```
-------------Question 2

a) Modularity
        A few of behaviours are associated with a few abstract concepts, e.g. abstract classes.
AN employee is too vague a concept for anyone to be just an employee. Tus as in the firs example
it is possible to split the properties and methods of all employed people away from their current
description. This allows you to change and inter-change the code pertinent to all employees without
even caring how many kinds of employeess are there.
        For example at first we wanted to distribute salary in cash: assign a day, another person
with the job title - cashier, to give it away in person, form queues and deliver lots of inconvenience
to everyone. Then we realised this is wasteful, so we've done away with the need for a cashier,
and changed the method of getting paid, for example to a bank transfer, we could do that just in
one place: the definition of the employee class. Thus all classes that inherit from it, would
also inherit the changes.
        Additionally thanks to dynamic polymorphism this means that we could have a pay method that
needs not be overloaded to take every single kind of employee separately.

b) In the second example, implementation of the abstract classes MotorDriven and HumanPowered allows
us to implement partiually, and in two diverse ways how a vehicle can be driven (e.g. it could be a
motor that powers the wheeels, or it could be also a human. THus the subclasses need not contain
duplicate code taht explains how for example an internal combustion engine works, they'll just inherit
that code

c) Encapsulation in the first example is the fact taht there are no items lying around that the
clientelle
can pick up withot paying for. THis increases the programmers ability to control what's going on. A
programmer can limit ways in which certain objects can be interacted with by encapsulating them inside
other objects whose methods allow interaction. THis allows the programmer to rule out a few potential
exceptions and not worry about them in the future. In the second example a programmer can be sure at
any
point in time that a car can only have 3 or 4 wheels, because encapsulation allows for objects of
class
car be both immutable and useful.

-------------Question 3
        A class is a grouping (or encapsulation) of an idea, it's properties and behaviours. Ususally a
 class should contain the Definitions of all member functions e.g. methods. Additionally it could have
member values which can and should be initialised either by default or inside a constructor (ideally
both).
        An abstract class describes some concept taht isn't independent on its own. An employee has a
speciality there are no general employees, they all do something, which is in everyone's case
different.
In the below example an animal is an abstract class. It describes an animal as something that has age,
legs and a name. All animals have different amounts of legs, age and names. All of them live, bue all
live
differently. All subclasses need to define what it is to live, otherwise a general animal can't be
created
because it doesn't know how to live.
        An Interface also allows for programmers to allow different ideas that behave the same way to
be
unified. For example: a pet may be different it may not even always be a mammal. THat's why we just
state
what beahiour should a pet implement and leave the programmers free to do it.

        Below code examples are given (also in Git Repository).



----------------------------
//file project/path-to-dir/animal.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;
public abstract class animal{
        protected int age;
        protected int numberOfLegs;
        protected String name;
        public animal (){
                age = 0;
                numberOfLegs = 4;
                name = "Homeless vagrant";
        }
        public animal (int mLegs, int mAge, String mName){
                age = mAge;
                numberOfLegs = mLegs;
                name = mName;
        }
        public int getAge(){
                return age;
        }
        public int getLegs(){
                return numberOfLegs;
        }
        public String getName(){
                return name;
        }
        protected abstract void live();

        }



----------------------------
//file project/path-to-dir/pet.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

public interface pet {
        public String getName();
        public void pet();
        public void bond();
}

----------------------------
//file project/path-to-dir/puppy.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

public class puppy extends animal implements pet {

        public puppy() {
                super(0,4,"Sparky"); //Creates a four legged animal named Sparky
        }

        public puppy(int mLegs, int mAge, String mName) {
                super(mLegs, mAge, mName);      //Could be a homeless cripple.
        }

        @Override
        public void pet() {
                //Allow the human to pet you.
                //Dogs generally enjoy this, as opposed to cats.
        }

        @Override
        public void bond() {
                System.out.println("Make puppy eyes etc. etc.");

        }

        @Override
        protected void live() {
                //breathe();
                //heartBeat (80);
                //etc.

        }

}
----------------------------
//file project/path-to-dir/cat.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

public class cat extends animal implements pet {
        public cat() {
                super (0,4,"Garfield");
        }

        public cat(int mLegs, int mAge, String mName) {
                super(mLegs, mAge, mName);

        }

        @Override
        public void pet() {
                //Scratch the eyes of the owner out
                //Cats generally don't like being pet
                //Or they do. Suggest Import java.util.random

        }

        @Override
        public void bond() {
                System.out.println("Sleep all day behave like a jerk");

        }

        @Override
        protected void live() {
                //breather();
                //heartBeat(50);
                //etc.
        }

}
```

----------Question 4

Polymorphism is the ability to resolve function calls of different object containing implementations of the
same virtual or abstract method. e.g. if we have a class catadded to the above example, we could have different implementations of the pet method. Have a list of pets and pet each and every one of them.

IF we have two separate data structures, for example arrays to hold all the cats and all the dogs separately,
we could do static polymorphism, e.g. typecast all cats into the cat array, all dogs into the dog array and it's known what kind of method will be called each time at compilation time.

Dynamic polymorphism allows you to cram different implementations of animals into a single datastrcuture of
type pet and iterate through it to call pet. Each time the result will be different. WHere the pet is a dog,
it will make puppy eyes, where a pet is a cat, it will try to put your eyes out or make purrring noise.

Dynamic polymorphism unbinds the coders hands and allows for the program to behave according to the situation
without necessarily knowing what's going to happen beforehand. We may want for the user to specify what kinds of pets he wants to put into our datastructure, and in what order. We might also like for other programmers to be able to implement other types of animals externally without us having to change our code several times. This saves lots of checking, which makes code much more legible to the viewer.

--------QUestion 5

```
----------------------------
//file project/path-to-dir/Employee.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

public abstract class Employee {
        int salary;
        int age;
        public Employee() {
                salary  = 20000;
                age     = 26;
        }
        public Employee(int mSalary, int mAge){
                age= mAge;
                salary = mSalary;
        }
        protected void work(){
                System.out.println("I'm busy");
        }
}

----------------------------
//file project/path-to-dir/Ninja.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

public interface Ninja {
        public void ThrowShuriken();
        public int      getHealth();
        public void receiveDamage(int damage);
}

----------------------------
//file project/path-to-dir/EmployeeNinja.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

public class EmployeeNinja extends Employee implements Ninja {
        protected int health;
        protected int damage;
        public EmployeeNinja() {
                super (40000, 22); //Ninja Recruits are usually higher paid and
                                                        //younger.
        }

        public EmployeeNinja(int mSalary, int mAge) {
                super(mSalary, mAge);
                health = 100;
        }
        public EmployeeNinja(int mSalary, int mAge, int mHealth) {
                super(mSalary, mAge);
                health = mHealth;
        }
        @Override
        public void ThrowShuriken() {
                //Throw the SHuriken bending hand inwards.
                //Or specify any other technique
                //Deal Damage damage.
                System.out.println("Jaguar claws away");
        }

        @Override
        public int getHealth() {

                return health;
        }

        @Override
        public void receiveDamage(int damage) {
                health -= damage;
        }
        public static void main(String [] args){
                EmployeeNinja shippuden = new EmployeeNinja(50000,18,2000);
                shippuden.work();                       //EMployee method
                shippuden.ThrowShuriken();      //Ninja method
        }
}
----------------------------
--------QUestion 6
```

As opposed to C++ and its sane, albeit dangerous solution of allowing the programmer to specify where and when do they want to free up memory, java is fitted with a garbage collection utility.

WHat it does is it periodically sweeps the heap by either reference counting or tracing or other more sophisticated methods periodically removes objects on the heap that aren't any more referenced by any call in the stack.

Sometimes removing an object may be straightforward, e.g. just freeing up memory will revert the machine
to its initial state without doing any damage. However, if the obhject in turn interacts with the system
itself, there's sometimes need for final touchups before the object is removed from memory.

C++ had destructors which were called when the object was deleted. However, since pointers in c/C++ could
be arbitratily assigned this caused trouble with potential damage to the system objects. Additionally,
sometimes the programmer might forget to delete the object, thus the destructor is never called.

Finaliser's are Oracles take on the problem. They're called when the java grabage collector is about to
delete the object. They aren't called destructors on purpose, because they cannot be invoked at will, and
there's absolutely no telling when if ever the garbage collector will call it.

--------Question 7

```
public void testOutput(){
        Person p = new Person ("Joe", "Bloggs");
        System.out.prinltn("PErson DEtails:" + p);
}
```

The above function contains implicit typecasting. All classes in Java extend the Object class. An Object
defines cloning and comparing, as well as conversion to String, If in class Person we have overridden
the Object's method toString then the above call can be arbitrarily interchanged.

FOr example this will output

```
        "PErson DEtails: BLoggs, John"
@Override
public String toString (){
        return Surname + ", "+ name;
}
```

or this

```
        "PErson DEtails: Johny Bloggsy"
@Override
public String toString (){
        return name+"y, "+ surname+"y";
}
```

--------Question 8

```
----------------------------
//file project/path-to-dir/HiloGame.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;

import java.io.IOException;

public class HiloGame {

        public static void main(String [] args) throws IOException{
                Game currentGame= new Game(args.length > 0?Integer.parseInt(args[0]):0);
                System.out.println("Welcome to the guessing game. Your current difficulty setting is:
" + currentGame.getDifficulty());
                System.out.println("You have to guess a random Number in the range from  0 to "+
currentGame.getRange());
                System.out.println("\n\n---------------------------------\n\n");
                int userInput= 0;
                while (!(currentGame.isOver() || userInput=='q')){
                        System.out.println("---------------------------------\n\n");
                        System.out.println("You have "+currentGame.getGuesses()+" guesses\n
\nGuess!!!");
                        userInput=System.in.read();
                        System.in.read();//Dummy to read the carriage return character
                        if(userInput!=10){
                        System.out.println("Your Guess was " + (userInput-48));
                        System.out.println(currentGame.guess(userInput-48));
                        }

                }
                if(currentGame.isLost()==false){
                        System.out.println("\n\n--------------------------------\n\n");
                        System.out.println("You have won. Congrats");
                }

        }

}

----------------------------
//file project/path-to-dir/Game.java
----------------------------
package uk.ac.cam.ap886.oopjava.supervision2;
import java.util.Random;
public class Game {
        int number;
        int difficulty;
        int numAttempts = 3;
        boolean over = false;
        boolean lost = false;

        public Game (int mDifficulty){
                difficulty = mDifficulty < 0?0:mDifficulty;
                Random randomGenerator = new Random();
                difficulty = difficulty <5?difficulty:4;
                number = randomGenerator.nextInt(5+difficulty);
                numAttempts = 2+difficulty;
        }
        public Game (int mDifficulty, int nAttempts){
                mDifficulty = mDifficulty < 0?-mDifficulty:mDifficulty;
                Random randomGenerator = new Random();
                number = randomGenerator.nextInt(5+difficulty);
                numAttempts = nAttempts;
        }
        public String guess(int mGuess){
                if(over){
                        return "Game Over. You have" + (lost?" lost":" won!!!");
                }
                else{
                        if(mGuess == number){
                                over = true;
                                return "Correct";
                        }
                        else{
                                if(numAttempts < 1){
                                        over = true;
                                        lost = true;
                                        return "Game Over. You've lost";
                                }
                                else{
                                        numAttempts --;
                                        return mGuess>number?"Go Lower":"Go Higher";
                                }

                        }
                }

        }
        public String getRange(){
                return Integer.toString((1<<(difficulty+3)));
        }
        public boolean isOver(){
                return over;
        }
        public boolean isLost(){
                return lost;
        }
        public int getGuesses(){
                return numAttempts;
        }
        public int getDifficulty(){
                return difficulty;
        }

}
```