Topics

• Object-Oriented Design: Inheritance
  – Know how to interpret UML class diagrams
  – Understand when the usage of inheritance is appropriate
  – Be able to refactor poorly-designed classes for better code maintenance by maximizing code reuse and taking advantage of inheritance
  – Be familiar with the `Object` class, what it provides for programmers, and its general purpose.

• Polymorphism
  – Be able to articulate the need/motivation for polymorphism.
  – Understand what a polymorphic variable is.
  – Be able to make good design decisions, i.e., determine whether a class, abstract class, and/or interface should be used for a particular problem.
  – Know when it is appropriate to down-cast.
  – Understand the dynamic method dispatch mechanism.
  – Understand the `Comparable` interface, and what it provides to classes that implement it.
  – Articulate the use of: `super()`, `super.method()`, `this()`, `@Override`

• Exception Handling
  – Understand the mechanism of exceptional execution.
  – Know the difference between a checked vs. an unchecked exception, and the compiler’s role in dealing with either.
  – Be familiar with the use of try-catch blocks and the throws command.
  – Know how to throw (raise) an exception.
  – Know how to create and use your own exception types.

• Reading/Writing Code
  – Expect to read and write short snippets of code on the exam.
  – Expect that examples (e.g., Library, TicTacToe, ...) used in lab and homework would show up on the exam.
  – Understand the cases in which Java automatically injects code for you and its consequences.
  – private vs. protected vs. public.
  – Given only the API documentation of a class, be able to use it when writing code.
• Complexity
  – Understand the formal definition of the Big-$O$ notation.
  – Be able to give a tight bound for a simple algorithm, though you don’t need to know how to find $c$ and $n_0$. 
Practice Problems

1. For each of the following Java keywords, describe what it is used for, and write a snippet of code demonstrating its use.

   (a) this(..)

   (b) abstract

   (c) implements

   (d) throws

   (e) @Override
2. Consider the following class, which represents a “pixel” value. A pixel is assumed to be monochromatic – it stores a single integer between 0 (black) and 255 (white).

```java
public class Pixel {
    public static final int MIN_VAL = 0;
    public static final int MAX_VAL = 255;
    private int val;

    public Pixel(int val) {
        setValue(val);
    }

    public int getValue() {
        return val;
    }

    public void setValue(int val) throws IllegalPixelException {
        if (val >= MIN_VAL && val <= MAX_VAL) {
            this.val = val;
        } else {
            throw new IllegalPixelException("Invalid pixel value "+ val);
        }
    }

    public void brighter() {
        this.val++;
    }

    @Override
    public String toString() {
        return "," + this.val;
    }
}
```

(a) Assuming `IllegalPixelException` is checked, the above code won’t compile. State why, and fix the code so that it does. In your fix, the program should recover gracefully and not crash.

(b) Why is `brighter()` unsafe? Fix the code so that it is safe to use.

(c) Define a `ColorPixel` class, which has three integer values that can range from 0 to 255. Each integer now correspond to a red value, blue value, and a green value. Define this class, inheriting from `Pixel` to maximizing code reuse and maintainability, and provide the following elements:

i. A default constructor that sets all three values to 0.

ii. Overloaded constructor that accepts inputs for all three values.

iii. A `toString()` method that returns the three values in order of red, blue, and green, as a triple. For instance, “(40, 100, 240)” if the color pixel has a red value of 40, blue value of 100, and green value of 240.

iv. `ColorPixels` need to be sortable. They are to be ordered by their red, blue, then green values, respectively. For instance, (0,100,200) would be ordered before (0,101,0).
3. Consider the code below for the next few problems:

```java
public class One {
    protected int i;
    public String toString() {
        return "" + i;
    }
}

public class Two extends One {
    public Two(int initial) {
        i = initial;
    }
}

public class Three extends Two {
    public Three() {
        i = 10;
    }
    public int getValue() {
        return i;
    }
}
```

(a) Can an instance of the class `One` be constructed via a default constructor? Explain.

(b) Can an instance of the class `Two` be constructed via a default constructor? Explain.

(c) Can an instance of the class `Two` be constructed via a one-argument constructor? Explain.

(d) Can an instance of the class `Three` be constructed via a default constructor? Explain.

(e) Can an instance of the class `Three` be constructed via a one-argument constructor? Explain.
4. What does the following method do? Write a $T(\cdot)$ function that represents the worst-case time complexity. What’s the $O(\cdot)$ for this method? You do not need to justify your answer by providing $c$ and $n_0$.

```java
/** assume n >= 0 */
public static int foo(int n) {
    int i = 0;
    while (n > 1) {
        n /= 10;
        i++;
    }
    return i;
}
```

5. What does the following method do? Assume that $x$ has $m$ elements. $T(\cdot)$ function that represents the worst-case time complexity. What’s the $O(\cdot)$ for this method?

```java
public static int[] tmp(int[] x, int n) {
    while (n > 0) {
        int tmp = x[x.length-1];
        for (int j = x.length-1; j > 0; j--) {
            x[j] = x[j-1];
        }
        x[0] = tmp;
        n--;
    }
    return x;
}
```