Name：إياد امين احمد احمد الفار

Student ID：202000172

Sheet: 4

**10.2**

(*The* **BMI** *class*) Add the following new constructor in the **BMI** class:

/\*\* Construct a BMI with the specified name, age, weight,

\* feet, and inches

\*/

**public** BMI(String name, **int** age, **double** weight, **double** feet,

**double** inches)

**10.3**

(*The* **MyInteger** *class*) Design a class named **MyInteger**. The class contains:

■An **int** data field named **value** that stores the **int** value represented by this object.

■A constructor that creates a **MyInteger** object for the specified **int** value.

■A getter method that returns the **int** value.

■The methods **isEven()**, **isOdd()**, and **isPrime()** that return **true** if the value in this object is even, odd, or prime, respectively.

■The static methods **isEven(int)**, **isOdd(int)**, and **isPrime(int)** that return **true** if the specified value is

even,odd, or prime, respectively.

■The static methods **isEven(MyInteger)**, **isOdd(MyInteger)**, and **isPrime(MyInteger)** that return **true** if the specified value is

even, odd, or prime, respectively.

■The methods **equals(int)** and **equals(MyInteger)** that return **true** if the value in this object is equal to the specified value.

■A static method **parseInt(char[])** that converts an array of numeric characters to an **int** value.

■A static method **parseInt(String)** that converts a string into an **int** value.

Write a client program that tests all methods in the class.

Sol:

public class Test{

public static void main(String[] args) {

    MyInteger n1= new MyInteger(3);

    MyInteger n2= new MyInteger(3);

    System.out.println(n1.equals(3));

    System.out.println(n1.equals(4));

    System.out.println(n1.equals(n2));

    System.out.println(n1.isEven());

    System.out.println(n1.isOdd());

    System.out.println(n1.isPrime());

    System.out.println(MyInteger.parseInt("5"));

}}

class MyInteger {

    private int value;

    public MyInteger(int value) {this.value = value;}

    public int getInt() {return value;}

    public boolean isEven(){return (value%2==0);}

    public boolean isOdd() {return (value%2==1);}

    public boolean isPrime() {

            if (value <= 1)

                return false;

            for (int i = 2; i < Math.sqrt(value); i++)

                if (value % i == 0)

                    return false;

            return true;

    }

    public static boolean isEven(int n){return (n%2==0);}

    public static boolean isOdd(int n) {return (n%2==1);}

    public static boolean isPrime(int n) {

        if (n <= 1)

            return false;

        for (int i = 2; i < Math.sqrt(n); i++)

            if (n % i == 0)

                return false;

        return true;

    }

    public static boolean isEven(MyInteger value){return (value.value%2==0);}

    public static boolean isOdd(MyInteger value) {return (value.value%2==1);}

    public static boolean isPrime(MyInteger value) {

        if (value.value <= 1)

            return false;

        for (int i = 2; i < Math.sqrt(value.value); i++)

            if (value.value % i == 0)

                return false;

        return true;

    }

    public boolean equals(int value) {return (this.value==value);}

    public boolean equals(MyInteger value) {return (this.value==value.value);}

    static int parseInt(char[] arr){

        String x=arr.toString();

        return Integer.parseInt(x);

    }

    static int parseInt(String arr){return Integer.parseInt(arr);}

}

**10.4**

(*The* **MyPoint** *class*) Design a class named **MyPoint** to represent a point with

**x**- and **y**-coordinates. The class contains:

■ The data fields **x** and **y** that represent the coordinates with getter methods.

■ A no-arg constructor that creates a point (**0**, **0**).

■ A constructor that constructs a point with specified coordinates.

■ A method named **distance** that returns the distance from this point to a specified point of the **MyPoint** type.

■ A method named **distance** that returns the distance from this point to another point with specified **x**- and **y**-coordinates.

■ A static method named distance that returns the distance from two MyPoint objects.

Write a test program that creates the two points (**0**, **0**) and (**10**, **30.5**) and displays the distance between them.

Sol:

public class Test{

public static void main(String[] args) {

MyPoint p1 = new MyPoint();

MyPoint p2 = new MyPoint(10,30.5);

System.out.println(p1.distance(p2));

}}

class MyPoint {

    private double x;

    private double y;

    public MyPoint() {

        this.x = 0;

        this.y = 0;

    }

    public MyPoint(double x,double y){

        this.x = x;

        this.y = y;

    }

    public double distance(MyPoint p2) {

    return  Math.sqrt(Math.pow((p2.x-x),2)+Math.pow((p2.y-y),2));

    }

    public double distance(double x,double y) {

    return  Math.sqrt(Math.pow((x-this.x),2)+Math.pow((y-this.y),2));

    }

    public static double distance(MyPoint p1,MyPoint p2) {

    return  Math.sqrt(Math.pow((p2.x-p1.x),2)+Math.pow((p2.y-p1.y),2));

    }

}

**\*10.10**

(*The* **Queue** *class*) Section 10.6 gives a class for **Stack**. Design a class named

**Queue** for storing integers. Like a stack, a queue holds elements. In a stack, the

elements are retrieved in a last-in first-out fashion. In a queue, the elements are

retrieved in a first-in first-out fashion. The class contains:

■ An **int[]** data field named **elements** that stores the **int** values in the queue.

■ A data field named **size** that stores the number of elements in the queue.

■ A constructor that creates a **Queue** object with default capacity **8**.

■ The method **enqueue(int v)** that adds **v** into the queue.

■ The method **dequeue()** that removes and returns the element from the queue.

■ The method **empty()** that returns true if the queue is empty.

■ The method **getSize()** that returns the size of the queue.

Implement the class with the initial array size set to 8.

The array size will be doubled once the number of the elements exceeds the size.

After an element is removed from the beginning of the array, you need to shift all elements in the array one position to the left. Write a test program that adds 20 numbers from 1 to 20 into the queue then removes these numbers and displays them.

public class Test{

public static void main(String[] args) {

Queue q1 = new Queue();

for(int i = 0;i<20;i++)

    q1.enqueue(i+1);

for(int i = 0;i<20;i++)

    System.out.println(q1.dequeue());

}}

class Queue {

    private int[] elements;

    private int size;

    private int capacity;

    public Queue(int capacity){

        this.capacity=capacity;

        elements = new int[capacity];

    }

    public Queue() {this(8);}

    public void enqueue(int v) {

        if (size >= elements.length) {

            int[] temp = new int[size \* 2];

            System.arraycopy(elements, 0, temp, 0, size);

            elements = temp;}

        elements[size++] = v;}

    public int dequeue() {

        int v = elements[0];

        size--;

        for (int i = 0; i < size; i++)

            elements[i] = elements[i + 1];

        return v;}

    public boolean empty(){

        return (size==0);

    }

    public int getSize(){

        return size;

    }}

**\*10.11**

(*Geometry: the* **Circle2D** *class*) Define the **Circle2D** class that contains:

■ Two **double** data fields named **x** and **y** that specify the center of the circle with getter methods.

■ A data field **radius** with a getter method.

■ A no-arg constructor that creates a default circle with (**0**, **0**) for (**x**, **y**) and **1** for **radius**.

■ A constructor that creates a circle with the specified **x**, **y**, and **radius**.

■ A method **getArea()** that returns the area of the circle.

■ A method **getPerimeter()** that returns the perimeter of the circle.

■ A method **contains(double x, double y)** that returns **true** if the

specified point (**x**, **y**) is inside this circle (see Figure 10.21a).

■ A method **contains(Circle2D circle)** that returns **true** if the specified

circle is inside this circle (see Figure 10.21b).

■ A method **overlaps(Circle2D circle)** that returns **true** if the specified

circle overlaps with this circle (see Figure 10.21c).

Write a test program that creates a **Circle2D** object **c1** (**new Circle2D(2, 2, 5.5)**),

displays its area and perimeter, and displays the result of **c1.contains(3,**

**3)**, **c1.contains(new Circle2D(4, 5, 10.5))**, and **c1.overlaps(new**

**Circle2D(3, 5, 2.3))**.

Sol:

public class Test{

public static void main(String[] args) {

Circle2D c1 = new Circle2D(2, 2, 5.5);

    System.out.println("Area: "+c1.getArea()+" Perimeter: "+c1.getPerimeter());

    System.out.println(" contains(3,3): "+c1.contains(3, 3)+" contains(4, 5, 10.5): "

    +c1.contains(new Circle2D(4, 5, 10.5))+ " overlaps(3, 5, 2.3): "+c1.overlaps(new Circle2D(3, 5, 2.3)));

}}

class Circle2D {

    private double x;

    private double y;

    private double radius;

    public Circle2D(double x,double y,double radius) {

        this.x=x;

        this.y=y;

        this.radius=radius;

    }

    public Circle2D(){

        this(0,0,1);

    }

    public double getX(){

        return this.x;

    }

    public double getY(){

        return this.y;

    }

    public double getRadius(){

        return this.radius;

    }

    public double getArea(){

        return (Math.PI\*radius\*radius);

    }

    public double getPerimeter(){

        return (Math.PI\*radius\*2);

    }

    public boolean contains(double x, double y) {

        return Math.sqrt(Math.pow(x-this.x,2)+Math.pow(y-this.y,2)) < radius;

    }

    public boolean contains(Circle2D c2) {

        return Math.sqrt(Math.pow(c2.x-x,2)+Math.pow(c2.y-y,2)) <= Math.abs(radius-c2.radius);

    }

    public boolean overlaps(Circle2D c2) {

        return Math.sqrt(Math.pow(c2.x-x,2) + Math.pow(c2.y-y,2)) <= radius+c2.radius;

    }

}

**\*10.13**

(*Geometry: the* **MyRectangle2D** *class*) Define the **MyRectangle2D** class that

contains:

■ Two **double** data fields named **x** and **y** that specify the center of the rectangle

with getter and setter methods. (Assume the rectangle sides are parallel to **x-** or **y-**axis.)

■ The data fields **width** and **height** with getter and setter methods.

■ A no-arg constructor that creates a default rectangle with (**0**, **0**) for (**x**, **y**) and **1** for both **width** and **height**.

■ A constructor that creates a rectangle with the specified **x**, **y**, **width**, and **height**.

■ A method **getArea()** that returns the area of the rectangle.

■ A method **getPerimeter()** that returns the perimeter of the rectangle.

■ A method **contains(double x, double y)** that returns **true** if the specified point (**x**, **y**) is inside this rectangle

■ A method **contains(MyRectangle2D r)** that returns **true** if the specified rectangle is inside this rectangle (see Figure 10.24b).

■ A method **overlaps(MyRectangle2D r)** that returns **true** if the specified

rectangle overlaps with this rectangle

Write a test program that creates a **MyRectangle2D** object **r1** (**new MyRectangle2D (2, 2, 5.5, 4.9)**),

displays its area and perimeter, and displays the result of **r1.contains(3, 3)**, **r1.contains(new MyRectangle2D(4, 5, 10.5, 3.2))**, and **r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4))**.

Sol:

public class Test{

public static void main(String[] args) {

    MyRectangle2D r1 = new MyRectangle2D(2, 2, 5.5, 4.9);

    System.out.println("Area: "+r1.getArea()+" Perimeter: "+r1.getPerimeter());

    System.out.println(" contains(3,3): "+r1.contains(3, 3)+" contains(4, 5, 10.5,3.2): "

    +r1.contains(new MyRectangle2D(4, 5, 10.5,3.2))+ " overlaps(3, 5, 2.3,5.4): "+r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4)));

}}

class MyRectangle2D {

    private double x;

    private double y;

    private double width;

    private double height;

    public MyRectangle2D(double x,double y,double width,double height) {

        this.x=x;

        this.y=y;

        this.width=width;

        this.height=height;

    }

    public MyRectangle2D(){

        this(0,0,1,1);

    }

    public double getX(){

        return this.x;

    }

    public double getY(){

        return this.y;

    }

    public double getWidth(){

        return this.width;

    }

    public double getHeight(){

        return this.height;

    }

    public double getArea(){

        return (width\*height);

    }

    public double getPerimeter(){

        return (2\*(width+height));

    }

    public boolean contains(double x, double y) {

        return Math.sqrt(Math.pow((x-this.x),2)+Math.pow((y-this.y),2)) <

        Math.sqrt(Math.pow((width-this.x),2)+Math.pow((height-this.y),2));

    }

    public boolean contains(MyRectangle2D r2) {

        return Math.sqrt(Math.pow((r2.x-this.x),2)+Math.pow((r2.y-this.y),2)) <

        Math.sqrt(Math.pow((width-this.x),2)+Math.pow((height-this.y),2)) &&

        r2.height< this.height && r2.width<this.width;

    }

    public boolean overlaps(MyRectangle2D r2) {

        return Math.sqrt(Math.pow((r2.x-this.x),2)+Math.pow((r2.y-this.y),2)) < this.width ||

        Math.sqrt(Math.pow((r2.x-this.x),2)+Math.pow((r2.y-this.y),2)) < this.height;

    }

}

**\*10.14**

(*The* **MyDate** *class*) Design a class named **MyDate**. The class contains:

■ The data fields **year**, **month**, and **day** that represent a date. **month** is 0-based, i.e., **0** is for January.

■ A no-arg constructor that creates a **MyDate** object for the current date.

■ A constructor that constructs a **MyDate** object with a specified elapsed time

since midnight, January 1, 1970, in milliseconds.

■ A constructor that constructs a **MyDate** object with the specified year, month, and day.

■ Three getter methods for the data fields **year**, **month**, and **day**, respectively.

■ A method named **setDate(long elapsedTime)** that sets a new date for the object using the elapsed time.

Write a test program that creates two **MyDate** objects (using **new MyDate()** and **new MyDate(34355555133101L)**)

and displays their year, month, and day. (*Hint*: The first two constructors will extract the year, month, and day

from the elapsed time. For example, if the elapsed time is **561555550000**

milliseconds, the year is **1987**, the month is **9**, and the day is **18**. You may

use the **GregorianCalendar** class to simplify coding.)

Sol:

import java.util.Date;

import java.util.GregorianCalendar;

public class Test{

public static void main(String[] args) {

    MyDate d1 = new MyDate();

    MyDate d2 = new MyDate(34355555133101L);

    MyDate d3 = new MyDate(561555550000L);

    System.out.println(d1.getYear()+" "+d1.getMonth()+" "+d1.getDay());

    System.out.println(d2.getYear()+" "+d2.getMonth()+" "+d2.getDay());

    System.out.println(d3.getYear()+" "+d3.getMonth()+" "+d3.getDay());

}}

class MyDate {

    private int year;

    private int month;

    private int day;

    private GregorianCalendar calendar = new GregorianCalendar();

    public MyDate(long elapsed) {

        calendar.setTimeInMillis(elapsed);

            year = calendar.get(GregorianCalendar.YEAR);

            month = calendar.get(GregorianCalendar.MONTH);

            day = calendar.get(GregorianCalendar.DAY\_OF\_MONTH);

    }

    public MyDate(){

        this.year = GregorianCalendar.YEAR;

        this.month = GregorianCalendar.MONTH;

        this.day = GregorianCalendar.DAY\_OF\_MONTH;

    }

    public MyDate(int year,int month,int day){

        this.year=year;

        this.month=month;

        this.day=day;

    }

    public int getYear(){

        return this.year;

    }

    public int getMonth(){

        return this.month;

    }

    public int getDay(){

        return this.day;

    }

    public void setDate(long elapsedTime) {

        calendar.setTimeInMillis(elapsedTime);

        day = (int)(elapsedTime/86400000);

        month = (int)((elapsedTime/30)%day);

        year = (int)((elapsedTime/12)%month);

    }

}

**\*10.17**

(*Square numbers*) Find the first 10 square numbers that are greater than **Long.**

**MAX\_VALUE**. A square number is a number in the form of *n*2. For example, 4, 9,

and 16 are square numbers. Find an efficient approach to run your program fast.

Sol:

import java.math.BigInteger;

public class Test{

public static void main(String[] args) {

    BigInteger squares = new BigInteger(Long.MAX\_VALUE+ "");

    squares = squares.add(BigInteger.ONE);

    BigInteger num = squares;

    for (int i = 0; i < 10; i++) {

        num = squares.multiply(squares);

        System.out.println(num);

        squares = squares.add(BigInteger.ONE);

    }}

}

**\*10.19**

(*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be

written in the form 2*p* - 1 for some positive integer *p*. Write a program that

finds all Mersenne primes with *p* … 100 and displays the output as shown

below. (*Hint*: You have to use **BigInteger** to store the number because it is

too big to be stored in **long**. Your program may take several hours to run.)

Sol:

import java.math.BigInteger;

class Test

{

   public static void main(String args[]){

      int status = 1;

      int num = 3;

      System.out.println("p      2^p-1");

      System.out.println("--------------");

      for ( int i = 2 ; i <=25 ;  ){

         for ( int j = 2 ; j <= Math.sqrt(num) ; j++ ){

            if ( num%j == 0 ){

               status = 0;

               break;

            }

         }

         if ( status != 0 ){

            BigInteger number = new BigInteger(num+"");

            number = BigInteger.TWO.pow(num).subtract(BigInteger.ONE);

            System.out.println(num + "      "+ number );

            i++;

         }

         status = 1;

         num++;

}

}

}