Com S 228
Spring 2018
Exam 1

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: _____

ISU NetID (username): _____

Recitation section (please circle one):

1. R 10:00 am (Xinxin and Tanmay)
2. R 2:10 pm (Mike P. and Arnoldo)
3. R 1:10 pm (Gabriel and Nirala)
4. R 4:10 pm (Christine and Nirala)
5. R 3:10 pm (Jason and Yuechuan)
6. T 9:00 am (Jacob and Waqwoya)
7. T 2:10 pm (Mike L. and Jason)
8. T 10:00 am (Andrew and Waqwoya)

Closed book/notes, no electronic devices, no headphones. Time limit 60 minutes. Partial credit may be given for partially correct solutions.

- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

If you have questions, please ask!

Question	Points	Your Score
1	27	
2	25	
3	24	
4	24	
Total	100	

1. (27 pts; 3 each) Refer to the class hierarchy on pages 15–18 to answer the questions below. (It helps to peel off pages 15–19 from your exam sheets for code lookup convenience and scratch purpose.) For each section of code, fill in the box stating one of the following:

   - the output, if any, *or*
   - that there is a compile error (briefly explain the error), *or*
   - the type of exception that occurs at runtime.

`MoneyMarketAccount sharpay =`
`  new MoneyMarketAccount("Sharpay␣Evans", 1,`
`  2000000.00, 4.25);`
`System.out.println(sharpay.getInterestRate());`	

`Account troy =`
`  new SavingsAccount("Troy␣Bolton", 2,`
`  3000.00, 1.125);`
`System.out.println(troy.getBalance());`	

`Account troy =`
`  new SavingsAccount("Troy␣Bolton", 2,`
`  3000.00, 1.125);`
`troy.withdraw(5000);`
`System.out.println(troy.getBalance());`	

`Account kelsi =`
`  new SavingsAccount("Kelsi␣Nielsen", 3,`
`  500.00, 1.125);`
`System.out.println(kelsi.getAccountInfo());`	

`CheckingAccount darbus =`
`  new MoneyMarketAccount("Mrs.␣Darbus", 4,`
`  10000.00, 4.25);`
`darbus.calculateInterest(365);`	

`InterestBearing chad =`
`  new CheckingAccount("Chad␣Danforth", 5,`
`  777.77);`
`System.out.println(chad.getInterestRate());`	

`InterestBearing ryan =`
`  new SavingsAccount("Ryan␣Evans", 6,`
`  400000.00, 1.125);`
`System.out.println(ryan.getInterestRate());`	

`InterestBearing gabriella =`
`  new MoneyMarketAccount("Gabriella␣Montez",`
`  10000, 7, 4.25);`
`System.out.println(`
`  ((MoneyMarketAccount)`
`  gabriella).getAccountInfo());`	

`CheckingAccount taylor =`
`  new CheckingAccount("Taylor␣McKessie", 8,`
`  7000.00);`
`System.out.println(`
`  ((InterestBearing) taylor).getInterestRate());`	

2. (25 pts) You are given two classes `Complex` and `ComplexTuple` on page 19. They implement complex numbers and tuples of complex numbers, respectively.

   Hint: In both a and b, below, you are given *significantly* more space than you should need.

   a) (10 pts) Add a `clone()` method to the `Complex` class; the method must override `java.lang.Object`'s `clone()` method. The signature is shown below; you just need to fill in the `try` and `catch` blocks.

```
@Override
public Object clone()
{
  try {
    Complex c = (Complex) super.clone();









  }
  catch (CloneNotSupportedException e) {









   }
}
```

b) (15 pts) Add a method to the `ComplexTuple` class that overrides the `equals()` method from `java.lang.Object` to perform a deep comparison between `this` object and an existing `ComplexTuple` object. Two `Complexes` are considered to be equal if they have the same real and imaginary parts. You can assume that `Complex` has a correctly implemented `equals()` method.

```
@Override
public boolean equals(Object o)
{
  // TODO




}
```

3. (24 pts; 6 each) Determine the worst-case execution time of each of the following methods as a function of the length of the input array(s). Express each of your answers as big-O of a simple function (which should be the simplest and slowest-growing function you can identify). For convenience, the analysis of each part has been broken down into multiple steps. For each step, you just need to fill in the blank a big-O function as the answer (in the ***worst case*** always).

a)

```
void methodA(int[] arr)
{
  int p = 0;
  int n = arr.length;
  for (int i = 0; i < n; i++)
  {
    for (int j = i; j < n; j++)
    {
      p += arr[j] % 2;
    }
  }
}
```

i) Number of iterations of the outer `for` loop: _____

ii) Number of iterations of the inner `for` loop: _____

iii) Worst-case execution time: _____

b) Assume that the method `foo()` takes $O(n^2)$ time and method `bar()` takes $O(n^3)$ time.

```
public static void methodB(int[] arr)
{
  int n = arr.length;
  while (n > 0)
  {
    foo(arr);
    n = n / 4;
  }

  bar(arr)
}
```

i) Number of iterations of the `while` loop: _____

ii) Time per iteration: _____

iii) Total time for the `while` loop: _____

iv) Total worst-case execution time for `methodB`: _____

c)

```
public static boolean hasI(int[] arr, int i)
{
   if (i == arr.length)
      return false;
   return arr[i] == 'i' ? true : hasI(arr, i + 1);
}
```

Suppose `arr` has length n, where n is at least 1. Assume that we call `hasI(arr, 0)`.

i) Number of recursive calls to `hasI`: _____

ii) Worst-case execution time: _____

d) Consider the following algorithm, which takes four `int` arrays `A`, `B`, `C`, and `D`, each of length n, and returns a new array that contains the combined elements of `A`, `B`, `C`, and `D` in sorted order. `Merge` is the algorithm for merging two sorted arrays that we saw in class.

```
QuadSort(A, B, C, D):
   sort A using MergeSort
   sort B using MergeSort
   sort C using MergeSort
   sort D using MergeSort
   p = A.length, q = B.length, r = C.length, s = D.length
   create an empty array E of length p + q
   E = Merge(A, B)
   create an empty array F of length r + s
   F = Merge(C, D)
   t = E.length, u = F.length
   create an empty array G of length t + u
   G = Merge(E, F)
   return G
```

What is the big-O time complexity of this algorithm? (For partial credit, to the right of each step of the algorithm write down the big-O time that it takes.)

4. (24 pts; 4 each) The following tables list the input array (first line), output array (last line), and internal array state in sequential order for each of the sorts that we have studied in class (SELECTIONSORT, INSERTIONSORT, MERGESORT, and QUICKSORT). The two $O(n^2)$ sorts print internal results as the last operation of their outer loops. MERGESORT prints the output array after each call to MERGE. QUICKSORT prints after each call to PARTITION. Array contents occupy rows in the following tables, with the top rows containing the input and proceeding down through time to the output on the bottom.

   There is exactly one right answer to each problem. a–d are multiple choice. In parts e and f, you must fill in the contents of the empty row.

a)

0	6	5	7	4	1	2	3
0	6	5	7	4	1	2	3
0	5	6	7	4	1	2	3
0	5	6	7	4	1	2	3
0	4	5	6	7	1	2	3
0	1	4	5	6	7	2	3
0	1	2	4	5	6	7	3
0	1	2	3	4	5	6	7

A) SELECTIONSORT B) INSERTIONSORT C) MERGESORT D) QUICKSORT

b)

5	4	7	1	2	6	0	3
4	5	7	1	2	6	0	3
4	5	1	7	2	6	0	3
1	4	5	7	2	6	0	3
1	4	5	7	2	6	0	3
1	4	5	7	2	6	0	3
1	4	5	7	0	2	3	6
0	1	2	3	4	5	6	7

A) SELECTIONSORT B) INSERTIONSORT C) MERGESORT D) QUICKSORT

c)

5	4	7	1	2	6	0	3
1	2	0	3	4	6	7	5
0	2	1	3	4	6	7	5
0	1	2	3	4	6	7	5
0	1	2	3	4	5	7	6
0	1	2	3	4	5	6	7

A) SELECTIONSORT B) INSERTIONSORT C) MERGESORT D) QUICKSORT

d)

0	6	5	7	4	1	2	3
0	6	5	7	4	1	2	3
0	1	5	7	4	6	2	3
0	1	2	7	4	6	5	3
0	1	2	3	4	6	5	7
0	1	2	3	4	6	5	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A) SELECTIONSORT B) INSERTIONSORT C) MERGESORT D) QUICKSORT

e) Fill in the missing row in this INSERTIONSORT output.

1	6	4	7	0	5	3	2
1	6	4	7	0	5	3	2
1	4	6	7	0	5	3	2
1	4	6	7	0	5	3	2
0	1	4	5	6	7	3	2
0	1	3	4	5	6	7	2
0	1	2	3	4	5	6	7

f) Fill in the missing row in this MERGESORT output.

1	6	4	7	0	5	3	2
1	6	4	7	0	5	3	2
1	6	4	7	0	5	3	2
1	4	6	7	0	5	3	2
1	4	6	7	0	5	3	2
1	4	6	7	0	5	2	3
0	1	2	3	4	5	6	7

Scratch paper

Scratch paper

Scratch paper

Scratch paper

Sample Code for Problem 1

```
interface InterestBearing {
  double getInterestRate();
  void setInterestRate(double newRate);
  void calculateInterest(int time);
}

public abstract class Account {
  protected String name;
  protected int number;
  protected double balance;

  public Account(String name, int number, double initialBalance)
  {
    this.name = name;
    this.number = number;
    balance = initialBalance;
  }

  public String getName()
  {
    return name;
  }

  public int getAccountNumber()
  {
    return number;
  }

  public double getBalance()
  {
    return balance;
  }

  public void deposit(double amount)
  {
    System.out.println("Deposited " + amount);
    balance += amount;
  }
```

```
  public void withdraw(double amount)
  {
    if (amount > balance) {
      throw new IllegalArgumentException();
    }
    System.out.println("Withdrew " + amount);
    balance -= amount;
  }

  public abstract String getAccountInfo();
}

class SavingsAccount extends Account implements InterestBearing {
  protected double rate;
  public SavingsAccount(String name, int number,
                        double initialBalance, double interestRate)
  {
    super(name, number, initialBalance);
    rate = interestRate;
  }

  @Override
  public double getInterestRate()
  {
    return rate;
  }

  @Override
  public void setInterestRate(double newRate)
  {
    System.out.println("Rate changed to " + newRate);
    rate = newRate;
  }

  @Override
  public void calculateInterest(int time)
  {
    System.out.println("Calculating interest on " + balance + " at " +
                       rate + "% over " + time + " days");
  }
```

```
  @Override
  public String getAccountInfo()
  {
    return "Savings: " + name + ", " + number;
  }
}

class CheckingAccount extends Account {
  public CheckingAccount(String name, int number, double initialBalance)
  {
    super(name, number, initialBalance);
  }


  public void writeCheck(double amount)
  {
    System.out.println("Wrote check for " + amount);
    balance -= amount;
  }

  public void checkOverdraft()
  {
    if (balance < 0) {
      System.out.println("Account is overdrawn");
    }
  }

  @Override
  public String getAccountInfo()
  {
    return "Checking: " + name + ", " + number;
  }
}

class MoneyMarketAccount extends CheckingAccount implements InterestBearing {
  protected double rate;
  public MoneyMarketAccount(String name, int number,
                            double initialBalance, double interestRate)
  {
    super(name, number, initialBalance);
    rate = interestRate;
  }
```

```
  @Override
  public double getInterestRate()
  {
    return rate;
  }

  @Override
  public void setInterestRate(double newRate)
  {
    System.out.println("Money␣market␣interest␣rate␣changed␣to␣" + newRate);
    rate = newRate;
  }

  @Override
  public void calculateInterest(int time)
  {
    System.out.println("Calculating␣money␣market␣interest␣on␣" +
                       balance + "␣at␣" + rate + "%␣over␣" + time + "␣days");
  }

  @Override
  public String getAccountInfo()
  {
    return "Money␣Market:␣" + name + ",␣" + number;
  }

}
```

Sample code for Problem 2

```
public class Complex implements Cloneable
{
   private int re;  // real part

   private int im;  // imaginary part

   public Complex(int re, int im)
   {
      this.re = re;
      this.im = im;
   }

   @Override
   public boolean equals(Object o)
   {
      // Assume this is already implemented; the
      // implementation details irrelevant.
   }
}

public class ComplexTuple
{
   private Complex c1;
   private Complex c2;

   public ComplexTuple(Complex c1, Complex c2)
   {
      this.c1 = c1;
      this.c2 = c2;
   }
}
```