CSE 122 Final Exam **KEY**
Winter 2024

Name of Student: ____

Section (e.g., AA):_________________ Student Number: _____________

The exam is divided into six questions with the following points:

#	Problem Area
1	Conceptual
2	Code Tracing
3	Debugging
4	Collections Programming
5	Objects Programming
6	Stacks/Queues Programming

Do not begin work on this exam until instructed to do so. Any student who starts early or who continues to work after time is called will receive U's on some problems as a penalty.

You are allowed one page of a reference sheet, front and back, as notes during the exam. Space is provided for your answers. There is also a reference sheet at the end that you should use. You are not allowed to access any other papers during the exam. You are NOT to use any electronic devices while taking the test, including calculators. Anyone caught using an electronic device will receive U's on some problems as a penalty.

The exam is not, in general, graded on code quality and you do not need to include comments. For the stack/queue and collections questions, however, you are expected to use generics properly and to declare variables using interfaces when possible. You may only use the methods on the cheat sheet for the data structures listed. For objects programming, you should declare all fields to be private. Problems may specify more specific requirements. You are not allowed to use programming constructs we haven't discussed in class such as break, continue, or returns from a void method on this exam.

Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.

You are allowed to ask for scratch paper to use as additional space when writing answers, but you must indicate on the original page for the problem that part of the solution is on scratch paper. Failure to do so may result in your work on scratch paper not being graded.

If you finish the exam early, please hand your exam to the instructor and exit quietly through the front door. During the last 5 minutes of the exam, please stay in your seats to avoid disrupting others during the end of the exam.

Each problem is graded on an E/S/N scale. In general, to earn an E on a problem your solution must work without error and meet all the problem requirements. To earn an S, there is allowance for minor errors in the solution, but the problem requirements must still be met to earn an S. Unless specified by the problem, we do not grade on code quality.

Initial here to indicate you have read and agreed to these rules: _________________

1. **Conceptual:** Each of these parts should be considered independent of the others

Part A: Consider the following code snippet:

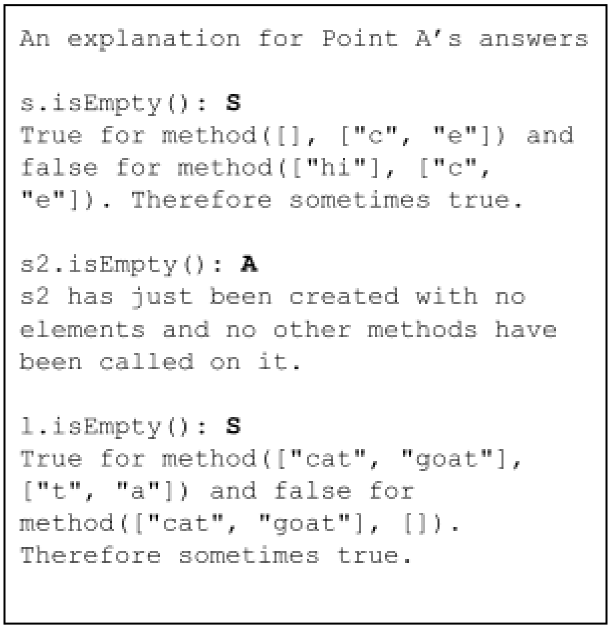
```
public static void removeNegatives(List<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        if (list.get(i) < 0) {
            list.remove(i);
        }
    }
}
```

The following code snippet was written to remove all negative values from the given list, but there is a bug. Select **all** lists that would show the bug if passed to removeNegatives.

- ☐ [1, 2, 3, 4, 5]
- ☑ **[-3, -3, -3]**
- ☐ [3, -8, 9, 9, -10,]
- ☐ []
- ☐ [-1, 2, -3, 4]
- ☑ **[-10, -5, 0, 5]**

Part B: Consider the following method. For each of the following commented Checkpoints, fill in the table for which conditions are always true (under any circumstance), only sometimes true, or never true at each comment. You can abbreviate **A**=always, **S**=sometimes and **N**=never.

```
public static Set<String> method(Set<String> s, List<String> l) {
    Set<String> s2 = new TreeSet<>();
    // Checkpoint A
    if (s.isEmpty()) {
        throw new IllegalArgumentException();
    }
    Iterator<String> it = s.iterator();
    // Checkpoint B
    while (it.hasNext()) {
        String str = it.next();
        boolean b = true;
        for (String tmp : l) {
            // Checkpoint C
            if (!str.contains(tmp)) {
                b = false;
            }
        }
        // Checkpoint D
        if (!b) {
            it.remove();
            s2.add(str);
        }
    }
    // Checkpoint E
    return s2;
}
```



An explanation for Point A's answers

s.isEmpty(): **S**
True for method([], ["c", "e"]) and false for method(["hi"], ["c", "e"]). Therefore sometimes true.

s2.isEmpty(): **A**
s2 has just been created with no elements and no other methods have been called on it.

l.isEmpty(): **S**
True for method(["cat", "goat"], ["t", "a"]) and false for method(["cat", "goat"], []). Therefore sometimes true.

	Checkpoint A	Checkpoint B	Checkpoint C	Checkpoint D	Checkpoint E
s.isEmpty()	**S**	**N**	**N**	**N**	**S**
s2.isEmpty()	**A**	**A**	**S**	**S**	**S**
l.isEmpty()	**S**	**S**	**N**	**S**	**S**

Part C: (Select one option) Consider the following method. Which of the following options is the best "plain-English" explanation of what the code is doing?

```
public static void method(File f) throws new FileNotFoundException {
    Scanner input = new Scanner(f);
    while (input.hasNextLine()) {
        Scanner scan = new Scanner(input.nextLine());
        while (scan.hasNext()) {
            double d = 0.0;
            if (scan.hasNextDouble()) {
                d += scan.nextDouble();
            } else {
                String s = scan.next();
            }
        }
        System.out.println(d);
    }
}
```

✔ **For each line in the given file, prints the sum of all numbers, ignoring non-numeric input.**

○ Prints the sum of all numbers in the file.

○ Creates a Scanner over the File parameter to read one line at a time, and then uses another Scanner to read through each line and add up all tokens.

○ For each line in the given file, prints the sum of all numbers up until the first non-numeric token.

○ For each line in the given file, prints all of the tokens concatenated together.

2. **Code Tracing:** Consider the method below.

```
public static void mystery(TreeMap<Character, TreeSet<String>> m) {
    TreeSet<String> s = new TreeSet<>();
    Iterator<Character> itr = m.keySet().iterator();
    while (!itr.hasNext()) {
        TreeSet<String> s2 = m.get(itr.next());
        if (s2.size() % 2 == 0) {
            s2 = s;
        } else {
            itr.remove();
            s = s2;
        }
    }
}
```

For each list below, indicate what it would hold after a call to mystery where it was passed as a parameter.

Before mystery call	**After mystery call**
{'a'={"a"}, 'b'={"b"}}	**{}**
{'c'={"c", “cc”}, 'd'={"d", "dd"}}	**{'c'={"c", "cc"}, 'd'={"d", "dd"}}**
{'e'={"e"}, 'f'={"f", "ff"}}	**{'f'={"f", "ff"}}**
{'g'={"p"}, 'h'={"m"}, 'i'={"e", "o"}, 'j'={"r", "w"}}	**{'i'={"e", "o"}, 'j'={"r", "w"}}**

3. **Debugging:** Consider the following buggy implementation of the **HostStaff** class.

The intended functionality of the HostStaff class is to represent the task of managing a waitlist and seating tables that host staff generally handles at a restaurant. It has the following methods defined:

- A **constructor** that takes no parameters and sets up the necessary state.
- **getWaitTime()** which returns the current wait time, which is equal to 10 minutes for each party on the waitlist.
- **getWaitList()** which returns the current wait list.
- **addName(String partyName, int partySize)** which adds a party to the waitlist under partyName and with partySize diners to the waitlist. If partySize is less than 1, throws an IllegalArgumentException.
- **seatParty(int tableCapacity)** which finds the first party on the waitlist who will fit at the table (so their party size is no larger than tableCapacity), removes them from the waitlist, and returns true. If no party on the waitlist can fit at the table, returns false.

Consider the following code, using the HostStaff class:

```
HostStaff hs = new HostStaff();
hs.addName("Mia", 7);
hs.addName("Ken", 10);
hs.addName("Ailsa", 4);
hs.addName("Elizabeth", 3);

System.out.println("wait list: " + hs.getWaitList());
System.out.println("wait time: " + hs.getWaitTime());
System.out.println("table for 2? " + hs.seatParty(2));
System.out.println("wait list: " + hs.getWaitList());
System.out.println("table for 6? " + hs.seatParty(6));
System.out.println("wait list: " + hs.getWaitList());
System.out.println("wait time: " + hs.getWaitTime());
```

Working with a ***correct*** implementation of the class, the statements above should produce the following output:

```
wait list: [Mia, Ken, Ailsa, Elizabeth]
wait time: 40
table for 2? false
wait list: [Mia, Ken, Ailsa, Elizabeth]
table for 6? true
wait list: [Mia, Ken, Elizabeth]
wait time: 30
```

However, **our buggy implementation produces the following output**:

```
wait list: [Mia, Ken, Ailsa, Elizabeth]
wait time: 40
table for 2? false
wait list: [Mia, Ken, Ailsa, Elizabeth]
table for 6? true
wait list: [Elizabeth, Mia, Ken]
wait time: 30
```

Problem continued on the next page…

Your task: Annotate (write on) the code below to indicate how you would fix the bug. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. However, the fix should not require a lot of work.

- You must *correctly identify* the bug for an S grade.
- You must *correctly identify and correctly fix* the bug for an E grade.

```
import java.util.*;

public class HostStaff {
    private Map<String, Integer> partyInfo;
    private Queue<String> waitList;
    private int currentWait;

    public HostStaff() {
        partyInfo = new HashMap<>();
        waitList = new LinkedList<>();
        currentWait = 0;
    }

    public int getWaitTime() {
        return currentWait;
    }

    public void addName(String partyName, int partySize) {
        if (partySize <= 0) {
            throw new IllegalArgumentException();
        }
        waitList.add(partyName);
        partyInfo.put(partyName, partySize);
        currentWait += 10;     }

    public Queue<String> getWaitList() {
        return waitList;
    }

    public boolean seatParty(int tableCapacity) {
        boolean foundParty = false;
        int currentWaitListSize = waitList.size();
        for (int i = 0; i < currentWaitListSize; i++) {
            String nextName = waitList.remove();
            if (partyInfo.get(nextName) <= tableCapacity && !foundParty) {
                currentWait -= 10;
                foundParty = true;
                ~~return foundParty;~~
            } else {
                waitList.add(nextName);
            }
        }
        return foundParty;
    }
}
```

4. **Collections Programming**: Write a method called aveRestaurants that takes a map indicating how each person rates various restaurants on the ave and a target rating and returns a map indicating all the foods each person has rated with at least the target rating.

The input map will have keys that are people's names (strings) and values which are maps with keys that are restaurants (strings) and values which are numbers in the range of 0.0 to 5.0 for the rating that person has given that restaurant. An example would be if we had a variable called ratings that stored the following map in the format just described:

```
    {"Colton"={"kong tofu"=3.5, "little thai"=4.9, "jack in the box"=0.0},
     "Arkita"={"chilis"=2.1, "pho shizzle"=3.3},
     "Ayush"={"sushi burrito"=3.4, "aladdin's"=3.9},
     "Claire"={}}
```

In this example, we see that Colton has rated little thai and kong tofu with a 4.9 and 3.5 respectively, and jack in the box as a 0.0, while Arkita has rated chilis as a 2.1 and pho shizzle as a 3.3.

The aveRestaurants method you are writing should take a ratings map described above and a target rating and should return a map indicating all the restaurants each person has rated with at least the target rating. The map you are to return should use the people's names as keys and the set of all the restaurants that person rated with at least the target value as values.

For example, suppose the following call is made:

```
    aveRestaurants(ratings, 3.5);
```

Given this call, the following map would be returned:

```
    {"Arkita"=[],
     "Ayush"=["aladdin's"],
     "Claire"=[],
     "Colton"=["kong tofu", "little thai"]}
```

Notice that the value for the key "Colton" is the set ["kong tofu", "little thai"] because he rated only those foods with at least a rating of 3.5. The value for the keys "Arkita" and "Claire" is [] because they rated no restaurants with a rating of at least 3.5. Note that foods rated with a 3.5 should be included (see Colton).

The map you return should have keys sorted alphabetically and the foods in the values should appear in alphabetical order as well.

Your method should not modify the provided map. You may assume that the map and none of its contents are null.

Write your solution on the next page

```
public Map<String, Set<String>> aveRestaurants(Map<String, Map<String, Double>> ratings,
                                                     double target) {
    Map<String, Set<String>> result = new TreeMap<>();
    for (String person : ratings.keySet()) {
        result.put(person, new TreeSet<>());

        Map<String, Double> ratingsFor = ratings.get(person);
        for (String restaurant : ratingsFor.keySet()) {
            if (ratingsFor.get(restaurant) >= target) {
                result.get(person).add(restaurant);
            }
        }
    }
    return result;
}
```

5. **Objects Programming:** Consider the following Garden interface. For this problem, write a class called VegetableGarden that implements the Garden interface and includes the required methods.

The VegetableGarden class should have a constructor that takes an int that represents the number of spots for plants. For example, new VegetableGarden(20) would represent a vegetable garden that has space for 20 plants. The constructor should throw an IllegalArgumentException if spots is negative, and otherwise should set up the necessary state.

```
// Represents a Garden in which users can plant items
public interface Garden {
    // Returns the maximum number of spots present within this garden.
    public int getMaxSpots();

    // Returns the total number of spots occupied with plants within this garden.
    public int getOccupiedSpots();

    // Plants 'num' instances of 'name' within the garden where 'temp' represents the
    // given plant's ideal temperature in degrees Fahrenheit. Should throw an
    // IllegalArgumentException if there aren't enough spots available within this garden
    // or if num is not positive. You may assume the temp for plants with the same name
    // will be equal.
    public void plant(String name, int num, double temp);

    // Removes a single instance of a plant named 'name' within this garden. Should throw
    // an IllegalArgumentException if no plants with the given 'name' exist.
    public void pick(String name);

    // Given a specific day's 'temp', returns a Set of all plants that require shade. A
    // plant needs shade if the current temp is greater than its ideal temperature + 10
    // degrees Fahrenheit. The returned Set should have all elements in sorted order.
    public Set<String> needsShade(double temp);

    // Copies all plants within this garden to the provided 'other' instance. Should throw
    // an IllegalArgumentException if there aren't enough spots present. This garden
    // should remain unmodified.
    public void copyInto(Garden other);
}
```

For example, if the following lines are executed:

```
        Garden gard = new VegetableGarden(5);       Garden gard2 = new VegetableGarden(10);
        gard.plant("Carrot", 1, 60.5);              gard2.plant("Broccoli", 2, 55.2);
        gard.plant("Lettuce", 2, 65.0);
        gard.plant("Celery", 1, 54.8);
```

Then the following calls to gard would return:

```
        gard.getMaxSpots();             5
        gard2.getMaxSpots();            10

        gard.getOccupiedSpots();        4
        gard.needsShade(71.0);          {"Carrot", "Celery"}

        gard.pick("Carrot");
        gard.getOccupiedSpots();        3
        gard.needsShade(71.0);          {"Celery"}

        gard2.copyInto(gard);
        gard.getOccupiedSpots();        5

        gard.plant("Okra", 1, 82.5);    IllegalArgumentException
```

Your fields should be properly encapsulated.

Write your solution on the next page.

```
public class VegetableGarden implements Garden {
    private int maxSpots;
    private Map<String, Integer> plantsToNum;
    private Map<String, Double> plantsToTemp;

    public VegetableGarden(int numSpots) {
        if (numSpots < 0) {
            throw new IllegalArgumentException();
        }
        maxSpots = numSpots;
        plantsToNum = new HashMap<>();
        plantsToTemp = new HashMap<>();
    }

    public void plant(String name, int num, double temp) {
        if (getTotalPlanted() + num > maxSpots) {
            throw new IllegalArgumentException();
        }
        if (!plantsToNum.containsKey(name)) {
            plantsToNum.put(name, 0)
        }
        plantsToNum.put(name, plantsToNum.get(name) + num);
        plantsToTemp.put(name, temp);
    }

    public int getOccupiedSpots() {
        int total = 0;
        for (int n : plantsToNum.values()) {
            total += n;
        }
        return total;
    }

    public void pick(String name) {
        if (!plantsToNum.containsKey(name)) {
            throw new IllegalArgumentException();
        }
        int newCount = plantsToNum.get(name) - 1;
        if (newCount > 0) {
            plantsToNum.put(name, newCount);
        } else {
            plantsToNum.remove(name);    // not strictly necessary - could check for 0 in other methods
            plantsToTemp.remove(name);  // not strictly necessary - could check for 0 in other methods
        }
    }

    public int getMaxSpots() {
        return this.maxSpots;
    }

    public Set<String> needsShade(double temp) {
        Set<String> result = new TreeSet<>();
        for (String plant : plantsToTemp.keySet()) {
            double plantTemp = plantsToTemp.get(plant);
            if (temp > plantTemp + 10) {
                result.add(plant);
            }
        }
        return result;
    }

    public void copyInto(Garden other) {
        if (other.getMaxSpots() < other.getTotalPlanted() + this.getTotalPlanted()) {
            throw new IllegalArgumentException(); // not strictly necessary - could do through plant call
        }
        for (String plant : plantsToNum.keySet()) {
            int num = plantsToNum.get(plant);
            int temp = plantsToTemp.get(plant);
            other.plant(plant, num, temp);
        }
    }
}
```

6. **Stacks/Queues Programming:** Write a method called **removeMin** that takes a stack of integers as a parameter and that removes and returns the smallest value from the stack. For example, if a variable called s stores the following sequence of values:

```
    bottom [2, 8, 3, 19, 7, 3, 2, 42, 9, 3, 2, 7, 12, -8, 4] top
```

and you make the following call:

```
    int n = removeMin(s);
```

the method removes and returns the value -8 from the stack, so that the variable n will be -8 after the call and s will store the following values:

```
    bottom [2, 8, 3, 19, 7, 3, 2, 42, 9, 3, 2, 7, 12, 4] top
```

If the minimum value appears more than once, all occurrences of the minimum should be removed from the stack. For example, given the ending value of the stack above, if we again call removeMin(s), the method would return 2 and would leave the stack in the following state:

```
    bottom [8, 3, 19, 7, 3, 42, 9, 3, 7, 12, 4] top
```

You are to use **one queue as auxiliary storage to solve this problem**. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. Use the Stack and Queue structures described in the cheat sheet and obey the restrictions described there. You may assume that the stack is not empty.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
    public void s2q(Stack<Integer> s, Queue<Integer> q) {
        while (!s.isEmpty()) {
            q.add(s.pop());
        }
    }

    public void q2s(Queue<Integer> q, Stack<Integer> s) {
        while (!q.isEmpty()) {
            s.push(q.remove());
        }
    }
```

Write your solution in the box on the next page. If you need additional space, please indicate that your solution is continued on scratch paper.

```
public static int removeMin(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<>();
    int min = s.pop();              // could set min to Integer.MAX_VALUE
    q.add(min);
    while (!s.isEmpty()) {
        int next = s.pop();
        if (next < min) {
            min = next;
        }
        q.add(next);
    }
    while (!q.isEmpty()) {
        int next = q.remove();
        if (next != min) {
            s.push(next);
        }
    }
    s2q(s, q);                      // not strictly necessary - order didn't matter
    q2s(q, s);
    return min;
}
```

(You may use the rest of this page as scratch paper if necessary)

^_^ CSE 122 Final Exam Reference Sheet ^_^

(DO NOT WRITE ANY WORK YOU WANTED GRADED ON THIS REFERENCE SHEET. IT WILL NOT BE GRADED)

Examples of Constructing Various Collections

```
List<Integer> list = new ArrayList<Integer>();
Queue<Double> queue = new LinkedList<Double>();
Stack<String> stack = new Stack<>();  // Diamond operator also permitted
Set<String> words = new HashSet<>();
Map<String, Integer> counts = new TreeMap<String, Integer>();
```

Methods Found in ALL collections (Lists, Stacks, Queues, Sets, Maps)

`equals(`**collection**`)`	Returns `true` if the given other collection contains the same elements
`isEmpty()`	Returns `true` if the collection has no elements
`size()`	Returns the number of elements in a collection
`toString()`	Returns a string representation such as `"[10, -2, 43]"`

Methods Found in both Lists and Sets (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**value**`)`	Adds value to collection (appends at end of list)
`addAll(`**collection**`)`	Adds all the values in the given collection to this one
`contains(`**value**`)`	Returns `true` if the given value is found somewhere in this collection
`iterator()`	Returns an Iterator object to traverse the collection's elements
`clear()`	Removes all elements of the collection
`remove(`**value**`)`	Finds and removes the given value from this collection
`removeAll(`**collection**`)`	Removes any elements found in the given collection from this one
`retainAll(`**collection**`)`	Removes any elements *not* found in the given collection from this one

`List<Type>` Methods

`add(`**index, value**`)`	Inserts given value at given index, shifting subsequent values right
`indexOf(`**value**`)`	Returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	Returns the value at given index
`lastIndexOf(`**value**`)`	Returns last index where given value is found in list (-1 if not found)
`remove(`**index**`)`	Removes/returns value at given index, shifting subsequent values left
`set(`**index, value**`)`	Replaces value at given index with given value

`Stack<Type>` Methods (only allowed methods plus `size` and `isEmpty`)

`pop()`	Removes the top value from the stack and returns it; `pop` throw an `EmptyStackException` if the stack is empty
`push(`**value**`)`	Places the given value on top of the stack
`peek()`	Returns the value at the top from the stack without removing it; throws a `EmptyStackException` if the stack is empty

`Queue<Type>` Methods (only allowed methods plus `size` and `isEmpty`)

`add(`**value**`)`	Places the given value at the back of the queue
`remove()`	Removes the value from the front of the queue and returns it; throws a `NoSuchElementException` if the queue is empty
`peek()`	Returns the value at the front of the queue without removing it; throws a `NoSuchElementException` if the queue is empty

Map<KeyType, ValueType> Methods

containsKey(**key**)	`true` if the map contains a mapping for the given key
get(**key**)	The value mapped to the given key (`null` if none)
keySet()	Returns a `Set` of all keys in the map
put(**key, value**)	Adds a mapping from the given key to the given value
putAll(**map**)	Adds all key/value pairs from the given map to this map
remove(**key**)	Removes any existing mapping for the given key
toString()	Returns a string such as `"{a=90, d=60, c=70}"`
values()	Returns a `Collection` of all values in the map

Iterator<Type> Methods

hasNext()	Returns `true` if there is another element in the iterator
next()	Returns the next value in the iterator and progresses the iterator forward one element
remove()	Removes the previous value returned by the next. Can only call once after each call to `next()`

String Methods

charAt(**i**)	The character in this String at a given index
contains(**str**)	`true` if this String contains the other's characters inside it
endsWith(**str**)	`true` if this String ends with the other's characters
equals(**str**)	`true` if this String is the same as *str*
equalsIgnoreCase(**str**)	`true` if this String is the same as *str*, ignoring capitalization
indexOf(**str**)	First index in this String where given String begins (-1 if not found)
lastIndexOf(**str**)	Last index in this String where given String begins (-1 if not found)
length()	Number of characters in this String
isEmpty()	`true` if this String is the empty string
startsWith(**str**)	`true` if this String begins with the other's characters
substring(**i, j**)	Characters in this String from index *i* (inclusive) to *j* (exclusive)
substring(**i**)	Characters in this String from index *i* (inclusive) to the end
toLowerCase(), toUpperCase()	A new String with all lowercase or uppercase letters

Math Methods

abs(**x**)	Returns the absolute value of `x`
max(**x, y**)	Returns the larger of `x` and `y`
min(**x, y**)	Returns the smaller of `x` and `y`
pow(**x, y**)	Returns the value of `x` to the `y` power
random()	Returns a random number between `0.0` and `1.0`
round(**x**)	Returns `x` rounded to the nearest integer

Object/Interface Syntax

```
public class Example implements InterfaceExample {
    private type field;
    public Example() {
       field = something;
    }
    public void method() {
       // do something
    }
}
```

```
public interface InterfaceExample {
    public void method();
}
```