CS 2110, FA24

Discussion 4: Prelim 1 Review

Topics

- Procedural programming in Java
- Compile-time and runtime
- Classes
- Testing
- Object-oriented programming
- Exceptions
- Data Structures
- Efficiency

Procedural programming in Java

Classify the following as either a *primitive type*, a *reference type*, or *not a type name*:

- `Object`
- `char`
- `5`
- `String`
- `null`
- `int[]`

Predict the result of running the below program

```
int[] arr = new int[] {1, 2, 4, 8, 16, 32, 64, 128};
for (int i = 0; i < arr.length; i += 1) {
    int temp = arr[arr.length - i - 1];
    arr[arr.length - i - 1] = arr[i];
    arr[i] = temp;
}
```

Complete this short method given the specification

```
/** Returns a new String with the characters of s in reverse order.
  * ex. reverseString("hello") => "olleh".
  * Requires: s is not null.
  * You may not use any Java methods or classes beyond length(),
  * charAt(), and concatenation operators. */

public static String reverseString(String s) {

    // Your code here!

}
```

Determine whether these lines of code run and the value of the variables.

```
int x = 10 + 12.5;

int y = 10 / 3;

double z = 1 + 10;

double e = 1 / 3;

int a = (double) 1 / 2;

double b = (int) 2.5;
```

Determine the types and results of the following expressions:

(double) 6/8

(int) 10.5 * 5.1

10 + 5 / 2 + (int) 11.1 - (double) 10

Compile-time and run-time

Given the following class hierarchy and code:

```
interface I1 { }

interface I2 { }

class A implements I2 { }

class B extends A implements I1, I2 { }
```

```
// Main Method

B b = new B();

I2 i2 = b;
```

Determine if the following code compiles, and if not, specify whether there is a runtime or compile-time error.

```
a)  I1 k = (I2) b;

b)  I1 k2 = b;

c)  I1 k3 = i2;

d)  String s = i2.toString();
```

Classes in Java

Class Diagrams

Given the following class, please draw a class diagram:

```
public class Student {
    private String name;
    private String netId;
    private int credits;

    public String name() {
       return name;
    }

    public String netId() {
       return netId;
    }

    public void modifyCredits(int creditChange) {
       credits += creditChange;
    }
}
```

Label the return type, parameters, specification, keywords, types and literals in the method below:

```
 /**
  * This method returns true if every character in String word consists of
  * lowercase english alphabet ('a' - 'z'), and false if otherwise.
  * Requires: word is not null or empty ("").
  */
 public static boolean isAllLowerCase(String word) {
   for (int i = 0; i < word.length(); i++) {
     char currentChar = word.charAt(i);
     if (currentChar < 'a' || currentChar > 'z') {
       return false;
     }
   }
   return true;
 }
}
```

Implement isSolved() according to the specification

```
/** A class representing a single row of cells in a Sudoku game */
public class SudokuRow {
    /** The values in each of the cells in the row.
     * Each element is either filled with a number 1-9 or is an empty cell, marked by a 0
     * Invariant: Only contains values in the range 0-9 inclusive.
     * Invariant: Each number in range 1-9 inclusive can only appear at most once in the row.
     */
    private int[] cells;

    // Other fields, constructors, and methods omitted

    /** Returns whether the row has been solved. A row has been solved if there are no empty cells
in the row
    */
    public boolean isSolved() {
        //TODO
    }
}
```

Testing

Given the method specification, write at least three **black box tests**, stating the input and expected output

Recap: Black box testing is a technique of testing where the functionality of the software is tested by only looking at the specifications and without looking at the code.

```
 /**
  * Returns the average sum of the first k elements of arr. If arr is empty,
  * returns 0, and if k > arr.length, returns the average sum of all elements in
  * arr.
  *
  * Requires: k > 0, arr is not null
  */
 public double averageOfFirstKElements(int[] arr, int k) {
     //implementation here
}
```

Object-oriented programming in Java

What will happen when we try to compile and run A and B?

```
public class Animal {
        public void makeNoise() {
                System.out.println("This animal is making its call");
                call();
        }

        public void call() {
                System.out.println("Grunt");
        }
}

public class Cat extends Animal {
        public void call() {
                System.out.println("Meow");
        }

        public void pet() {
                System.out.println("Purr");
        }
}
```

A

```
public static void main(String args[]) {
        Animal oliver = new Cat();
        oliver.makeNoise();
}
```

B

```
public static void main(String args[]) {
        Animal oliver = new Cat();
        oliver.pet();
}
```

Does the following equals() method for the Player class satisfy all the properties of an equivalence relation? If not, which ones does it violate

```
public class Player {
    public String playerName;
    public int jerseyNo;
    public String team;

    public boolean equals(Object obj) {
        if (!obj instance of Player) {return false;}
        Player pl = (Player) obj;
        if (this.jerseyNo > pl.jerseyNo) {
          return this.playerName.equals(pl.playerName)
              && this.team.equals(pl.team);
        }
        return this.playerName.equals(pl.playerName);
    }
}
```

Does Class SuperSonics implement Interface NBATeam? Are there any compile-time errors?

(There are no specifications, so we can't say whether the implementation is *correct*; we're just interested in whether it compiles for now.)

```
public interface NBATeam {

    public double winPercent();

    public String nextGame();

}
```

```
public class SuperSonics implements NBATeam {
    int gamesPlayed;
    double winPercent;
    String[] schedule;
    public SuperSonics(){
        gamesPlayed = 0;
        this.winPercent = 0.0;
        this.schedule = null;
        // the team no longer exists, so the schedule will
        always be null
    }
    public double winPercent() {
        return winPercent;
    }
    public String nextGame() {
        return schedule[gamesPlayed];
    }
}
```

Given the type hierarchy and variable declarations, state whether the following lines of code will compile and if so, whether they will always succeed.

Goose <: Waterfowl <: Bird

Duck <: Waterfowl <: Bird

Robin <: SongBird <: Bird

Bird b;

Waterfowl w;

Robin r;

Goose g;

```
Waterfowl x = (Waterfowl) b;
Bird y = (Waterfowl) w;
Robin a = (SongBird) b;
Duck d = (Duck) g;
Songbird s = (Songbird) w;
```

Given the following subtype relations:

Fish <: Animal
Mammal <: Animal
Cat <: Mammal
Dog <: Mammal

And the code on the right, answer the following questions:

1. Would this code compile? Would it fully execute at run-time?

2. If not, which line can you remove to fix the error?

3. After that change, what are the dynamic and static types of
   a. `d`
   b. `c`
   c. `animals[0]`
   d. `animals[1][1]`

```
public static void main(String[] args) {
    Animal[][] animals = new Animal[2][];

    animals[0] = new Mammal[2];
    animals[1] = new Fish[2];

    Mammal d = new Dog();
    Mammal c = new Cat();
    Fish f = new Fish();

    animals[0][0] = c;
    animals[1][0] = f;
    animals[1][1] = d;
}
```

Given the classes on the right side, what will the code below print?

```
public static void main(String [] args) {
        Vehicle[] vehicles = new Vehicle[3];
        vehicles[0] = new Bike();
        vehicles[1] = new Car();
        vehicle[2] = new Plane();
        int sum = 0;

        for (Vehicle v : vehicles) {
                sum += v.countSeats();
                if (v instanceof Bike) {
                        ((Bike) v).ride(); }
                if (v instanceof Car) {
                        ((Car) v).drive(); }
                if (v instanceof Plane) {
                        ((Plane) v).pilot(); }
        }

        System.out.println("These vehicles can hold a total of
        " + sum + " people.");
}
```

```
public class Vehicle {
        public int countSeats() { return 0; }
}

public class Bike extends Vehicle {
        public int countSeats() { return 1; }
        public void ride() {
                System.out.println("zoom");
        }
}

public class Car extends Vehicle {
        public int countSeats() { return 4; }
        public void drive() {
                System.out.println("vroom");
        }
}

public class Plane extends Vehicle {
        public int countSeats() { return 100; }
        public void pilot() {
                System.out.println("nyoom");
        }
}
```

Given the classes to the right, and variables initialized as follows:

```
Foo f1 = new Foo(1,2);
Foo f2 = new Foo(2,1);
Foo f3 = new Foo(1,2);
Bar b1 = new Bar(1,2);
Bar b2 = new Bar(2,1);
Bar b3 = new Bar(2,1);
```

What would the following statements evaluate to?

1. f1.equals(f2);
2. f1.equals(f3);
3. f1 == f3;
4. b1 == b1;
5. b1 == b2;
6. b2 == b3;
7. b2.equals(b3);

```
public class Foo {
	int x;
	int y;

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

	@Override
	public boolean equals(Object obj) {
		if (!(obj instanceof Foo)) {
			return false;
		}
		return this.x == ((Foo)obj)x &&
			this.y == ((Foo)obj).y;
	}
}
public class Bar {
	int x;
	int y;

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

(Challenge problem) Given the class to the right (assume it compiles with only one of the given implementations of `.equals()`), for each of the implementations determine which of these properties they satisfy: reflexive, symmetric, transitive. **Assume no strict subtype of `Foo` is passed in.**

If they do not satisfy one of the properties, give a counterexample.

1. Implementation A

2. Implementation B

3. Implementation C

```
public class Foo {
	int x;
	int y;
	boolean z;

	// Implementation A
	public boolean equals(Object o) {
		if (! (o instanceof Foo)) { return false; }
		return this.x == ((Foo)o).x &&
			this.y == ((Foo)o).y
			&& this.z == ((Foo)o).z;
	}

	// Implementation B
	public boolean equals(Object o) {
		if (! (o instanceof Foo)) { return false; }
		return this.x == ((Foo)o).y &&
			this.y == ((Foo)o).x
			&& this.z != ((Foo)o).z;
	}

	// Implementation C
	public boolean equals(Object o) {
		if (! (o instanceof Foo)) { return false; }
		if (((Foo)o).x == 0 || ((Foo)o).y == 0) {
			return false;
		}
		return (this.x % ((Foo)o).x == 0) &&
			(this.y % ((Foo)o).y == 0) &&
			this.z == ((Foo)o).z
	}
}
```