Letterkenny Institute of Technology

Course code: OOPR CP603

YEAR 2 COMPUTING

(Common paper for all streams)

Subject:	Object Oriented Programming	**Stage:**	2
Date:	January 2016	**Examiners:**	Mr. D. Hegarty Dr. A. Belatreche
Time allowed:	3 hours		

INSTRUCTIONS

Answer any **FOUR** questions. All questions carry equal marks.

NOTE: It may be useful to remove the appendices from the questions portion of the paper – i.e. you can have the questions side-by-side with the relevant code/diagrams, and don't need to go back and forth.

Question 1

Appendix A provides classes to implement a basic grid-based game where a player can move around the grid based on user input and the enemies move in random directions, shooting the player when they are within range.

- The Game Class controls the game.
- The World Class gives us a basic world which is a grid-based arrangement that contains a Player object and an arraylist of enemies. Importantly, the world provides us with methods for updating the state of our game and also drawing our world.
- The Player class represents a player – it has a location (within the co-ordinates of the world) and a health. It can be moved in one of eight directions.
- The Enemy class represents an enemy which has a location, a number (which is used for display purposes), and a shooting range. It moves in random directions, and can shoot the player if he is in range.
- The Location class is used by the Enemy and Player classes to store and update their position.

A sample screen – with the player in his initial position – is given:

```
                         Initial World
                         -------------
                         |P| | | | | |
                         -------------
                         | | | | |2| |
                         -------------
                         | | | | | | |
                         -------------
                         | | | | | | |
                         -------------
                        | | |13| | | |
                         -------------
                         | | | | | | |
                         -------------
Direction to move? 0:NORTH 1:NORTH EAST 2:EAST 3:SOUTH EAST 4:SOUTH
5:SOUTH WEST 6: WEST 7: NORTH WEST
```

Answer the following:

a) Specify how you would change the code in Game.java so that the player initially appears in the bottom right-hand corner rather than the top-left.
Note that you should not use magic numbers (hint: use the constants from World.java). Also note that in the x-y coordinate system x would represent the column and y would represent the row.
(5 marks)

b) You are given the code which will allow the Player and Enemy classes to move *south*. Provide additional code snippets to:
 i. Move a Player/Enemy *north*. (3 marks)
 ii. Move a Player/Enemy *south-east*. (4 marks)

c) The Player and Enemy classes share certain characteristics and behaviours; therefore there is code duplication.
Identify the shared components and explain what would be required to avoid this. (6 marks)

d) Provide a method called `printLocationInfo` which would be a member of the World Class and would print the coordinates of each character (Players and Enemies) to the screen.
(7 marks)

Question 2

A basic Person class is given below. Answer the subsequent questions based on this class.
Note 1: You may find it useful to refer to appendix B for this question.
Note 2: You can assume that all code snippets are part of a valid tester class containing a `main()` method.

```
public class Person
{
   private String name;
   private int age;

   public Person(String name, int age)
   {
      this.name = name;
      this.age = age;
   }

}
```

a) Override the toString() method.

(2 marks)

b) Amend the Person class so the code snippet below will work properly:

```
      ArrayList<Person> people = new ArrayList<Person>();

      //Assume Person objects have been added to the list

      if (people.contains(new Person("Adam Ant", 48)))
      {
         //Do something
      }
      else
      {
         //Do something else
      }
```

(8 marks)

c) Assuming that I want to use the `sort` method of the Collections class to sort the `people` list alphabetically by name - `Collections.sort(people);` - provide the required changes to the Person class.

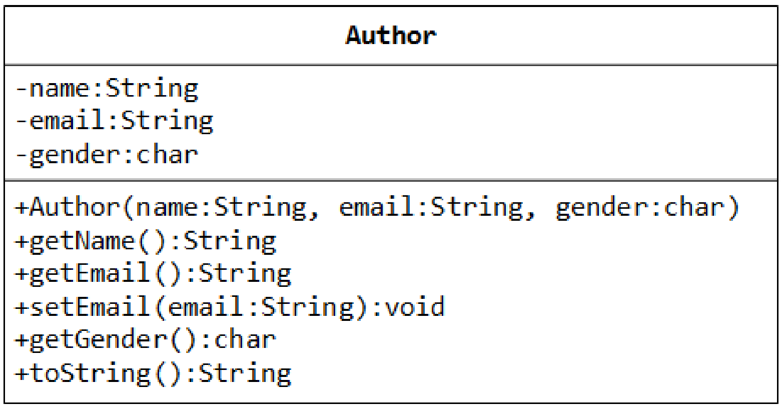
(6 marks)

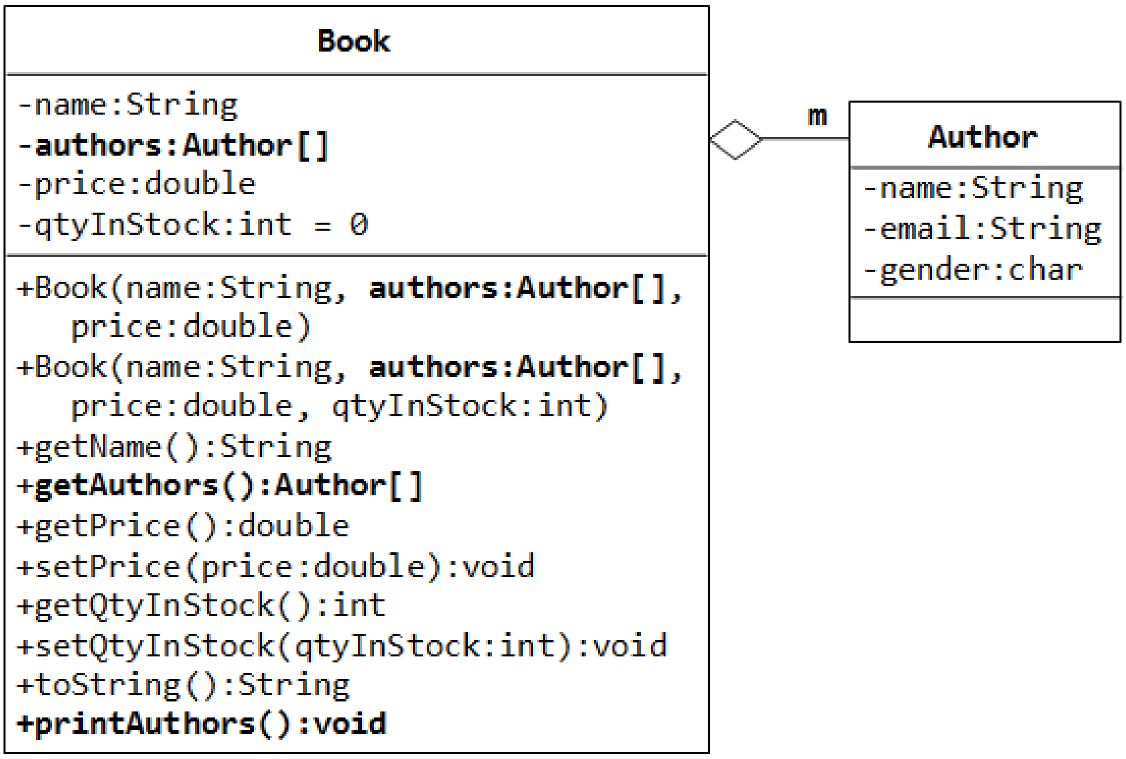
d) If I also wish to provide the additional capability of sorting by age, provide a mechanism that will allow this additional sorting capability. Show how you would invoke the Collections.sort() method for this.

(9 marks)

Question 3

The Book and Author classes shown below are an example of composition; *a book is composed of a number of authors*.





a) Provide the code for each of the Book class constructors.

(6 marks)

b) Provide the code for the `printAuthors()` method. Assume that all other methods are complete.

(5 marks)

c) Show how you would create a Book object in a tester class.
The book should be: “The Best American Short Stories”, price: 9:80, quantity: 3.
The authors should be: “Heidi Pitior, hpitior@gmail.com, *female*”,
“Jennifier Egan, jegan@yahoo.com, *female*”,
“Mike Atwell, matwell@hotmail.com, *male*”.

(6 marks)

d) Continuing your tester in part c), show how you could then search for the author named Jennifer Egan and, if found, change the associated email address to jennifer.egan@gmail.com.
Note: Your search must retrieve the author information from the book object.

(8 marks)

Question 4

A basic Person class is given below:

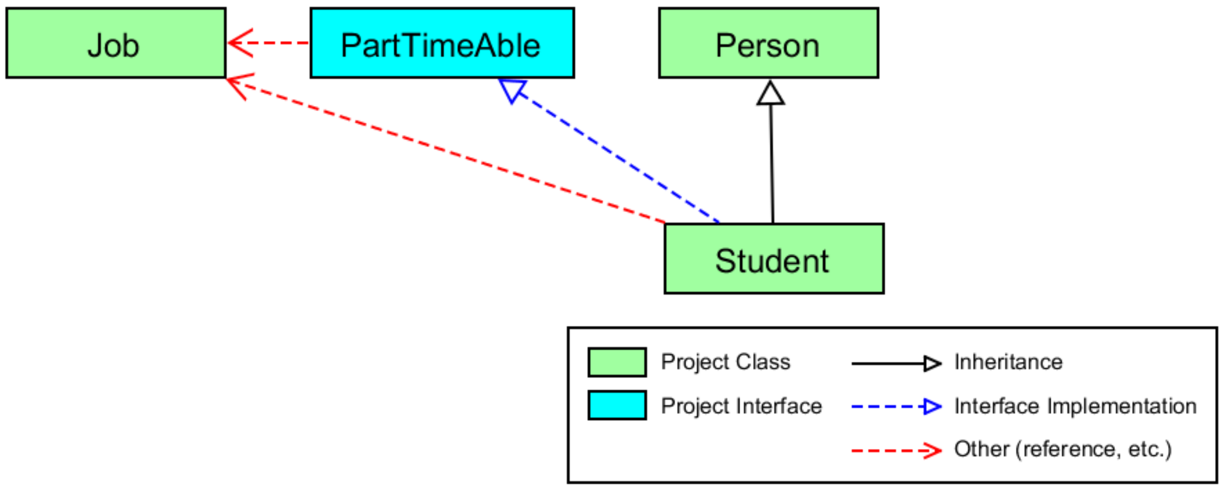
```
public class Person{
   private String name;
   private int age;

   public Person(String name, int age){
      this.name = name;
      this.age = age;
   }

}
```

An interface called PartTimeable is defined as:

```
public interface PartTimeAble {
     public boolean doJob(Job j);
}
```

A description of the Job class is given in Appendix C. You should assume that this class is already provided, i.e. you do not have to provide any code for this class.

The following diagram illustrates the relationships between the various classes.



The Student class should have the following instance fields:
name (String), age (int), course(String), spendingMoney (double).

Additionally the class should maintain an ArrayList<Job> instance field which should store each of the jobs done by a student.

a) Create the basic Student class including an overloaded constructor for the class which allows for student objects to be created, e.g. Student s1 = new Student("Jim Jones", 19, "Applied Computing");.
All other instance fields should be initialised in this constructor: spendingMoney initialised to 0.0, arraylist initialised to empty list.

(7 marks)

b) Override the interface method so that a students's spending money is updated with the money earned from the job and the job details are stored in the list (note that your method should simply return *true*; in other implementations there may a possibility that the implementing class may not be able to carry out the job.)

(6 marks)

c) Provide a snippet of test code which creates a student object and has him/her do some part-time work.

(5 marks)

d) The following partially complete class represents an electronic device capable of doing jobs. If there is enough charge it will simply deplete its batteryLevel by the required amount; if there is not enough charge it will return false (since it determines that it cannot carry out the task/job).

Provide the requisite interface code.

```
public class Automaton implements PartTimeAble{

   private double batteryLevel;
   private final int BATTERY_UNITS_PER_HOUR = 3;

   private final int INITIAL_CHARGE = 24;

   public Automaton()
   {
         batteryLevel = INITIAL_CHARGE;
   }

   public void recharge()
   {
         batteryLevel = INITIAL_CHARGE;
   }

   public double getBatteryLevel()
   {
         return batteryLevel;
   }
}
```

(7 marks)

Question 5

Appendix D consists of two parts: A Sorter class with selectionSort and bubbleSort methods; A Searcher class with a binarySearch method.

(a) The bubbleSort algorithm could potentially quit sorting if it realised that there were no swaps on the previous pass. Amend the method to provide this functionality (hint: use a boolean flag)

(5 marks)

(b) For a list with 7 elements in it, determine how many *comparisons* would be required by the selectionSort algorithm.

(3 marks)

(c)

I. The sort methods work on integer arrays. Briefly outline the changes required to the methods so that they would work on an array of objects, such as Strings.

(4 marks)

II. What requirement must the class - *String*, as mentioned above, or indeed a class you might have created yourself such as *Person* – fulfil so that your sort methods can work?

(3 marks)

(d) For the following array (13 elements), determine the indices that binarySearch will look at when searching for the search-key 10.

[2, 5, 7, 13, 15, 16, 18, 20, 22, 24, 30, 34, 39]

Your answer should identify what occurs on each pass in terms of the variables **`start, mid, end`**.

(7 marks)

(e) How many passes are required in a binarySearch for an array/list containing 40 elements? You should explain your answer.

(3 marks)

<u>Question 6</u>

Given the BankAccount class and skeleton code for the CheckingAccount class in Appendix E, answer the following:

(a) “Shadowing of instance fields is a common mistake for programmers who are new to inheritance”. Explain what this means. (4 marks)

With the above statement in mind, correct the mistake that is contained in the instance fields declared in the CheckingAccount class.
(2 marks)

(b) Supply the code for the empty methods in the Checking Account class.
(6 marks)

(c) Give an illustration of how the `super` keyword prevents infinite recursion. (4 marks)

(d) For the tester code given at the bottom of the page :-

- add code to print the total amount of money in harrysChecking at the end. (2 marks)
- calculate the total amount of money in harrysChecking at the end.
(2 marks)

(e) Provide a method in the BankAccount class called `transfer` that will allow for transfer of money from one BankAccount to another (5 marks)

```
public class AccountTester
{
   public static void main(String[] args)
   {
      CheckingAccount harrysChecking
            = new CheckingAccount(125);

      harrysChecking.withdraw(70);
      harrysChecking.deposit(50);
      harrysChecking.deposit(45);
      harrysChecking.deposit(16);
      harrysChecking.withdraw(50);
      // Simulate end of month
      harrysChecking.deductFees();

     // Print out how much is in the account

   }
}
```

Appendix A - Game Class

```
/**
 * This basically represents our game loop
 *
 */
public class Game
{
       public static void main(String[] args)
       {
              World gameWorld = new World(new Player(new Location(0,0)), 3);
              System.out.println("Initial World");
              gameWorld.drawWorld();

              Scanner userInput = new Scanner(System.in);

              String userChoice;

              int iterationCounter = 0;//Keeps track of how long you survive

              //GAME LOOP
              do
              {
                     //All games boil down to this – update world and render
                     gameWorld.update(userInput);
                     gameWorld.drawWorld();

                     iterationCounter++;


                     if(!gameWorld.getPlayer().isAlive())
                     {
                            System.out.println("You're Dead! You lasted " +
                                                 iterationCounter + "turns");
                            break;
                     }


                     System.out.println("Continue y/n");
                     userChoice = userInput.nextLine();
                     userChoice = userInput.nextLine();
              } while (!userChoice.equalsIgnoreCase("n"));

              System.out.println("Bye Bye");
              userInput.close();
       }

}
```

Appendix A - Location Class

```
/**
 * Encapsulates a grid co-ordinate, with some methods to move and set
 *
 * To illustrate - in our sample 6*6 world, "P" is at location 0,0,
 * "1" is location 5,2 (x = 5, y = 2) etc.
 * -------------
 * |P| | | | | |
 * -------------
 * | | | | | | |
 * -------------
 * | | | | | |1|
 * -------------
 * | | | | | | |
 * -------------
 * | | | | | | |
 * -------------
 * |3| | | |2| |
 * -------------
 *
 */
public class Location{
        private int x;
        private int y;

        public Location(int x, int y){
                this.x = x;
                this.y = y;
        }

        public int getX(){
                return x;
        }

        public int getY(){
                return y;
        }

        public void setX(int x){
                this.x = x;
        }

        public void setY(int y){
                this.y = y;
        }

        public void changeX(int amountToChange){
                this.x += amountToChange;
        }

        public void changeY(int amountToChange){
                this.y += amountToChange;
        }

        public String toString(){
                return this.getClass().getSimpleName() + "[X: " + x + " Y: " +
                                                                  y + "]";
        }
}
```

Appendix A – Enemy Class

```
/**
 * The Enemy class encapsulates the notion of an enemy having
 * a number (for display), a location and a shooting range.
 * The enemy can move can attempt to move in a random direction and
 * can "shoot" the player if he is in range.
 */
public class Enemy{
      private int enemyNumber; //useful to identify the enemy
      private Location location;

      private double defaultRange = 3.0; //If enemy is <= to this then he
                                        //will shoot
      public Enemy(int enemyNumber, Location location){
            this.enemyNumber = enemyNumber;
            this.location = location;
      }

      public int getEnemyNumber(){
            return enemyNumber;
      }

      public Location getLocation(){
            return this.location;
      }

      /**
       * Move the enemy in a random direction by one position, if possible
       * @return whether the move was possible or not.
       */
      public boolean move(){
            Random rand = new Random();

            int direction = rand.nextInt(World.NUM_DIRECTIONS);
            return moveDirection(direction);

      }

      /**
       * TO BE COMPLETED
       * @param directionToMove
       * @return whether a move was made or not.
      */
      public boolean moveDirection( int directionToMove){
            switch (directionToMove){
                  case (World.SOUTH):
                        if(location.getY() < World.WORLD_HEIGHT - 1)
                        {
                              location.changeY(1);
                              return true;
                        }
                        return false; //couldn't move in this direction

                  default:
                        return false;

            }

      }
	/**
	 *
	 * @param player the player's character
	 * @return whether the player is in range of the enemy or not.
	 */
	public boolean inRange(Player player){
		if( computeDistance(player) <= defaultRange)
			return true;
		else
			return false;
	}

	/**
	 * Note how the Player is passed in so the enemy can interact with
	 * him (the alternative would be for the enemy to have a Player
	 * reference as an instance field)
	 * @param player
	 */
	public void shootPlayer(Player player)
	{
		if (inRange(player))
			player.takeHit(1);
	}

	/**
	 * Computes the distance to the player using Pythagoras.
	 * Note that the unit distance between "cells" is 1, so the diagonal
	 * distance between cells would be 1.41 (square root of 2)
	 * @param p
	 * @return the straight-line distance to the player (from the enemy,
	 * i.e. "this" object)
	 */
	private double computeDistance(Player p)
	{
		int x1 = this.location.getX();
		int x2 = p.getLocation().getX();
		int deltaX = x1 - x2;

		int y1 = this.location.getY();
		int y2 = p.getLocation().getY();
		int deltaY = y1 - y2;
		return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
	}

}
```

```
/**
 * The Player class represents the player, with its main characteristics
 * being a location and a health indicator.
 * In the game, the user will attempt to move a player in the optimum
 * direction to avoid being shot by enemies.
*/
public class Player{
      private Location location;
      private int health;

      public Player(Location location){
            this.health = 5;
            this.location = location;
      }

      public void takeHit(int healthHit){
            health -= healthHit;
      }

      public int getHealth(){
            return health;
      }

      public boolean isAlive(){
            return (health > 0);
      }

      public Location getLocation(){
            return location;
      }


      public boolean moveDirection( int directionToMove)
      {
            switch (directionToMove)
            {
                  case (World.SOUTH):
                        if(location.getY() < World.WORLD_HEIGHT - 1)
                        {
                              location.changeY(1);
                              return true;
                        }
                        return false;
                  default:
                        return false;

            }

      }

}
```

Appendix A - World Class

```
/**
 * The "World".
 * Contains the objects in our world (composition), an update method (where
 * things move), and a draw method to "render" our world.
*/
public class World
{
	public static final int WORLD_WIDTH = 6;
	public static final int WORLD_HEIGHT = 6;

	public static final int NORTH = 0;
	public static final int NORTH_EAST = 1;
	public static final int EAST = 2;
	public static final int SOUTH_EAST = 3;
	public static final int SOUTH = 4;
	public static final int SOUTH_WEST = 5;
	public static final int WEST = 6;
	public static final int NORTH_WEST = 7;

	public static final int NUM_DIRECTIONS = 8;

	//The instance fields are the contents of our world
	private ArrayList<Enemy> enemies = new ArrayList<Enemy>();
	private Player player;

	/**
	 *
	 * @param player Player object is created outside the class and
	 *               passed in
	 * @param noOfEnemies Enemies are created internally
	 */
	public World(Player player, int noOfEnemies)
	{
		this.player = player;
		for(int i = 0; i < noOfEnemies-1; i++)
		{
			createEnemy(i+1);
		}
		Random rand = new Random();
		int xVal = rand.nextInt(WORLD_WIDTH);
		int yVal = rand.nextInt(WORLD_HEIGHT);
		enemies.add(new SuperEnemy(noOfEnemies,
				new Location(xVal, yVal), player));
	}

	private void createEnemy(int enemyNumber)
	{
		Random rand = new Random();
		int xVal = rand.nextInt(WORLD_WIDTH);
		int yVal = rand.nextInt(WORLD_HEIGHT);

		enemies.add(new Enemy(enemyNumber, new Location(xVal, yVal)));

	}
```

```
/**
 * This corresponds to a single frame of our game (a game loop
   merely calls this in a loop):
 * Move the Player (via input from the user)
 * Move the enemies
 * Enemies will shoot if they are in range of the player.
 * @param sc this will be the same scanner as used in the Game class
                 (cannot nest them)
 */
public void update(Scanner sc){
      //Game loop continues as long as there is something to play,
      //i.e. you're not dead.
      if(player.isAlive())
      {
            //move the player based on user input
            player.moveDirection(getPlayerMovementDirection(sc));

            //Attempt to move each of the enemies and then check
            //to see if they're currently in range to shoot.
            for(Enemy currEnemy: enemies)
            {
                  if(!currEnemy.move())
                  {
                        System.out.println("Enemy " +
                              currEnemy.getEnemyNumber() + " can't
                              move this iteration");
                  }

                  if(currEnemy.inRange(player))
                  {
                        currEnemy.shootPlayer(player);
                        System.out.println("Enemy " +
                              currEnemy.getEnemyNumber() + "
                              shoots Player");
                        System.out.println("Player Health down to
                                    " + player.getHealth());
                  }
            }
      }
}


/**
 * Private utility method to ask for user input and return the value
 * @param sc the active scanner object
 * @return value representing one of eight directions.
 */
private int getPlayerMovementDirection(Scanner sc)
{
      System.out.println("Direction to move? 0:NORTH 1:NORTH EAST
                        2:EAST 3:SOUTH EAST 4:SOUTH 5:SOUTH WEST
                        6: WEST 7: NORTH WEST");

      return sc.nextInt();

}
```

```
        /**
         * Draw the grid world
         * NOTE: I’ve left this out in the interests of brevity
         */
        public void drawWorld()
        {
                //code to draw the world...

        }


        /**
         * Other classes will also need to communicate with the player
         * object.
         * @return reference to the Player object
         */
        public Player getPlayer()
        {
                return player;
        }
}
```

Appendix B - Information on the Comparable and Comparator interfaces (Question 2)

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

Method Summary

Methods

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

This is how the Comparable interface would be written:

```
public interface Comparable<T>
{
   public int compareTo(T o);
}
```

Interface Comparator<T>

Type Parameters:

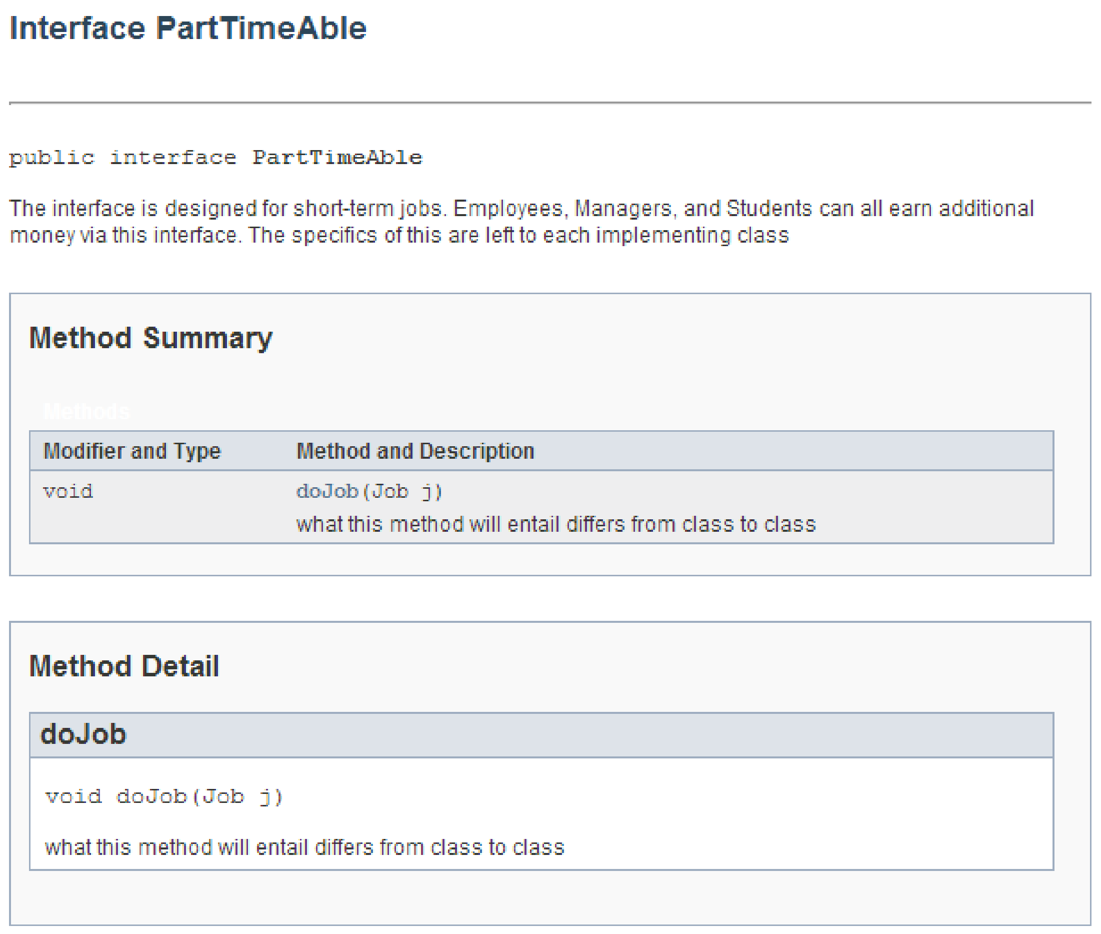
T - the type of objects that may be compared by this comparator

Method Summary

Methods

Modifier and Type	Method and Description
`int`	`compare(T o1, T o2)` Compares its two arguments for order.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this comparator.

Appendix C - PartTimeAble interface and Job class (Question 4)



Interface PartTimeAble

```
public interface PartTimeAble
```

The interface is designed for short-term jobs. Employees, Managers, and Students can all earn additional money via this interface. The specifics of this are left to each implementing class

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`doJob(Job j)` what this method will entail differs from class to class

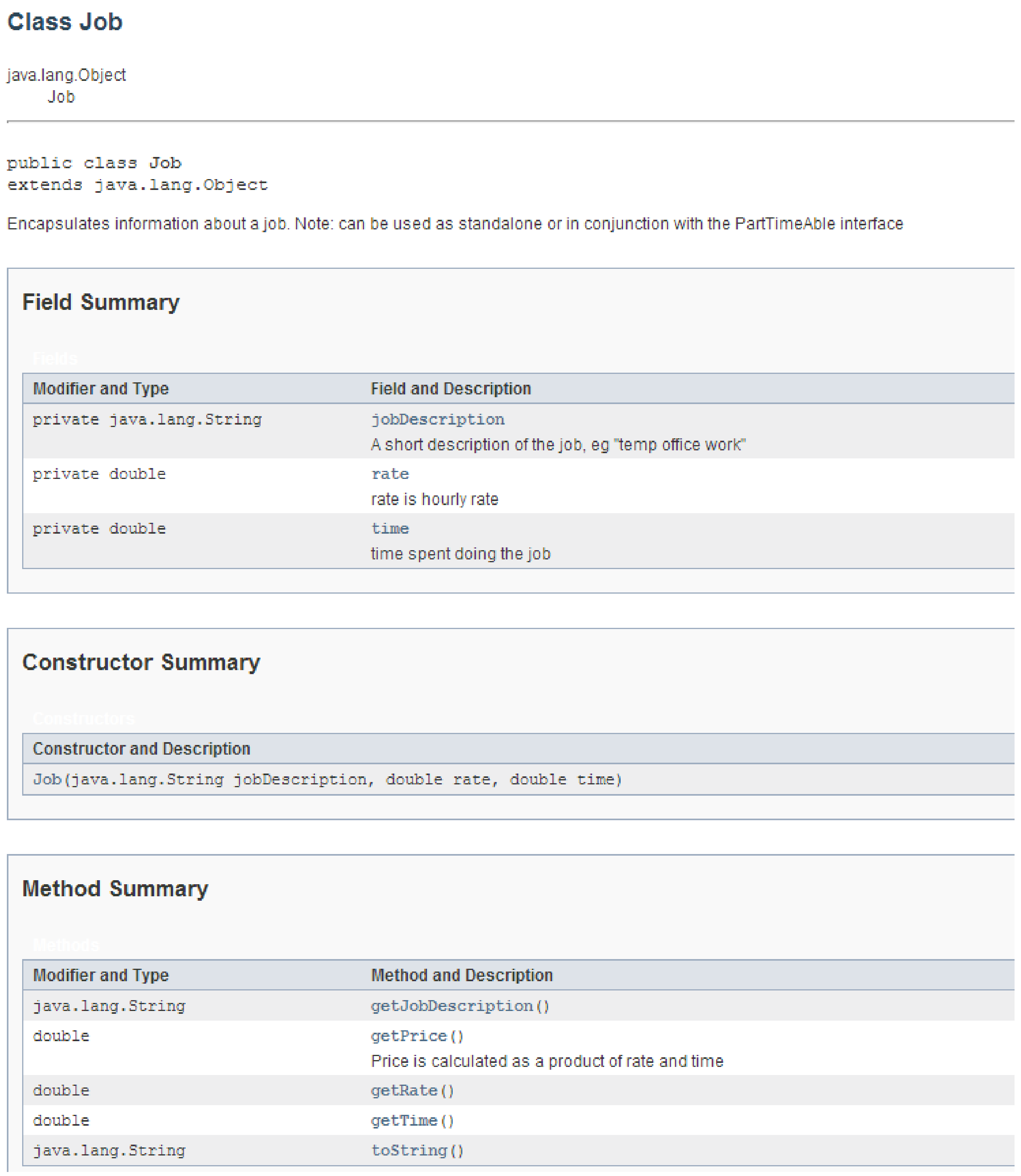
Method Detail

doJob

```
void doJob(Job j)
```

what this method will entail differs from class to class

Job class is specified on the next page.



Class Job

java.lang.Object
    Job

```
public class Job
extends java.lang.Object
```

Encapsulates information about a job. Note: can be used as standalone or in conjunction with the PartTimeAble interface

Field Summary

Modifier and Type	Field and Description
`private java.lang.String`	`jobDescription` A short description of the job, eg "temp office work"
`private double`	`rate` rate is hourly rate
`private double`	`time` time spent doing the job

Constructor Summary

Constructor and Description
`Job(java.lang.String jobDescription, double rate, double time)`

Method Summary

Modifier and Type	Method and Description
`java.lang.String`	`getJobDescription()`
`double`	`getPrice()` Price is calculated as a product of rate and time
`double`	`getRate()`
`double`	`getTime()`
`java.lang.String`	`toString()`

Appendix D: Part 1 - Sorter class (Question 5)

```
/* Sort Utility Class*/

public class Sorter
{
  /**  Uses Selection Sort to sort
  *      an integer array in ascending order
  *    @param the array to sort
  */
  public static void selectionSort( int [] array )
  {
   int max;  // index of maximum value in subarray

    for ( int i = 0; i < array.length; i++ )
    {
      // find index of largest value in subarray
      max = indexOfLargestElement( array, array.length - i );

      //Swap the elements at the index of the largest (max)
      // and the last index in our sub-array (see notes)
      swap(array, max, array.length - i - 1);
     }
  }

  /**  Performs a Bubble Sort on an integer array,
  *     Note: this version does not stop once the array is sorted
  *    @param array to sort
  */
  public static void bubbleSort( int [] array )
  {
     for ( int i = 0; i < array.length – 1; i++ )
     {

        for ( int j = 0; j < array.length - i - 1; j++ )
        {
           if ( array[j] > array[j + 1] )
           {
              // swap the adjacent elements
              swap(array, j+1, j);
           }
        }
     }
  }


//Continued...
```

```
    /**  Finds index of largest element
    *    @param    size  the size of the subarray
    *    @ return  the index of the largest element in the subarray
    */
    private static int indexOfLargestElement(int[] array,int size )
    {
      int index = 0;
      for( int i = 1; i < size; i++ )
      {
         if ( array[i] > array[index] )
            index = i;
      }
      return index;
    }

    /**  Swaps 2 elements in a given array
    *   @param array  the array on which we are to perform the swap
    *    @param    index1  the location of the 1st element
    *    @param    index2  the location of the 2nd element
    */
    private static void swap( int[] array, int index1, int index2)
    {
      int temp = array[index1];
      array[index1] = array[index2];
      array[index2] = temp;
    }

  }
```

Appendix D: part 2 - Searcher class

```
public class Searcher
{

  //This method will return the index of the searchItem in the array
  //If it doesn't find the searchItem, it will return -1 to indicate
  //this
  public static int binarySearch(int[] list, int searchItem)
  {
    int start=0;
    int end = list.length - 1;

    int mid = 0;

    boolean found = false;

    //Loop until found or end of list.
    while(start <= end && !found)
    {
        mid = (start + end) /2;

        if(list[mid] == searchItem)
        {
          found = true;
        }
        else
        {
            if(list[mid] > searchItem)
            {
               end = mid -1;
            }
            else
            {
               start = mid + 1;
            }
        }
    }

    if(found)
    {
       return mid;
    }
    else
    {
       return(-1);
    }
  }
}
```

Appendix E – BankAccount class (Question 6)

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount{

   // declare instance variables
   private double balance;

   /**
      Constructs a bank account with a zero balance
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constructs a bank account with a given balance
      @param initialBalance the initial balance
  */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
     return balance;
   }

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

  /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
  */
   public void withdraw(double amount)
   {

        balance = balance - amount;
   }
}
```

Appendix E – CheckingAccount class (Question 6)

```
/**
   A checking account that charges transaction fees.
*/
public class CheckingAccount extends BankAccount
{
   private int transactionCount;
   private double balance;

   private static final int FREE_TRANSACTIONS = 2;
   private static final double TRANSACTION_FEE = 1.5;

   /**
      Constructs a checking account with a given balance.
      @param initialBalance the initial balance
   */
   public CheckingAccount(double initialBalance)
   {
   }

   /**
      Deposit into the account. This is a transaction.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
   }

   /**
      Withdraw from the account. This is a transaction.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
   }

   /**
      Deducts the accumulated fees and resets the
      transaction count.
   */
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE *
               (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
}
```