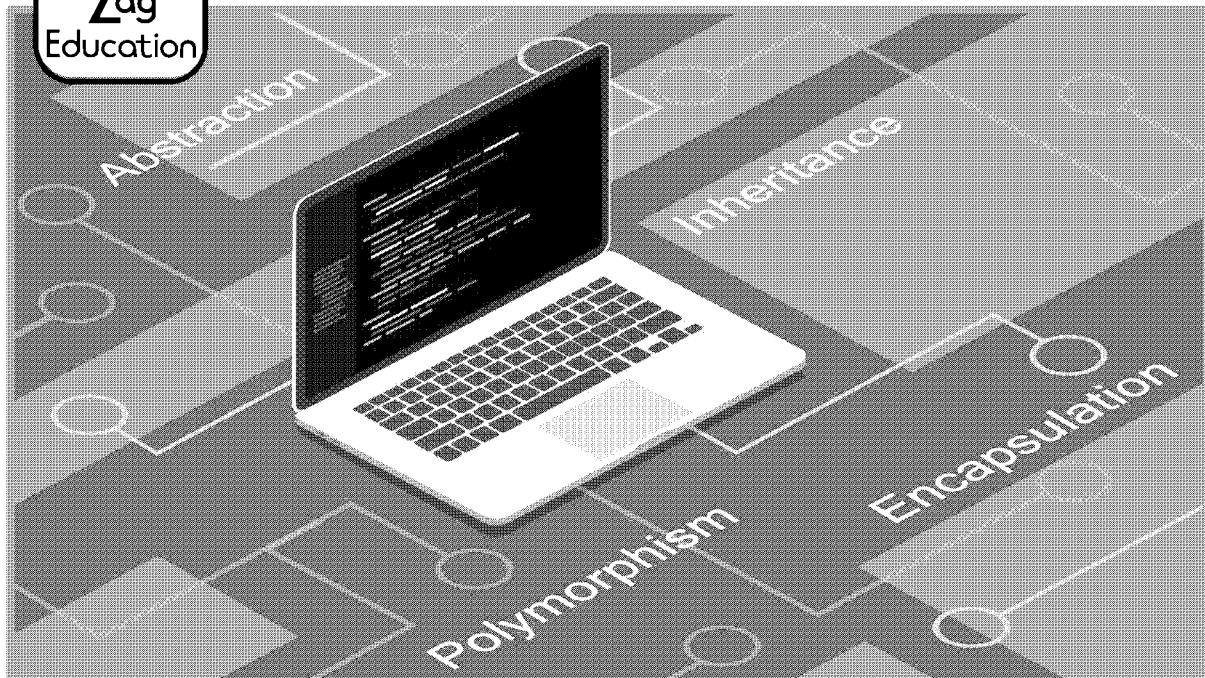




**Computer Science**



# Introduction to Object-Oriented Programming (Python Edition)

Cameron Standing



**DA9/  
9752**

**POD  
9752**

Follow on Twitter:  
**@ZigZagComputing**

**zigzageducation.co.uk**

Publish your  
own work...  
Write to a brief...  
**Register at**  
**publishmenow.co.uk**

# Contents

|                                                              |            |
|--------------------------------------------------------------|------------|
| <b>Thank You for Choosing ZigZag Education .....</b>         | <b>ii</b>  |
| <b>Teacher Feedback Opportunity.....</b>                     | <b>iii</b> |
| <b>Terms and Conditions of Use .....</b>                     | <b>iv</b>  |
| <b>Teacher's Introduction.....</b>                           | <b>1</b>   |
| <b>1. Fundamentals of Object-Oriented Programming .....</b>  | <b>2</b>   |
| Introduction to object-oriented programming.....             | 2          |
| Objects and classes.....                                     | 3          |
| Questions (Fundamentals of Object-Oriented Programming)..... | 5          |
| Python Task 1.....                                           | 6          |
| <b>2. Encapsulation .....</b>                                | <b>8</b>   |
| Encapsulation in object-oriented programming.....            | 8          |
| Private attributes and methods.....                          | 8          |
| Questions (Encapsulation).....                               | 11         |
| Python Task 2.....                                           | 12         |
| <b>3. Inheritance and Abstract Methods .....</b>             | <b>15</b>  |
| Inheritance.....                                             | 15         |
| Abstract classes.....                                        | 17         |
| Questions (Inheritance and Abstract Methods).....            | 20         |
| Python Task 3.....                                           | 22         |
| <b>4. Polymorphism.....</b>                                  | <b>27</b>  |
| Polymorphism.....                                            | 27         |
| Overriding .....                                             | 27         |
| Overloading.....                                             | 28         |
| Questions (Polymorphism) .....                               | 30         |
| Python Task 4.....                                           | 31         |
| <b>5. Class Relationships .....</b>                          | <b>33</b>  |
| Class diagrams .....                                         | 33         |
| Composition and aggregation.....                             | 34         |
| Questions (Class Relationships) .....                        | 36         |
| UML Class Diagram Tasks.....                                 | 37         |
| <b>Python Projects .....</b>                                 | <b>38</b>  |
| Project 1: Four in a Row.....                                | 38         |
| Project 2: Sinking Ships.....                                | 40         |
| Project 3: Chess.....                                        | 42         |
| <b>Crossword (OOP Concepts).....</b>                         | <b>46</b>  |
| <b>Answers .....</b>                                         | <b>47</b>  |
| Questions (Chapters 1–5).....                                | 47         |
| UML Class Diagrams .....                                     | 50         |
| Crossword (OOP Concepts).....                                | 54         |
| <b>Glossary .....</b>                                        | <b>55</b>  |

# 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. 'Python notes' are also included throughout to highlight how Python specifically deals with object-oriented concepts.

Each topic contains pseudocode examples written in these boxes

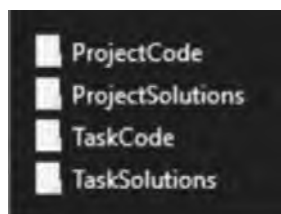
These black boxes show where Python code is being used instead of pseudocode

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).

**Python code files** are provided electronically on the accompanying CD.



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



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

The answer files for both the Python 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 Standing, July 2019

## Free Updates!

Register your email address to receive any future free updates\* made to this resource or other ICT/Computing resources your school has purchased, and details of any promotions for your subject.

\* resulting from minor specification changes, suggestions from teachers and peer reviews, or occasional errors reported by customers

Go to [zzed.uk/freeupdates](http://zzed.uk/freeupdates)

# 1. FUNDAMENTALS OF OBJECT-ORIENTED 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) are
- ✓ What constructors are and how they are used to instantiate a class
- ✓ How to write basic object-oriented programs

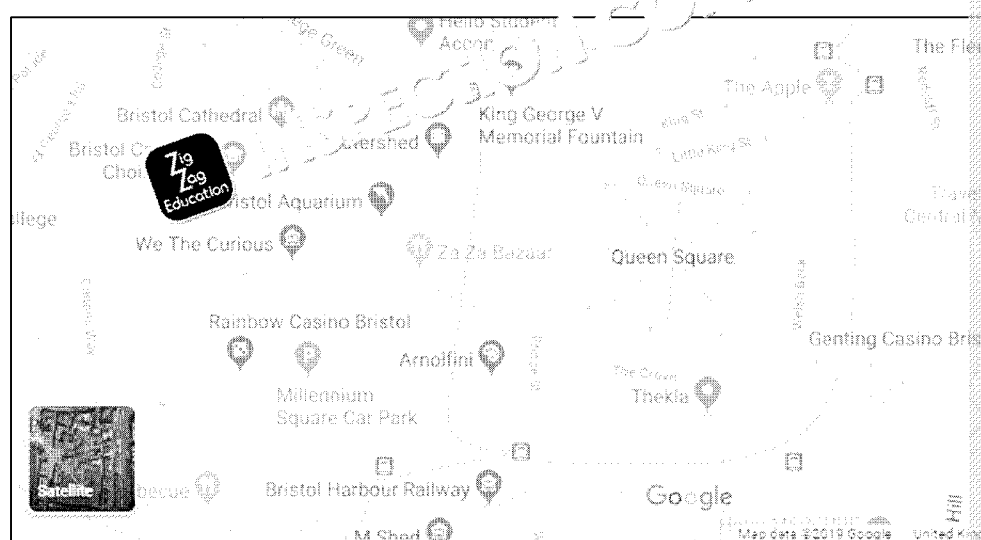


## Introduction to object-oriented programming

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 programming**, every variable, constant and subroutine is defined separately and the relationships between each other.

Another commonly used programming paradigm is **object-oriented programming**, where you define separate **objects** that have their own associated values and subroutines. These 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 (e.g., a restaurant). Each pin object would also have its own associated subroutines, such as a method to get detailed information if the pin is clicked, allowing for the pin's information to be displayed.

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

**COPYRIGHT  
PROTECTED**



## Objects and classes

As object-oriented programs can have many different objects, many of which share common properties, it is not necessary to write code to define the properties of each individual object. Instead, a **class** (known as a **class**) is created.

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

```
class Pin
    private name //Data belonging to this Pin object
    private location
    private markerType

    public procedure new(pinName, pinLocation, pinMarker)
        this.name = pinName //Method to create a new Pin object
        this.location = pinLocation
        this.markerType = pinMarker
    endprocedure
...
endclass
```

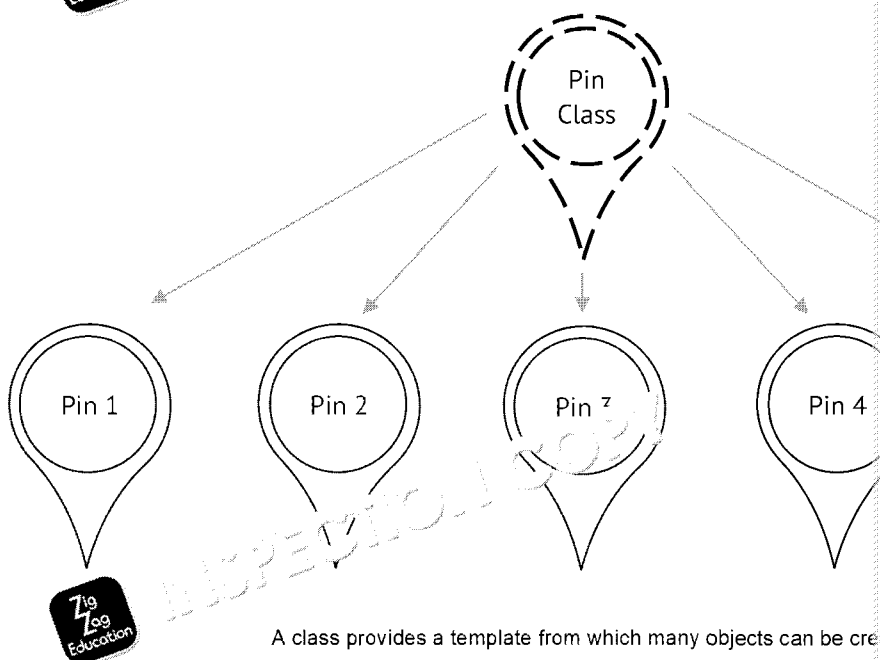
The keyword 'this' is used by an object to refer to itself, so when an object runs the 'new' subroutine, 'this' refers to the value of 'pinName'.

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 and markerType) and the subroutines that a `Pin` object can perform. The values associated with an object are known as **attributes**, and the subroutines associated with an object are known as **methods**.

The `new` method is a **constructor**, a method that creates an object of a particular class. The process of creating an object from a class is known as **instantiation**, and an instance of a class is known as an **object**. We can think of classes as blueprints, and objects as the individual buildings.



A class provides a template from which many objects can be created.

Most attributes and methods are only relevant to a particular object. However, some attributes and methods are relevant to the class as a whole. These are known as **class attributes** and **class methods**.

**COPYRIGHT  
PROTECTED**



For example, the `Pin` class could include a static attribute to count the number of pins created.

```
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 to the class. Similarly, static methods are called by '`ClassName.method()`' whereas non-static methods are called by '`object.method()`'.

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

### Python Note

In Python, non-static attributes are defined in the constructor and do not need to be declared as static, so instead of writing a class as:

```
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)
```

...it would be written as:

```
class Pin:
    public noOfPins = 0

    def __init__(self, pinName, pinLocation, pinMarker):
        self.__name = pinName
        self.__location = pinLocation
        self.__markerType = pinMarker
        Pin.noOfPins = Pin.noOfPins + 1

    ...

pinObject = Pin(name, location, marker)
```

Note that the static attribute `noOfPins` is still declared outside of the constructor. In Python, the '`static`' keyword is not used because any attribute written in the main body of the class is static. The first parameter of any non-static Python method is the object itself (usually named `self`, but you can give this any name you like and it will still work). While the method must declare the first parameter, you do not need to pass it when calling the method (note that the constructor is defined with one parameter, `self`, and it is passed when it is called).

**COPYRIGHT  
PROTECTED**



## Questions (Fundamentals of Object-Oriented Programming)

1. Define the term *programming paradigm*.

.....

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

79  
100



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

.....

.....

.....

.....

.....

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

.....

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



www.zigzageducation.com



INSPECTION COPY

**COPYRIGHT  
PROTECTED**



## Python Task 1

The *Task 1* skeleton code (*Task1.py*) is part of a program that allows a user to create their account, check their balance and deposit or withdraw money from their account. It contains a *Bank* class, that stores information about the individual bank accounts, and a *BankAccount* class that stores information about individual accounts and performs operations on individual accounts when asked to by the user.

Add the missing attributes and method logic to the *Task1* skeleton code to complete the program.

No changes should be made to methods that are already defined, and the methods that are missing should be defined, and the methods that are already defined should be changed, deleted or added to.

### Task1.py

```
class Account:
    def __init__(self, number, password, balance): #A new bank account is defined with a given account number, password and balance
        self.__accountNumber = number
        self.__password = password
        self.__balance = balance

    def getNumber(self):
        #This method should return the account number of this account

    def checkPassword(self, password):
        #This method should check if a given password is equal to the account password

    def getBalance(self):
        #This method should return the balance of this account

    def setBalance(self, balance):
        #This method should set the balance of this account to the given balance

class Bank:
    def __init__(self): #A new bank is defined with a list of accounts that keeps track of the account number of the most recent account
        self.__accounts = []
        self.__latestAccount = -1

    def login(self):
        #This method should ask the user to give their account number and password, and return the account number if they match, or returning -1 if they don't

    def deposit(self, number):
        #This method should ask the user how much money they want to deposit, and correctly update the balance of their account

    def withdraw(self, number):
        #This method should ask the user how much money they want to withdraw, and correctly update the balance of their account

    def checkBalance(self, number):
        #This method should show a message telling the user how much money they have in their account

    def addAccount(self):
        #This method should create a new account with an account number, a password, and a balance of 0. The account should be added to the bank's list of accounts
```

**COPYRIGHT  
PROTECTED**





```

def main():
    bank = Bank()
    loggedIn = False
    quitting = False
    while not loggedIn and not quitting:
        response = input("Do you have an account? (y/n/quit):")
        if response == "y":
            account = bank.login()
            if account != -1:
                loggedIn = True
        elif response == "n":
            bank.addAccount()
        elif response == "quit":
            quitting = True
    while quitting:
        option = input("Press 1 to check your balance\nPress 2 to deposit money\nPress 3 to withdraw money\nPress 4 to exit\n")
        if option == "1":
            bank.checkBalance(account)
        elif option == "2":
            bank.deposit(account)
            bank.checkBalance(account)
        elif option == "3":
            bank.withdraw(account)
            bank.checkBalance(account)
        elif option == "4":
            quitting = True
        else:
            print("Invalid option selected")

if __name__ == '__main__':
    main()

```

**COPYRIGHT  
PROTECTED**



## 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 to work on and understand. In object-oriented programming, encapsulation is achieved by making a class only contain the attributes and methods that it needs, and none of the logic on the internal processing in another class.

Imagine a company has a system that stores various information about different employees. If we use encapsulation, any data can be used or altered in any part of the program. This can lead to:

1. If any errors occur it will be much harder to identify the source of the error, as it could be anywhere. In a properly encapsulated program, any errors will originate either from a part of the program 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, while others won't. In the example of a company's employee information system, an employee should be able to update some of their own records (such as their name or phone number), but 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
```

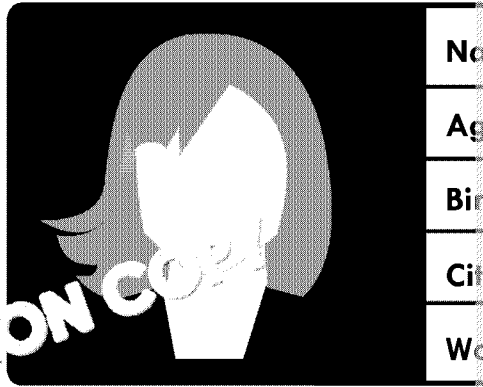
Take the above program from *Task 1*. Notice how the **private** and **public** keywords are used. If a method or attribute is private, it can only be accessed within the class. If an attribute or method is public, it can be accessed by other classes. In *Task 1*, the `accountPassword` attribute in the `Account` class and the `accounts` attribute in the `Bank` class cannot be accessed by other classes. Instead, the `checkPassword` method in the `Account` class is used 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 in a program, 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 using the data).

**COPYRIGHT  
PROTECTED**



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



No  
Ag  
Bir  
City  
Wo

Private attributes are hidden from other parts of the system. There are many situations where you might need to store information about a user object.



When an attribute from another class is needed, instead of making that attribute a public attribute, you can create a public method that returns the value of the attribute. Similarly, to change the value of a public attribute, you can create a public method to change its value rather than directly altering it. Methods that return the value of an attribute are known as **accessors** (or 'getters'), and methods that alter the value of an attribute are known as **mutators** (or 'setters'). These may at first appear unnecessary, but can be useful for the encapsulation and 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 showTime()
    print(clock.currentTime) //currentTime is public so can be called in Display
  endprocedure
endclass
```



The `Display` class directly accesses the clock's `currentTime` attribute to display the time. If you wanted to make a change to how the clock's time is displayed (e.g. by making a 24-hour clock, or changing whether seconds or milliseconds are displayed) and the `currentTime` attribute is used in multiple places, then formatting or other checks would need to be repeated 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 method gives access to currentTime
    return this.currentTime
  endfunction
endclass

class Display()
  clock = new Clock()
  ...
  public procedure showTime()
    print(clock.getTime()) //The public method is called in Display
  endprocedure
endclass
```



INSPECTION COPY

**COPYRIGHT  
PROTECTED**



With this version of the program, the change could be made to the `getTime` accessors and mutators should not just be used to get and set values but to hide information from other classes or limit the ability of other classes to access them.

## Python Note

In Python, there are no 'private' or 'public' keywords; in fact, all methods and attributes are public. However, if the name of an attribute or method begins with a double underscore, it is considered private.

So, for the class:

```
class Class:
    def __init__(self, publicAttribute, privateAttribute):
        self.publicAttribute = publicAttribute
        self.__privateAttribute = privateAttribute
```

If you tried to access the attributes from outside of the class:

```
class = Class("Public", "Private")
print(class.publicAttribute)
print(class.__privateAttribute)
```

... the program would print the value of `class.publicAttribute` ('Public') but not the value of `class.__privateAttribute`. The '\_\_' doesn't make `__privateAttribute` private; instead it changes the name of the attribute when it is called outside of the class. So:

```
print(class._Class__privateAttribute)
```

... would print the value of `class.__privateAttribute` ('Private'). While this attribute can be accessed from outside of the class in which they are defined, you may treat attributes that begin with a double underscore as private.



**COPYRIGHT  
PROTECTED**



## Questions (Encapsulation)

1. Define the term *encapsulation*.

.....

.....

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

.....

.....



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*.

.....

.....



**COPYRIGHT  
PROTECTED**



## Python Task 2

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

Recreate the *Task 2* non-encapsulated code (*Task2-Non-Encapsulated.py*) skeleton code functionality but is properly encapsulated. The skeleton code should be classes for Hotel, Room, Receptionist and Cleaner. The Hotel class should be responsible for processing bookings, the Receptionist should be responsible for checking rooms in and out, and the Cleaner should be made private (although you may add any methods you like).

You may use the provided *Task 2* encapsulated skeleton code (*Task2-Encapsulated.py*) as a reference. It provides a converted main method and constructors for each class that does not need to be implemented.

### Task2-Non-Encapsulated.py

```
def addOccupant(room, occupantIn):
    if len(room[2]) < room[1]:
        room[2].append(occupantIn)
        occupantIn[2] += 1
    else:
        occupantIn[2] -= 1
        return
    if room[3] == True:
        occupantIn[2] += 1
    else:
        occupantIn[2] -= 1
    room[3] = False

def removeOccupant(room, occupantOut):
    index = -1
    for i, occupant in enumerate(room[2]):
        if occupant[1] == occupantOut[1]:
            index = i
    if index != -1:
        del room[2][index]

def takeFeedback(manager, customer):
    if customer[2] > 0:
        print(manager + " says:\n" + customer[1] + " was happy")
    elif customer[2] < 0:
        print(manager + " says:\n" + customer[1] + " was unhappy")
    else:
        print(manager + " says:\n" + customer[1] + " found the room clean")

def cleanRooms(cleaner, hotel):
    for room in hotel:
        if room[2] == []:
            room[3] = True
            print(cleaner + " cleaned room " + str(room[0]))

def checkIn(receptionist, hotel, customer):
    room = hotel[customer[0] - 1]
    addOccupant(room, customer)
    print(receptionist + " checked in " + customer[1])

def checkOut(receptionist, hotel, customer, manager):
    room = hotel[customer[0] - 1]
    removeOccupant(room, customer)
    print(receptionist + " checked out " + customer[1])
    takeFeedback(manager, customer)
```

**COPYRIGHT  
PROTECTED**



```

def main():
    room1 = [1, 1, [], False]
    room2 = [2, 2, [], True]
    room3 = [3, 1, [], False]
    hotel = [room1, room2, room3]
    customer1 = [1, "Mrs. White", 0]
    customer2 = [2, "Mr. Green", 0]
    customer3 = [2, "Miss. Scarlett", 0]
    customer4 = [3, "Mrs. Peacock", 0]
    customer5 = [2, "Prof. Plum", 0]
    customer6 = [3, "Col. Mustard", 0]
    receptionist = "Jan"
    cleaner = "Mick"
    manager = "John"

    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)
    input()

if __name__ == '__main__':
    main()

```

## Task2-Encapsulated.py

```

class Customer:
    def __init__(self, roomBooking, name):
        self.__roomBooking = roomBooking
        self.__name = name
        self.__feedback = 0

class Room:
    def __init__(self, number, size, clean):
        self.__number = number
        self.__size = size
        self.__occupants = []
        self.__clean = clean

class Hotel:
    def __init__(self, name, rooms):
        self.__name = name
        self.__rooms = rooms

    def cleanRooms(self):
        for room in self.__rooms:
            room.__clean = True

class Manager:
    def __init__(self, name):
        self.__name = name

```

**COPYRIGHT  
PROTECTED**



```

class Cleaner:
    def __init__(self, name):
        self.__name = name

class Receptionist:
    def __init__(self, name):
        self.__name = name

def main():
    room1 = Room(1, 1, False)
    room2 = Room(2, 2, True)
    room3 = Room(3, 1, False)
    hotel = Hotel([room1, room2, room3])
    customer1 = Customer(1, "Mrs. White")
    customer2 = Customer(2, "Mr. Green")
    customer3 = Customer(2, "Miss. Scarlett")
    customer4 = Customer(3, "Mrs. Peacock")
    customer5 = Customer(2, "Prof. Plum")
    customer6 = Customer(3, "Col. Mustard")
    receptionist = Receptionist("Jane")
    cleaner = Cleaner("Michael")
    manager = 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)
    input()

if __name__ == '__main__':
    main()

```

**COPYRIGHT  
PROTECTED**





### 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

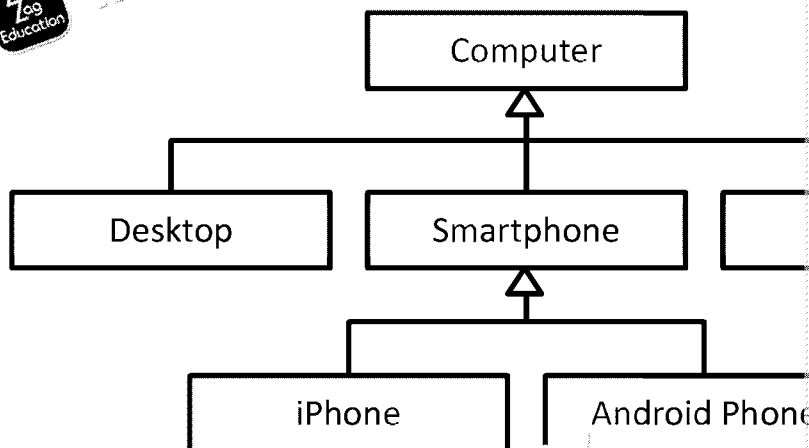
When creating classes, you may begin to realise that some classes have certain similarities, with code being duplicated across classes.

For example, in *Task 2*, the *Manager*, *Receptionist*, *Cleaner* and *Customer* all have 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 you have multiple identical attributes, or even identical methods, a lot of time can be spent copying code 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 of characteristics from a generic computer class. Desktops, smartphones and laptops are all specific types of computer. iPhones and Android phones are specific types of smartphone. Inheritance only works when you have a generic class. If you come up with a definition for a smartphone, then that definition will also work for a laptop, as it is a type of computer.

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

**COPYRIGHT  
PROTECTED**



```

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 attribute is protected, it can only be accessed from within that class or from a class that inherits from it. The `Bird` class defines the attributes and abilities that every bird has. However, we can create a 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 on.

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

In the following code:

```

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

```

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

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

Inheritance is not just used to add to a parent class, it can also be used to change the way a parent class act in the child class:

```

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

```

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

**COPYRIGHT  
PROTECTED**



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

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

A parent class can have multiple child classes that inherit from it, along with the ability to have multiple parent classes.

```
class FlightlessBird inherits Bird, SwimmingBird
  public procedure fly()
    print("This bird cannot fly")
  endprocedure
endclass
```

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

## Abstract classes

While a class can have multiple child classes without causing any problems, if a class has more than one class (known as **multiple inheritance**) there can be issues regarding what methods should inherit. For example, if a class tried to inherit from multiple classes as follows:

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

class B inherits A //Inherits 'method' from A
endclass

class C inherits A //Inherits 'method' from A
  public procedure method() //Changes 'method'
    print("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'. Python won't allow a class to inherit multiple classes, while others try to solve this issue by allowing a class to inherit from multiple parent classes depending on how the classes are arranged (but this can be confusing).

One way to get around the issues with multiple inheritance is to use **abstract classes**. An abstract class is a class that declares methods without specifying how they work. These methods are known as **abstract methods**. For example, the following class would be an abstract class:

```
class AbstractClass
  public procedure concreteMethod() //This method is not abstract
    print("Concrete method")
  endprocedure

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

**COPYRIGHT  
PROTECTED**



An abstract method only defines the method's name and parameters (and the data parameters in some programming languages) of the method. Any class that contains an abstract method must be an 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 from an abstract class must implement all of its parent class' abstract methods using the specified parameters for each method. If a class inherits from an abstract class and does not implement any of the abstract methods undefined, then it will be an abstract class itself.

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

`ConcreteClass` defines the abstract method in `AbstractClass`, and as it has implemented all abstract methods, `ConcreteClass` objects can be created.

Abstract methods are useful because they tell the programmer what functionality is required without defining a generic method in the parent class that may not be useful. For example, in a 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 if `Labrador` 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, which can have multiple interfaces but only one actual class. This avoids the issues with multiple inheritance where multiple classes have implementations that can cause conflicts with each other.

**COPYRIGHT  
PROTECTED**



## Python Note

Python allows for multiple inheritance, using a Method Resolution Order (MRO) to be used in the case of a conflict. For example, with the following program

```
class First:
    def __init__(self):
        self.number = 1

class Second:
    def __init__(self):
        self.number = 2

class Third(First, Second):
    def __init__(self):
        super().__init__()
```

The super constructor method will call the constructor from `First` because the classes are listed from left to right. The MRO becomes much more complicated when the resolution order becomes more complex.

In Python, abstract methods are not supported by default. However, if the `ABC` module is imported, a method can be made abstract as follows:

```
from abc import ABCMeta, abstractmethod
class Dog(metaclass=ABCMeta):
    ...
    @abstractmethod #This line makes the following method abstract
    def thisDog():
        print("This is a dog")
```

By declaring `thisDog` as an abstract method, a `Dog` object cannot be created, and any `Dog` must implement a non-abstract `thisDog` method before an object of that class can be created. Note that the abstract method can still have some functionality (in this case printing "This is a dog"). Child classes in Python can still call the super method on abstract methods, like so:

```
class Husky(Dog):
    ...
    def thisDog():
        super.thisDog()
        print("This dog is a husky")
```

Because Python doesn't have 'public', 'private' or 'protected' keywords, there is no way to indicate the names of any 'protected' attributes or methods should begin with a single underscore. This is a note to the programmers and does not actually make the attribute or method protected.

INSPECTION COPY

**COPYRIGHT  
PROTECTED**



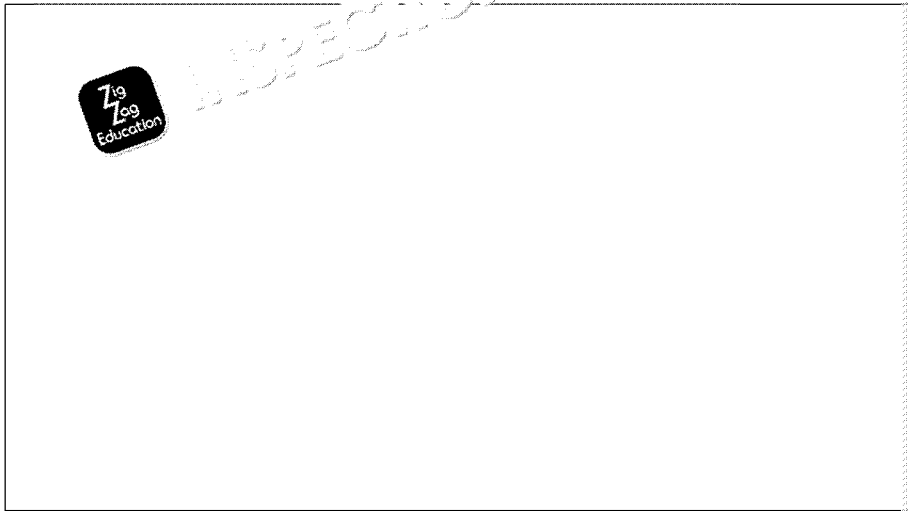
## Questions (Inheritance and Abstract Methods)

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 = 5
    public procedure holdFret()
    ...
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`.

.....

.....

.....

**COPYRIGHT  
PROTECTED**



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*

.....

.....

.....

.....



**COPYRIGHT  
PROTECTED**



## Python Task 3

The *Task 3* skeleton code (`Task3.py`) contains classes for various animals, describe them 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 methods.

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

### Task3.py

```
class Tortoise:
    def __init__(self):
        self._coldBlooded = True
        self._skinType = "scales"
        self._tail = True
        self._legs = 4
        self._arms = 0
        self._wings = 0

    def move(self):
        print("This animal walks")

    def eat(self):
        print("This animal is a herbivore")

    def birth(self):
        print("This animal lays eggs")

    def hibernate(self):
        print("This animal hibernates")

    def get_info(self):
        print("Info:")
        if self._coldBlooded:
            print("This animal is cold-blooded")
        else:
            print("This animal is warm-blooded")
        if self._skinType != None:
            print("This animal is covered in " + self._skinType)
        if self._tail:
            print("This animal has a tail")
        if self._legs > 0:
            print("This animal has " + str(self._legs) + " legs")
        if self._arms > 0:
            print("This animal has " + str(self._arms) + " arms")
        if self._wings > 0:
            print("This animal has " + str(self._wings) + " wings")
        self.move()
        self.eat()
        self.birth()
        self.hibernate()
        print()

class Turtle:
    def __init__(self):
        self._coldBlooded = True
        self._skinType = "scales"
        self._tail = True
        self._legs = 4
        self._arms = 0
        self._wings = 0
```

**COPYRIGHT  
PROTECTED**





```

def move(self):
    print("This animal crawls and swims")

def eat(self):
    print("This animal is an omnivore")

def birth(self):
    print("This animal lays eggs")

def hibernate(self):
    print("This animal hibernates")

def getInfo(self):
    print("Snake:")
    if self._coldBlooded:
        print("This animal is cold-blooded")
    else:
        print("This animal is warm-blooded")
    if self._skinType != None:
        print("This animal is covered in " + self._skinType)
    if self._tail:
        print("This animal has a tail")
    if self._legs > 0:
        print("This animal has " + str(self._legs) + " legs")
    if self._arms > 0:
        print("This animal has " + str(self._arms) + " arms")
    if self._wings > 0:
        print("This animal has " + str(self._wings) + " wings")
    self.move()
    self.eat()
    self.birth()
    self.hibernate()
    print()

```

```
class Snake:
```

```

    def __init__(self):
        self._coldBlooded = True
        self._skinType = "scales"
        self._tail = True
        self._legs = 0
        self._arms = 0
        self._wings = 0

    def move(self):
        print("This animal slithers")

    def eat(self):
        print("This animal is a carnivore")

    def birth(self):
        print("This animal lays eggs")

    def hibernate(self):
        print("This animal hibernates")

    def getInfo(self):
        print("Snake:")
        if self._coldBlooded:
            print("This animal is cold-blooded")
        else:
            print("This animal is warm-blooded")
        if self._skinType != None:
            print("This animal is covered in " + self._skinType)
        if self._tail:
            print("This animal has a tail")

```

**COPYRIGHT  
PROTECTED**



```

if self._legs > 0:
    print("This animal has " + str(self._legs) + " legs")
if self._arms > 0:
    print("This animal has " + str(self._arms) + " arms")
if self._wings > 0:
    print("This animal has " + str(self._wings) + " wings")
self.move()
self.eat()
self.birth()
self.hibernate()
print()

```

**class Otter:**

```

def __init__(self):
    self._coldBlooded = False
    self._skinType = "fur"
    self._tail = True
    self._legs = 4
    self._arms = 0
    self._wings = 0

def move(self):
    print("This animal walks and swims")

def eat(self):
    print("This animal is an omnivore")

def birth(self):
    print("This animal gives birth to live young")

def getInfo(self):
    print("Otter:")
    if self._coldBlooded:
        print("This animal is cold-blooded")
    else:
        print("This animal is warm-blooded")
    if self._skinType != None:
        print("This animal is covered in " + self._skinType)
    if self._tail:
        print("This animal has a tail")
    if self._legs > 0:
        print("This animal has " + str(self._legs) + " legs")
    if self._arms > 0:
        print("This animal has " + str(self._arms) + " arms")
    if self._wings > 0:
        print("This animal has " + str(self._wings) + " wings")
    self.move()
    self.eat()
    self.birth()
    print()

```

**class Gorilla:**

```

def __init__(self):
    self._coldBlooded = False
    self._skinType = "fur"
    self._tail = False
    self._legs = 2
    self._arms = 2
    self._wings = 0

def move(self):
    print("This animal walks and climbs")

def eat(self):
    print("This animal is a herbivore")

```

**COPYRIGHT  
PROTECTED**



```

def birth(self):
    print("This animal gives birth to live young")

def getInfo(self):
    print("Gorilla:")
    if self._coldBlooded:
        print("This animal is cold-blooded")
    else:
        print("This animal is warm-blooded")
    if self._skinType != None:
        print("This animal is covered in " + self._skinType)
    if self._tail:
        print("This animal has a tail")
    if self._legs > 0:
        print("This animal has " + str(self._legs) + " legs")
    if self._arms > 0:
        print("This animal has " + str(self._arms) + " arms")
    if self._wings > 0:
        print("This animal has " + str(self._wings) + " wings")
    self.move()
    self.eat()
    self.birth()
    print()

```

```

class Bat:
    def __init__(self):
        self._coldBlooded = False
        self._skinType = "fur"
        self._tail = True
        self._legs = 2
        self._arms = 0
        self._wings = 2

    def move(self):
        print("This animal is flying")

    def eat(self):
        print("This animal is an omnivore")

    def birth(self):
        print("This animal gives birth to live young")

    def hibernate(self):
        print("This animal hibernates")

    def getInfo(self):
        print("Bat:")
        if self._coldBlooded:
            print("This animal is cold-blooded")
        else:
            print("This animal is warm-blooded")
        if self._skinType != None:
            print("This animal is covered in " + self._skinType)
        if self._tail:
            print("This animal has a tail")
        if self._legs > 0:
            print("This animal has " + str(self._legs) + " legs")
        if self._arms > 0:
            print("This animal has " + str(self._arms) + " arms")
        if self._wings > 0:
            print("This animal has " + str(self._wings) + " wings")
        self.move()
        self.eat()
        self.birth()
        self.hibernate()
        print()

```

**COPYRIGHT  
PROTECTED**



```
def main():
    tortoise = Tortoise()
    turtle = Turtle()
    snake = Snake()
    otter = Otter()
    gorilla = Gorilla()
    bat = Bat()

    tortoise.getInfo()
    turtle.getInfo()
    snake.getInfo()
    otter.getInfo()
    gorilla.getInfo()
    bat.getInfo()
    input()

if __name__ == '__main__':
    main()
```

INSPECTION COPY

**COPYRIGHT  
PROTECTED**



## 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. 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 multiple implementations depending on how it is being used, instead of declaring for each different implementation. This is useful because it allows different parts of a program to be used without needing to know which specific implementation is required, allowing for more flexibility in a program.

### Overriding

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

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

```
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 defined in the `Lizard` class by default, except for the constructor method `new`.

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

**COPYRIGHT  
PROTECTED**



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

```
class Lizard
...
    public procedure move()
        print("The lizard walks")
    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 inclusion of the `final` keyword in the definition for the `move` method in the `SlowWorm` class means that if another class inherits from `SlowWorm` it would not be able to override `move` because it is defined as a non-virtual method.

## Overloading

The other type of polymorphism is overloading. Overloading allows multiple methods to have the same name but different parameters to use the same name. For example, you can have a method to make some variable an object:

```
class Cat
    private legs
    private age

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

    public procedure new(age, legs) //A second 'new' method takes 2 arguments
        this.age = age
        this.legs = legs
    endprocedure
endclass

...
mia = new Cat(6)
percy = new Cat(12, 3) //Both 'new' methods are called
```

The `Cat` class is defined with two different constructors. The constructor to be used is determined by the arguments that are passed to it; so, when `mia` is instantiated, the first constructor is used (because one argument is passed), but when `percy` is instantiated the second constructor is used (because two arguments are passed).

Unlike overriding, which allows one method to act in place of another, overloading allows multiple methods which simply share a name, so overloading is not considered to be 'true' polymorphism.

**COPYRIGHT  
PROTECTED**



## Python Note

Python doesn't use multiple method definitions for overloading, but instead allows parameter values to make certain parameters optional. So, instead of writing a procedure

```
class Cat
  private age
  private legs

  public procedure new(age)
    this.age = age
    this.legs = 4
  endprocedure

  public procedure new(age, legs)
    this.age = age
    this.legs = legs
  endprocedure
endclass
```

It would be written as:

```
class Cat:
  def __init__(self, age, legs=4):
    self.__age = age
    self.__legs = legs
```

This sets the default value of the `legs` parameter to 4, allowing the method to be called with one argument (which will use the default value of `legs`) or two arguments (which will use the provided values). Parameters with default values should be declared after all parameters without default values.

For example:

```
class Cat:
  def __init__(self, legs=4, age):
    self.__age = age
    self.__legs = legs
```

This method would throw an error because if only one argument is passed it is not clear which argument should be used for the first non-default parameter or for the first parameter (this is why Python methods start using multiple parameters with default values).

As Python uses duck typing (meaning that the type of a variable is not important, as long as the methods on the variable are valid), method overloading is not required to allow a method to accept a different number of parameters (i.e. the constructors above will work no matter if the argument given is an integer or another data type).

INSPECTION COPY

**COPYRIGHT  
PROTECTED**



## 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) State why 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 method

.....

.....

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

.....

.....

**COPYRIGHT  
PROTECTED**





## Python Task 4

The *Task 4* skeleton code (*Task4.py*) provides a series of calls to various methods and provides expected results for each method call.

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

### Task4.py

```
import math

class Shape:
    def __init__(self, string):
        if string == "one":
            return 1
        elif string == "two":
            return 2
        elif string == "three":
            return 3
        elif string == "four":
            return 4
        elif string == "five":
            return 5
        elif string == "six":
            return 6
        elif string == "seven":
            return 7
        elif string == "eight":
            return 8
        elif string == "nine":
            return 9
        else:
            return 0

    def perimeter(self):
        pass

    def area(self):
        pass

def main():
    #Circles have one value: radius
    circle1 = Shape(2)
    circle2 = Shape("three")

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

    #Triangles have three values: the lengths of each side of the triangle
    triangle1 = Shape("four", "six", "nine")
    triangle2 = Shape(3, 6, 5)

    #You can assume that shapes are either given only integer values or
    #with one of the following values: "one", "two", "three", "four", "five",
    #seven", "eight", "nine"

    #The perimeter of a circle is 2 * pi * radius
    #The area of a circle is pi * radius^2
    #You can use 'math.pi' as the value of pi
    #You can use 'math.pow' to square a value

    circle1.perimeter() #Should print "This circle has a perimeter of 12.566370614359172"
    circle1.area() #Should print "This circle has an area of 6.283185307179586"

    circle2.perimeter() #Should print "This circle has a perimeter of 18.84955592153879"
    circle2.area() #Should print "This circle has an area of 28.274333882308136"
```

INSPECTION COPY

**COPYRIGHT  
PROTECTED**



```
#The perimeter of a rectangle is: 2 x (width + height)
#The area of a rectangle is: width x height
```

```
rectangle1.perimeter() #Should print "This rectangle has a
rectangle1.area() #Should print "This rectangle has an area
```

```
rectangle2.perimeter() #Should print "This rectangle has a
rectangle2.area() #Should print "This rectangle has an area
```

```
#The perimeter of a triangle with sides of length a, b and c is
#The area of a triangle with sides of length a, b and c is
#square root of: p/2 x (p/2-a) x (p/2-b) x (p/2-c).
#You can use 'math.sqrt(value)' to get the square root of
```

```
triangle1.perimeter() #Should print "This triangle has a
triangle1.area() #Should print "This triangle has an area
```

```
triangle2.perimeter() #Should print "This triangle has a
triangle2.area() #Should print "This triangle has an area
```

```
input()
```

```
if __name__ == '__main__':
    main()
```

INSPECTION COPY



**COPYRIGHT  
PROTECTED**



## 5. CLASS RELATIONSHIPS

In this chapter you will learn:

- ✓ What class diagrams are, why they are used, and how to create and understand them
- ✓ What composition and aggregation are
- ✓ When composition should be used over inheritance

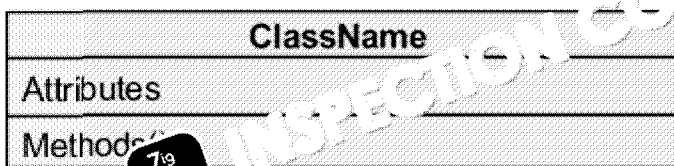
### Class diagrams

A very important aspect of object-oriented programs is the relationships between classes. There are many different types of relationship in object-oriented programs, and as a result, it can be difficult to understand the relationships between different classes by using Unified Modelling Language (UML) class diagrams – visualisations that show the classes, methods and relationships that form systems.

In a UML class diagram, the following symbols represent the following visibility an 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*:

```

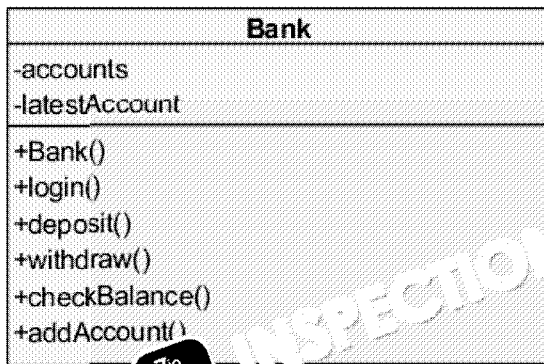
class Bank
    private accounts
    private latestAccount

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

**COPYRIGHT  
PROTECTED**

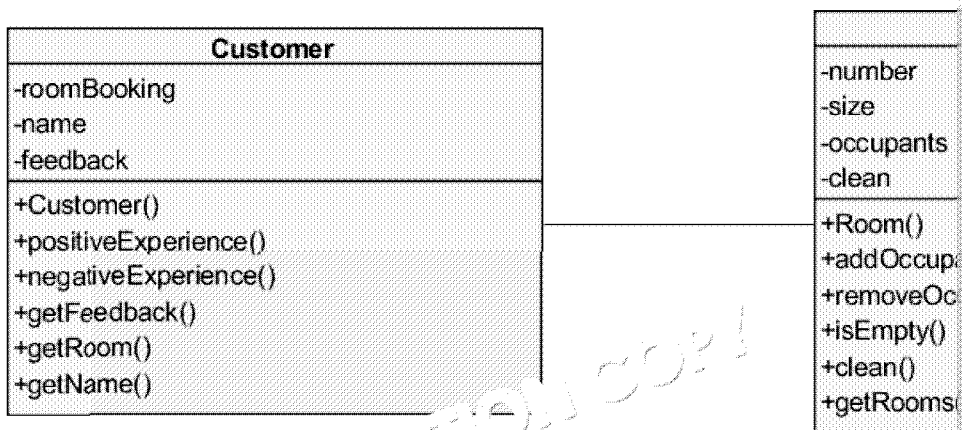


A UML class diagram would represent the `Bank` class as:

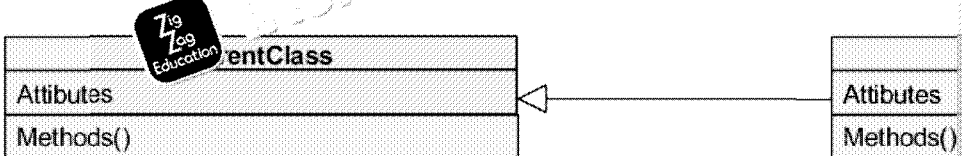


UML class diagrams can sometimes include associations (e.g. "has a" or "owns"), but this is not strictly necessary. Whether or not to include the associations is a matter of understanding the system.

UML class diagrams can also show association between classes. For example, in the `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 different bookings). This is demonstrated by a line connecting these classes in the diagram as follows:



As well as general associations, UML class diagrams can include inheritance relationships.



In a UML class diagram, a child class does not need to display the methods and attributes of the parent class, although it can be helpful to display any inherited methods that have been overridden.

## Python Note

Python allows for 'abstract' methods that contain functionality; however, because they are not implemented, it may be more useful to show them as non-abstract in a UML diagram. To show a method as abstract if it does not have any functionality but is just used as a placeholder, you can use a dashed line. The choice depends on what you find most useful.

## Composition and Association

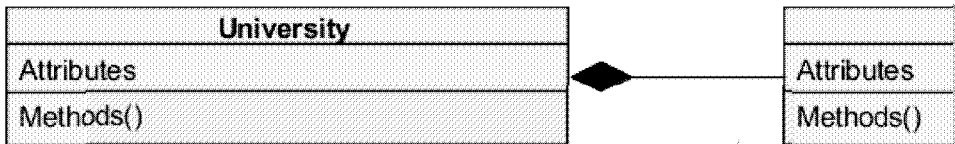
Another type of relationship between classes that is often seen is called **composition**. A composite object is formed from a collection of different component objects, where the component objects can only exist as part of a composite object.

For example, a university is formed by a collection of different departments. Each department will only exist for as long as the university does. If the university closes, all departments will also close.

COPYRIGHT  
PROTECTED

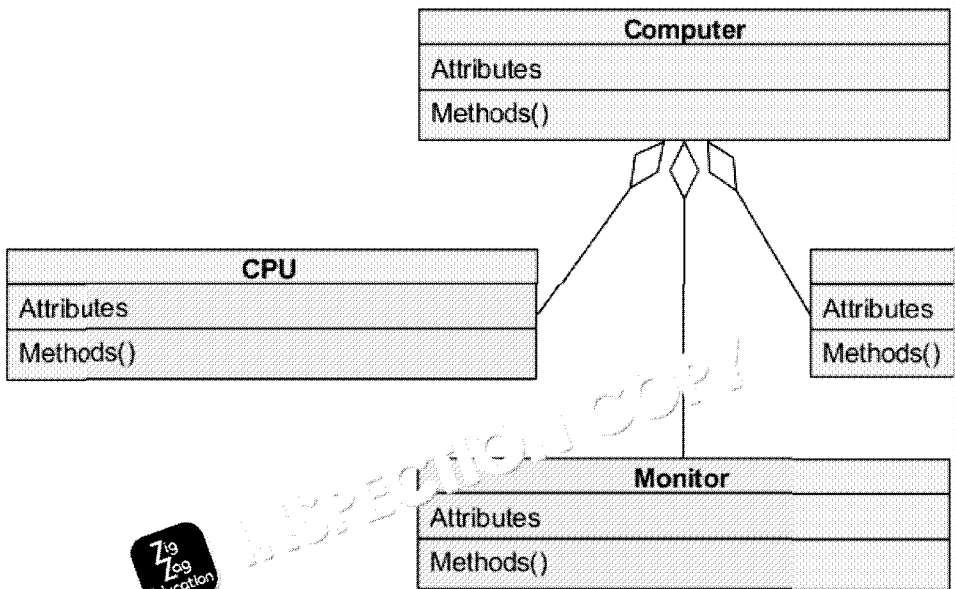


This relationship would be shown as follows:



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

There is another type of relationship whereby several component objects combine to form a new object. This relationship is very similar to composition, but the component objects continue to exist as objects in their own right; therefore, if you disassemble the composite object, the component objects continue to exist even though the composite object no longer does. This relationship is called aggregation.



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

There is not always a clear distinction between composition and aggregation. For example, the relationship between lecturers and universities is a composition, as a lecturer is no longer a lecturer if they are not part of a university. However, it is an aggregation, because a lecturer still exists as a person without a university. The model is whatever model is most useful for the system that you are designing.

When designing an object-oriented program, there are some basic principles that you should follow:

- **Encapsulate what varies** – if the implementation of a particular aspect of a program is likely to change, then it should be encapsulated in a separate module.
- **Favour composition over inheritance** – inheritance relationships can get complicated, especially if you have multiple inheritance. Instead, it is often better to use composition. You can use composition to reuse code from different components while avoiding this complexity.
- **Program to interfaces, not implementation** – use abstract methods wherever possible. This allows you to implement a method from a parent class so that you don't need to change child classes when the parent class changes.

**COPYRIGHT  
PROTECTED**



## Questions (Class Relationships)

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

.....

.....

.....

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

.....

.....

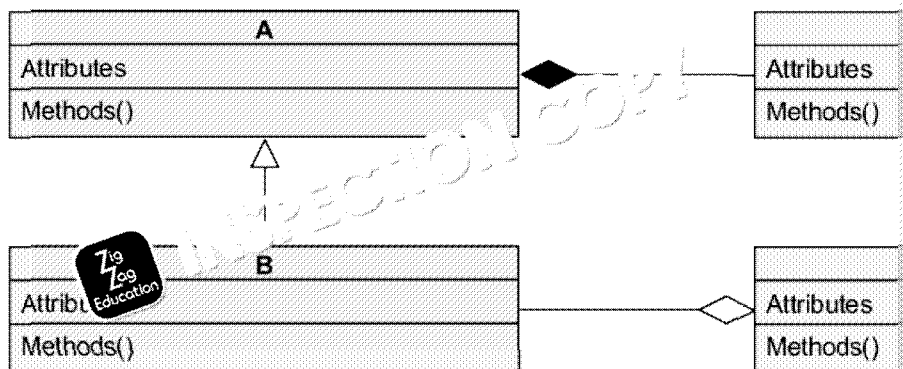
3. Explain the difference between *composition* and *aggregation*.

.....

.....

.....

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.

.....

.....

**COPYRIGHT  
PROTECTED**



## UML Class Diagram Tasks

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



**COPYRIGHT  
PROTECTED**



# 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 chosen column that does not already have a token in it. Once one of the players has placed four of their tokens in a straight line (either vertically, horizontally or diagonally), they win the game.*

*If the board is full and no player has won, then the game ends in a draw.*



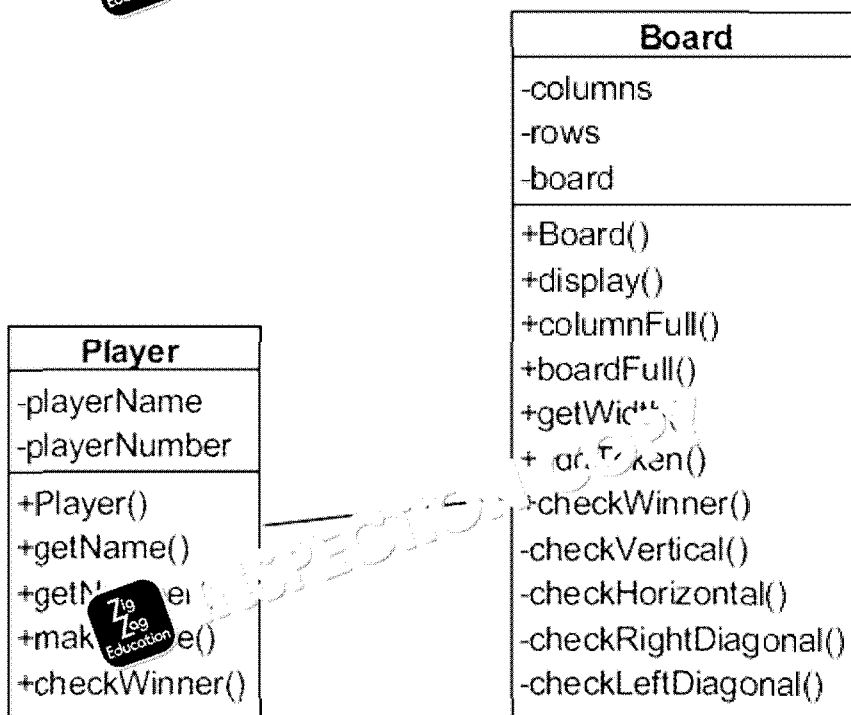
## Task

Using the following UML class diagram and class descriptions to help you in a Row.

- The game must allow for a minimum of two and a maximum of ten players.
- The game must allow each player to enter their name (duplicate names accepted)
- The game should give the players the ability to choose how many rows (between four and 10), and how many columns (between four and 10) the game board should be.

You may use the Four in a Row starter code to help you.

UML class diagram



**COPYRIGHT  
PROTECTED**





## 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 playerName.                                                                            |
| getNumber()      | Accessor for playerNumber.                                                                          |
| makeMove()       | Asks the player to pick a column to place their token in, and adds their token to the given column. |
| checkWin()       | Returns the player's name if they have won, or "Nobody" if there is no winner.                      |

## 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 which player's token (if any) is stored in each cell of the 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 columns.                                                                                                                                                          |
| addToken()           | Adds a given token to a given column.                                                                                                                                          |
| checkWinner()        | Checks the board for a winner, returning the winner's name if there is no winner.                                                                                              |
| checkVertical()      | Checks for vertical lines of four matching tokens, returning the playerNumber of the player who made the line if there are lines of four.                                      |
| checkHorizontal()    | Checks for horizontal lines of four matching tokens, returning the playerNumber of the player who made the line if there are horizontal lines of four.                         |
| checkRightDiagonal() | Checks for left-to-right diagonal lines of four matching tokens, returning the playerNumber of the player who made the line if there are left-to-right diagonal lines of four. |
| checkLeftDiagonal()  | Checks for right-to-left diagonal lines of four matching tokens, returning the playerNumber of the player who made the line if there are right to low-left lines of four.      |

**COPYRIGHT  
PROTECTED**



# 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 or more lines on their opponent's board. Their opponent then tells them whether the shot hit or missed any of the ships.*

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

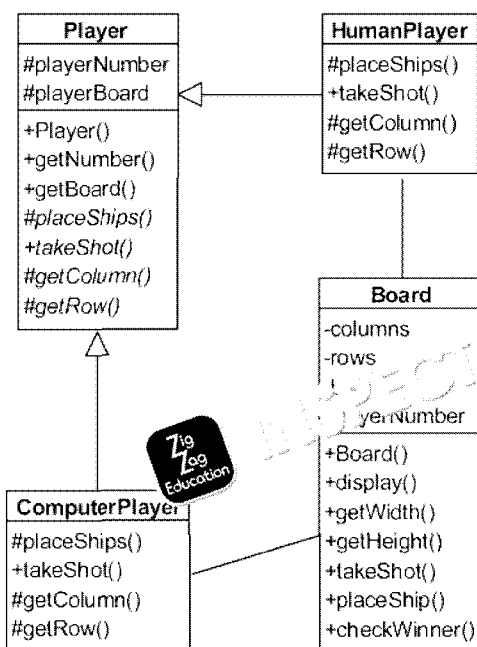
## Task

Using the following UML class diagram and class descriptions to help you design the Sinking Ships game.

- The game must have one human player, and one player controlled by a computer (the computer player can be made to target random tiles).
- The player should be able to choose how many rows (between 10 and 20) and columns (between 10 and 26) the game boards should have.
- Players cannot shoot tiles that they have already shot, and before taking a shot, they should have the option to look at their own board (to see their ships and the tiles that the computer player has taken shots at), or the opponent's board (showing the tiles that have been 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



**COPYRIGHT  
PROTECTED**



## 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 location and shot locations on the board.                                                                                                                                            |
| playerNumber     | Specifies the number of the player that the board belongs to.                                                                                                                                                |
| Board()          | Creates a new Board object.                                                                                                                                                                                  |
| display()        | Displays the current state of the board, only showing ship locations 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 orientation for a ship of a given length until a valid location and orientation ship on the board, and then adds the ship to the board as a new ship. |
| checkWinner()    | Checks whether all of the ships on the board have been sunk.                                                                                                                                                 |

### 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, displaying the board and asking which ship they are placing and to confirm that they have placed it.              |
| takeShot()       | Gets a valid location on a board and takes a shot at it, displaying the board and giving the player another chance to take a shot if they select an invalid location. |
| getColumn()      | Gets a valid column on a board from player input, displaying the board and giving player another chance to select a column if they select an invalid column.          |
| getRow()         | Gets a valid row on a board from player input, displaying the board and giving another chance to select a row if they select an invalid row.                          |

### ComputerPlayer

| Attribute/Method | Description                                                                                                                                              |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| placeShips()     | Places all of this player's ships onto their board, displaying the board and asking which ship they are placing and to confirm that they have placed it. |
| 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.                                                                                                                             |

**COPYRIGHT  
PROTECTED**



# PROJECT 3: CHESS

## Introduction

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

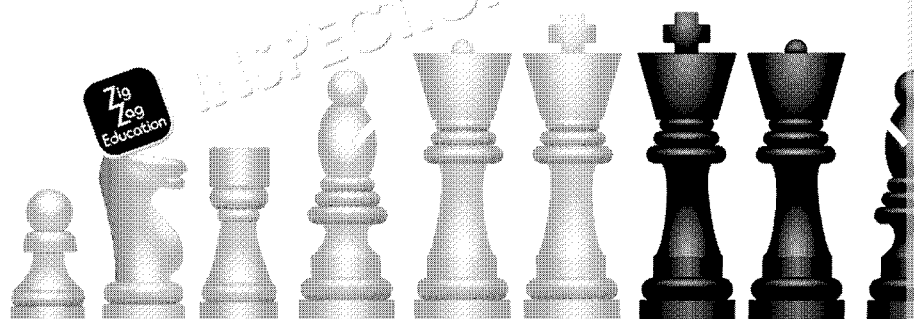
Each piece has different rules for how it can move around the board, and if it moves onto a tile containing one of the opponent's pieces, the opponent's piece is removed.

The winner is the player who manages to take their opponent's King.

## Task

Using the following UML class diagram and class descriptions to help you, design a Chess game.

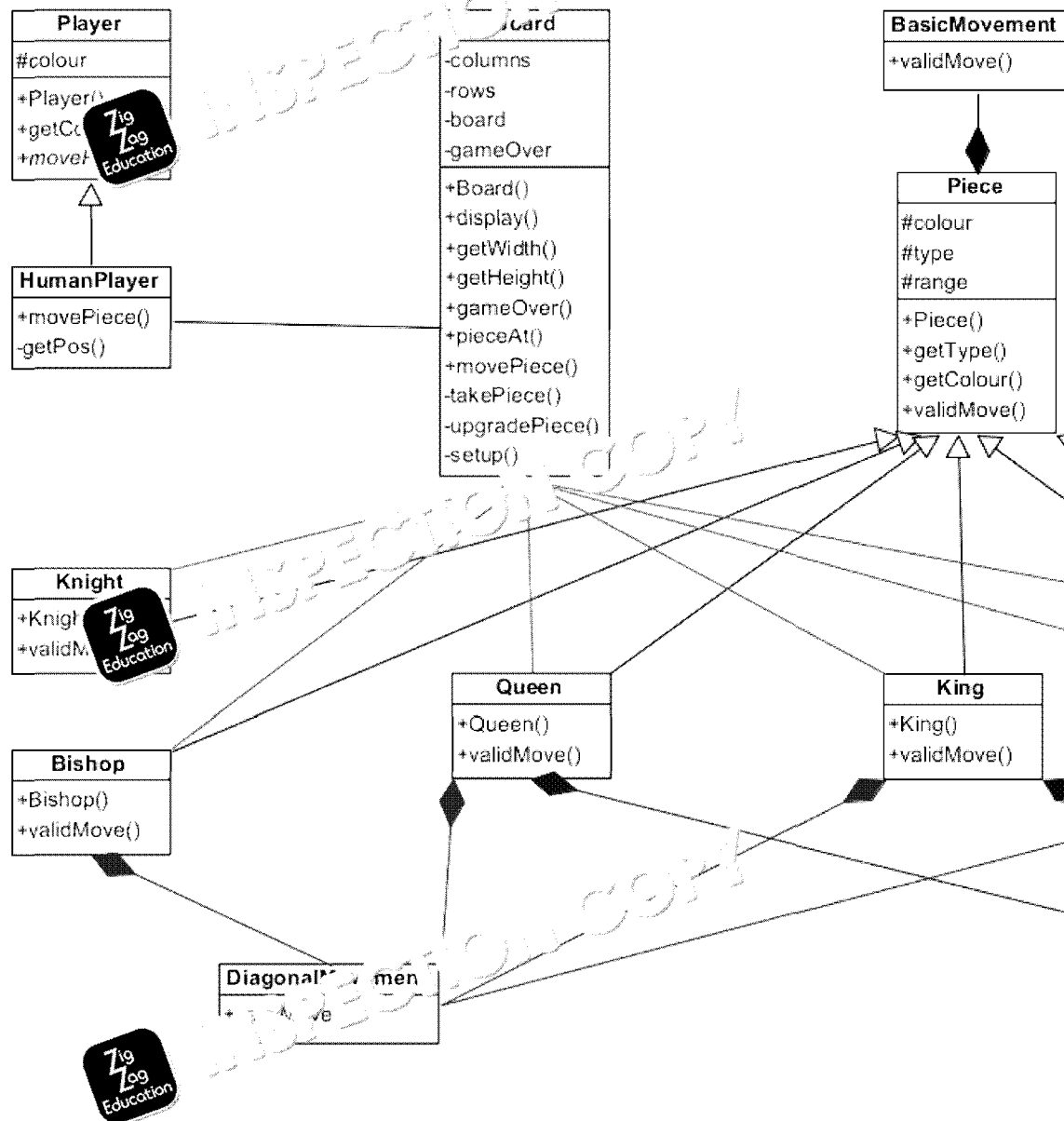
- When it is a player's turn, they must select the tile with the piece they want to move (if this tile doesn't contain one of their pieces, a message should be displayed to the player that they don't have a piece on that tile and they should select the tile again) and then select the tile that they would like to move the piece to. If this is not a valid move, a message should be displayed to tell the player that it is not a valid move and they should be asked for their move again).
- Player input should be given as a number, to indicate the column (first column is 1, second column is 2, etc.) and a letter, to indicate the row (first row is A, second row is B, etc.).
- You may use the Chess skeleton code to help you.



COPYRIGHT  
PROTECTED



## UML Class Diagram



INSPECTION COPY

COPYRIGHT  
PROTECTED



## 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 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 location is empty.                                                                                    |
| movePiece()      | Tries to make a given move, displaying a message saying success or failure, or a message saying that the move is invalid.                                    |
| takePiece()      | Displays a message to say which piece has been taken by the player. Sets gameOver = True if the piece which has been taken is the last piece of that colour. |
| upgradePiece()   | Replaces a pawn with a queen if it reaches the end of the board.                                                                                             |
| 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, then makes the move and returns whether or not the move was successful.                                                 |
| getPos()         | Checks that the player has given a valid location, returning a list of numbers if one location has been passed to getPos or a list of two numbers if 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, cannot start on, and cannot land on a tile containing a friendly piece. |

**COPYRIGHT  
PROTECTED**



**Pawn**

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

**Knight**

| Attribute/Method         | Description                                                                                                                                         |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Knight()</code>    | Creates a new Knight object, with <code>type = "knight"</code> .                                                                                    |
| <code>validMove()</code> | A knight can move two tiles vertically and one tile horizontally, or one tile vertically and two tiles horizontally. A knight can jump over pieces. |

**Rook**

| Attribute/Method         | Description                                                                                                        |
|--------------------------|--------------------------------------------------------------------------------------------------------------------|
| <code>Rook()</code>      | Creates a new Rook object, with <code>type = "rook"</code> .                                                       |
| <code>validMove()</code> | A rook can move any number of tiles in a straight line as long as there are no pieces between it and the end tile. |

**Bishop**

| Attribute/Method         | Description                                                                                                          |
|--------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>Bishop()</code>    | Creates a new Bishop object, with <code>type = "bishop"</code> .                                                     |
| <code>validMove()</code> | A bishop can move any number of tiles in a diagonal line as long as there are no pieces between it and the end tile. |

**Queen**

| Attribute/Method         | Description                                                    |
|--------------------------|----------------------------------------------------------------|
| <code>Queen()</code>     | Creates a new Queen object, with <code>type = "Queen"</code> . |
| <code>validMove()</code> | A Queen can make any move that a rook or bishop can make.      |

**King**

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

**BasicMovement**

| Attribute/Method         | Description                                                               |
|--------------------------|---------------------------------------------------------------------------|
| <code>validMove()</code> | Makes sure that the given move meets the basic criteria for a valid move. |

**StraightMovement**

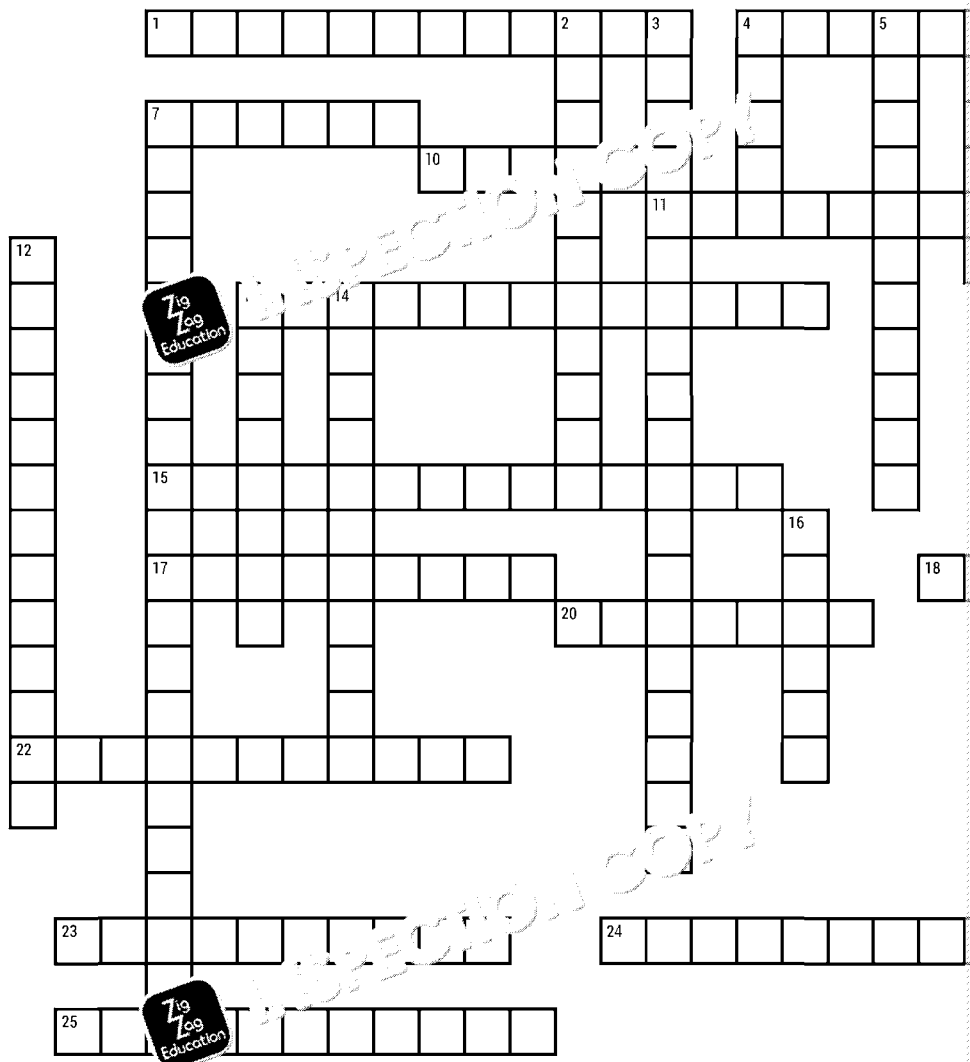
| Attribute/Method         | Description                                                                       |
|--------------------------|-----------------------------------------------------------------------------------|
| <code>validMove()</code> | Makes sure that the given move meets the criteria for a valid straight line move. |

**DiagonalMovement**

| Attribute/Method         | Description                                                                       |
|--------------------------|-----------------------------------------------------------------------------------|
| <code>validMove()</code> | Makes sure that the given move meets the criteria for a valid diagonal line move. |

**COPYRIGHT  
PROTECTED**

# CROSSWORD (OOP CONCEPTS)



INSPECTION COPY

**COPYRIGHT  
PROTECTED**



## Across

- 1 Allowing different implementations of a method to use the same method name (12)
- 4 Forming one larger object from multiple smaller objects, where the smaller objects cannot exist separately to the larger object (11)
- 7 A method or attribute of a particular class that can be called by any other class (6)
- 10 An instance of a data structure that has its own attribute values and associated methods (6)
- 11 The process of creating an object from a particular class (13)
- 13 A class that contains abstract methods and cannot be instantiated (8,5)
- 15 A method whose name and parameters are defined but that does not have any implementation (8,6)
- 17 A method or attribute of a particular class that can only be called within that class or its child classes (7)
- 18 A public method that changes the value of a private attribute (7)
- 20 A method or attribute of a particular class that can only be called within that class (7)
- 22 Creating multiple implementations of the same method that take different arguments (11)
- 23 A class that inherits from another class (5,5)
- 24 Grouping together related data and subroutines into classes, and providing controlled access to that class's private attributes (13)
- 25 Forming one larger object from multiple smaller objects, where the smaller objects can exist separately to the larger object (11)

## Down

- 2 When one type of object is replaced by a different type of object (8,11)
- 3 When a child class inherits from a parent class (8,11)
- 4 A template defining the structure of an object from which other objects are created (8,11)
- 5 A class that is inherited by other classes (8,11)
- 6 Class method that can be called on the class object (8,11)
- 7 A programming paradigm that involves a series of steps that are repeated (8,11)
- 8 A variable or constant that is shared by all objects of a class (9)
- 9 A method that can be called on the class object (8,11)
- 12 Any method that can be called on the class object (8,11)
- 13 A public method that can be called on the class object (8,11)
- 14 When a child class inherits from a parent class (8,11)
- 16 A subroutine belonging to a class (8,11)
- 19 Superseding the implementation of a method in its child class (10,11)
- 21 A particular approach to programming (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 allows to store values and subroutines as objects. When 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 have. **(1)**
4. A static method may be used to perform an operation that corresponds to the class as a whole, rather than a particular object of that class, **(1)** or when the method may be used even if the class has no objects. **1 mark**
5. **1 mark** for suitable attributes; **1 mark** for including a constructor method; **1 mark** for including a method to get the time; **1 mark** for including a method to display the time; **1 mark** for including a method that updates the display time each minute. **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
        endif
    endprocedure

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

INSPECTION COPY

**COPYRIGHT  
PROTECTED**

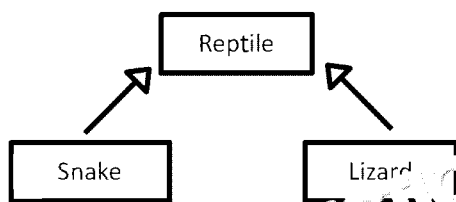


## 2 – Encapsulation

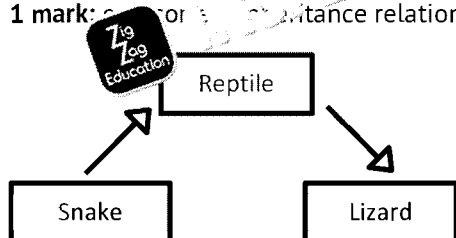
1. Encapsulation is the grouping of data and subroutines that relate to each other.
2. If an attribute or method is public, it will be available to any part of the program; if private, it can only be accessed from within the class in which it is defined. **(1)**
3. An attribute may be made private to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

## 3 – Inheritance and Abstract Methods

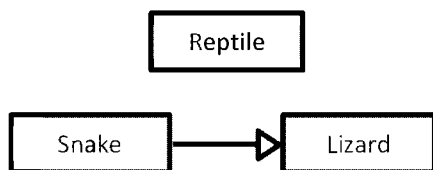
1. Inheritance is when one class uses another class as a base which can then be extended.
2. 1 mark for showing a correct inheritance relationship; 1 mark if all inheritance relationships are correct. Accept any diagram that shows a clear inheritance structure between at least three things. **2 marks:** all inheritance relationships are correct between at least three things, e.g.:



1 mark: correct inheritance relationship shown, e.g.:



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



3.
  - a) The parent class is Guitar and the child class is ElectricGuitar. **(1 mark for the parent class, 1 mark for the child class)**
  - b) ElectricGuitar inherits the attribute noOfStrings **(1)** and the method play **(1)** from Guitar. **(Max. 2 marks if adjustVolume is given as an inherited method)**
4. When a super method is called, the version of the method in the current class's superclass is used. **(1)**
5. Multiple inheritance can cause conflicts if a particular method has one implementation in one parent class and a different implementation in another parent class, **(1)** as the program may be unclear as to which method should be inherited. **(1)**

**COPYRIGHT  
PROTECTED**



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

#### 4 – Polymorphism

- Polymorphism is a way of allowing a particular method to have multiple different implementations. Polymorphism is useful for allowing methods to be called on different data types, where the specific implementation of the method is different. **(1)**
- Object cannot override any of the methods in Number because Object is an abstract class. **(1)**
  - type is an overloaded method **(1)** because the implementation in the child class is different from the parent class Object. **(1)**
  - type/add is an overloaded method **(1)** because it has multiple implementations. **(1)**
  - type/add is a virtual method **(1)** because it can be overridden in a child class. **(1)**
- A method would be made final to prevent child classes that inherit the method from overriding it. **(1)**

#### 5 – Class Relationships

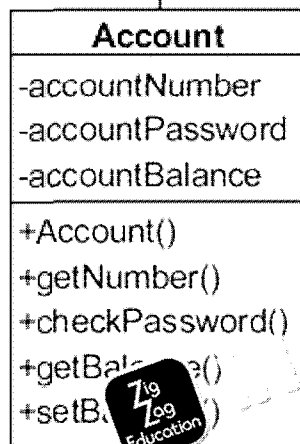
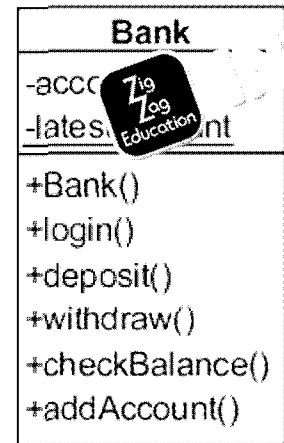
- UML class diagrams are visual representations of object-oriented systems **(1)** that help in the design and understanding of object-oriented systems. **(1)**
- Composition and aggregation both form an object from multiple smaller objects. **(1)**
- The component objects that form a composite object only exist for as long as the composite object exists, whereas the component objects that form an aggregated object can exist as separate objects even if the aggregated object is destroyed. **(1)**
- Class B inherits from Class A. Class A is the parent class, Class B is the child class. This is an inheritance relationship, i.e. 'inheritance'. **1 mark for getting classes the correct way round, i.e. 'A inherits from B'.**
  - Class A is a composite class formed with Class C component objects. **(1 mark for getting classes the correct way round, i.e. 'C forms A').**
  - Class A is an aggregated class formed with Class B component objects. **(1 mark for getting classes the correct way round, i.e. 'B forms A').**

**COPYRIGHT  
PROTECTED**



## UML Class Diagrams

### Task 1



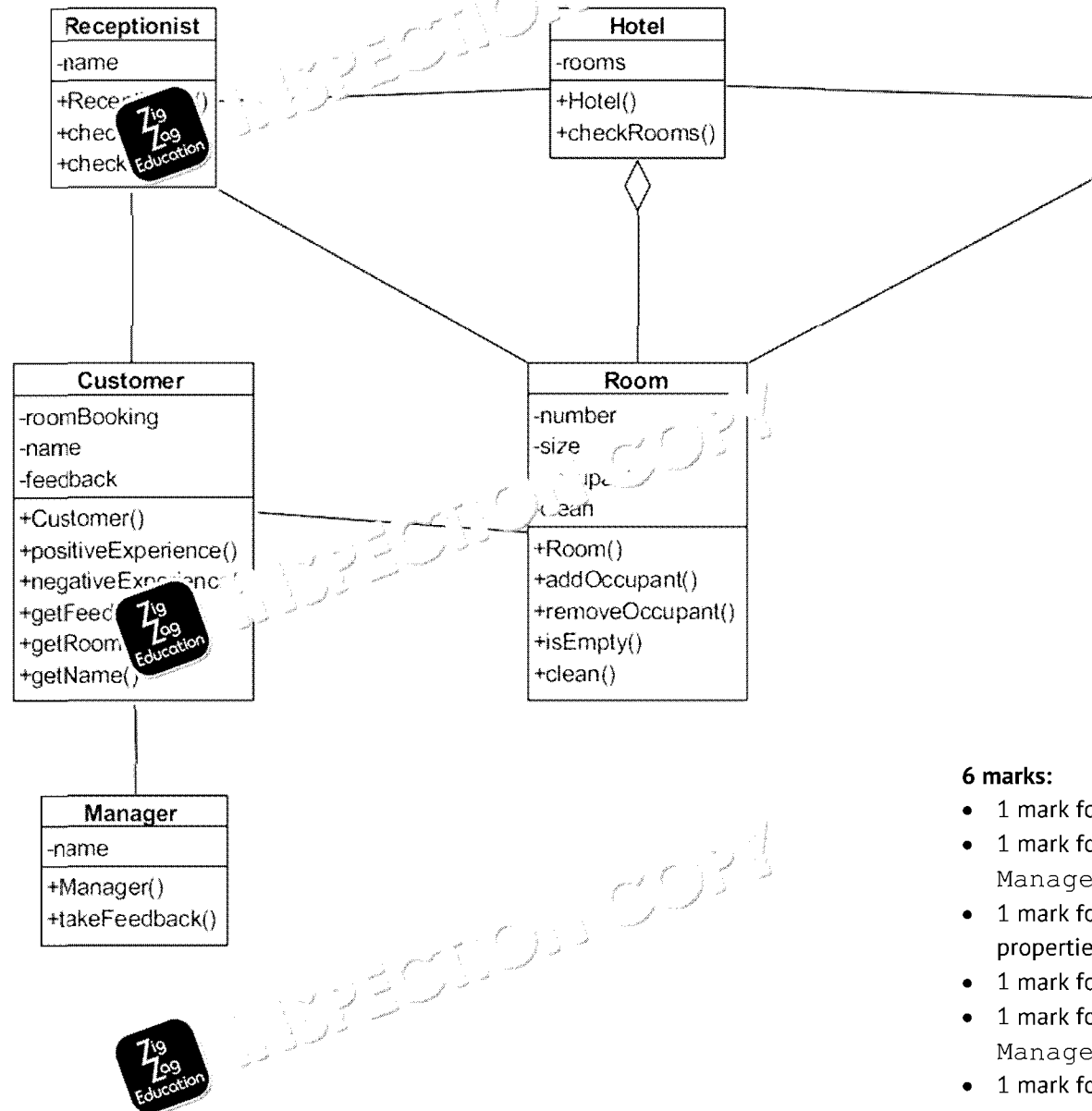
- 1 mark for showing all classes
- 1 mark for showing all attributes and methods in Bank
- 1 mark for showing all attributes and methods in Account
- 1 mark for showing the aggregation relationship between Bank and Account

INSPECTION COPY

COPYRIGHT  
PROTECTED



## Task 2



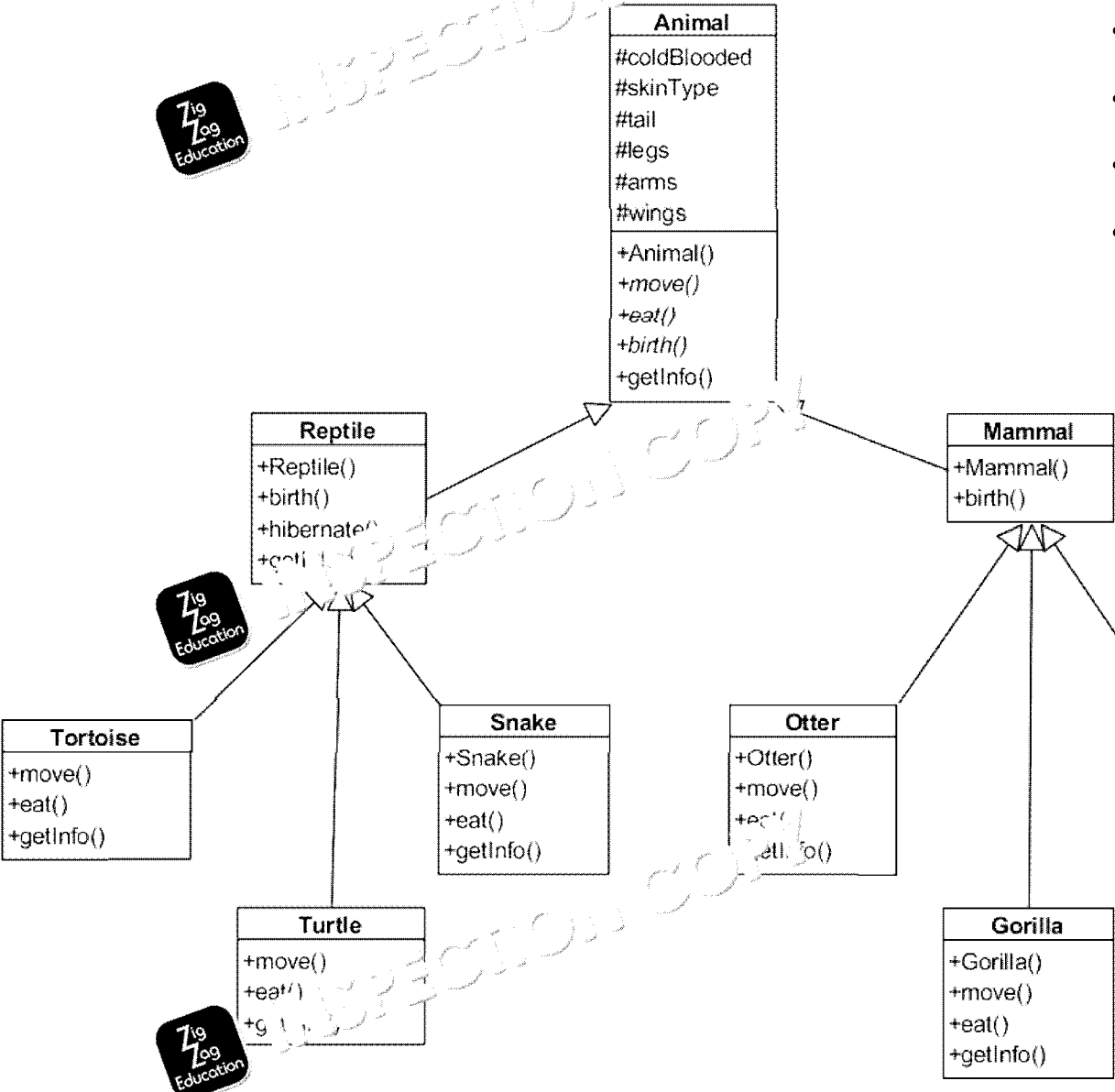
### 6 marks:

- 1 mark for
- 1 mark for
- 1 mark for
- 1 mark for
- 1 mark for
- 1 mark for

INSPECTION COPY

**COPYRIGHT  
PROTECTED**



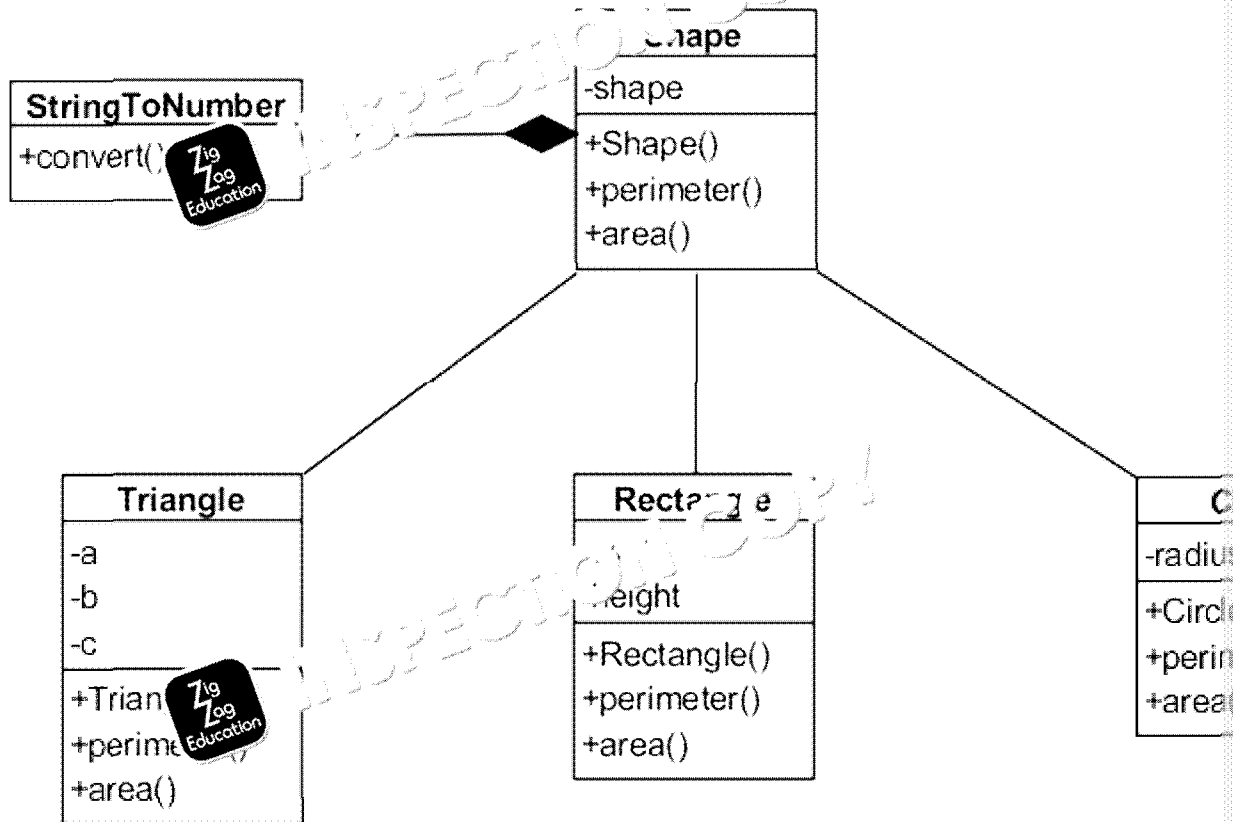


INSPECTION COPY

COPYRIGHT  
PROTECTED



## Task 4



5 marks:

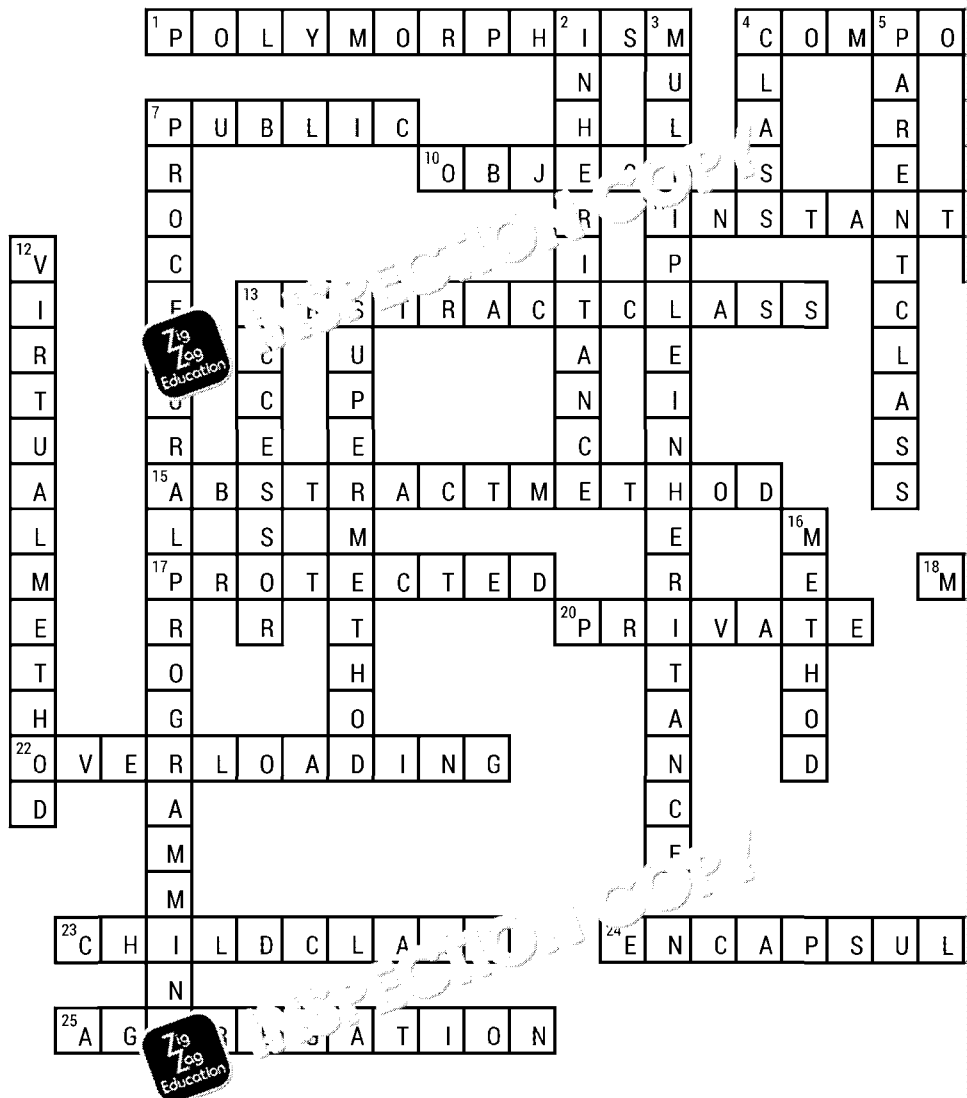
- 1 mark for showing all classes
- 1 mark for showing all attributes and methods
- 1 mark for showing all attributes and methods
- 1 mark for showing all relationships involved
- 1 mark for showing the composition relationship

INSPECTION COPY

COPYRIGHT  
PROTECTED



## Crossword (OOP Concepts)



### Across

- 1 Allowing different implementations of a method to use the same method name (12)
- 4 Forming one larger object from multiple smaller objects, where the smaller objects cannot exist separately to the larger object (11)
- 7 A method or attribute of a particular class that can be called by any other class (6)
- 10 An instance of a data structure that has its own attribute values and associated methods (6)
- 11 The process of creating an object from a particular class (13)
- 13 A class that contains abstract methods and cannot be instantiated (8,5)
- 15 A method whose name and parameters are defined but that doesn't have any implementation (8,6)
- 17 A method or attribute of a particular class that can only be called within that class or its child classes (9)
- 18 A public method that changes the value of a public attribute (7)
- 20 A method or attribute of a particular class that can only be called within that class (7)
- 22 Creating multiple implementations of the same method that take different types of arguments (11)
- 23 A class that inherits from another class (5,5)
- 24 Grouping together related data and subroutines into classes, and providing controlled access to that class's private attributes (13)
- 25 Forming one larger object from multiple smaller objects, where the smaller objects can exist separately to the larger object (11)

### Down

- 2 When one type of object is replaced by a different type of object (11)
- 3 When a child class inherits from a parent class (8,11)
- 4 A template defining an object from which other objects are created (11)
- 5 A class that inherits from another class (11)
- 6 Class method that is used to create an object (11)
- 7 A programming paradigm that uses a series of steps to create an object (11)
- 8 A variable or constant object (9)
- 9 A method that creates an object (11)
- 12 Any method that can be called from outside the class (11)
- 13 A public method that can be called from outside the class (11)
- 14 When a child class inherits from a parent class (11)
- 16 A subroutine belonging to a class (11)
- 19 Superseding the implementation in its child class (11)
- 21 A particular approach to solving a problem (11)

INSPECTION COPY

**COPYRIGHT  
PROTECTED**





# GLOSSARY

|                               |                                                                                                                     |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <b>Abstract Class</b>         | A class that contains abstract methods and cannot be instantiated                                                   |
| <b>Abstract Method</b>        | A method whose name and parameters are defined but no implementation                                                |
| <b>Accessor</b>               | A public method that returns a value relating to a private attribute                                                |
| <b>Aggregation</b>            | Forming one larger object from multiple smaller objects that can exist separately to the larger object              |
| <b>Attribute</b>              | A variable or constant belonging to a particular class or object                                                    |
| <b>Child Class</b>            | A class that inherits from another class                                                                            |
| <b>Class</b>                  | A template defining the attributes and methods of objects that can be created                                       |
| <b>Composition</b>            | Forming one larger object from multiple smaller objects that cannot exist separately to the larger object           |
| <b>Constructor</b>            | A method that creates an object of a particular class                                                               |
| <b>Encapsulation</b>          | Grouping together related data and subroutines in a class with controlled access to that class's private attributes |
| <b>Inheritance</b>            | When one type of object or class adopts functional characteristics of another object or class                       |
| <b>Instantiation</b>          | The process of creating an object from a particular class                                                           |
| <b>Method</b>                 | A subroutine belonging to a particular class or object                                                              |
| <b>Multiple Inheritance</b>   | When a child class inherits from multiple parent classes                                                            |
| <b>Mutator</b>                | A public method that changes the value of a private attribute                                                       |
| <b>Object</b>                 | An instance of a data structure that has its own attributes and associated methods                                  |
| <b>Overloading</b>            | Creating different implementations of the same method with different argument types                                 |
| <b>Overriding</b>             | Superseding the implementation of a parent class method                                                             |
| <b>Parent Class</b>           | A class that is inherited by another class                                                                          |
| <b>Polymorphism</b>           | Allowing different implementations of a method to share the same name                                               |
| <b>Private</b>                | A method or attribute of a particular class that can only be accessed within that class                             |
| <b>Procedural Programming</b> | A programming paradigm that structures a program as a sequence of steps to be followed in sequence                  |
| <b>Paradigm</b>               | A particular approach to designing and creating programs                                                            |
| <b>Protected</b>              | A method or attribute of a particular class that can be accessed by the class and its child classes                 |
| <b>Public</b>                 | A method or attribute of a particular class that can be accessed by any class                                       |
| <b>Static</b>                 | Class method that can be called even if no objects of that class are instantiated                                   |
| <b>Super Method</b>           | When a child class calls its parent class' implemented method                                                       |
| <b>Virtual Method</b>         | Any method that can be overridden by a child class                                                                  |

INSPECTION COPY

**COPYRIGHT  
PROTECTED**

