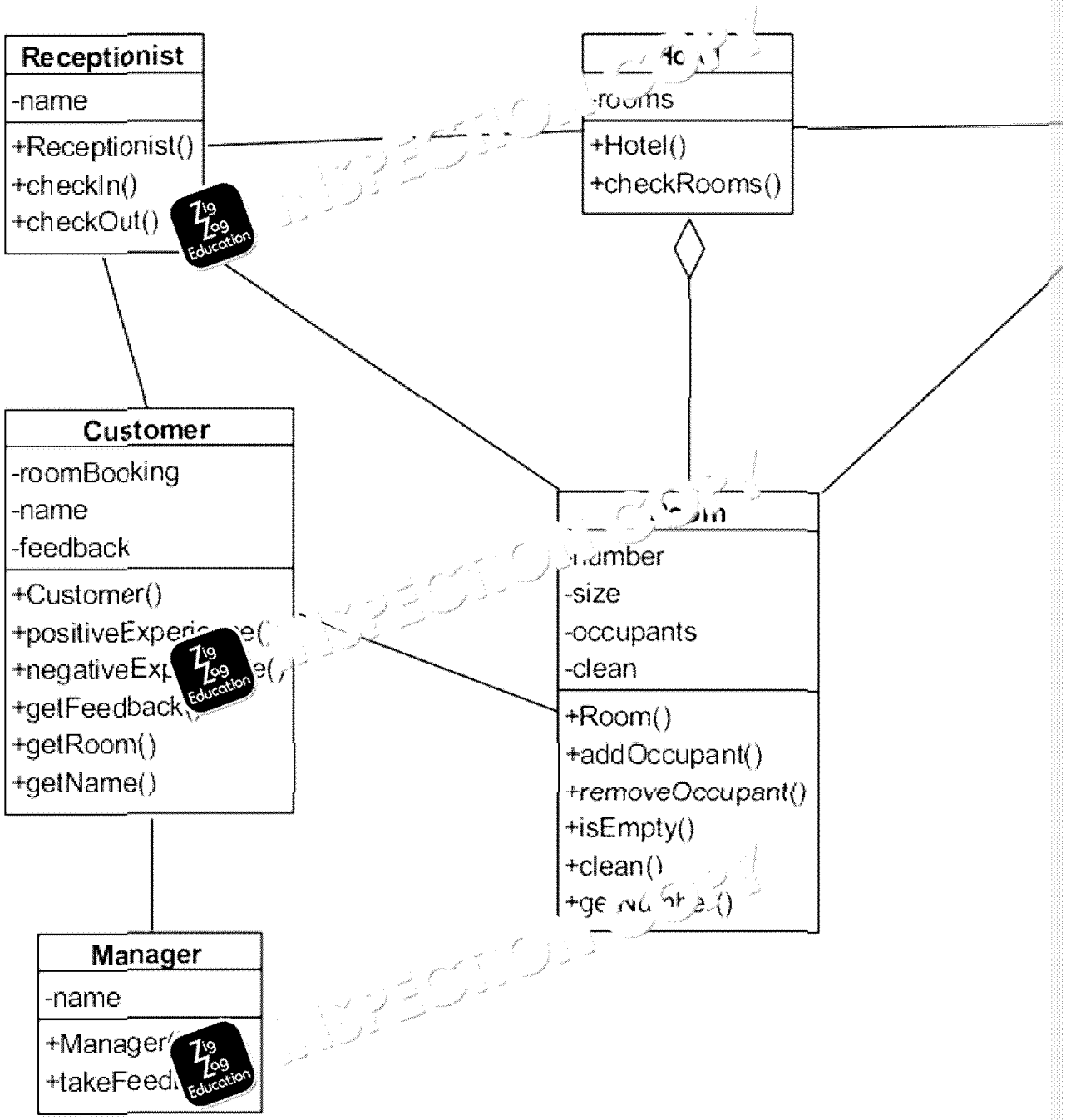
4. a) Class B inherits from Class A (Class A is the parent class, Class B is the chil **relationship, i.e. 'inheritance'; 1 mark for getting classes the correct way ro**
   b) Class A is a composite class formed with Class C component objects. **(1 ma 'composition'; 1 mark for getting classes the correct way round, i.e. 'C form**
   c) Class B is an aggregated class formed with Class B component objects. **(1 m 'aggregation'; 1 mark for getting classes the correct way round, i.e. 'B form**

UML Class Diagram Solutions

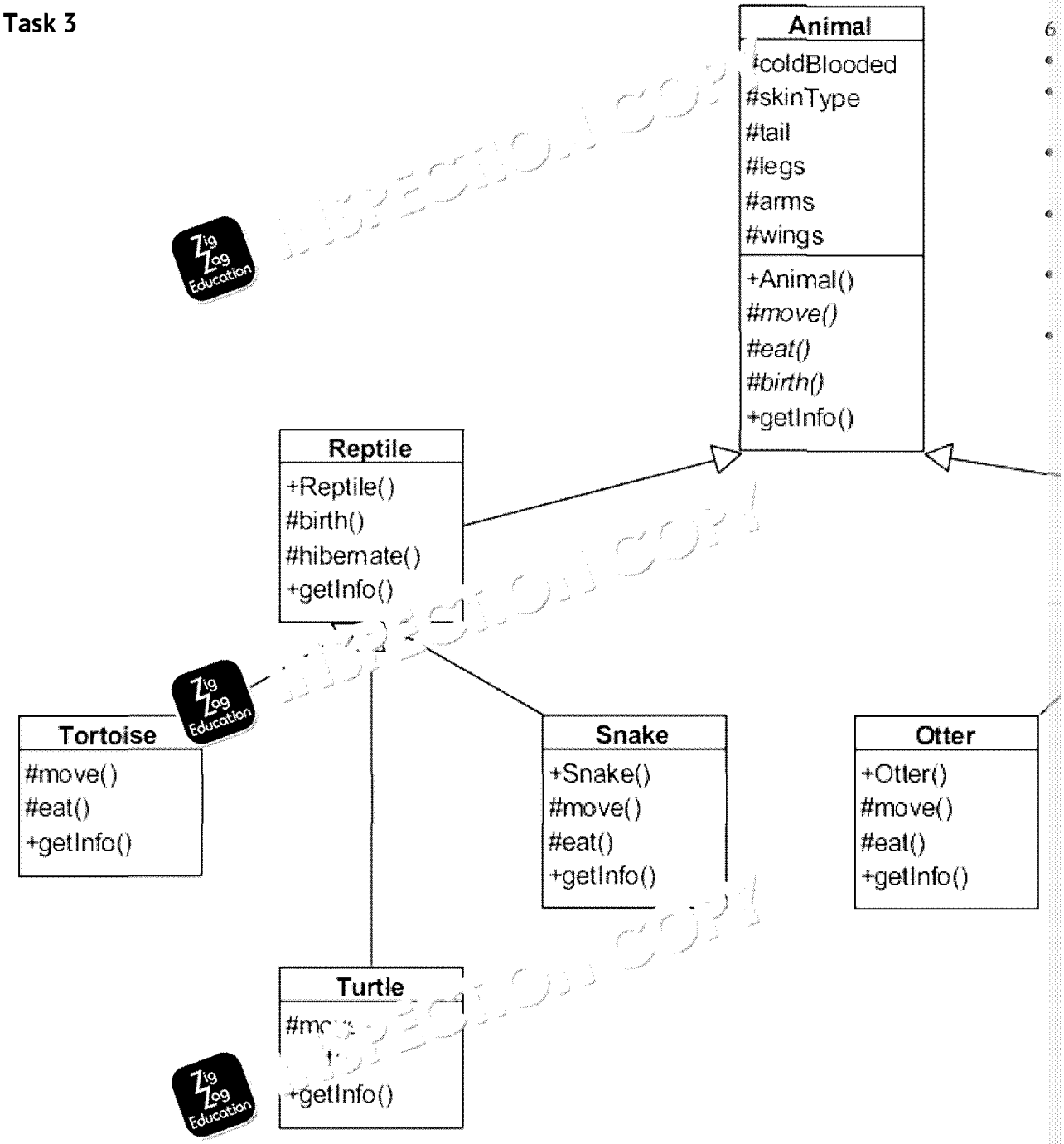
Task 1



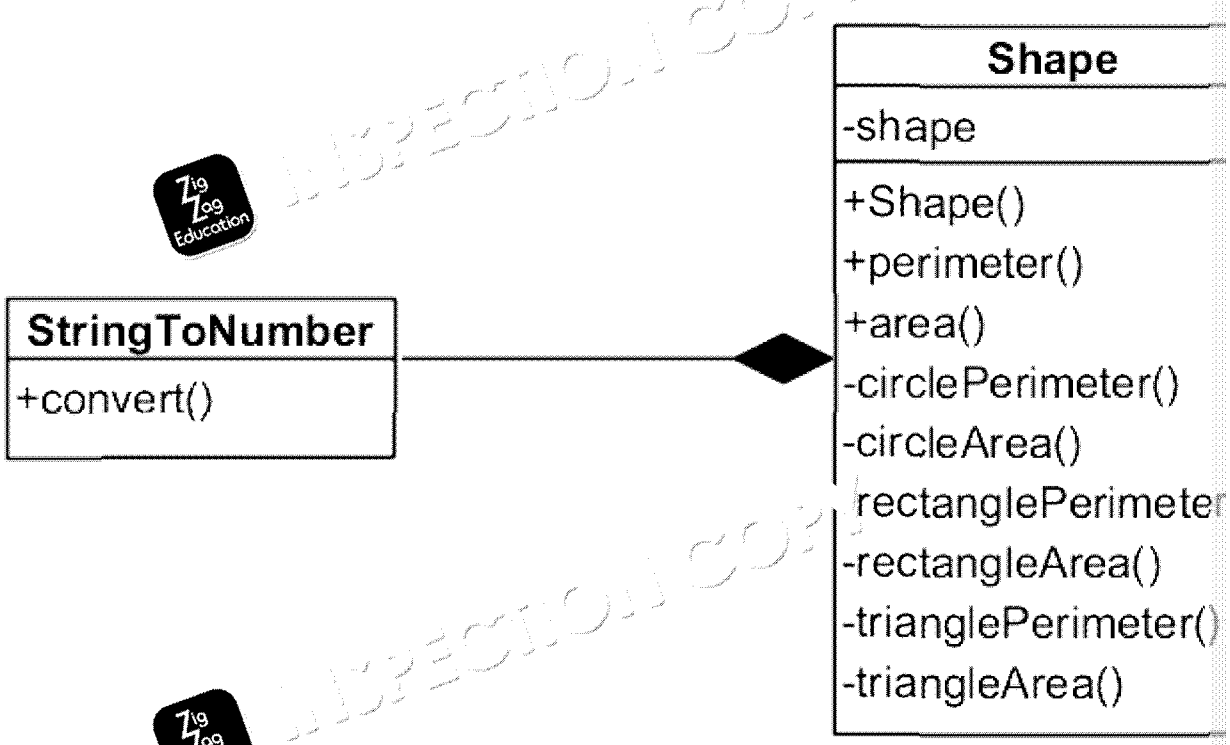
Task 2



Task 3



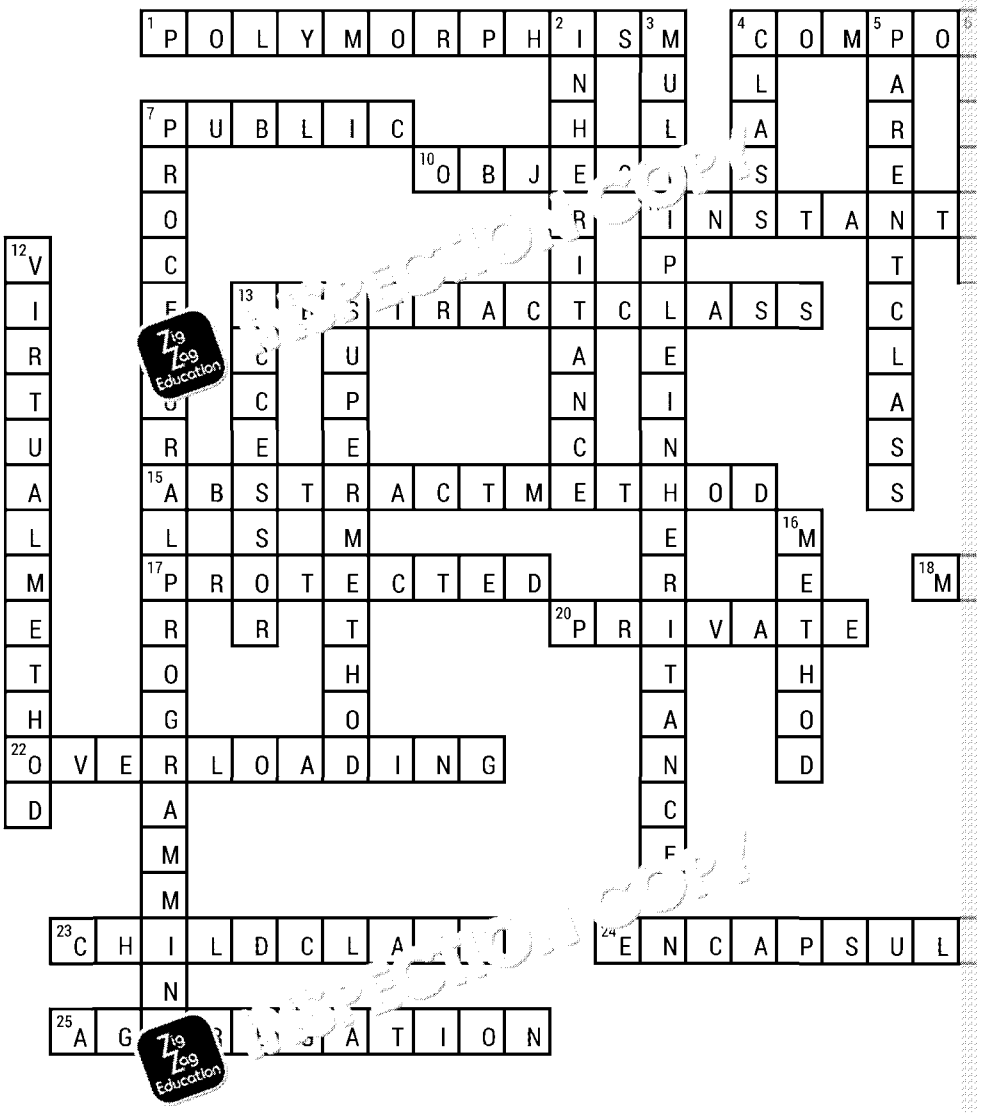
Task 4



4 marks:

- 1 Mark for showing all classes
- 1 Mark for showing all attribute
- 1 Mark for showing all attribute
- 1 Mark for showing the compos

Crossword (OOP Concepts)



Across	Down
1 Allowing different implementations of a method to use the same method name (12)	**2** When one type of o a different type of o
4 Forming one larger object from multiple smaller objects, where the smaller objects cannot exist separately to the larger object (11)	**3** When a child class (8,11)
7 A method or attribute of a particular class that can be called by any other class (6)	**4** A template defining object from which o
10 An instance of a data structure that has its own attribute values and associated methods (6)	**5** A class that is inher
11 The process of creating an object from a particular class (13)	**6** Class method that c class have been ins
13 A class that contains abstract methods and cannot be instantiated (8,5)	**7** A programming par series of steps that
15 A method whose name and parameters are defined but that doesn't have any implementation (8,6)	**8** A variable or const object (9)
17 A method or attribute of a particular class that can only be call within that class or its child classes (9)	**9** A method that creat
18 A public method that changes the value of a private attribute (7)	**12** Any method that ca
20 A method or attribute of a particular class that can only be called within that class (7)	**13** A public method the attribute (8)
22 Creating different implementations of the same method that take different argument types (11)	**14** When a child class a method (5,6)
23 A class that inherits from another class (5,5)	**16** A subroutine belong
24 Grouping together related data and subroutines into classes, and providing controlled access to that class's private attributes (13)	**19** Superseding the im in its child class (10
25 Forming one larger object from multiple smaller objects, where the smaller objects can exist separately to the larger object (11)	**21** A particular approa (8)

GLOSSARY

Abstract Class	A class that contains abstract methods and canno
Abstract Method	A method whose name and parameters are define implementation
Accessor	A public method that retu[illegible] relating to a pri
Aggregation	Forming one la[illegible] from multiple smaller o objects [illegible] separately to the larger object
Attribute	[illegible] or constant belonging to a particular cl
Child Clas	A class that inherits from another class
Class	A template defining the attributes and methods o objects can be created
Composition	Forming one larger object from multiple smaller o objects cannot exist separately to the larger objec
Constructor	A method that creates an object of a particular cla
Encapsulation	Grouping together related data and subroutines in controlled access to that class's private attributes
Inheritance	When one type of object or class adopts functiona object or class
Instantiation	The process of creating an object from a particula
Method	A subroutine belonging to a particular class or obj
Multiple Inheritance	When a child class inh[illegible] m multiple parent c
Mutator	A public method that changes the value of a priva
Object	[illegible]stance of a data structure that has its own a associated methods
Overloading	Creating different implementations of the same m argument types
Overriding	Superseding the implementation of a parent class
Parent Class	A class that is inherited by another class
Polymorphism	Allowing different implementations of a method t name
Private	A method or attribute of a particular class that ca class
Procedural Programming	A programming paradigm that structures a progra are followed in sequence
Paradigm	A particular approach [illegible]ning and creating p
Protected	A method [illegible] of a particular class that ca class [illegible] child classes
Public	[illegible]method or attribute of a particular class that ca class
Static	Class method that can be called even if no objects instantiated
Super Method	When a child class calls its parent class' impleme
Virtual Method	Any method that can be overridden by a child clas