**Software Testing**

**Assignment#1**

**Nada Mohamed Fathy Omar** [20186031](mailto:s.makady@fci-cu.edu.eg)

**Ayat Hany Ahmed**  20186007

**Task#2**

**Class#1**

**(de.tilman\_neumann.util.StringUtil )**

* **Method1(repeat)**:
* **Partitions**:

**\*For string s we have 3 partitions:**

* B1:empty string
* B2:null string
* B3:non empty string

\***For output :**

* C1:empty string
* C2:null string
* C3: repeated string

**\*For integer n(number of repetition) we have 3 partitions:**

* D1: negative(<0)
* D2:zero(=0)
* D3: positive(>0)
* **Boundaries**:

\***For string s we have**

* There are no upper bounds for s and txt (i.e., no upper limit on their length)
* The lower bound on their length is 0.

**\*For output**

* a string resulting from n repetitions of s
* null if n <= 0 or s=null or empty if s= empty string

\***For integer n**

* valid for n>0, invalid for n<=0
* **The chosen coverage criteria with complete explanation:**
* we used ACOC because we had used all the combination of partitions from all characteristics, that results 3\*3=9 test Cases :
* t1: s=” “ , n=positive (covers B1,C1,D3)
* t2: s=” “ , n=negative (covers B1,C2,D1)
* t3: s=” “ , n=zero (covers B1,C2,D2)
* t4: s= null , n=positive (covers B2,C2,D3)
* t5: s=null, n=negative (covers B2,C2,D1)
* t6: s=null, n=zero (covers B2,C2,D2)
* t7: s=non empty string , n=positive (covers B3,C3,D3)
* t8: s=non empty string, n=negative (covers B3,C2,D1)
* t9: s=non empty string, n=zero (covers B3,C2,D2)
* **Method2(formatleft)**:
* **Partitions**:

**\*For string s we have 5 partitions**:

* B1:empty string
* B2:null string
* B3:non empty string (same size as mask)
* B4:non empty string (> mask)
* B5:non empty string (< mask)

**\*For output :**

* C1:s left-aligned in mask
* C2:first string s
* C3: second string mask
* C4:empty string

\***For string mask we have 5 partitions:**

* D1: empty string
* D2: null string
* D3: non empty string (same size as s)
* D4:non empty string (> s)
* D5:non empty string (< s)
* **Boundaries**:

\***For string s we have:**

* There are no upper bounds for s and txt (i.e., no upper limit on their length)
* The lower bound on their length is 0.

\***For output:**

* s left-aligned in mask or first string s or second string mask
* empty string if s= empty string and mask = empty string or s,mask =null

\***For string mask:**

* There are no upper bounds for s and txt (i.e., no upper limit on their length)
* The lower bound on their length is 0.
* **The chosen coverage criteria with complete explanation:**
* we used ACOC because we had used all the combination of partitions from all characteristics, that results 5\*5=25testCases:
* t1: s=” “ , mask=non empty string (covers B1,C3,D3,D4,D5)
* t2: s=” “ , mask=null (covers B1,C4,D2)
* t3: s=non empty string ,mask=” “ (covers B3,B4,B5,C2,D1)
* t4: s=” “ , mask=” “ (covers B1,C4,D1)
* t5: s= null , mask=null (covers B2,C4,D2)
* t6: s= null , mask=empty (covers B2,C4,D1)
* t7: s=null, mask =non empty string (covers B2,C3,D3,D4,D5)
* t8: s= non empty string, mask =null (covers B3,B4,B5,C2,D2)
* t9: s=non empty string ,mask =non empty string(same size) (covers B3,C2,D3)
* t10: s=non empty string, mask =non empty string (s>mask) (covers B4,C2,D5)
* t11: s=non empty string, mask =non empty string (s<mask) (covers B5,C1,D4)

**\*\*Note:**

-The other test cases are not logic

* **Method3 (formatright)**:
* **Partitions**:

\***For string s we have 5 partitions:**

* B1: empty string
* B2: null string
* B3:non empty string (same size as mask)
* B4:non empty string (> mask)
* B5:non empty string (< mask)

\***For output :**

* C1: s right-aligned in mask
* C2: first string s
* C3: second string mask
* C4: empty string

\***For string mask we have 5 partitions:**

* D1: empty string
* D2: null string
* D3: non empty string (same size as s)
* D4:non empty string (> s)
* D5:non empty string (< s)
* **Boundaries**:

\***For string s we have:**

* There are no upper bounds for s and txt (i.e., no upper limit on their length).
* The lower bound on their length is 0.

**\*For output:**

* right-aligned in mask or first string s or second string mask.
* empty string if s ,mask = empty string or

s , mask =null .

\***For string mask:**

* There are no upper bounds for s and txt (i.e., no upper limit on their length).
* The lower bound on their length is 0.
* **The chosen coverage criteria with complete explanation:**
* we used ACOC because we had used all the combination of partitions from all characteristics, that results 5\*5=25 test Cases:
* t1: s=” “ , mask=non empty string (covers B1,C3,D3,D4,D5)
* t2: s=” “ , mask=null (covers B1,C4,D2)
* t3: s=non empty string ,mask=” “ (covers B3,B4,B5,C2,D1)
* t4: s=” “ , mask=” “ (covers B1,C4,D1)
* t5: s= null , mask=null (covers B2,C4,D2)
* t6: s= null , mask=empty (covers B2,C4,D1)
* t7: s=null, mask =non empty string (covers B2,C3,D3,D4,D5)
* t8: s= non empty string, mask =null (covers B3,B4,B5,C2,D2)
* t9: s=non empty string , mask =non empty string(same size) (covers B3,C2,D3)
* t10: s=non empty string, mask =non empty string (s>mask) (covers B4,C2,D5)
* t11: s=non empty string, mask =non empty string (s<mask) (covers B5,C1,D4)

**\*\*Note:**

-The other test cases are not logic

* **Report for all testcases**:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Params** | **Returns** | **Values** | **Exception** | **Ch ID** | **Character -istic** | **Covered by** |
| **Repeat** | String s  Int n | string | Repeated string, empty string ,null string. |  | C1 | a string resulting from n repetitions of s, or null if n <= 0 or s=null or empty string if n>0 or s=empty string |  |
| **formatleft** | String s  String mask | string | Empty string,s left-aligned into a mask,first string ,second string |  | C2 | Return left-aligned string or null or empty string |  |
| **formatright** | String s  String mask | string | Empty string,sright-aligned into a mask,firststring ,second string |  | C3 | Return left-aligned string or null or empty string |  |

* **We put which methods relevant for each characteristic :**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Character -istic** | **Repeat** | **FormatLeft** | **FormatRight** | **Partitions** |
| C1 | a string resulting from n repetitions of s, or null if n <= 0 or s=null or empty string if n>0 or s=empty string | x |  |  | {true,false} |
| C2 | Return left-aligned string or null or empty string |  | x |  | {true,false} |
| C3 | Return right-aligned string or null or empty string |  |  | x | {true,false} |

**Class#2**

**(de.tilman\_neumann.util. Multiset\_HashMapImpl)**

* **Method1(addWithOneArgument)**:
* **Partitions**:

\***For T entery we have 5 partations:**

* A1: string
* A2:integer
* A3: double
* A4: null
* A5: empty string

\***For output :**

* C1: 0 if this element added as first time at hash table
* C2: >0 if this elementadded but not first time at hash table.
* **Boundaries**:

\***For T entry we have**

* There are no limitation for the input

\***For output**

* The return value will be >=0
* **The chosen coverage criteria with complete explanation:**
* we used BCC because we choose to use the base choice for each characteristic , that results 5 testCases:
* t1: T entry =string(base case/happy path) (covers A1 ,C1/C2)
* t2:T entry=int (covers A2,C1/C2)
* t3:T entry=double (covers A3,C1/C2)
* t4: T entry=null (covers A4,C1/C2)
* t5: T entry=empty string (covers A5,C1/C2)
* **Method2 (addWithTwoArgument)**:
* **Partitions**:

\***For T entry we have 5 partitions:**

* A1: string
* A2:integer
* A3: double
* A4: null
* A5: empty string

\***For integer mult:**

* B1: negative
* B2: zero
* B3: positive

**\*For output:**

* C1: zero if this element added as first time at hash table
* C2: >zero if this element added but not first time at hash table.
* **Boundaries**:

\***For T entry we have :**

* There are no limitation for the input

\***For output:**

* The return value will be >=0

\***For integer mult:**

* >= 0 is valid < 0 invalid
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 7testCases:
* t1: T entry =string ,intmult =positive (base case/happy path) (covers A1 ,B3,C1/C2)
* t2:T entry=string ,intmult =negative (covers A1,B1,C1)
* t3:T entry=string ,intmult =0 (covers A1,B2,C1)
* t4:T entry=int ,intmult =positive (covers A2,B3,C1/C2)
* t5:T entry=double ,intmult =positive (covers A3,B3,C1/C2)
* t6:T entry=null ,intmult =positive (covers A4,B3,C1/C2)
* t7:T entry=empty string ,intmult =positive (covers A5,B3,C1/C2)
* **Method3 (addAll(Multiset<T> other))**:
* **Partitions**:

**\*For Multiset we have 6 partitions:**

* A1: Multiset ofempty string element
* A2: Multiset ofstring element
* A3: Multiset ofinteger element
* A4: Multiset ofnull element
* A5: Multiset ofdouble element
* A6: Multiset ofdifferent types of elements

\***For output:**

* The function is void (there is no output).
* **Boundaries**:

\***For Multiset we have :**

* There are no limitation for the input
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 6testCases:
* t1: multiset is collection of string elements (base case/happy path) (covers A2)
* t2:multiset is collection of empty string elements (covers A1)
* t3:multiset is collection of integer elements (covers A3)
* t4:multiset is collection of null elements (covers A4)
* t5:multiset is collection of double elements (covers A5)
* t6:multiset is collection of different types of elements (covers A6)
* **Method4 (addAll(Collection<T> values)))**:
* **Partitions**:

\***For collection we have 6 partitions:**

* A1: Collectionof empty string element
* A2: Collectionof string element
* A3: Collectionof integer element
* A4: Collectionof null element
* A5: Collectionof double element
* A6: Collectionofdifferent types of elements

\***For output:**

* The function is void (there is no output).
* **Boundaries**:

\***For collection we have :**

* There are no limitation for the input
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 6 testCases:
* t1: collection is collection of string elements (base case/happy path) (covers A2)
* t2: collection is collection of empty string elements (covers A1)
* t3: collection is collection of integer elements (covers A3)
* t4: collection is collection of null elements (covers A4)
* t5: collection is collection of double elements (covers A5)
* t6: collectionis collection of different types of elements (covers A6)
* **Method5 (addAll(T[] values)))**:
* **Partitions**:

\***For objectArray we have 6 partitions:**

* A1: objectArrayof empty string element
* A2: objectArrayof string element
* A3: objectArrayof integer element
* A4: objectArrayof null element
* A5: objectArrayof double element
* A6: objectArrayofdifferent types of elements

\***For output:**

* The function is void (there is no output).
* **Boundaries**:

\***For objectArraywe have :**

* There are no limitation for the input
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 6 testCases:
* t1: objectArrayis collection of string elements (base case/happy path) (covers A2)
* t2: objectArrayis collection of empty string elements (covers A1)
* t3: objectArrayis collection of integer elements (covers A3)
* t4: objectArrayis collection of null elements (covers A4)
* t5: objectArrayis collection of double elements (covers A5)
* t6: objectArrayis collection of different types of elements (covers A6)
* **Method6 (removeWithOneArgument)**:
* **Partitions**:

\***For T entry we have 6 partitions:**

* A1: string
* A2: integer
* A3: double
* A4: null
* A5: empty string
* A6:element not in hashTable

\***For output :**

* C1: return >=1 if this element is found
* C2: return null if this elements not found
* **Boundaries**:

\***For T entry we have**

* There are no limitation for the input

\***For output**

* The return value will be >=1 if found and return null if not found.
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 6testCases:
* t1: T entry =string(base case/happy path) (covers A1 ,C1)
* t2:T entry=int (covers A2,C1)
* t3:T entry=double (covers A3,C1)
* t4: T entry=null (covers A4,C1)
* t5: T entry=empty string (covers A5,C1)
* t6: T entry(any partition) (covers A6,C2)
* **Method7 (removeWithTwoArgument)**:
* **Partitions**:

\***For T entry, we have 6 partitions:**

* A1: string
* A2: integer
* A3: double
* A4: null
* A5: empty string
* A6: element not on hashTable

**\*For integer mult:**

* B1: negative
* B2: zero
* B3: positive

\***For output:**

* C1: return >=1 if this element is found and removing it.
* C2: return null if this element is not found.
* C3: return >=1 if this element is found and not removing it.
* **Boundaries**:

**\*For T entry we have :**

* There are no limitation for the input

\***For output:**

* The return value will be >=1 if found and return null if not found.

**\*For integer mult:**

* >= 0 is valid < 0 invalid
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 8 testCases:
* t1: T entry =string,intmult =positive (base case/happy path) (covers A1,B3,C1)
* t2:T entry=string,intmult =negative (covers A1,B1,C3)
* \*\*for t2 : this test case is failed because the result of this case is to removing the element if exists or not removing it if dose not exist but the actual result was triples this elements and this wrong for the expected result that we want.
* t3:T entry=string,intmult =0 (covers A1,B2,C3)
* t4:T entry=int,intmult =positive (covers A2,B3,C1)
* t5:T entry=double,intmult =positive (covers A3,B3,C1)
* t6:T entry=null,intmult =positive (covers A4,B3,C1)
* t7:T entry=empty string,intmult =positive (covers A5,B3,C1)
* t8:T entry(any partition),intmult=positive (covers A6,B3,C2)
* **Method8 (removeAll(T key))**:
* **Partitions**:

\***For T key we have 6 partitions:**

* A1: string
* A2: integer
* A3: double
* A4: null
* A5: empty string
* A6:element not in hashTable

**\*For output :**

* C1: return >=1 if this element is found
* C2: return 0 if this elements not found
* **Boundaries**:

**\*For T entry we have**

* There are no limitation for the input

\***For output**

* The return value will be >=1 if found and return 0 if not found.
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 6testCases:
* t1: T key =string(base case/happy path) (covers A1 ,C1)
* t2:T key =int (covers A2,C1)
* t3:T key =double (covers A3,C1)
* t4: T key =null (covers A4,C1)
* t5: T key =empty string (covers A5,C1)
* t6: T key (any partition) (covers A6,C2)
* **Method 9(totalCount):**
* **Partitions:**
* A1: hashTable with no elements
* A2: hashTable with element

**\*Output:**

* C1: return number of element if there is element in table
* C2: return zero if there is no element
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 2testCases:
* t1: hashTable has element (covers A2,C1)
* t2: hashTable has no element (covers A1,C2)
* **Method 10 (equals):**
* **Partitions:**
* A1: hashTable and object with the same size and element
* A2: hashTable and object with different sizes
* A3: hashTable and object with same sizes

**\*Output:**

* C1: return true if the same size and element
* C2: otherwise return false
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 3testCases:
* t1: object O = collection of element same size and same element (covers A1,C1)
* t2: object O = collection of element with different size of hashTable (covers A2,C2)
* t3: object O = collection of element with same size within hashTable (covers A3,C2)
* **Method 11 (intersect):**
* **Partitions:**
* A1: intersect with multiSet and hashTable
* A2: no intersect with multiSet and hashTable

**\*Output:**

* C1: return the intersected multiSet if there is intersect with multiSet and hashTable
* C2: otherwise return null
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 2 testCases:
* t1: multiset has intersection with hashTable(covers A1,C1)
* t2: multiset has no intersection with hashTable(covers A2,C2)
* **Method 12(toString):**
* **Partitions:**
* A1: hashTables has element
* A2: if hashTable is empty

\***Output:**

* C1: return all hashTables as string
* C2: return { } if there is no element
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 2 testCases:
* t1: hashTable has element (covers A1,C1)
* t2: hashTable is empty (covers A2,C2)
* **Method 13 (toList):**
* **Partitions:**
* A1: hashTables has element
* A2: if hashTable is empty

**\*Output:**

* C1: return all hashTables as list
* C2: otherwise return empty list
* **The chosen coverage criteria with complete explanation:**
* We used BCC because we choose to use the base choice for each characteristic, that results 2testCases:
* t1: hashTable has element (covers A1,C1)
* t2: hashTable is empty (covers A2,C2)
* **Report for all testCases:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Params** | **Returns** | **Values** | **Exception** | **Ch ID** | **Character -istic** | **Covered by** |
| **add** | T entry | int | Int number >= 0 |  | C1 | Add if there is element or not |  |
| **add** | (T entry, intmult) | int | Int number >= 0 |  | C2 | Add if there is element or not |  |
| **addAll** | (Multiset<T> other) |  |  |  |  |  | C2 |
| **addAll** | (Collection<T> values) |  |  |  |  |  | C1 |
| **addAll** | (T[] values) |  |  |  |  |  | C1 |
| **remove** | (Object key) | int | >=1 or null |  | C3 | Remove element if exist |  |
| **remove** | (T key, intmult) | int | >=1 or zero |  | C4 | Remove element if exist |  |
| **removeAll** | (T key) | int | >=1 or zero |  |  |  | C3 |
| **intersect** | (Multiset<T> other) | Multiset<T> | return the intersected multiSet |  | C5 | Return intersect multiset |  |
| **totalCount()** | Use hashTable | int | Number of elemnts in hashTable |  | C6 | Return Number of elemnts in hashTable |  |
| **toList()** | Use hashTable | List<T>List | hashTable as list |  | C7 | Return hashTable as list |  |
| **toString()** | Use hashTable | string | hashTable as string or { } |  | C8 | hashTable as string or { } |  |
| **equals** | (Object o) | boolean | True or false |  | C9 | Return true if the object and hashTable is equal otherwise false |  |
| **hashCode()** | Use hashTable | int | Hash number or IllegalStateException |  | C10 | Return Hash number |  |
| IllegalStateException | C11 | Throw exception as SortedMultisets are not ready to be used in hash structures |  |

* **We put which methods relevant for each characteristic :**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ID** | **Character -istic** | **add** | **add** | **addAll** | **addAll** | **addAll** | **remove** | **remove** | **removeAll** | **partition** |
| **C1** | Add if there is element or not | x |  |  | x | x |  |  |  | {true,false} |
| **C2** | Add if there is element or not |  | x | x |  |  |  |  |  | {true,false} |
| **C3** | Remove element if exist |  |  |  |  |  | x |  | x | {true,false} |
| **C4** | Remove element if exist |  |  |  |  |  |  | x |  | {true,false} |
|  |  | **intersect** | **totalCount** | **toList** | **toStrig** | **equals** | **hashCode** |
| **C5** | Return intersect multiset | x |  |  |  |  |  |  |  | {true,false} |
| **C6** | Return Number of elemnts in hashTable |  | x |  |  |  |  |  |  | {true,false} |
| **C7** | Return hashTable as list |  |  | x |  |  |  |  |  | {true,false} |
| **C8** | hashTable as string or { } |  |  |  | x |  |  |  |  | {true,false} |
| **C9** | Return true if the object and hashTable is equal otherwise false |  |  |  |  | x |  |  |  | {true,false} |
| **C10** | Return Hash number |  |  |  |  |  | x |  |  | {true,false} |
| **C11** | Throw exception as SortedMultisets are not ready to be used in hash structures |  |  |  |  |  | x |  |  | {true,false} |

\*\***Node: Java package with two test classes with name Test in the same file**