

Assignment 1

Subject: Software Testing

Subject code: SCS357

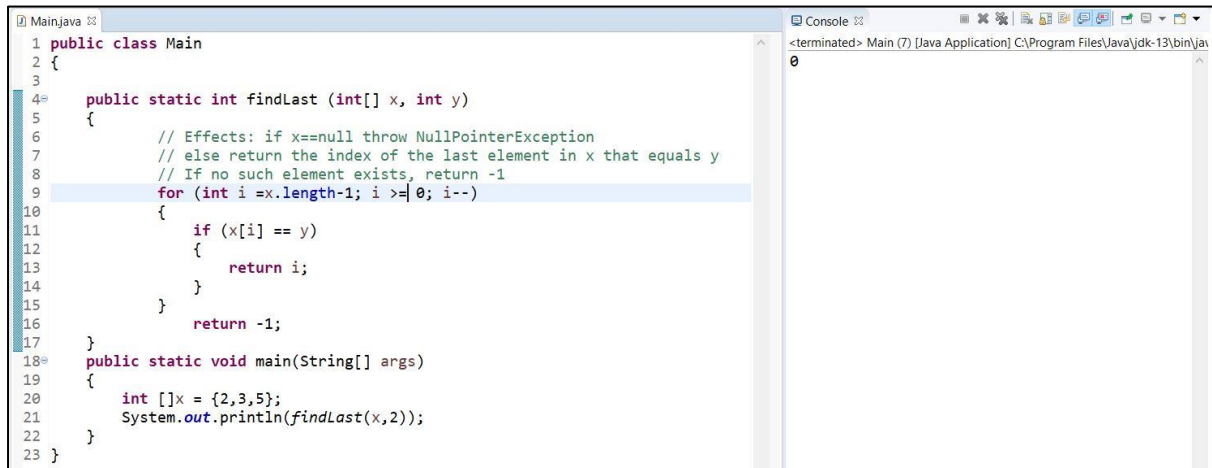
Under the supervision of: Dr. Soha Makady

TA. Hassan Mourad

ID	Name	Group
20186008	Sarah Khaled	S1
20186043	Mark Rofaeel	

Task 1:

a) The wrong in the code that in the for loop ($i > 0$) which is false, and the modification is that ($i \geq 0$) to take the first element in the array, as shown below.



The screenshot shows a Java IDE with two panes. The left pane is the code editor, showing a Java class named 'Main'. The right pane is the console, showing the output of the program.

```
1 public class Main
2 {
3
4     public static int findLast (int[] x, int y)
5     {
6         // Effects: if x==null throw NullPointerException
7         // else return the index of the last element in x that equals y
8         // If no such element exists, return -1
9         for (int i =x.length-1; i >= 0; i--)
10        {
11            if (x[i] == y)
12            {
13                return i;
14            }
15        }
16        return -1;
17    }
18    public static void main(String[] args)
19    {
20        int []x = {2,3,5};
21        System.out.println(findLast(x,2));
22    }
23 }
```

The console output shows the result of the program execution:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jdk-13\bin\jav
0
```

b) There is no test case that does not execute the fault. Every time, the program will eventually execute the fault, so there is no test case. The fault is in the for loop ($i > 0$) which is false.

c) There is no test case that executes the fault, but it does not result in an error state. Because if the fault is executed, then the error state is already made.

d) Here, the input of y was 5, and the output was 2 which is right, and it did execute the fault but there is not failure.

```

1 public class Main
2 {
3
4     public static int findLast (int[] x, int y)
5     {
6         // Effects: if x==null throw NullPointerException
7         // else return the index of the last element in x that equals y
8         // If no such element exists, return -1
9         for (int i =x.length-1; i > 0; i--)
10        {
11            if (x[i] == y)
12            {
13                return i;
14            }
15        }
16        return -1;
17    }
18    public static void main(String[] args)
19    {
20        int []x = {2,3,5};
21        System.out.println(findLast(x,5));
22    }
23 }

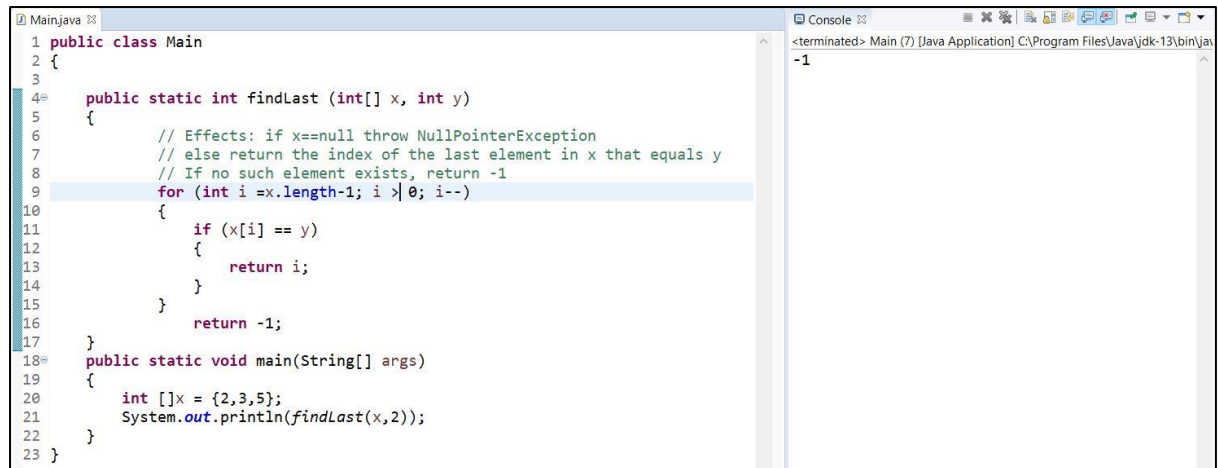
```

Console: <terminated> Main (7) [Java Application] C:\Program Files\Java\jdk-13\bin\javaw.exe (May 7, 2020) 2

Line number	i	First checked element array	array
1	2		[2,3,5]
2	2	(X[2]=5 ==2) False	[2,3,5]
1	1		[2,3,5]
2	1	(x[1]=3 ==2) False	[2,3,5]
1	0		[2,3,5]
4		Return -1	[2,3,5]

e) Here, the input of y was 2, and the output was -1 which is wrong, and this is the error state.

The state is X{2,3,5}, fault executed, error, failure.



The screenshot shows a Java IDE with two panels. The left panel is the code editor, titled 'Main.java', containing the following code:

```
1 public class Main
2 {
3
4     public static int findLast (int[] x, int y)
5     {
6         // Effects: if x==null throw NullPointerException
7         // else return the index of the last element in x that equals y
8         // If no such element exists, return -1
9         for (int i =x.length-1; i >= 0; i--)
10        {
11            if (x[i] == y)
12            {
13                return i;
14            }
15        }
16        return -1;
17    }
18    public static void main(String[] args)
19    {
20        int []x = {2,3,5};
21        System.out.println(findLast(x,2));
22    }
23 }
```

The right panel is the console window, titled 'Console', showing the output of the program:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jdk-13\bin\jav
-1
```

Task 2:

Class de.tilman_neumann.util.StringUtil:

Method	Params	Returns	Values	Exception	Ch ID	Characteristics	Covered by
Repeat	s,n	String	String		C1	Returns repeated string, else null or empty string.	
FormatLeft	s, mask	String	String		C2	Returns left-aligned string, else null or empty string.	
FormatRight	s, mask	String	String		C3	Returns right-aligned string, else null or empty string.	

ID	Characteristics	Repeat	FormatLeft	FormatRight	Partitions
C1	Returns repeated string, else null or empty string.	X			{true,false}
C2	Returns left-aligned string, else null or empty string.		X		{true,false}
C3	Returns right-aligned string, else null or empty string.			X	{true,false}

We used **all combinations** coverage criteria in the below functions because all characteristics must be used.

(We used zero, -1, and 1 as a boundary value)

Repeat function:

There are 3 blocks in s which are text, null, empty string. And there are 3 blocks in n which are negative, positive and zero.

s: {E1: text, E2: null, E3: empty string}

n: {E4: negative, E5: positive, E6: zero}

T1: s=text, n=negative number (covers E1, E4)

T2: s=text, n=positive number (covers E1, E5)

T3: s=text, n=zero (covers E1, E6)

T4: s=null, n=positive number (covers E2, E5)

T5: s= empty string, n=positive number (covers E3, E5)

T6: s=null, n=negative number (covers E2, E4)

T7: s=null, n=zero (covers E2, E6)

T8: s= empty string, n=negative number (covers E3, E4)

T9: s= empty string, n=zero (covers E3, E6)

FormatLeft function:

There are 3 blocks in s which are text, null, empty string. And there are 3 blocks in mask which are text, null, and empty string.

s: {E1: text, E2: null, E3: empty string}

mask: {E4: text, E5: null, E6: empty string}

T1: s=text, n=text (covers E1, E4)

T2: s= empty string, n=text (covers E3, E4)

T3: s= text, n= empty string (covers E1, E6)

T4: s= text, n= null (covers E1, E5)

T5: s= null, n= text (covers E2, E4)

T6: s= null, n= null (covers E2, E5)

T7: s= null, n= empty string (covers E2, E6)

T8: s= empty string, n= null (covers E3, E5)

T9: s= empty string, n= empty string (covers E3, E6)

FormatRight function:

There are 3 blocks in s which are text, null, empty string. And there are 3 blocks in mask which are text, null, and empty string.

s: {E1: text, E2: null, E3: empty string}

mask: {E4: text, E5: null, E6: empty string}

T1: s=text, n=text (covers E1, E4)

T2: s= empty string, n=text (covers E3, E4)

T3: s= text, n= empty string (covers E1, E6)

T4: s= text, n= null (covers E1, E5)

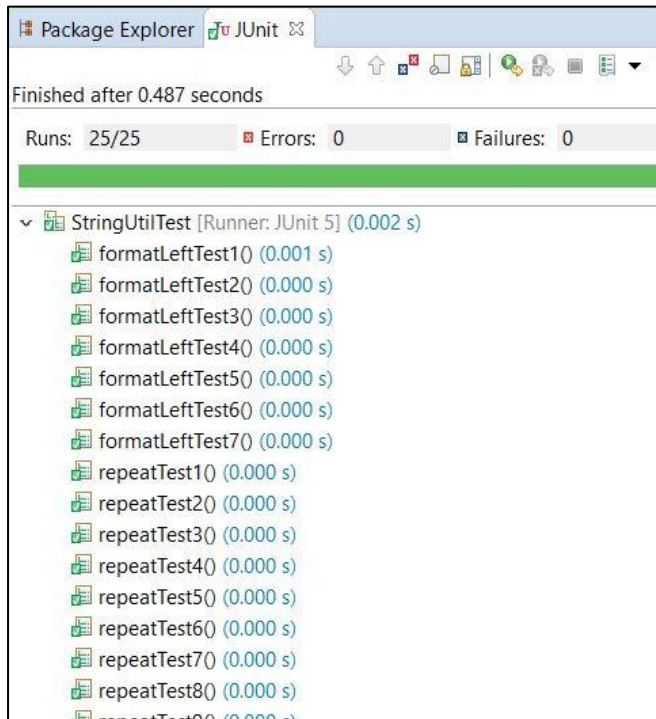
T5: s= null, n= text (covers E2, E4)

T6: s= null, n= null (covers E2, E5)

T7: s= null, n= empty string (covers E2, E6)

T8: s= empty string, n= null (covers E3, E5)

T9: s= empty string, n= empty string (covers E3, E6)



Class de.tilman_neumann.util.Multiset HashMapImpl:

Method	Params	Returns	Values	Exception	Ch ID	Characteristics	Covered by
int add(T entry)	entry	Integer	Integer		C1	Adds a single occurrence of the specified element.	C3,C4,C5
int add(T entry, int mult)	entry, mult	Integer	Integer	IllegalArgumentExc eption	C2	Adds several occurrences of an element to this multiset.	C1 (if mult ==1)
void addAll(Multiset< T> other)	other	void			C3	Adds all the elements in the specified multiset to this multiset.	
void addAll(Collection<T> values)	values	void		IllegalStat eExceptio n	C4	Adds all the elements in the specified collection to this collection.	
void addAll(T[] values)	values	void			C5	Adds all the elements in the specified array to this array.	
Integer remove(Object key)	Key	Integer	Integer		C6	Removes a single occurrence of the specified element.	C7, C8
int remove(T key, int mult)	Key, mult	int	Integer		C7	Removes several occurrences of the specified element.	C6 (if mult ==1)
int removeAll (T key)	key	In t	Integer		C8	Removes all the elements with the number of its occurrences.	

ID	Characteristics	<code>int add(T entry)</code>	<code>int add(T entry, int mult)</code>	<code>void addAll(Multiset<T> other)</code>	<code>void addAll(Collection<T> values)</code>	<code>void addAll(T[] values)</code>	Partitions
C1	Adds a single occurrence of the specified element to this multiset.	X		X	X	X	{true,false}
C2	Adds several occurrences of an element to this multiset.	X	X				{true,false}
C3	Adds all the elements in the specified multiset to this multiset.			X			{true,false}
C4	Adds all the elements in the specified collection to this collection.				X		{true,false}
C5	Adds all the elements in the specified array to this array.					X	{true,false}

ID	Characteristics	Integer remove(Object key)	int remove(T key, int mult)	int removeAll(T key)	Partitions
C6	Removes a single occurrence of the specified element.	X	X	X	{true,false}
C7	Removes several occurrences of the specified element	X	X		{true,false}
C8	Removes all the elements with the number of its occurrences.			X	{true,false}

We used **base choice** coverage criteria in the below function, as we hold all but one base choice constant and using each non base choice in each other characteristic.

```
int add(T entry, int mult)
```

The base choice test is [string, positive], and the following additional tests would be needed:

(We used zero, -1, and 1 as a boundary value)

There are 6 blocks in entry which are double, string, negative, null, and empty string. There are 3 blocks in multi which are negative number, positive number and zero.

entry: {E1: double, E2: string, E3: negative, E5: null, E6: empty string}

mult: {E7: negative, E8: positive, E9: zero}

T1: entry=**string**, mult=negative (covers E3, E7)

T2: entry=**string**, mult=zero (covers E3, E9)

T3: entry=double, mult=**positive** (covers E1, E8)

T4: entry=negative, mult=**positive** (covers E3, E8)

T5: entry=null, mult=**positive** (covers E5, E8)

T6: entry=empty string, mult= **positive** (covers E6, E8)

We used **all combinations** coverage criteria in the below functions because all characteristics must be used.

```
int add(T entry)
```

There are 5 blocks in entry which are double, string, negative, null, and empty string.

entry: {E1: double, E2: string, E3: negative, E4: null, E5: empty string}

T1: entry=double (covers E1)

T2: entry= string (covers E2)

T3: entry= negative (covers E3)

T4: entry=null (covers E4)

T5: entry= empty string (covers E5)

```
void addAll(Multiset<T> other)
```

There are 3 blocks in other which are doubles, strings and null.

other: {E1: Doubles, E2: Strings, E3: null}

T1: other = doubles (covers E1)

T2: other = strings (covers E2)

T3: other = null (covers E3)

```
void addAll(Collection<T> values)
```

There are 3 blocks in values which are doubles, strings and null.

values: {E1: Doubles, E2: Strings, E3: null}

T1: values = doubles (covers E1)

T2: values = strings (covers E2)

T3: values = null (covers E3)

```
void addAll(T[] values)
```

There are 5 blocks in values which are mix of doubles and strings, strings, doubles, null and empty strings.

values: {E1: mix of doubles and strings, E2: strings, E3: doubles, E4: null, E5: empty strings}

T1: values = mix of doubles and strings (covers E1)

T2: values = strings (covers E2)

T3: values = doubles (covers E3)

T4: values = null (covers E4)

T5: values = empty strings (covers E5)

We used **base choice** coverage criteria in the below function, as we hold all but one base choice constant and using each non base choice in each other characteristic.

```
int remove(T key, int mult)
```

The base choice test is [valid, positive], and the following additional tests would be needed:

There are 4 blocks in key which are valid element, invalid element, null, and empty string. There are 3 blocks in multi which are positive number, negative number, and zero.

(We used zero, -1, and 1 as a boundary value)

When we used any negative number to remove T key(valid), It did not remove the key but increased the number of it **(It is written in the documentation that it will throw an exception, but that did not happen).**

Key: {E1: valid, E2: invalid, E3: null, E4: empty string}

Multi: {E5: positive, E6: negative, E7: zero}

T1: entry=valid, mult=positive (covers E1, E5)

T2: entry=valid, mult=negative (covers E1, E6)

T3: entry=valid, mult=zero (covers E1, E7)

T4: entry=invalid, mult=positive (covers E2, E5)

T5: entry=null, mult=positive (covers E3, E5)

T6: entry= empty string, mult=positive (covers E4, E5)

We used **all combinations** coverage criteria in the below functions because all characteristics must be used.

There are 4 blocks in key which are valid element, invalid element, null, and empty string.

```
Integer remove(Object key)
```

Key: {E1: valid, E2: invalid, E3: null, E4: empty string}

T1: key = valid element (covers E1)

T2: key = invalid element (covers E2)

T3: key = null (covers E3)

T4: key = empty string (covers E4)

There are 4 blocks in key which are valid element, invalid element, null, and empty string.

```
int removeAll(T key)
```

Key: {E1: valid, E2: invalid, E3: null, E4: empty string}

T1: key = valid element (covers E1)

T2: key = invalid element (covers E2)

T3: key = null (covers E3)

T4: key = empty string (covers E4)

Method	Params	Returns	Values	Exception	Ch ID	Characteristics	Covered by
Intersect	other	Multiset	Multiset		C9	The intersection of those two lists.	
totalCount	state	int	Integer		C10	Total elements in hash.	
toList	state	list	List		C11	Hash elements into list.	
toString	state	String	String		C12	A string representation of the unsorted multiset like collection.	
equals	o	Boolean	True or false		C13	Compares list with list.	
hashCode	State	integer	Integer	IllegalStateException	C14	Throws an exception.	

ID	Characteristics	Intersect	totalCount	toList	toString	equals	hashCode	Partitions
C9	The intersection of those two lists.	X						{true,false}
C10	Total elements in hash.		X					{true,false}
C11	Hash elements into list.			X				{true,false}
C12	A string representation of the unsorted multiset like collection.				X			{true,false}
C13	Compares list with list.					X		{true,false}
C14	Throws an exception.						X	{true,false}

We used **all combinations** coverage criteria in the below functions because all characteristics must be used.

Intersect:

There are 3 blocks in key which are strings, integers, and empty string.

Collection, multiset: {E1: strings, E2: integers, E3: empty string}

T1: Collection, multiset = Strings (covers E1)

T2: Collection, multiset = integers (covers E2)

T3: Collection, multiset = empty strings (covers E3)

totalCount():

There are 4 blocks in key which are strings, integers, and empty string.

Collection, multiset: {E1: strings, E2: integers}, E3: empty, E4: null

T1: Collection, multiset = Strings (covers E1)

T2: Collection, multiset = integers (covers E2)

T3: empty hash (covers E3)

T4: null object (covers E4)

toList:

There are 2 blocks in key which are objects, empty.

{E1: empty, E2: objects}

T1: empty hash (covers E1)

T2: objects (covers E2)

toString:

The hash may be empty, may have strings, may have doubles.

There are 3 blocks in key which are strings, doubles, or empty.

{E1: strings, E2: doubles, E3: empty}

T1: strings (covers E1)

T2: doubles (covers E2)

T3: empty (covers E3)

Equals:

There are 3 blocks in key which are strings, integers, or null.

Collection, multiset: {E1: strings, E2: integers}, E3: null

T1: Collection, multiset = Strings (covers E1)

T2: Collection, multiset = integers (covers E2)

T3: null object (covers E3)

hashCode:

Throws an `IllegalStateException`

Finished after 0.375 seconds

Runs: 51/51 Errors: 0 Failures: 1

▼ Multiset_HashMapImplTest [Runner: JUnit 5] (0.208 s)

- removeMultiTest1() (0.035 s)
- removeMultiTest2() (0.002 s)
- removeMultiTest3() (0.008 s)
- removeMultiTest4() (0.002 s)
- removeMultiTest5() (0.002 s)
- addEntryTest1() (0.001 s)
- addEntryTest2() (0.003 s)
- addEntryTest3() (0.002 s)
- addEntryTest4() (0.004 s)
- addEntryTest5() (0.000 s)
- toListTest1() (0.002 s)
- toListTest2() (0.002 s)
- addAllArrayTest1() (0.054 s)
- addAllArrayTest2() (0.007 s)
- addAllArrayTest3() (0.006 s)
- addAllArrayTest4() (0.006 s)
- addAllArrayTest5() (0.005 s)
- totalCountTest1() (0.001 s)
- totalCountTest2() (0.002 s)
- totalCountTest3() (0.001 s)
- totalCountTest4() (0.001 s)
- totalCountTest5() (0.001 s)
- addTest2() (0.001 s)
- addTest3() (0.001 s)
- addTest4() (0.001 s)
- addTest5() (0.001 s)
- addTest6() (0.001 s)
- addTest() (0.001 s)
- equalsTest1() (0.001 s)
- equalsTest2() (0.000 s)
- equalsTest3() (0.001 s)
- removeTest1() (0.001 s)
- removeTest2() (0.001 s)
- removeTest3() (0.001 s)