

CS61B SPRING 2016 SECRET SECTION 1 WORKSHEET

CS61B Tutors

Week 1

THIS IS A SOLUTION FILE.

1 Access Control

The goal is to learn how access control works in Java, across classes, files, and packages. Use the following table as a quick reference. Remember that if an access modifier is omitted, it is **default** by default.

	Same Class	Same Package	Subclass (different package)	World
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

```
1 package com.cs61b.animal;
2
3 public class Dog {
4     private String name = "Wolf";
5     int age = 35;
6     protected double weight = 123.4;
7     public String nickname = "DumDum";
8
9     public void bark() {
10        System.out.println(name + " barks, Woof!");
11    }
12
13    void sleep() {
14        System.out.println("zzz");
15    }
16
17    protected String getName() {
18        return name;
19    }
20
21    private void changeNickname(String n) {
22        nickname = n;
23    }
24 }
```

```
1 package com.cs61b.animal;
2
3 public class Cat {
4     private Dog d = new Dog();
5
6     public void olderDog() {
7         d.age += 5;
8     }
9 }
```

```
9
10 public void heavierDog() {
11     d.weight += 10;
12 }
13
14 public void changeDogNickname(String newNickname) {
15     d.changeNickname(newNickname);
16 }
17
18 }
```

```
1 package com.cs61b.robot;
2
3 import com.cs61b.animal.Dog;
4
5 public class RobotDog extends Dog {
6
7     public void sleepMode() {
8         sleep();
9     }
10
11     public void sayName1() {
12         System.out.println("My name is " + getName());
13     }
14
15     public void sayName2() {
16         System.out.println("My name is " + name);
17     }
18
19     public Cat getCat() {
20         return new RobotCat();
21     }
22 }
```

```
1 package com.cs61b.robot;
2
3 import com.cs61b.animal.Cat;
4
5 class RobotCat extends Cat {
6     public void meow() {
7         System.out.println("MEOW!");
8     }
9 }
```

```
1 package com.cs61b;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Dog dog = new Dog();
7         Cat cat = new Cat();
8         RobotDog robotDog = new RobotDog();
9         RobotCat robotCat = new RobotCat();
10        Cat catFromDog = robotDog.getCat();
11        catFromDog.meow();
12
13        dog.name = "Fluffy";
14        dog.age = 125;
```

```

15     dog.weight = 0.23;
16     dog.nickname = "Hamburger";
17
18     dog.bark();
19     dog.sleep();
20     System.out.println(dog.getName());
21     dog.changeNickname("Burrito");
22
23     cat.olderDog();
24     cat.heavierDog();
25     cat.changeDogNickname("Hamstring");
26
27     robotDog.sleepMode();
28     robotDog.sayName1();
29     robotDog.sayName2();
30 }
31
32 }

```

Your Job: Cross out all lines in above code (Dog, Cat, RobotDog, RobotCat, and Main) that are ILLEGAL, and be able to give brief justifications why.

In Dog.java, all lines are valid.

In Cat.java:

Line 15 is invalid. Since Dog's `changeNickname` method is private, and Cat is not allowed to access it.

In RobotDog.java:

Line 8 is invalid. Dog's `sleep` method is default, so RobotDog, which is in a different package, cannot access the method.

Line 16 is invalid. Dog's name field is private, so RobotDog cannot access it.

In RobotCat.java, all lines are valid.

In Main.java:

Line 9 is invalid. Since RobotCat.java is in a different package and RobotCat is not a public class, Main cannot directly reference RobotCat.

Line 11 is invalid. While `catFromDog`'s dynamic type is RobotCat, its static type is Cat. The compiler determines that Cat does not have the method `meow`.

Line 13-15 are invalid. They are all private, default, and protected fields of Dog, which Main is not allowed to access.

Line 19-21 are invalid. They are all private, default, and protected methods of Dog, which Main cannot access.

2 Iterator

```

1 public interface Iterator<T> {
2     boolean hasNext();
3     T next();
4 }
5
6 public interface Iterable<T> {
7     Iterator<T> iterator();
8 }
9
10 public class Dog {
11     private String name = ...;
12     public void bark() {
13         System.out.print("Woof " + name + "!");
14     }
15 }

```

Question 1. Implement the following method, `barkCombinations`, which prints ALL bark combinations of dogs from two dog sets in form of “Woof DOG1! Woof DOG2!”. The dogs in `dogs1` should bark first. For 3 dogs in `dogs1` and 5 dogs in `dogs2`, there must be 15 bark combinations printed. (Hint: `Set<T>` implements `Iterable<T>`)

```
1 public static void barkCombinations(Set<Dog> dogs1, Set<Dog> dogs2) {
2     // Your code here.
3     for (Dog d1 : dogs1) {
4         for (Dog d2 : dogs2) {
5             d1.bark();
6             System.out.print(" ");
7             d2.bark();
8             System.out.println();
9         }
10    }
11
12
13
14
15 }
```

Question 2. Implement the following method `printAllTwice` which prints each element returned by given `String` iterator TWICE. To see how it should work, refer to the main method posted below.

```
1 public static void printAll(Iterator<String> iterator) {
2     // Your code here.
3     while (iterator.hasNext()) {
4         String s = iterator.next();
5         System.out.println(s);
6         System.out.println(s);
7     }
8
9
10
11
12
13 }
14
15
16 public static void main(String[] args) {
17     List<String> dogs = new ArrayList<>();
18     dogs.add("Helium");
19     dogs.add("Silver");
20     dogs.add("Neumann");
21     printAll(dogs.iterator());
22 }
```

main should print:

```
Helium
Helium
Silver
Silver
Neumann
Neumann
```

3 Exceptions

```
1 public class IntList {
2     private int head;
3     private IntList tail;
4
5     /* Returns the index of an element in the list */
6     public int getIndex(int item){
7         int index = 0;
8         IntList temp = this;
9         while(temp.head != item){
10            temp = temp.tail;
11            index++;
12        }
13        return index;
14    }
15 }
```

Question 1. What happens when you call `getIndex(int item)` on an element that is not in the list? `NullPointerException` is thrown with `temp.head`.

Question 2. Write `getIndexThrowException`, which attempts to get the index of an item, but throws an `IllegalArgumentException` with a useful message if no such item exists in the list. Do *NOT* use `if` statements, `while` loops, `for` loops, or recursion. (Hint: you can use `get(int item)`)

Question 3. Write `getIndexDefaultNegative`, which attempts to get the index of an item, but returns `-1` if no such item exists in the list. Again, do not use `if` statements, `while` loops, `for` loops, or recursion.

```
1 public class IntList {
2     private int head;
3     private IntList tail;
4
5     /* Returns the index of an element in the list */
6     public int getIndex(int item){
7         int index = 0;
8         IntList temp = this;
9         while(temp.head != item){
10            temp = temp.tail;
11            index++;
12        }
13        return index;
14    }
15
16    public int getIndexThrowException(int item) {
17        // Your code here.
18        try {
19            return getIndex(item);
20        } catch(NullPointerException ex) {
21            throw new IllegalArgumentException("No such element!");
22        }
23
24
25
26
27
28
29    }
30
31    public int getIndexDefaultNegative(int item) {
32        // Your code here.
33        try {
34            return getIndex(item);
35        } catch(NullPointerException ex) {
36            return -1;
37        }
38
39
40
41
42
43
44
45
46
47    }
48 }
```

4 More Exceptions

```
1 public class FactorialException extends RuntimeException {
2     ...
3 }
4
5 public class WrongNumberException extends Exception {
6     ...
7 }
```

Question 1. Is `FactorialException` a checked exception? What about `WrongNumberException`? Why? Why not? `FactorialException` is NOT a checked exception since it is a subclass of `RuntimeException`. However, `WrongNumberException` is a checked exception since it is a subclass of `Exception` and not `RuntimeException`.

```
1 public static int factorial(int n) throws WrongNumberException {
2     if (n < 0) {
3         throw new WrongNumberException();
4     }
5     if (n <= 1) {
6         return 1;
7     } else {
8         return factorial(n - 1);
9     }
10 }
11
12 public static int weirdFactorial(int n) {
13     if (n == 50) {
14         throw new FactorialException();
15     }
16     if (n <= 1) {
17         return 1;
18     } else {
19         return weirdFactorial(n - 1);
20     }
21 }
```

Question 2. Is `factorial`'s definition legal? Is there something to be changed? If there is any, fix it! It is NOT legal since it throws `WrongNumberException`, which is a checked exception, but does not declare it in the method signature.

Question 3. Repeat Question 2 for `weirdFactorial`. Even though it throws `FactorialException`, the code is legal since it is an unchecked exception.

Question 4. Referring to the definition on the previous page for `factorial` and `weirdFactorial`, answer what is printed for each blocks of code below. If the program terminates with an exception, specify which exception occurred. If illegal (doesn't compile), explain why. For numeric results, feel free to just put down "number" instead of performing tedious calculations by hand.

```
1 // (a)
2 int answer = factorial(50);
3 System.out.println(answer);
4 // ANSWER: ILLEGAL. Should wrap in try/catch block
5
6 // (b)
7 try {
8     int answer = factorial(50);
9     System.out.println(answer);
10 } catch (WrongNumberException ex) {
11     System.out.println("Wrong Number!");
12 }
13 // ANSWER: Number
14
15
16 // (c)
17 int answer = weirdFactorial(30);
18 System.out.println(answer);
19 // ANSWER: Number
20
21
22 // (d)
23 int answer = weirdFactorial(60);
24 System.out.println(answer);
25 // ANSWER: FactorialException
26
27
28 // (e)
29 try {
30     int answer = weirdFactorial(50);
31     System.out.println(answer);
32 } catch (FactorialException ex) {
33     System.out.println("Factorial Exception!");
34 } catch (Exception ex) {
35     System.out.println("Some Exception!");
36 }
37 // ANSWER: prints Factorial Exception!
38
39
40 // (f)
41 try {
42     int answer = weirdFactorial(50);
43     System.out.println(answer);
44 } catch (Exception ex) {
45     System.out.println("Some Exception!");
46 } catch (FactorialException ex) {
47     System.out.println("Factorial Exception!");
48 }
49 // ANSWER: prints Some Exception!
```