CS 2110, FA24

Discussion 4: Prelim 1 Review

Topics

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[]`

Solution

- `Object`: `reference type`
- `char`: `primitive type`
- `5`: `NOT a type name - this is a value.`
- `String`: `reference type`
- `null`: `NOT a type name - this is a value.`
- `int[]`: `reference type`

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;
}
```

Solution

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

For each pair of `[i, arr.length - 1 - i]` (`e.g. [0, 7], [1, 6], … [7, 0]`),

The loop body swaps the values. It starts iterating at `i = 0, ending at i = 7.`

Notice: They are actually swapped twice! For example, `i = 0 & i = 7` swap back and forth.

So, `arr` is actually unchanged.

Complete this short method given the specification

```java
/** 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!

}
```

Solution

```
public static String reverseString(String s) {
    String result = "";
    for (int i = s.length() - 1; i >= 0; i--) {
        result = result + s.charAt(i);
    }
    return result;
}
```

Intuition: This iterates through s in the reverse order. It then rebuilds the string character by character, and so since the iteration is reversed, the result is reversed.

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;
```

Solution

int x = 10 + 12.5;	Incompatible types (cast required)
int y = 10 / 3;	Runs; y is assigned to 3
double z = 1 + 10;	Runs; z is assigned to 11.0 (automatically widened)
double e = 1 / 3;	Runs; e is assigned to 0.0 (widens after evaluation)
int a = (double) 1 / 2;	Incompatible types
double b = (int) 2.5;	Runs; b is assigned to 2.0 (widens after evaluation)

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

Solution

(double) 6/8	0.75, double
(int) 10.5 * 5.1	51.0, double
10 + 5 / 2 + (int) 11.1 - (double) 10	13.0, double

Compile-time and run-time

Given the following class hierarchy and code:

```java
interface I1 { }

interface I2 { }

class A implements I2 { }

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

```java
// 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();`

Solution

a) **Compile-time error**: A static-typed `I2` value cannot fit into a variable with a static type of `I1`.

b) **Compiles**!

c) **Compile-time error**: A static-typed `I2` value cannot fit into a variable with a static type of `I1`.

d) **Compiles**!

Classes in Java

Class Diagrams

Given the following class, please draw a class diagram:

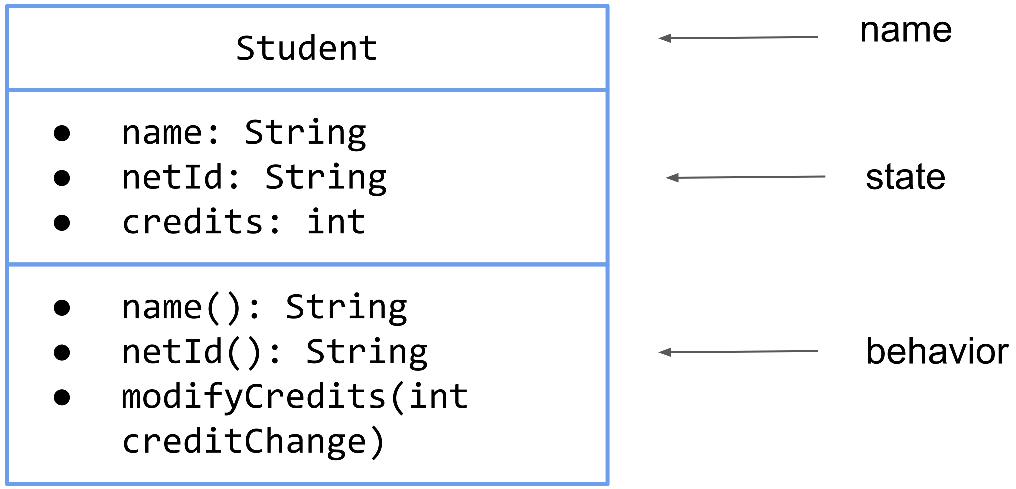
```
public class Student {
    private String name;
    private String netId;
    private int credits;

    public String name() {
       return name;
    }

    public String netId() {
       return netId;
    }

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

Solution


Student
name: String
netId: String
credits: int
name(): String
netId(): String
modifyCredits(int creditChange)
name
state
behavior

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;
 }
}
```

Solution

```
 /**
  * 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;
 }
}
```

Red: Return Type
Blue: Parameters
Green: Specification
Keywords: Purple
Static types: Magenta
Literals: Orange

Implement isSolved() according to the specification

```java
/** 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
    }
}
```

Solution

```
 public boolean isSolved() {
   for (int i = 0; i < cells.length; i++) {
     if (cells[i] == 0) {
       return false;
     }
   }
  return True;
}
```

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
}
```

Solution

1. Case 1: arr is empty:
   a. input: averageOfFirstKElements([], 3)
   b. expected output: 0
2. Case 2: k < arr.length
   a. input: averageOfFirstKElements([3,2,1,4,5], 2)
   b. expected output: 2.5
3. Case 3: k == arr.length
   a. input: averageOfFirstKElements([3,2,1,4,5], 5)
   b. expected output: 3

Note: Test cases should not violate preconditions (k should not be negative, arr should not be null)!

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();
}
```

Solution

A will compile since Cat is a subtype of Animal, and Animal contains a method called “makeNoise()”. Upon running, we invoke Animal’s method “makeNoise()”, printing “This animal is making its call”, then we invoke the method “call()”. However, since the object oliver is an instance of a Cat, we reference the bottom-up rule and use Cat’s definition of call(), yielding the printed output:

```
This animal is making its call
Meow
```

B will not compile since the bottom-up rule only applies at runtime. Since the static type of oliver is Animal, oliver.pet() is not a legal call (as the Animal class doesn’t contain a “pet()” method). To fix this problem, we would need to cast oliver to a Cat object.

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);
    }
}
```

Solution

No. The equivalence relation is not symmetric nor transitive.

- Consider a player p named Rui, with jersey number 1, who plays for Roma. Consider another player q also named Rui, with jersey number 6, but plays for Napoli. Then p.equals(q) is true, since 1 is not greater than 6 and player p and player q have the same name. But, q.equals(p) is false, since 6 is greater than 1 but players p and q don't play for the same team. Hence, this relation is not symmetric

- Consider a player m named Kim, with jersey number 9, who plays for Napoli. Consider another player t named Kim, with jersey number 7, who also plays for Napoli. Consider another player v named Kim, with jersey number 8, but plays for Roma. m.equals(t) is true since player m and player t have the same name and play for the same team (Napoli). t.equals(v) is also true since player t's jersey number (7) is less than player v's jersey number (8) and player t and player v have the same name. However, m.equals(v) is false. Player m's jersey number (9) is greater than player v's jersey number (8), but player m plays for Napoli whereas player v plays for Roma. Hence, this relation is not transitive.

- This relation is reflexive. Every player equals themselves in this relation

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];
    }
}
```

Solution

Short answer: Yes

- For a class to implement an interface, it must match the interface’s specification
    - Method signatures, return types, parameters, and specs in the implementing class must match the interface
    - For more details, see lecture 5 slides
- The methods winPercent() and nextGame() are implemented according to the interface’s specification
    - NextGame() will always throw a NullPointerException
    - However, this is a runtime exception, not a compile time error
- Therefore, Yes, the SuperSonics class technically implements the NBATeam Interface, albeit very badly

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;
```

Solution

Waterfowl x = (Waterfowl) b;	May sometimes fail at runtime
Bird y = (Waterfowl) w;	An unnecessary cast, but will always work
Robin a = (SongBird) b;	Compile time error; A SongBird cannot be assigned to a Robin variable.
Duck d = (Duck) g;	Compile time error; Cannot cast Goose to Duck
Songbird s = (Songbird) w;	Compile time error; Cannot cast Waterfowl to Songbird

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 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?
   Would compile, wouldn't fully execute at run-time
2. If not, which line can you remove to fix the error?
   Remove `animals[1][1] = d`.
3. After that change, what are the dynamic and static types of
   a. `d` Mammal, Dog
   b. `c` Mammal, Cat
   c. `animals[0]` Animal[], Mammal[]
   d. `animals[1][1]` Animal, none (it's null)

```
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 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.”);
}
```

```
zoom
vroom
nyoom
These vehicles can hold a total of 105 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;
        }
}
```

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

```
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
        }
}
```

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
   reflexive, symmetric, and transitive
2. Implementation B
   symmetric, neither reflexive nor transitive
(transitive counterexample: (1,0,true) = (0,1,false) and (0,1,false) = (1,0,true), but (1,0,true) != (1,0,true))
3. Implementation C
   reflexive, transitive, not symmetric
(counterexample: (2,2,false) != (4,4,false) but (4,4,false) = (2,2,false))

```
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
          }
}
```